2023 IEEE Symposium Series on Computational Intelligence (SSCI)
Mexico City, Mexico. December 5-8, 2023

Breaking the Cycle: Exploring the Advantages of Novel Evolutionary Cycles

Braden N. Tisdale
BONSAI Lab
Department of Computer Science and Software Engineering
Auburn University
Auburn, Alabama, United States of America
bnt0008@auburn.edu

Daniel R. Tauritz
BONSAI Lab
Department of Computer Science and Software Engineering
Auburn University
Auburn, Alabama, United States of America
dtauritz@acm.org

***Abstract*—There have been many different forms of evolutionary algorithms (EAs) designed by humans over the past 50 years, with many variants optimized for specific classes of problems. As computational resources grow, the automated design of EAs has become an increasingly viable method for improving performance. However, many components of EAs have been treated as largely immutable, for both human and automated designers, dramatically constraining design space. In particular, the evolutionary cycle (the repeating pattern of reproduction and survival) has little or no differences between most popular forms of EA. In a previous paper, we proposed a technique for automatically designing evolutionary cycles using directed graphs, greatly increasing the explorable design space. In this paper, we showcase an improved representation and evolutionary process, provide preliminary experiments demonstrating that EAs produced by this process can outperform those with a traditional cycle, and explore the phenotype landscape to show that the new space explored by our technique may contain better EAs than traditional cycles allow.**

***Index Terms*—automated design, hyperheuristics, evolutionary algorithms**

I. Introduction

Through its history, the evolutionary computation (EC) community has utilized a minimalist, streamlined representation for the evolutionary process: nearly every instance is based on an evolutionary cycle consisting of parent selection, reproduction, and survival selection. In previous papers [1], [2] it was proposed that this traditional structure may not always be an optimal choice, and it was demonstrated that a methodology for evolving novel evolutionary cycles was competitive with parameter tuning in its ability to produce performant EAs. In this paper, we present an improved methodology for the automatic design of evolutionary cycles, and explore the fitness landscape of cycles produced by this process. We believe the results strengthen the assertion that the traditional evolutionary cycle is often suboptimal for specific problem classes.

II. Related Work

Parameter tuning has a rich history in the EC community, being an essential factor of EA performance [3]–[9]. More recently, the concept of meta-optimization has been extended to the broader concept of automatically designing EAs or components used in EAs. This includes the creation of individual operators such as selection or recombination [10]–[12]. Other works focus on the order in which operations are conducted, or act as more problem-specific tuning algorithms that choose from available components [13]–[17]. These works typically rely on linear orders of operations, leading to simplistic population structures that are largely identical to traditional EA cycles. A linear sequence of parent selection, reproduction, and survival selection can be found unmistakably in nearly every EA. Some works, however, do implement more complex structures that diverge significantly from normal populations [2], [18]–[22]. There are existing techniques for applying genetic programming to the optimization of graphs [23], [24]. However, their generic representations have shortcomings when representing EAs, as discussed in [1].

A desire to explore a broader range of evolutionary cycles served as motivation for previous work using a directed graph representation for evolutionary processes [1]. Experiments with that representation showed novel evolutionary cycles were capable of competitive performance with traditional EAs. To enable further exploration, we have made numerous changes to that representation in order to explore a broader range of designs. In the process, we have also resolved several shortcomings of the previous approach.

That previous methodology used a variable-length vector for its nodes. Recombination was conducted by 1-point crossover that produced an output vector with size bounded by the parents' sizes. This may have been biased towards the creation of shorter vectors and decoupled genes from loci, which is problematic as the recombination operation depended on the index of genes in the vector. The mutation operation performed poorly, which we speculate to be a result of self-adaptivity involved in the mutation process. The self-adapted genes may have adapted poorly, since a solution's fitness may not accurately represent the performance of a variation operator, especially when mutation is not always performed. The new algorithm foregoes self-adaptivity to allay these concerns. The old representation required that the entire graph be strongly-connected, and that every node participate in execution. This precluded distinct subpopulations, and prevented some strate-

This work was supported by Los Alamos National Laboratory via the Cyber Security Sciences Institute under subcontract 570204.

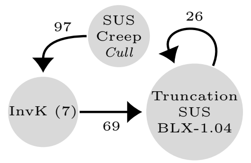

Fig. 1. The best graph from one of the runs in our experiment, which could not have been represented with the previous methodology. The EA it represents initializes 97 random solutions, samples parents using stochastic universal sampling, produces 97 children using creep mutation, selects 69 of those children using inverse k-tournaments (sometimes called death selection) with *k*=7, selects 26 of those using truncation, and then behaves identically to a (26 + 1)-EA using stochastic universal sampling and blend crossover. Only information that functionally affects the algorithm represented by the the graph is shown.

gies such as nodes that only execute once, prior to a cycle. Fig. 1 shows such a graph, which was the best graph found during one of the runs of the experiment presented here.

III. Methodology

A. Primitives

The first step in creating a generic representation for an EA is to identify what the basic primitives for an EA actually are. We formulate them as selection operators and variation/reproduction operators, as well as some mechanism for transferring individuals between them. This formulation lends itself to a graphical representation (specifically, a directed graph), where the operators are encoded in nodes and the flow of individuals is encoded in the edges. The algorithms that can be represented by such a graph include many that are structurally very different from traditional EAs, but the graphs constructed with this primitive set produce search algorithms that are broadly analogous to EAs.

The selection operators are uniform-random, fitness-proportional (parent selection only), k-tournament(parent selection only), stochastic universal sampling (parent selection only), truncation (survival selection only), and inverse k-tournaments (sometimes known as death selection, survival selection only). Reproduction operators depend on the representation of the underlying problem; our experiments involved floating-point vectors, and the operators we have implemented are uniform crossover, whole arithmetic crossover (averaging), blend crossover (BLX-α), reset mutation, and creep mutation. The crossovers are performed on two parents and produce one child, while the mutations are performed on one parent and produce one child. No reproduction operators modify the parent (mutation creates a new individual).

B. Genotype

The first component of the genotype is a fixed-length vector of nodes, avoiding issues previously mentioned with variable-length vectors, but requires tuning the number of nodes for each problem class. In order to ensure all nodes can execute regardless of whether population size must be increased or decreased, each node simultaneously encodes an algorithm for each of three separate operations: parent selection, reproduction, and survival selection. Each node also encodes any necessary parameters for its operations, as well as three boolean flags *cull*, *shuffle*, and *segregate*, which change the behavior of the node and will be detailed in Subsection III-C. Our graphical representation also requires edges to represent the flow of individuals between its nodes. All edges in our representation are unidirectional, and each one encodes a positive integer representing the number of individuals that will flow across it (the edge's "weight"), a simplification from the previous work which included some percentage-based edge weights. When a single node has multiple out-edges, these edges have a fixed total order, which is defined independently of any other parameters (i.e., each edge has a unique index assigned arbitrarily). This ordering determines how individuals are partitioned across out-edges, which will be explained in greater detail in the next subsection.

C. Execution

There are three possible behaviors when a node is executed, and the choice of which is executed depends on the sum of the weights on its inbound edges (the insize), versus the sum of the weights on its outbound edges (the outsize). If a node's insize is less than its outsize, then in order to reach its outsize, individuals must be created. Conversely, if its insize is greater, it must remove individuals. Based on this comparison, a node will either perform parent selection and reproduction, or survival selection. If these values are equal, the node will produce one individual by reproduction then conduct survival selection; otherwise, some graphs (such as one node with a loop) would never generate new solutions. An exception is the first boolean flag, *cull*, which causes a node to perform parent selection and reproduction, then discard its input population. This behavior simulates a generational (μ, λ)-EA.

After execution, the resulting population is partitioned across the node's out-edges. The edges are ordered by their arbitrary indices, while the order of the individuals is defined by the remaining two boolean flags. If the *shuffle* flag is set, the output population is shuffled randomly. Otherwise, the population is sorted in descending order of fitness. The last flag, *segregate*, is only relevant for nodes that perform reproduction and do not have the *cull* flag set. If it is set, all individuals that were part of the input are ordered before any individuals that were generated by reproduction. These two subsets are then shuffled or sorted separately. This allows for behavior similar to traditional EAs which typically mutate offspring (but not existing adults) before survival selection.

In most graphs, only a subset of the nodes in the genotype vector will be executed. The nodes which will be executed, and the order in which they are executed, depends on the graph's edges. The first node to be executed is called the root, and it is chosen as the first node in the genotype vector with at least one out-edge. Any nodes that are reachable from the root will also be executed. The root, and the nodes reachable from it, induce a subgraph that we will call the executed graph (E-graph). As we will explain later, out methodology ensures that the E-graph has at least one cycle. Due to the fixed-length node vector, there will likely be nodes and edges not in the E-graph which are not executed (i.e., introns).

Nodes are ordered by a depth-first search (DFS) across the E-graph, similar to the DFS algorithm for topological sorting in a directed acyclic graph. However, as our representation is cyclic, no true topological sort can exist. Rather, our algorithm produces a topological sort for a spanning tree of the E-graph. The search begins at the root. DFS proceeds (preferring lower-index nodes when possible) until it reaches a node whose neighbors are all already in the search path or the ordering. This node is prepended to the ordering and the search continues from its predecessor. This continues until every node is in the ordering.

Execution iterates through this ordering and executes each node one-by-one. We will refer to this as one pass through the E-graph. On reaching the end of a pass, execution begins again from the root. During the first pass, the out-edges of nodes that have not yet been executed do not have any individuals to send to their target node. This happens at any edges $\{(u, v)|u_{index} \geq v_{index}\}$. We call these backwards edges, and they serve as the random initialization for the search. As an example, the loop with weight 26 in Fig. 1 is a backwards edge. The E-graph's ordering is designed to minimize the number of such backwards edges. Prior to the first pass, these edges are filled with randomly-initialized individuals according to their weight. The root may also have no incoming edges, in which case it will randomly initialize a number of individuals equal to the maximal edge weight in the E-graph, then execute as normal.

D. Representable Algorithms

This methodology can represent several different canonical types of EAs. Since the E-graph is guaranteed to have at least one cycle, execution can be continued indefinitely as in a typical EA, supporting arbitrary termination criteria. Traditional $(\mu + \lambda)$- or (μ, λ)-EAs are trivial to construct. A cycle consisting of a single node with a loop can act as a simple hill climber. As the only requirement for inclusion in the E-graph is to be reachable from the root, there may be multiple strongly-connected components serving as separate evolutionary cycles. They could be entirely disconnected except for being reachable from the root, essentially forming parallel searches, or there may be unidirectional flow between them. These can roughly approximate simple island models.

The ability to represent canonical algorithms is useful since it can be used to investigate if these standard forms are good choices for solving a given problem. However, the intent of this methodology is to search areas of algorithmic space that have not been considered by the EC community. Notably, for types of EAs that our methodology can exactly represent, our search space is a strict superset over that of traditional parameter tuning, which poses a more difficult search problem but may permit superior designs. Our methodology has the potential to produce novel algorithms that behave very differently from existing techniques. While these may potentially be superior EAs for specific problems or problem classes, they also present an opportunity for insights about overall EA behaviors. They may contain components that can be reused elsewhere, or which indicate the usefulness of specific strategies on that problem class.

TABLE I
META-EA PARAMETERS

Name	Function	Value
μ	population size	50,000
λ	generation size	5,000
mutation rate	probability to mutate	0.5
parent select	parent selection algorithm	5-tournament
survival select	survival selection algorithm	truncation
num nodes	number of nodes in each graph	10
max edge	maximal weight for a single edge	250
min density	min initialization edge density	0.05
max density	max initialization edge density	0.4
seg chance	prob a node segregates	0.5
shuf chance	prob a node shuffles	0.5
cull chance	prob a node culls	0.2

Values for selected meta-EA parameters.

E. Meta-EA

A meta-EA is responsible for optimizing these graphs for performance on a given problem. This meta-EA is based on a standard $(\mu + \lambda)$-EA. Our meta-EA is a form of hyper-heuristic, as it is searching through algorithmic space with a combinatorial representation. This search necessitates the inclusion of all the usual representation-specific components of an EA, namely initialization and variation operators. The meta-EA uses several parameters, shown in Table I. Values were selected by manual tuning.

1) Initialization: Each node is initialized with a uniform random selection of operators and any parameters for the chosen operators. Each node also has its boolean flags set to true or false according to configured probabilities. Edges are generated by the Erdős-Rényi-Gilbert model. Future work should investigate the impact of different random graph generators. The weights of each edge are uniform random and independent. The out-edges for each node are in the same order as their target nodes in the genotype vector.

After initialization, a repair function is run to ensure the graph is valid. If the graph has no edges, it adds one between two random nodes (which may be the same node). Then, if the E-graph does not already have a cycle, an edge is added from the last node in the ordering to the root node.

2) Variation: Reproduction during a run of the meta-EA involves crossover and mutation. After reproduction, the resulting graphs are repaired, as in initialization.

For crossover, a random choice is made between taking in-edges or out-edges. Crossover is then conducted normally on the genotype vectors, using either one-point or uniform crossover. Nodes are copied along with their incident edges of the chosen type. That is, if a parent contributes a node during crossover, and in-edges were chosen to be taken, its incident in-edges are also taken (or vice versa for out-edges).

TABLE II
TRADITIONAL EA PARAMETER SPACE

Parameter	Possible Values
μ & λ[a]	1, 2, 3, 5, 10, 25, 50, 75, 100, 175, 250, 375, 500, 750, 1000, 1750, 2500, 3750, 5000
mutation rate	0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1
locus rate[b]	0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1
exclusive mutation[c]	false, true
generational	false, true
parent select	FPS, SUS, Uniform Random, 2-, 5-, 10-, & 25-tournament
survival select	Truncation, Uniform Random, 2-, 5-, 10-, & 25-inverse-tournament
recombination	Uniform, Whole Arithmetic, BLX-(-0.25, 0, 0.25, 0.5, 0.75, 1, & 1.5)[d]
mutation	Reset, Creep (with 4 different creep rates)

[a]These have the same possible values, but vary independently
[b]Probability of mutation at each locus in the solution
[c]Whether mutation replaces or follows recombination
[d]Blend crossover with each of these 7 possible values for α

Mutation can perform several different operations. Each of these has a configurable probability, and geometric distributions are sampled using these probabilities during each mutation to determine the number of times each operation will occur. These operations, in the order they are performed, are: reinitializing a node (with no effect on incident edges); swapping the positions of two random nodes in the genotype vector (with a chance of also swapping their incident edges); adding new edges to the graph; removing existing edges from the graph; and shuffling the order of out-edges in a random node. In addition to these operations, there are mutation operations that are run on each individual node and edge with configurable probabilities. For edges, their weight is multiplied by a value sampled from a normal distribution with a mean of one. For nodes, each flag may be flipped, the chosen algorithm for each operation may be changed, and parameters for each algorithm may be changed.

IV. EXPERIMENT

Select functions defined by the BBOB test suite [25] will serve as benchmarks. We will compare the performance of EAs optimized for these benchmarks. The comparisons will be between our evolved graphs versus graphs found by random search (using the random initialization process for the meta-EA). We will also perform an exhaustive search over a large parameter space using a standard $(\mu + \lambda)$- or (μ, λ)-EA, to determine if our meta-EA and the graphs it produces outperform finely-tuned traditional EAs. The traditional EA parameter space searched through is shown in Table II. Invalid or redundant configurations were skipped, leading to 153.7 million configurations tested.

For clarity, for the remainder of this paper we will use the following vocabulary. We will say *meta-run* to mean an independent run of the meta-EA or the random search to generate EAs, whereas just *run* refers to an independent run of an EA on a benchmark function, and *evaluations* are executions of a benchmark function. A meta-run contains many runs, and each run contains many evaluations.

TABLE III
PERFORMANCE OF META-EA VS RANDOM SEARCH

Benchmark	Runs Counted	p	Algorithm	μ	σ
f_1	150	0.5904	Meta-EA	1371	161
			Random	1354	41
	135	0.0000	Meta-EA	1112	20
			Random	1159	21
f_3	150	0.9850	Meta-EA	4998	1462
			Random	5003	644
	135	0.0000	Meta-EA	2985	347
			Random	3623	240
f_7	150	0.3045	Meta-EA	3637	979
			Random	3874	785
	135	0.0005	Meta-EA	1211	158
			Random	1340	103

Each problem instance has a target output value (the global optimum plus 10^{-8}), and each EA is run until it finds a solution that reaches that target. Each EA is run 33 times, and its fitness is the inverse of the mean number of evaluations taken across its 30 shortest runs[1]. To save computation, runs that surpass $5 * 10^6$ evaluations are considered failures, and an EA that has more than 3 such runs is assigned arbitrarily poor fitness and not evaluated any further. Each run initializes a new random problem instance to avoid overfitting.

We use three benchmarks from the BBOB specifications: 5-dimensional Sphere (f_1), 3-dimensional Rastrigin (f_3), and 3-dimensional Step Ellipsoidal (f_7). There were 30 meta-runs on each benchmark of the meta-EA and the random search, with five million graphs per meta-run. f_1 was chosen as a very simple problem that encourages rapid convergence. f_7 was chosen as a problem which requires a focus on exploration and mutation due to having large areas in the search space with no gradient. f_3 was chosen as a more typical problem.

After a meta-run, the 64 best EAs of the five million evaluated are run 150 times to obtain more robust data for statistical analysis. As the fitness function used during a meta-run discarded the worst 10% of runs, our analysis will also discard the worst 10% of runs unless otherwise noted.

We categorize EAs based on the number of nodes and number of edges in their E-graph. For each unique $(num\ nodes, num\ edges)$ pair, the best EA seen during each meta-run is saved, as well as a count of how many EAs had that number of nodes and edges. This is similar to MAP-Elites [26], though it does not influence the search during a meta-run, it is simply for post-hoc analysis. Each of these EAs is run 150 times after a meta-run, similar to the overall best EAs. Note that these runs are separate from the runs of the overall best EAs, meaning values may differ slightly (e.g., in the following section, tables use different data than figures).

[1]In the previous paper on this technique [1], it was found that the resulting graphs had a much lower rate of failure than traditional EAs, but much worse performance during successful runs. In other words, since even a single failure was disastrous to fitness, using a simple mean placed an unintended emphasis on consistency. Discarding the worst 3 runs alleviates this and may help reduce sampling error by omitting outliers.

TABLE IV
COMPARISON OF BEST EAS FOUND

Benchmark	Algorithm	μ	σ
f_1	Meta-EA	1051	148
	Random	1103	155
	Traditional	1183	127
f_3	Meta-EA	2441	688
	Random	3229	777
	Traditional	3045	781
f_7	Meta-EA	958	214
	Random	1172	180
	Traditional	1277	876

V. RESULTS

30 meta-runs were conducted with both of the meta-EA and random search on each of the benchmark problems. Table III shows measures of central tendency for the average evaluation count of the best EA found during each meta-run ($n = 30$). "Runs Counted" refers to how many runs of each EA are used during analysis. p-values are from Welch's t-test. When considering the 135 best runs, our meta-EA outperformed random search by a statistically-significant margin: our meta-EA is, on average, superior to random search in its ability to produce high-performance graph EAs. This demonstrates that our graphical representation is conducive to iterative optimization, and that our meta-EA implementation succeeds in doing so. When counting all 150 runs, the meta-EA was indistinguishable from random search; this is not surprising since the meta-EA was optimizing for performance when discarding outliers, but it may indicate that EAs could be optimized for different levels of consistency.

Table IV shows similar measures for the best graph EAs found by the two algorithm, as well as the best traditional EAs found by the exhaustive parameter search. As opposed to Table III, this analyzes the performance of the best EAs found during the entire experiment (one per method per problem), discarding the 15 worst runs for each EA ($n = 135$). Pairwise Welch's t-tests show the EAs found by our meta-EA outperformed the other EAs on every problem ($p = 0.0051, 1.5 * 10^{-16}, 1.3 * 10^{-16}$, respectively). The EA found by random search was statistically better than the traditional EA on $f_1 (p = 7.4 * 10^{-6})$, but the difference was negligible for the $f_3 (p = 0.053)$ and $f_7 (p = 0.17)$ EAs. We believe it is notable that the random search and the traditional search each produced EAs with lower means on f_1 than on f_7, while the opposite is true for the meta-EA. f_1 can be rapidly solved using a very simple EA, whereas f_7 can be difficult to solve due to its landscape having large areas with no change in fitness. The meta-EA found an algorithm that overcomes the challenges of this particular problem very effectively.

VI. ANALYSIS

Phenotype maps are generated by selecting the best EA in each $(num\ nodes, num\ edges)$ cell across all 30 meta-runs of the algorithm. The maps stop at 25 edges in this paper, though the experiment did not limit graphs to 25 edges. All values

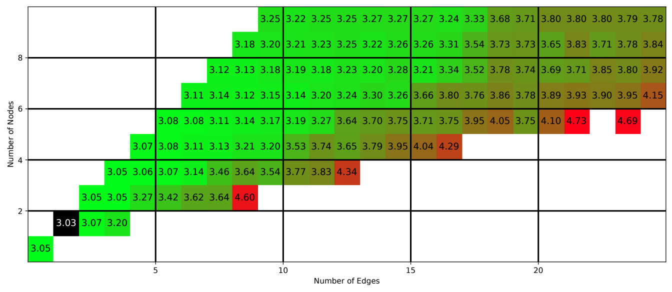

Fig. 2. Performance of best EAs found by the meta-EA on f_1 across 30 meta-runs. Values are the lowest mean evaluations out of all graphs in each cell, calculated using the 135 best out of 150 runs. Values are log_{10}.

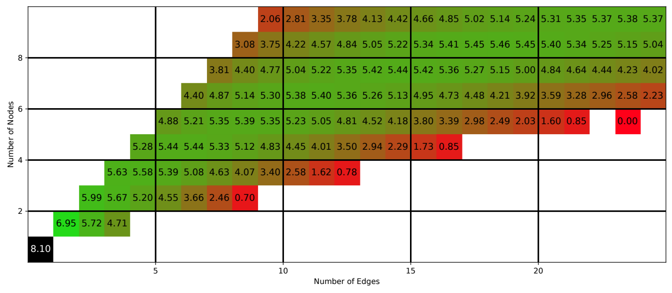

Fig. 3. Distribution of EAs generated on f_1, of $1.5 * 10^8$ generated in total.

are log_{10}. In any graph, $(1, 1)$ may represent either a $(\mu + 1)$-EA or a (μ, μ)-EA, with either recombination or mutation but not both. The further away from $(1, 1)$, the more complicated the represented EA. A traditional $(\mu + \lambda)$- or (μ, λ)-EA with both recombination and mutation can be represented as $(2, 3)$, $(3, 3)$, or $(3, 4)$.

The phenotype map for the meta-runs of our meta-EA on f_1 is seen in Fig. 2. Again, smaller values represent better performance. The best graph found overall is in $(2, 2)$, though it behaves as a $(\mu + 1)$-EA with a node serving to seed the search (structurally similar to Fig. 1, minus the first node). All of the best-performing graphs on f_1 follow this strategy, which is an intuitive result since the problem is symmetric and convex. This shows the potential of our methodology to produce algorithms that exploit characteristics of the underlying problem, and shows that (at least in this instance) it can consistently find such strategies. Notably, such a strategy was not representable under our old representation [1]. Fig. 3 shows the distribution for the number of graphs generated over the course of the 30 meta-runs of the meta-EA. Notably, the amount of graphs generated by the algorithm for each cell does not correlate very strongly with the fitness of graphs found in that cell. This could indicate our variation operators may bias new graphs towards simpler representations.

Perhaps most interesting is the phenotype map for f_7 as found by our meta-EA, seen in Fig. 4. This problem has large areas in the search space with no gradient. Thus, an EA must be able to avoid premature convergence in order to solve this problem consistently. This may explain why very simple graphs, which performed well on f_1, performed poorly

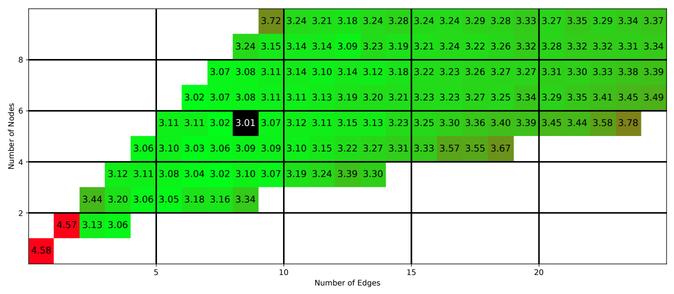

Fig. 4. Performance of best EAs found by the meta-EA on f_7.

here. Conversely, more complicated graphs performed very well here, even surpassing the minimum evaluations on f_1.

VII. Conclusion

Our experiments show that our meta-EA is capable of using a graphical representation to automatically design EAs with novel evolutionary cycles that can outperform traditional EAs. Our data also supports the hypothesis that cycles outside of the traditional design space are worth considering, may offer advantages over traditional EAs, and can be created (automatically or manually) to exploit problem-specific characteristics. Our approach can be utilized alongside or in lieu of traditional parameter tuning, leveraging a priori computation to find high-performance graphical EAs that can be reused indefinitely.

VIII. Future Work

Further analysis on trends in the graphs found by the meta-EA could discover commonly-occurring patterns. The trend of using complex seeding strategies before entering a $(\mu+1)$-EA-style cycle was noted on f_1, but other recurring strategies may exist. In addition, other dimensions in the design space (besides the number of edges and nodes in the E-graph) may offer more insight into the fitness landscape.

Further improvements to the representation may be made, namely in improving the variation operators so as to not bias so heavily towards simple graphs. Different random graph generators for initialization may also improve the meta-EA's performance. We also believe that the use of memetic optimization on numeric parameters could improve the performance of generated EAs.

References

[1] B. Tisdale, D. Seals, A. S. Pope, and D. R. Tauritz, "Directing evolution: The automated design of evolutionary pathways using directed graphs," in *2021 Genetic and Evolutionary Computation Conference (GECCO '21), July 10-14, 2021, Lille, France*. ACM, 2021, p. 9 pages.

[2] M. A. Martin and D. R. Tauritz, "Evolving black-box search algorithms employing genetic programming," in *Proceedings of the 15th Genetic and Evolutionary Computation Conference Companion*. New York, NY, USA: Association for Computing Machinery, 2013, pp. 1497–1504.

[3] D. J. Cavicchio, Jr., "Adaptive search using simulated evolution," University of Michigan, Ann Arbor, MI, USA, Tech. Rep., Aug. 1970.

[4] J. H. Holland, "Genetic algorithms and the optimal allocation of trials," *SIAM Journal on Computing*, vol. 2, no. 2, pp. 88–105, 1973.

[5] H.-G. Beyer and H.-P. Schwefel, "Evolution strategies - a comprehensive introduction," *Natural Computing*, vol. 1, no. 1, pp. 3–52, Mar. 2002.

[6] S. K. Smit and A. E. Eiben, "Comparing parameter tuning methods for evolutionary algorithms," in *2009 IEEE Congress on Evolutionary Computation*. IEEE Press, Jun. 2009, pp. 399–406.

[7] D. Fogel, L. Fogel, and J. Atmar, "Meta-evolutionary programming," in *Conference Record of the Twenty-Fifth Asilomar Conference on Signals, Systems & Computers*. Los Alamitos, CA, USA: IEEE Computer Society, Nov. 1991, pp. 540–545.

[8] T. Bäck and H.-P. Schwefel, "An overview of evolutionary algorithms for parameter optimization," *Evolutionary Computation*, vol. 1, no. 1, pp. 1–23, 1993.

[9] R. Ugolotti, L. Sani, and S. Cagnoni, "What can we learn from multi-objective meta-optimization of evolutionary algorithms in continuous domains?" *Mathematics*, vol. 7, no. 3, 2019. [Online]. Available: https://www.mdpi.com/2227-7390/7/3/232

[10] E. Smorodkina and D. Tauritz, "Toward automating EA configuration: the parent selection stage," in *2007 IEEE Congress on Evolutionary Computation*. IEEE Press, Sep. 2007, pp. 63–70.

[11] S. N. Richter and D. R. Tauritz, "The automated design of probabilistic selection methods for evolutionary algorithms," in *Proceedings of the 2018 Genetic and Evolutionary Computation Conference Companion*. Association for Computing Machinery, 2018, pp. 1545–1552.

[12] B. Edmonds, "Meta-genetic programming: Co-evolving the operators of variation," *Turkish Journal of Electrical Engineering and Computer Sciences*, vol. 9, no. 1, pp. 13–29, 2001.

[13] L. Dioşan and M. Oltean, "Evolutionary design of evolutionary algorithms," *Genetic Programming and Evolvable Machines*, vol. 10, no. 3, pp. 263–306, Sep. 2009.

[14] M. Oltean and C. Groşan, "Evolving evolutionary algorithms using multi expression programming," in *Advances in Artificial Life*, W. Banzhaf, J. Ziegler, T. Christaller, P. Dittrich, and J. T. Kim, Eds. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 651–658.

[15] M. Oltean, "Evolving evolutionary algorithms using linear genetic programming," *Evolutionary Computation*, vol. 13, no. 3, pp. 387–410, Sep. 2005.

[16] S. van Rijn, H. Wang, M. van Leeuwen, and T. Bäck, "Evolving the structure of evolution strategies," in *2016 IEEE Symposium Series on Computational Intelligence*. IEEE Press, 2016, pp. 1–8.

[17] N. Lourenço, F. Pereira, and E. Costa, "Evolving evolutionary algorithms," in *Proceedings of the 14th Genetic and Evolutionary Computation Conference Companion*. Association for Computing Machinery, 2012, pp. 51–58.

[18] L. Spector, N. F. McPhee, T. Helmuth, M. M. Casale, and J. Oks, "Evolution evolves with autoconstruction," in *Proceedings of the 2016 Genetic and Evolutionary Computation Conference Companion*. Association for Computing Machinery, 2016, pp. 1349–1356.

[19] L. Spector and A. Robinson, "Genetic programming and autoconstructive evolution with the push programming language," *Genetic Programming and Evolvable Machines*, vol. 3, no. 1, pp. 7–40, 2002.

[20] L. Spector, *Towards Practical Autoconstructive Evolution: Self-Evolution of Problem-Solving Genetic Programming Systems*. Berlin, Heidelberg: Springer-Verlag, Oct. 2010, vol. 8, pp. 17–33.

[21] J. Bim, G. Karafotias, S. K. Smit, A. Eiben, and E. Haasdijk, "It's fate: A self-organising evolutionary algorithm," in *Proceedings of the 12th International Conference on Parallel Problem Solving from Nature*. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 185–194.

[22] A. E. Avramiea, G. Karafotias, and A. Eiben, "Fate agent evolutionary algorithms with self-adaptive mutation," in *Proceedings of the 2014 Genetic and Evolutionary Computation Conference Companion*. Association for Computing Machinery, 2014, pp. 191–192.

[23] T. Atkinson, D. Plump, and S. Stepney, "Evolving graphs by graph programming," in *Genetic Programming*, M. Castelli, L. Sekanina, M. Zhang, S. Cagnoni, and P. García-Sánchez, Eds. Cham, Germany: Springer International Publishing, 2018, pp. 35–51.

[24] J. Miller, *Cartesian Genetic Programming*. Berlin, Heidelberg: Springer-Verlag, Jun. 2003, vol. 43.

[25] N. Hansen, S. Finck, R. Ros, and A. Auger, "Real-parameter black-box optimization benchmarking 2009: Noiseless functions definitions," Ph.D. dissertation, INRIA, 2009.

[26] J. Mouret and J. Clune, "Illuminating search spaces by mapping elites," *CoRR*, vol. abs/1504.04909, 2015. [Online]. Available: http://arxiv.org/abs/1504.04909