DISCOVERING GRAPH GENERATION ALGORITHMS

Anonymous authors
Paper under double-blind review

ABSTRACT

We provide a novel approach to construct generative models for graphs. Instead of using the traditional probabilistic models or deep generative models, we propose to instead find an algorithm that generates the data. We achieve this using evolutionary search and a powerful fitness function, implemented by a randomly initialized graph neural network. This brings certain advantages over current deep generative models, for instance, a higher potential for out-of-training-distribution generalization and direct interpretability, as the final graph generative process is expressed as a Python function. We show that this approach can be competitive with deep generative models and under some circumstances can even find the true graph generative process, and as such perfectly generalize.

1 INTRODUCTION

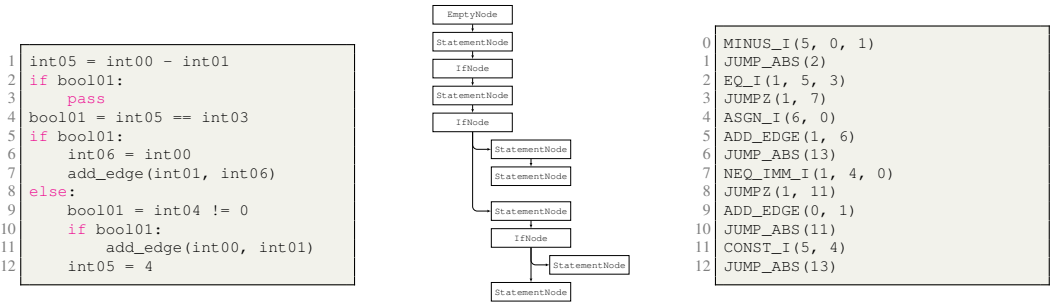
Generating new samples of graphs similar to a given set of graphs is a long-standing problem, which initially was tackled with various statistical models, such as the Erdős/Rényi model (Erdös & Rényi, 1959; Holland et al., 1983; Eldridge et al., 2017). While such models lend themselves well to formal analysis, they do not closely fit real-world graph distributions. More recently, deep generative models have proven to fit graph distributions well (You et al., 2018; Liao et al., 2020; Simonovsky & Komodakis, 2018; Martinkus et al., 2022; Haefeli et al., 2022; Vignac et al., 2022). However, similar to other deep models, they are not interpretable and struggle to generalize to graph sizes outside of the training distribution.

In this work, we propose an alternative approach. Given a set of graphs, we aim to learn a single algorithm, represented as a Python function, that can generate that given set of graphs as well as new examples. If we find an algorithm that fits the data well we can directly inspect it and potentially learn about the generative process that originally created the data. This can be quite useful for example in social sciences. Additionally, the produced algorithm will have a predictable behavior if the input parameters go beyond those observed during training.

```
1  def main():
2    for i in range(N):
3      int04 = i % W
4      bool00 = W != 0
5      call()
6
7  def call():
8      for j in range(i):
9          int05 = i - j
10         if bool01:
11             pass
12         bool01 = int05 == W
13         if bool01:
14             int06 = i
15             add_edge(j, int06)
16         else:
17             bool01 = int04 != 0
18             if bool01:
19                 add_edge(i, j)
```

Figure 1: An example program from the search space that generates a grid graph. `i, j, N` and `W` are aliases for `int00-03`, where `N` is the number of nodes in the grid and `W` is the width of the grid. Lines 1, 2, 7, and 8 are hard-coded, while the contents of the two for loops are what we are searching over.

We achieve this by factorizing the graph construction algorithm into two loops (Figure 1) over nodes and their potential neighbors. Evolutionary search is then used to assemble the logic inside these loops to define when graph edges are added or removed. This setup is closely related to genetic programming (Sobania et al., 2022) where programs are evolved to match input-output examples, especially the grammar-guided (Whigham et al., 1995; Forstenlechner et al., 2016; Fenton et al., 2017) and linear (Lalejini & Ofria, 2019; Hernandez et al., 2019; Dolson et al., 2019; Ferguson et al., 2020) genetic programming approaches. In contrast to stack-based approaches (Perkis, 1994; Spector et al., 2005) they also aim to generate code in a standard programming language such as Python. In general, evolutionary search has proven to be a powerful method for code search. It is sometimes even able to discover neural network architectures and their training functions (Real et al., 2020). Our approach also relates to ap-



(a) Python code (b) Code tree (c) Compiled instructions

Figure 2: The internal representations of a program. Left: Python code. Middle: internal representation as code tree. Right: compiled instructions used for running the code tree.

proaches that aim to learn a program that draws a figure (Ellis et al., 2018; 2020). However, we want a program that captures a family of graphs, not just a single figure.

Note that there are some genetic programming packages that allow generating Python code, such as PonyGE2 (Fenton et al., 2017). However, these proved to be too slow and not well suited for our problem.

We implement our method as a highly-optimized C++ framework and experimentally validate it on synthetic and real-world datasets. When applied to grid graphs with known grid sizes, our search finds algorithms that generalize perfectly to arbitrary grid sizes.

2 METHOD

In this section, we explain the representation our method uses, and how the evolutionary search is performed.

2.1 ALGORITHM REPRESENTATION

Ideally, the representation should be fast to execute, easy to mutate, and human-readable. Unfortunately, some of these goals can be hard to align. Instead, we choose to build a representation that is easy to mutate and can be easily converted to either a human-readable or an efficiently executable format. This representation consists of a tree of nodes (Figure 2b), in which each node corresponds (with few exceptions) to one line of code in a Python program (Figure 2a), where the lines contain either simple operations or if-statements. The structure of the tree matches the logical structure of the code that is implied through indentation: the tree branches out at if-statements and is chain-like otherwise. This allows us to execute the lines of code in the correct sequence by simply correctly traversing the tree (Appendix A.1). As shown in Figure 1, our algorithms consist of two for-loops, which can be nested. The code tree can represent the contents of such a for-loop, but the loop itself is hard-coded in the function that executes the individual.

To respect the code's branching behavior, we define three types of nodes: statement nodes, if-nodes, and empty/root nodes. Each node has a fixed number of named slots for its children, where each slot contains either a pointer to the child node or a marker, the `NULL` pointer if the child is missing.

A *statement-node* contains an instruction to be executed. They have a single child slot, named `nextInBranch`, which points to the next line of code in the current branch (i.e. at the current level of indent for Python code). *Root-nodes* are similar to statement-nodes but do not contain any instructions. They are at the root of a code tree, and their purpose is to simplify inserting new code at the beginning of the program. *If-nodes* contain the address of a boolean condition variable, which decides which branch to take, and they have three child slots. The slots correspond to the "then" branch, to the "else" branch, and to the next line of code at the same level of indent and are named `thenBranch`, `elseBranch` and `nextInBranch` respectively.

Given this structure, we convert the code to a Python-like representation as follows: We traverse the tree in the order `currentNode` → `thenBranch` → `elseBranch` → `nextInBranch` and, keeping track of the current nesting level, we simply output the associated lines of code of each individual.

Similarly to conversion into Python, it is possible to traverse the tree and directly execute it. However, this has some drawbacks and can be slow. Instead, we propose to use a simple compilation step (Figure 2c) to improve evaluation speed inside the for-loops and thus the search speed. We discuss this and other representation implementation details in Appendix A.

The instructions allowed in our representation are listed in Appendix A.2. They cover the usual arithmetic, relational and Boolean operations, if-else statements, and function calls for generating random numbers. We restrict the available variables to a limited set that is strongly typed. Note that in our setup all variables are global. For constructing the graph, we use an adjacency matrix which the algorithm can manipulate and interrogate through dedicated functions. Notably, we disallow while-loops, additional for-loops (except the two hard-coded ones), and jumps to ensure all algorithms have finite running time.

2.2 Evolutionary Search

The evolutionary search is started by initializing each individual as an empty algorithm. Every round we use tournament selection based on the graph fitness. The tournaments are stochastic, where softmax is applied on the fitness values and the winners are then randomly sampled. Taking inspiration from simulated annealing (Kirkpatrick et al., 1983) we gradually decay the softmax temperature during the search to ensure better exploration during the initial phases. The children are generated only through mutation of the parents, without crossovers, as done by Real et al. (2020). The mutations are effectively point-wise. We either delete, insert or change a command by changing one of its variables. In each round, the whole generation is replaced with the children, with some number of the best (elite) individuals being preserved. More information is provided in Appendix B.

To determine the fitness, we use the individual to generate a set of graphs and compare those to the training set graphs. Determining if two graph distributions are the same is a hard task (O'Bray et al., 2022). For our fitness function, we chose to use a randomly initialized GIN graph neural network (Xu et al., 2019) as a powerful feature extractor (Thompson et al., 2022) to embed the training graphs and graphs produced by the generated code and then compare the two resulting distributions (Appendix B.3). The fitness value is computed by using a Radial basis function (RBF) kernel and computing Maximum Mean Discrepancy (MMD) between the two embedding distributions. The use of a randomly initialized model helps us avoid cumbersome training of the graph neural network in an adversarial setting, but still provides us with a well-encompassing feature extractor, that can implicitly capture various graph structures and even node and edge features. This contrasts with the alternative of manually computing various pre-defined graph metrics, such as clustering coefficient and node degree distributions. Note, that Dziugaite et al. (2015) have shown that it is possible to train generative adversarial networks for image generation by solely relying on MMD instead of a trained discriminator. So the MMD can be a well-encompassing metric for generative model training.

3 Experiments

Our work has similarities to autoregressive deep graph generative models (Liao et al., 2020; You et al., 2018), as our search is biased towards iterative algorithms. Thus, we use the experimental setup of GRAN (Liao et al., 2020), and compare to GRAN, GraphRNN (You et al., 2018) and GraphVAE (Simonovsky & Komodakis, 2018) as baselines for our experiments. The datasets and their 80/20 split between training and test data match the ones used by GRAN Liao et al. (2020). We do not make use of a validation set, as in our setup it is redundant, since we do not expect overfitting. During the search procedure, similarly to neural network training, we follow a mini-batch approach, where in every round of search a random subset of 16 test graphs is used to compute the fitness scores. This helps to avoid performing expensive MMD computation over the whole training set at once. See Appendix B.4 for other search hyperparameters.

Table 1 shows that our discovered graph algorithms perform quite well on the simpler grid and lobster graph datasets. While the deep learning models achieve slightly better statistical similarity,

	Grids				Lobsters				Proteins			
	Deg.	Clus.	Orbit	Spec.	Deg.	Clus.	Orbit	Spec.	Deg.	Clus.	Orbit	Spec.
GraphVAE	$7.07e^{-2}$	$7.33e^{-2}$	0.12	$1.44e^{-2}$	$2.09e^{-2}$	$7.97e^{-2}$	$1.43e^{-2}$	$3.94e^{-2}$	0.48	$7.14e^{-2}$	0.74	0.11
GraphRNN	$1.12e^{-2}$	$7.73e^{-5}$	$\mathbf{1.03e^{-3}}$	$\mathbf{1.18e^{-2}}$	$\mathbf{9.26e^{-5}}$	**0.0**	$\mathbf{2.19e^{-5}}$	$\mathbf{1.14e^{-2}}$	$1.06e^{-2}$	0.14	0.88	$1.88e^{-2}$
GRAN	$\mathbf{8.23e^{-4}}$	$3.79e^{-3}$	$1.59e^{-3}$	$1.62e^{-2}$	$3.73e^{-2}$	**0.0**	$7.67e^{-4}$	$2.71e^{-2}$	$\mathbf{1.98e^{-3}}$	$\mathbf{4.86e^{-2}}$	**0.13**	$\mathbf{5.13e^{-3}}$
Ours	$5.10e^{-3}$	**0.0**	$3.38e^{-3}$	0.1	$2.77e^{-3}$	**0.0**	$2.31e^{-2}$	$3.41e^{-2}$	0.42	1.07	1.14	0.31

Table 1: Comparison with deep graph generative models. The results for the baseline models are taken from Liao et al. (2020).

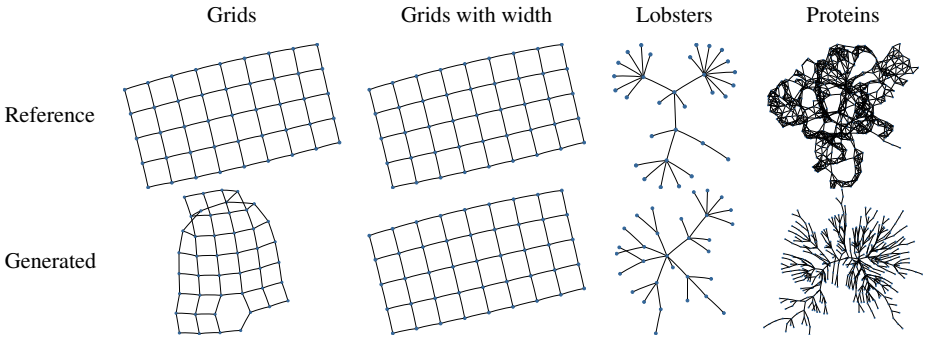


Figure 3: Comparison of reference graphs from the test set to graphs generated by algorithms found with our method. Graphs in the same column have the same number of nodes.

the discovered algorithms are much better at respecting some constraints, such as graphs having no triangles (with clustering coefficient 0). However, the approach struggled with more complex protein graphs. This might indicate insufficient exploration under some circumstances.

Even though we are able to find an algorithm that produces statistically similar graphs to grids, when supplied just with node count, the graphs are not true grids (Figure 3). In our setup, the algorithm can also be conditioned on additional input values. If we instead perform our search over algorithms that take both the node count and the grid width as inputs, our method manages to recover the true generative algorithm (Figure 7d). While it was discovered only using a dataset with up to 361 nodes, it can generalize to produce perfect grids of any size, showcasing the potential benefit of using program synthesis for the graph generation. This is impossible with current deep generative models, which are also incapable of generating perfect, rectangular grids even without considering extrapolation (Liao et al., 2020). This suggests that providing extra inputs to the algorithm can provide a user with additional control over the discovered function's outputs, and also makes the search simpler. The discovered algorithm in Figure 7d is also quite interpretable. This is the case for most of the algorithms we have discovered (Appendix C).

```
def main():
  for i in range(N):
    remove_edge(int06, int10)
    add_edge(i, int06)
    int07 = i % W
    int09 = i - W
    add_edge(int09, i)
    int10 = int07 + int06
    int06 = i + int05
```

Figure 4: The discovered algorithm for grids, when the grid width W is supplied alongside the number of nodes N.

4 Conclusion

In this work, we show that it is possible to use program synthesis to discover interpretable graph generation algorithms. In certain cases even the true generative processes can be discovered, resulting in ideal extrapolation. The performance on more complex graph families can likely be improved, by more carefully tuning the fitness function to make it smoother. As shown by O'Bray et al. (2022) MMD metrics can be quite sensitive to their hyperparameters. Another strong extension could be a compilation of a library of primitive functions that are useful for graph construction. For example, if we provided a function to factorize a number, discovering an algorithm for grid generation when no additional inputs are given would likely be as easy as when the grid width is provided. In principle, the approach can also be applied to attributed graph generation, which is left for future work.

REFERENCES

Emily Dolson, Alexander Lalejini, and Charles Ofria. Exploring Genetic Programming Systems with MAP-Elites. In Wolfgang Banzhaf, Lee Spector, and Leigh Sheneman (eds.), *Genetic Programming Theory and Practice XVI*, pp. 1–16. Springer International Publishing, Cham, 2019. ISBN 978-3-030-04734-4 978-3-030-04735-1. doi: 10.1007/978-3-030-04735-1_1.

Gintare Karolina Dziugaite, Daniel M Roy, and Zoubin Ghahramani. Training generative neural networks via maximum mean discrepancy optimization. *arXiv preprint arXiv:1505.03906*, 2015.

Justin Eldridge, Mikhail Belkin, and Yusu Wang. Graphons, mergeons, and so on!, May 2017.

Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. Learning to infer graphics programs from hand-drawn images. *Advances in neural information processing systems*, 31, 2018.

Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sable-Meyer, Luc Cary, Lucas Morales, Luke Hewitt, Armando Solar-Lezama, and Joshua B. Tenenbaum. DreamCoder: Growing generalizable, interpretable knowledge with wake-sleep Bayesian program learning, June 2020.

P. Erdös and A. Rényi. On random graphs I. *Publicationes Mathematicae Debrecen*, 6:290, 1959.

Michael Fenton, James McDermott, David Fagan, Stefan Forstenlechner, Michael O'Neill, and Erik Hemberg. PonyGE2: Grammatical Evolution in Python. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pp. 1194–1201, July 2017. doi: 10.1145/3067695.3082469.

Austin J. Ferguson, Jose Guadalupe Hernandez, Daniel Junghans, Alexander Lalejini, Emily Dolson, and Charles Ofria. Characterizing the Effects of Random Subsampling on Lexicase Selection. In Wolfgang Banzhaf, Erik Goodman, Leigh Sheneman, Leonardo Trujillo, and Bill Worzel (eds.), *Genetic Programming Theory and Practice XVII*, pp. 1–23. Springer International Publishing, Cham, 2020. ISBN 978-3-030-39957-3 978-3-030-39958-0. doi: 10.1007/978-3-030-39958-0_1.

Stefan Forstenlechner, Miguel Nicolau, David Fagan, and Michael O'Neill. Grammar Design for Derivation Tree Based Genetic Programming Systems. In Malcolm I. Heywood, James McDermott, Mauro Castelli, Ernesto Costa, and Kevin Sim (eds.), *Genetic Programming*, Lecture Notes in Computer Science, pp. 199–214, Cham, 2016. Springer International Publishing. ISBN 978-3-319-30668-1. doi: 10.1007/978-3-319-30668-1_13.

Arthur Gretton, Karsten M. Borgwardt, Malte J. Rasch, Bernhard Schölkopf, and Alexander Smola. A Kernel Two-Sample Test. *Journal of Machine Learning Research*, 13(25):723–773, 2012. ISSN 1533-7928.

Kilian Konstantin Haefeli, Karolis Martinkus, Nathanaël Perraudin, and Roger Wattenhofer. Diffusion models for graphs benefit from discrete state spaces. *arXiv preprint arXiv:2210.01549*, 2022.

Jose Guadalupe Hernandez, Alexander Lalejini, Emily Dolson, and Charles Ofria. Random subsampling improves performance in lexicase selection. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pp. 2028–2031, Prague Czech Republic, July 2019. ACM. ISBN 978-1-4503-6748-6. doi: 10.1145/3319619.3326900.

Paul W. Holland, Kathryn Blackmond Laskey, and Samuel Leinhardt. Stochastic blockmodels: First steps. *Social Networks*, 5(2):109–137, June 1983. ISSN 0378-8733. doi: 10.1016/0378-8733(83)90021-7.

S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220 (4598):671–680, 1983. ISSN 0036-8075.

Alexander Lalejini and Charles Ofria. Tag-accessed memory for genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pp. 346–347, Prague Czech Republic, July 2019. ACM. ISBN 978-1-4503-6748-6. doi: 10.1145/3319619.3321892.

Renjie Liao, Yujia Li, Yang Song, Shenlong Wang, Charlie Nash, William L. Hamilton, David Duvenaud, Raquel Urtasun, and Richard S. Zemel. Efficient Graph Generation with Graph Recurrent Attention Networks, July 2020.

Karolis Martinkus, Andreas Loukas, Nathanaël Perraudin, and Roger Wattenhofer. SPECTRE: Spectral Conditioning Helps to Overcome the Expressivity Limits of One-shot Graph Generators, June 2022.

Leslie O'Bray, Max Horn, Bastian Rieck, and Karsten Borgwardt. Evaluation Metrics for Graph Generative Models: Problems, Pitfalls, and Practical Solutions, March 2022.

Timothy Perkis. Stack-based genetic programming. In *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*, pp. 148–153. IEEE, 1994.

Esteban Real, Chen Liang, David R. So, and Quoc V. Le. AutoML-Zero: Evolving Machine Learning Algorithms From Scratch, June 2020.

Martin Simonovsky and Nikos Komodakis. GraphVAE: Towards Generation of Small Graphs Using Variational Autoencoders, February 2018.

Dominik Sobania, Dirk Schweim, and Franz Rothlauf. A Comprehensive Survey on Program Synthesis with Evolutionary Algorithms. *IEEE Transactions on Evolutionary Computation*, pp. 1–1, 2022. ISSN 1941-0026. doi: 10.1109/TEVC.2022.3162324.

Lee Spector, Jon Klein, and Maarten Keijzer. The Push3 execution stack and the evolution of control. In *Proceedings of the 2005 Conference on Genetic and Evolutionary Computation - GECCO '05*, pp. 1689, Washington DC, USA, 2005. ACM Press. ISBN 978-1-59593-010-1. doi: 10.1145/1068009.1068292.

Rylee Thompson, Boris Knyazev, Elahe Ghalebi, Jungtaek Kim, and Graham W. Taylor. On Evaluation Metrics for Graph Generative Models, April 2022.

Clement Vignac, Igor Krawczuk, Antoine Siraudin, Bohan Wang, Volkan Cevher, and Pascal Frossard. Digress: Discrete denoising diffusion for graph generation. *arXiv preprint arXiv:2209.14734*, 2022.

Peter A Whigham et al. Grammatically-based genetic programming. In *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, volume 16, pp. 33–41. Citeseer, 1995.

Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How Powerful are Graph Neural Networks?, February 2019.

Jiaxuan You, Rex Ying, Xiang Ren, William L. Hamilton, and Jure Leskovec. GraphRNN: Generating Realistic Graphs with Deep Auto-regressive Models, June 2018.

A Details on Algorithm Representation and Execution

Similarly to Real et al. (2020) we implement our search procedure in C++ to ensure efficiency. As our chosen representation and implementation are tailored to our problem setting of finding graph generation algorithms it is easier to ensure that our search procedure is efficient and satisfies our goals of fast selection, mutation, and execution procedures, as well as a human-readable output format. Our implementation will be made publicly available.

A.1 Efficient execution

As noted in the main text, we can traverse the tree and run it using the naive algorithm in Listing 5. While this algorithm functions correctly, it has some drawbacks. Note that the traversal presented here uses recursion, which can have significant overhead, especially when the amount of useful processing done per call is small. We can solve this by using an explicit stack (instead of the implicit call stack) to keep track of which nodes should be visited next. In practice, we have noticed that the traversal-based execution described above has suboptimal performance, even after making the stack explicit and even by reusing the same stack across runs in order to avoid memory allocations. The likely cause of the slowdown is the layout of the nodes in memory: the nodes are not at sequential memory locations, so the traversal will not use the CPU cache optimally.

```
def run_code_tree(node, memory):
    if node is None:
        return

    if type(node) is IfNode:
        if memory.bools[node.conditionVar]:
            run_code_tree(node.thenBranch, memory)
        else:
            run_code_tree(node.elseBranch, memory)
    elif type(node) is StatementNode:
        exec_instruction(node.instruction, memory)

    # For all node types go to nextInBranch
    run_code_tree(node.nextInBranch, memory)

```

Figure 5: Executing a code tree through recursive traversal (pseudocode). `memory` contains the current execution state.

Our chosen solution is to perform a "compilation" step, converting the code tree to an array of instructions (bytecode) which can then be efficiently executed. Of course, compiling the tree incurs an overhead, but it is offset by more quickly executing the code tree as part of a for-loop.

The idea behind the compilation step is to use jump instructions to convert the if-statements into linear code. Whenever we encounter an if-node with a `thenBranch` or `elseBranch`, we insert a conditional jump that skips to the appropriate location in the program. At the same time, we store the node that comes after the if statement in a stack so that we know where to jump once the execution of a branch has finished. The complete algorithm is provided in Listing 6.

Figure 2 illustrates how a piece of Python code is represented as a code tree and how this can then be converted to a linear sequence of instructions. The Python code is similar to that in the `call` function from Figure 1, but the variables' internal names are used, and we have added another statement on line 12. We made this last modification to highlight that the compilation algorithm inserts redundant instructions, such as the jump at location 10. It is practically a no-op since the program flow would continue to that instruction by itself. It would have been possible to eliminate such instructions, but since they only appear at the end of branches (which will likely not make up a significant proportion of the code), we decided not to pursue such an optimization.

```
        def compile_code_tree(tree)
            jump_target_stack = [None]
            jump_list = []
            program = []

            def add_jump(jump_op, target_node, cond_var=None):
                jump_list.append((len(program), target_node))
                if cond_var is None:
                    program.append([jump_op, None])
                else:
                    program.append([jump_op, cond_var, None])

            for node in tree:
                node.instrIdx = len(program)
                if type(node) is IfNode:
                    if node.nextInBranch:
                        next_node = node.nextInBranch
                    else:
                        next_node = jump_target_stack.pop()

                    if node.thenBranch and node.elseBranch:
                        jump_target_stack.extend([next_node]*2)
                        add_jump("JUMPZ", node.elseBranch,
                                 node.conditionVar)
                    elif node.thenBranch:
                        jump_target_stack.append(next_node)
                        add_jump("JUMPZ", next_node,
                                 node.conditionVar)
                    elif node.elseBranch:
                        jump_target_stack.append(next_node)
                        add_jump("JUMPNZ", next_node,
                                 node.conditionVar)
                    else:
                        add_jump("JUMP_ABS", next_node)
                elif type(node) is StatementNode:
                    program.append(node.instruction)
                    if not node.nextInBranch:
                        next_node = jump_target_stack.pop()
                        add_jump("JUMP_ABS", next_node)

            for jump_instr_idx, target in jump_list:
                if target is None:
                    target_instr_idx = len(program)
                else:
                    target_instr_idx = target.instrIdx

                program[jump_instr_idx][-1] = target_instr_idx

            return program

```

Figure 6: Python pseudocode for compiling a code tree. For simplicity, we denote instructions as lists of one op-code and operands. Assumes a code tree has an iterator for traversing it in the order `currentNode` → `thenBranch` → `elseBranch` → `nextInBranch`.

A.2 Execution environment and instructions

So far, we have looked at how to compile a code tree into a sequence of instructions but skimmed over the details of the instructions and the execution environment (the memory). This subsection fills those gaps.

To make the concepts clear, it is best to start with an overview. We represent the memory as an object with different fields, where each field contains a different type of data. The instructions are similar to the machine instructions of a CPU and consist of an operation (op-code) and operands. Depending on the op-code, the operands can be interpreted as addresses inside one of the fields or as constants. Moreover, addresses refer to specific fields in the memory depending on the op-code. For example, `LT_IMM_F(5, 3, 0.125)` is the representation of `bool05 = float03 < 0.125`, while `PLUS_I(1, 2, 3)` corresponds to `int01 = int02 + int03`, where `bool`, `int` and `float` indicate different memory fields.

Memory. In detail, the memory consists of: (1) a program counter; (2) three sets of "registers" (i.e. arrays), one for integers, booleans, and floating-point numbers, respectively; (3) the state of a pseudo-random number generator (PRNG); (4) per-node storage in the form of an array of integers; (5) the representation of the graph being built;

The *program counter* is the index of the instruction being executed as part of the compiled program. It is incremented by one automatically to progress the flow of the program but can be manipulated by jump instructions to implement branching.

The *registers* act as the main inputs and outputs for the instructions, and their size can be configured. These memory accesses are not bounds-checked, but the mutations which construct programs are designed never to insert invalid accesses.

The *PRNG state*, the *per-node storage*, and the *graph representation* are all accessed, used, and modified by dedicated instructions. In the case of the per-node storage and the graph, the contents of registers are used as addresses, so we introduce bounds-checking. Invalid write accesses do nothing, while invalid reads return 0.

Graph format. The details of the graph storage format require special attention. Since we are interested in being able to add and remove edges quickly ($O(1)$), it is best to represent the graph through its adjacency matrix. Unfortunately, the naive approach of storing the complete matrix has a space requirement of $O(N^2)$, where N is the number of nodes, which becomes prohibitive for large graphs. Our solution is to make use of the sparsity structure of the graphs of interest and employ a sparse-matrix format. The format we choose is based on the dictionary of keys. We use a set based on a hash table to store the row-column index pairs of non-zero entries in the matrix, giving us amortized $O(1)$ performance for adding and removing edges. In addition, to enforce the symmetry of the matrix (remember that we are only working with undirected graphs), we sort the indices in each index pair so that only the elements above the main diagonal have to be stored.

Instructions. With the structure of the memory explained, let us shift our focus to the instructions. As mentioned earlier, an instruction consists of an op-code and operands. Specifically, we represent an instruction as a struct with one op-code, one output operand, and two input operands. Each of these components uses 4 bytes. Depending on the op-code, the operands are interpreted as floating point constants, integer constants, or addresses (which are also integers). The op-code also dictates the type of register towards which an address points.

The complete list of op-codes is presented in Table 2. Generally, the suffix indicates the input type, except for `B_TO_I` (conversion from bool to int), and the random sampling functions. Op-codes that contain the "`_IMM_`" substring are operations where one of the input operands, usually the second, is a constant (an "immediate" operand). One crucial design decision is how to respond to invalid operations, such as division by zeros. Since we expect it is hard to generate code that has no such operations, we decided that they will simply be skipped when encountered during execution.

Individual execution. With the details of the instructions and the memory in mind, running an individual to generate graphs is straightforward. We first do some setup, which consists of compiling the code trees, emptying the memory (values for all possible variables are initially set to 0), and loading all input data into memory. Then, for each node, we iterate over and execute the compiled instructions of the first code tree, which corresponds to the `main` function in our code listings. To ensure the instructions can access the loop index and the program counter, the loop variables of the for-loops over nodes and instructions are references to entries in the memory object.

Type	Operations
Integer arithmetic	PLUS_I, MINUS_I, TIMES_I, DIV_I, MOD_I PLUS_IMM_I, MINUS_IMM_I, TIMES_IMM_I, DIV_IMM_I, MOD_IMM_I
Integer relational operations	LT_I, LTE_I, EQ_I, GTE_I, GT_I, NEQ_I LT_IMM_I, LTE_IMM_I, EQ_IMM_I, GTE_IMM_I, GT_IMM_I, NEQ_IMM_I NZERO_I, ZERO_I
Floating-point relational operations	LT_F, LTE_F, GTE_F, GT_F LT_IMM_F, LTE_IMM_F, GTE_IMM_F, GT_IMM_F
Boolean operations	AND_B, NAND_B, OR_B, NOR_B, XOR_B, XNOR_B, NOT_B
Conversion operations	B_TO_I
Randomness	RND_UNIF_F, RND_UNIF_I RND_UNIF_IMM_I
Assignment	ASGN_I, ASGN_F, ASGN_B CONST_I, CONST_F
Edge operations	ADD_EDGE, REMOVE_EDGE, FLIP_EDGE, IS_EDGE
Jumps	JUMP_ABS, JUMP_REL, JUMPZ, JUMPNZ
Call inner loop function	CALL
Per-node storage access	STORE_I, LOAD_I

Table 2: Op-codes considered in our search.

Executing a `CALL` instruction represents a special case. For it, we save the program counter value, execute the `call()` function in a similar fashion to the current one, and then restore the value of the program counter to continue execution.

At the end of the run, the generated graph is inside the memory object and can be converted to a PyTorch sparse tensor for calculating the loss function.

B Evolutionary Search Details

In this section, we further detail our evolutionary search setup described in the main paper.

B.1 Stochastic Tournament Selection

In classical tournament selection, the winner of a tournament is decided as the individual with the best score, even if that individual is only marginally better than the others in the tournament. This behavior can lead to locally optimal traits spreading too fast through the population and trapping the search. The problem is especially evident if we add a regularization factor to the cost function to direct the search toward shorter programs. In this case, it is challenging for the search to build a complex solution since there is strong selection pressure against adding lines of code that do not lead to an immediate improvement of the cost function.

We propose to generalize the tournament by introducing stochastic sampling to mitigate this issue. For this, we pass the individuals' scores through a softmax function, which induces a probability distribution over the individuals. The softmax has a temperature parameter which, similarly to the learning rate used for other optimization methods, we gradually decrease according to an annealing schedule as the search progresses. We then take one sample from this distribution and declare the resulting individual the winner.

Given individuals $(I_1, \ldots, I_T)$ with associated loss values $(l_1, \ldots, l_T)$, the probability of individual I_i of winning the tournament at temperature $\mathcal{T} > 0$ is

$$p(I_i) = \frac{\exp\left(-\frac{l_i}{\mathcal{T}}\right)}{\sum_{j=0}^{T} \exp\left(-\frac{l_j}{\mathcal{T}}\right)}. \tag{1}$$

As $\mathcal{T} \to \infty$, the distribution tends to the uniform distribution, which removes selection pressure and leads to a random walk through the search space. When $\mathcal{T} \to 0$, the sampling procedure

reverts to being an argmin over the losses, with the difference that it samples uniformly over the best-performing individuals if more than one achieves the lowest loss.

By following an annealing schedule that decreases from a high value of $\mathcal{T}$ to 0, we can explore larger parts of the loss landscape in the beginning and switch to an exploiting and fine-tuning behavior towards the end.

Since it uses the score differences between individuals and not just the rank order, the softmax also lets us combine multiple cost functions in useful ways. In particular, we can add a regularization loss that penalizes long programs to the main loss. By choosing a small scaling factor for the regularization loss, we enable the search to explore programs of all sizes, but still, have a preference for shorter programs with equivalent functionality when they are discovered.

B.2 MUTATIONS

The choice of mutation operations dictates how easy it is to transition between candidate solutions and therefore is critical for the performance of the search. We base some of our mutations on those used by Real et al. (2020), but adapt them to our search space and implement several others.

The list of mutations is as follows:

1. Insertion Mutation: insert a random line of code at a random location in the program.
2. Knockout Mutation: remove a random line of code from the program.
3. Operation Change Mutation: select a random line of code and replace its operation with a new random one with the same inputs and outputs.
4. Parameter Change Mutation: select a random line and change one of the input or output arguments with a new random one.
5. Randomization Mutation: randomize the contents of the program while keeping the size of each function, `main` and `call`, the same.
6. No-Op Mutation: leave the program as it is.

We sample the mutation types at different rates. We chose knockout mutation at triple the baseline rate, while the parameter change mutation happens at double the baseline rate. When a mutation type is sampled, we sample the particular change uniformly at random, from the available ones for that mutation type (e.g. one line is removed uniformly at random). There are two exceptions to this: (1) when we are inserting a line, we choose to insert an if-statement with a probability of 0.2, while otherwise, we insert a random command (Table 2); (2) when we are updating a parameter value, we use Brownian motion to ensure that parameter change is gradual. For integer variables, we use the discrete Gaussian distribution with a standard deviation of 1.0 for the Brownian motion, while for floating point variables we use the Gaussian distribution with a standard deviation of 0.1.

B.3 FITNESS

As with all optimization procedures, the choice of the loss function is one of the most important design choices and is decisive for the algorithm's performance.

To evaluate an individual's performance, we sample batches of B examples from the dataset, iterate over them and run the individual once for each, using the graph size and the auxiliary data as input. The functions that the individual can call for generating random numbers make use of a random number generator state that is not reset between runs. This implies that the resulting graphs can be considered a random variable with an associated probability distribution. By comparing this probability distribution to the distribution of the graphs in the batch, we can calculate a score for the individual.

As mentioned before, we use MMD to compute the difference between the distributions:

$$\mathrm{MMD}_b^2\left[k, X, Y\right] = \frac{1}{m^2}\sum_{i,j=1}^{m} k(x_i, x_j) - \frac{2}{mn}\sum_{i,j=1}^{m,n} k(x_i, y_j) + \frac{1}{n^2}\sum_{i,j=1}^{n} k(y_i, y_j), \tag{2}$$

where X and Y are samples from two distributions, and $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is a kernel function (Gretton et al., 2012).

We follow the usual practice of setting $\mathcal{X} = \mathbb{R}^d$ and applying a feature extraction function $\phi : \mathcal{G} \to \mathbb{R}^d$ to the reference and predicted graphs and using the MMD on the resulting embeddings.

The main feature extraction step that we use is a randomly initialized, untrained GIN neural network (Xu et al., 2019). We set the number of message passing rounds to 3, the number of layers in each MLP to 2, and the embedding dimension to 35, as suggested by Thompson et al. (2022). As node input features, we use the one-hot encoding of the node degrees after clamping them to the range $[0, 19]$. We deviate from the original GIN model in three ways.

First, we remove the batch normalization layers. Untrained batch norm layers default to a mean of 0 and a variance of 1, so they act as no-ops in our model.

Second, we do not use biases in the MLP layers. Since they were untrained and our input (a one-hot encoding) can never be zero, we considered them redundant. An advantage of doing this is that, when padding the graphs and embedding vectors with zeros to match a fixed graph size, zero elements stay unmodified throughout the GIN's processing, which leads to easier pooling over the graph.

Third, we add a custom normalization step after each messag-passing round. Concretely, for each node, we divide the results of message aggregation by the square root of the number of neighbors. The goal is to reduce the output variance and compensate for the fact that the batch norm layers cannot be used.

With the developed framework, we also have the option to use classical feature extractors, such as histograms of node degrees, histograms of the eigenvalues of the graph Laplacian, and histograms of node clustering coefficients. In principle, if one would assume that the ordering of the graphs is known, a powerful alternative feature extraction would be to directly compute the Hamming distance between the generated adjacency matrix and the true adjacency matrix. However, in our experiments, we assume that the ordering is unknown, and effectively leave it to the search procedure to find a node ordering with which it is easy to build a graph construction algorithm.

As kernel functions for the MMD, we implement the Gaussian kernel, also known as the radial basis function kernel:

$$k_{Gauss}(x, y) = \exp\left(-\frac{\|x - y\|_2^2}{2\sigma^2}\right) \tag{3}$$

$$\tag{4}$$

B.4 Hyperparameters

For each dataset, we train models that have access to all operations, with a population of 1000 individuals, tournaments of size 4, and keep an elite set of 10 individuals when performing the generational replacement. We anneal the temperature exponentially from a value of 10 towards 0 with a factor of 0.9998 per generation. The cost function is the MMD with a GIN feature extractor and using the Gaussian kernel with $\sigma = 1$, onto which we add a regularization cost of 10^{-8} for each node in an individual's code trees. The code length is capped to a maximum of 50 nodes in the tree. All experiments are run on multi-core CPU machines without GPU acceleration. The search experiments are run for 89 hours or 1 million generations, whichever comes first.

C Examples of Discovered Algorithms

Here in Figure 7 we show the algorithms our method discovered for the graph classes we have considered. We can see that the discovered algorithms are mostly interpretable, except the algorithm for grids (Figure 7d). Which likely grows in complexity because it is quite hard to determine a possible width of a grid from the desired number of nodes without having a function for number factorization. The function discovered for lobster graph construction produces graphs very close to lobster graphs, as the nodes with low IDs are prioritized for attachment, forming a sort of a backbone. Hover the strict limit of having at most two children outside of the backbone was not

discovered. This can potentially be alleviated by performing the search with more individuals and running it for longer. However, we did not test this due to the time limit. The search procedure only ended up using one loop for these datasets, as we apply some pressure on the program length and the solutions with two loops did not prove to be better. This can be adjusted, either by decreasing the pressure on program length or forcing it to execute the inner loop.

```
def main():
  for i in range(N):
    flip_edge(int03, i)
    int03 = uniform(i)
```

(a) Proteins

```
def main():
  for i in range(N):
    int05 = uniform(i)
    int08 = int05 // 2
    add_edge(i, int08)
```

(b) Lobsters

```
def main():
  for i in range(N):
    remove_edge(int06, int10)
    add_edge(i, int06)
    int07 = i % W
    int09 = i - W
    add_edge(int09, i)
    int10 = int07 + int06
    int06 = i + int05
```

(c) Grids with Input

```
def main():
  for i in range(N):
    bool05 = int07 >= 0
    int04 = i - int08
    if bool04:
      bool05 = int07 >= int06
      int03 = int08 + 3
      int07 -= 2
    add_edge(i, int03)
    bool02 = bool05 or bool02
    if not bool02:
      int06 = int05 + 1
      flip_edge(int08, int04)
    int08 = int05 - int04
    int03 = i
    bool00 = int06 < 1
    flip_edge(int06, i)
    remove_edge(int08, int06)
    int08 = int07 + 4
    int06 /= int05
    if bool00:
      int07 = int05 + int03
      int08 = j
      int04 = i // 5
      int06 = j // 3
      bool05 = is_edge(N, i)
      bool01 = int04 > int05
      int05 = int04 * int07
    else:
      int07 = int05 + N
      if bool01:
        int08 += 5
        int04 = 5
        remove_edge(int08, int07)
        int03 = 5
      bool01 = int06 >= 4
    bool02 = int06 < int04
    int05 += 3
    if bool01:
      int05 = int04 // 3
      add_edge(int05, int04)
    else:
      bool04 = int05 == int06
      int05 = int03 % int06
    int06 = int03 - 5
    bool00 = int04 >= int05
    if not bool05:
      bool01 = is_edge(int06, int08)
      if bool01:
        int03 = int03 % i
      bool04 = bool00 and bool05
    bool02 = bool00 != bool02
    int05 = int04 + int03
```

(d) Grids

Figure 7: Generated algorithms for each experiment. Algorithms for Lobsters and Grids were manually cleaned, by removing redundant lines.