C190I Final Project Report

Manu Kondapaneni
Computer Science
UC Santa Barbara
mkondapaneni@ucsb.edu

Abstract

This paper introduces two tools. The first expands Graphs based on Graph classes and the second synthesizes programs made up of graph algorithms to satisfy a set of input output examples on small graphs. We use a subset of networkx methods to build these programs and graphs. Furthermore, we use random enumeration, relaxed equality measures, and other heuristic measures to give good quality solutions while also speeding up the synthesis process. We then measure the speed of synthesis for both the graph expansions and the program generator. Finally, we qualitatively analyze the graphs generated and discuss possible future work.

Keywords: Program synthesis

1 Introduction

In this paper we will introduce a rudimentary tool Grapher , that expands Graphs and also generates Graph algorithms. The two features are expanded upon below.

1.1 Graph Expansion

Computer Scientists and students constantly use graphs in many ways. Often times they use toy examples for many reasons such as teaching and learning new concepts and trying out new algorithms on small test cases. However, the graphs used often times don't have any randomness about them. This means that it might be harder for a human to think of a counterexample for their algorithm. Similarly, a teacher giving multiple graphs to students for a problem set might fall into the trap of making each graph too similar. To help researchers, programmers, teachers, and students in this scenario we introduce graph expansion. Graph expansion as the process of taking an input graph and increasing it's size while maintaining an intuitive sense of structure within the graph. This sense of structure is vague and informal so this problem is very hard to express, let alone solve. In this paper we will formulate this problem in one of the many possible ways based on graph classes. Then, we will give our solution to this problem relying on random enumeration. Finally, we will evaluate the time it takes to synthesize solutions for this problem and also give a qualitative analysis of the graphs outputted by our tool.

1.2 Graph Algorithm Synthesis

Often times programmers have an algorithm they want to run on a graph but they don't know how to implement it or even the name of the algorithm they want to run. Our goal with graph algorithm synthesis is to take in a set of input output examples where the input is a graph and the output is an integer output related to the graph in some way. Then, Grapher will enumerate the possible graph algorithms to find an algorithm that satisfies all the examples.
To implement these graph algorithms we use networkx [2]. networkx is a python library that allows for graph manipulation and provides a large set of graph algorithms that can be run. This means grapher can both find graph algorithms for programs and also serves as an educational introduction to networkx a valuable tool.

1.3 Summary

To summarize Grapher does the following tasks.

- Graph expansion
- Graph Algorithm Synthesis

, ,
2021.

- Introduction to networkx

2 Motivating Example

In this section we will introduce some motivating examples that we will refer to in the rest of the paper.

2.1 Professors and Researchers

Consider a researcher trying to solve a graph problem or a professor that wants to give an exercise to her students. In this case they may have some graph structure they want to maintain but don't want to "fill in the blanks" of their graph under some constraints. Furthermore, in the case of the researcher she may not want to bias the rest of her graph with her thought process about the algorithm she is developing. This induces the niche for a graph expander that maintains the structure of the graph as an input along with some in variants given by the user and adds nodes and edges in a non-deterministic way to be used for testing and other purposes.

2.2 Developers

For developers there is both an educational and practical use for Grapher .

2.2.1 Educational. Consider a developer with limited experience with networkx. However, they want to learn more about the library. They would want some way to generate graphs quickly and play with them or even try out algorithms. Using Grapher this is possible they could use a graph representation they are more comfortable with such as an adjacency list/matrix and convert it to a graph in networkx to both learn more about the library and quickly get started with the library.

2.2.2 Practical. Developers may want to write a graph algorithm to solve a problem but don't know how to write the algorithm or don't know where to look. For example, they may want to find the number of vertex disjoint paths in their graph but they don't know how to write the algorithm themselves. Instead, they could give some examples of outputs they want on small input graphs and let a synthesizer handle the actual algorithm. This would both save time and allow developers to not need to have esoteric knowledge about algorithms.

These motivating examples cover many use cases and introduce the niche for Grapher . In the rest of the paper we will refer to the first problem as *Expansion example* and the practical problem in the second section as *Synthesizer example*.

3 Problem Formulation

In this section we will give our definition of both problems we are trying to solve along with the advantages and drawbacks of our approach. Finally, we will discuss ideas for future iterations of Grapher that address the drawback of our problem formulation.

3.1 Graph Expansion

3.1.1 Formulation. Expanding a graph is a vague problem. What does it mean to maintain structure in a graph under an expansion. There are metrics we can use like edit distance or even embedding the graph into $\mathbb{R}^d$ space and then using euclidean distance metrics. However, these methods are difficult to implement and have the drawback of not satisfying specifications given by the user. In this paper we formulate the expansion problem as a problem under graph family. Graph families or classes capture a subset of general graphs. Trees, Bipartite Graphs, and Chordal Graphs are examples of graph families. Many problems can only be solved efficiently (as far as we know) on specific graph families like the independent set on Trees. Furthermore, many times teachers want to give graphs to students in specific graph families as a homework problem. This connection to our motivating example is the rationale to why we chose to formulate this problem in this way. Currently, we have only implemented three graph families but we plan to do more in the future.

3.1.2 Drawbacks. The drawbacks of formulating the problem in this way are twofold. First, users must know about graph classes and exactly what graph class they want. This means they cannot use Grapher to find structure in a graph and expand it based on that structure. We hope to also implement this feature using the approaches outlined above such as edit distance and graph embedding to make Grapher more general and easier to use. Second, many graph families cannot be quickly recognized. This means to verify that it a graph is part of a specific graph family can be hard. This means the synthesizer will not be able to run quickly with certain graph families.

3.2 Graph Algorithm Synthesis

3.2.1 Formulation. We formulate the problem of synthesizing graph algorithms as a search problem. Essentially, we need to lift a subset of graph algorithms and try all of them on the input. Furthermore, we do not need to try multiple combinations of algorithms because most graph algorithms have clear inputs and outputs. This implies that all we need is to do a simple search over the search space that can be done quickly with or without pruning the search space. However, it is not clear what the measure of equality to find a correct solution should be. This ambiguity can be seen through example. Consider a maximum matching on a graph defined in the following way. A matching is a set of edges such that no two edges share a vertex. A maximum matching is the maximum number of edges such that no two edges share a vertex. The ambiguity between correct solutions arises because there can be multiple maximum matchings on a graph as can be seen in Figure 1

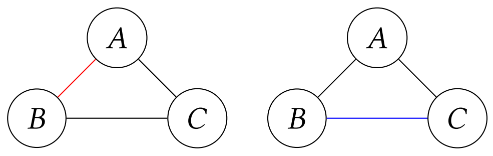

Figure 1. A graph G. A maximum matching M_1 in the graph highlighted in red and a different maximum matching M_2 on the same graph highlighted in blue

This means there are multiple correct solutions to this problem with no way to distinguish a best solution. To resolve this ambiguity for graph optimization problems we verify equality between the program we generate and the output solution given by the user using the size of the solution instead of the exact solution.

3.2.2 Drawbacks. There are three drawbacks of this formulation. First, we need to explicitly lift graph algorithms that can be used for synthesis. There is no way to automatically lift methods that can be used. This means to extend Grapher the source code must be edited. In the future this process could be crowd-sourced. Another drawback is that the programs outputted are not very complex. This is because the depth of the search is very shallow as we only try one graph algorithm at a time. The last drawback is that our measure of equality is very strict. This means that users must be very specific and apply the algorithm they have mind correctly every time on their input graphs. For example, if they make a mistake on their example inputs Grapher will not find the correct program. Another drawback, related to this same issue is that Grapher cannot currently use approximation algorithms because of the inherently looser definition of equality. Approximation algorithms give a guarantee that the solution found is at most some factor worse than the optimal solution. A solution that we want to implement in the future is to try the stricter version of equality first. Then, loosen the equality to a solution that satisfies the most examples. Furthermore, we will try to add a measure of equality for approximation algorithms that checks to see if the solution outputted by the program is at most worse than the solution given by the user by the approximation factor.
Both our problem formulations have advantages and drawbacks. The biggest drawback is that both problem formulations rely on technical expertise from the user. We hope to make Grapher more accessible in the future.

4 Interactive Synthesis Algorithm

In this section we will introduce the DSL's (Domain Specific Language) we use for graph expansion and graph algorithm synthesis along with the algorithms we use. We will then connect the DSL with our motivating examples for both problems.

Algorithm 1 Graph Expansion algorithm given DSL D, graph $G = (V, E)$, graph family $\mathcal{F}$ and additional number of nodes n.

```
Add $|V| + n$ nodes to $D$.
$P \leftarrow \text{root}(S)$
$d := (|V| + n)^2$
while true do
    if $P(G)$ is connected and $P(G) \in \mathcal{F}$ then
        return $P$
    else
        $P \leftarrow \text{rand}(P, D, S)$
    if $|P| \geq d$ then
        backtrack
```

4.1 Graph Expansion

First, we will discuss the DSL for graph expansion in Figure 2. Note, that the DSL is not completely defined before the example graph is given. Namely, there is not a fixed number of nodes. This allows for Grapher to generate a DSL for every graph based on the number of nodes in the input graph and the number of nodes the user wants to be added. Finally, the only method in the DSL is to add an edge between two nodes in the graph. This is the only need for synthesis in graph expansion because the edges added give the structure to the graph. To see why we do not need to enumerate nodes recall the *expansion example* given in section 2. In this example we had a teacher giving a graph to their students for an exercise or a researcher generating graphs to test their algorithm. In both cases the user will know the small number of nodes they want in their graph in advance. For example, a teacher will want to give a graph that is analyzable to their students for exercises . Similarly, a researcher will want to be able to see where their algorithm goes wrong on a given graph. With too many nodes this becomes impossible. Therefore, we expect the graphs we generate with Grapher to be relatively small. Now, consider Algorithm 1. This is the algorithm for graph expansion. Note that in this algorithm P is the program, the rand function chooses a random branch of the AST to add to P and d is the depth of the search. Note, that the depth of the search is $(|V| + n)^2$ because there can be at most $(|V| + n)^2$ edges in our graph.

```
enum Node{
    ...
}
value Graph;
value Empty;

#synthesized program
program G(Graph) -> Graph;
#production rules
func empty: Empty -> Empty;
func add_edge: Graph n ->
    Graph a, Node v, Node u;
```

Figure 2. DSL for graph expansion in tyrell (from Neo) syntax with an unspecified number of nodes

(Each edge can be matched with $|V| + n - 1$ other edges.) Generally, random enumeration is a poor choice for enumeration. However, in this use case we argue that it allows for users to actually get non-deterministic graphs that are inherently random. Furthermore, because graphs are small as shown above random enumeration will not take too long. This can be seen in our evaluation section. We now will discuss the DSL and algorithm for graph algorithm synthesis.

```
enum Node{
    ...
}
value Graph;
value Empty;
value Output;

#synthesized program
program G(Graph) -> Output;

#production rules
func empty: Empty -> Empty;
func num_conn_comp: Graph g -> Output o;
func num_node_disjoint_paths: Output o ->
Graph g, Node s, Node t;
func num_edge_disjoint_paths: Output o ->
Graph g, Node s, Node t;
func maximum_matching: Output o -> Graph g;
func minimum_edge_cut: Output o -> Graph g;
func max_core: Output o -> Graph g;
func shortest_path: Output o ->
Graph g, Node s, Node t;
func min_cut: Output o ->
Graph g, Node s, Node t;
```

Figure 3. DSL for graph algorithm synthesis in tyrell (from Neo) syntax with an unspecified number of nodes

Algorithm 2 Graph Algorithm Synthesis algorithm given partial DSL D, graph input examples I, and corresponding output O.

```
Add $\min_{i \in I}\{|i|\}$ nodes to $D$.
$P \leftarrow \text{root}(S)$
$d := 1$
while $true$ do
    for $i \in I$ do
        Check if $P(i) = O_i$.
    if $\forall i \in I(P(i) = O_i)$ then
        return $P$
    else
        $P \leftarrow \text{rand}(P, D, S)$
    if $|P| \geq d$ then
        backtrack
```

4.2 Graph Algorithm Synthesis

First, we will discuss the Graph Algorithm Synthesis DSL in figure 3. Similar to the first DSL for graph expansion we also use a variable number of nodes in the Graph Algorithm Synthesis DSL. This means users can once again use any graph size with any number of nodes. However, we enforce the invariant that the nodes are labelled consistently. By consistently we mean that if the user wants to compute the shortest path between two nodes they must use the same label for those two nodes in all their examples. This allows us to avoid the hard graph isomorphism problem while not putting too much of a burden on the user. Finally, the DSL has a number of lifted networkx methods. These are the methods we will enumerate to synthesize a program for the user. In the future, we hope to add more lifted methods to Grapher . Now, recall the *Synthesis example* from Section 2. Using the DSL the developer can learn about the algorithms networkx offers and also provide input output examples of any size to the synthesizer. This illustrates the connection between our DSL for graph algorithm synthesis and the motivating example for the problem.

We can now consider Algorithm 2, the algorithm for graph algorithm synthesis. This algorithm first enumerates only the minimum number of nodes possible. This is because we maintain the invariant that algorithms that use specific vertices as part of their input can only use nodes that appear in all the input examples. Therefore, all possible nodes we can use occur in the graph example with the minimum number of nodes. Then, we enumerate programs up to a depth of 1 and choose the first program that satisfies all the test cases. We only need to enumerate programs up to a depth 1 because all the programs we have immediately give an output (most likely an integer) that cannot be used with another algorithm. This is because most of the complexity for a graph algorithm takes place within the algorithm itself and not within the composition of algorithms.

5 Implementation

In this section we will discuss the implementation decisions we make. We used Neo and python to implement both tools in Grapher .

5.1 Graph Expansion

To perform enumeration we used the random enumerator in Neo [1] for the graph expansion problem. Furthermore, we added our own definition of equality based on graph families and connectivity as described above. The pipeline for our implementation is given in Figure 4.

5.2 Graph Algorithm Synthesis

To perform enumeration we used the SMT enumerator in Neo for the graph algorithm synthesis problem because we don't need to have a non-deterministic algorithm that takes advantage of

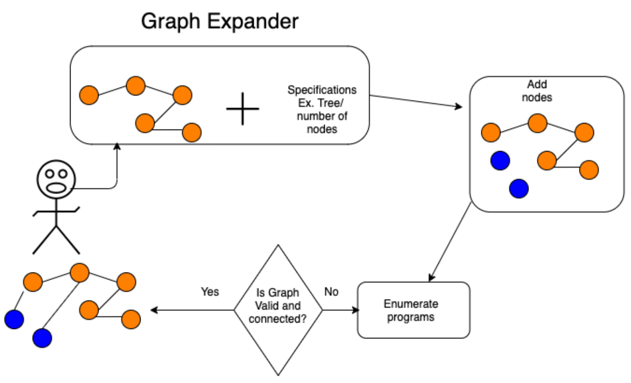
Graph Expander
Specifications Ex. Tree/ number of nodes
Add nodes
Enumerate programs
Is Graph Valid and connected?
Yes
No

Figure 4. Pipeline for Graph Expansion (top) and Graph Algorithm Synthesis (bottom)

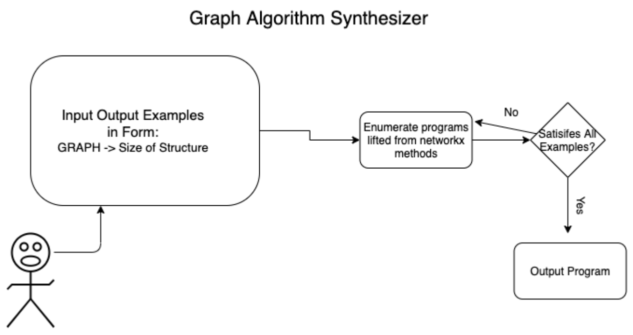
Graph Algorithm Synthesizer
Input Output Examples in Form: GRAPH -> Size of Structure
Enumerate programs lifted from networkx methods
Satisifes All Examples?
No
Yes
Output Program

Figure 5. Pipeline for Graph Expansion (top) and Graph Algorithm Synthesis (bottom)

randomness. Furthermore, we added our own definition of equality based on the size of the output as described above. The pipeline for our implementation is given in Figure 5.

Our code can be found at https://github.com/konman2/Grapher_CS190I

6 Evaluation

6.1 Graph Expansion

For graph expansion, our goal for evaluation was to answer the following two questions.

- How fast does Grapher generate small graphs.
- How "good" are the graphs Grapher generates.

6.1.1 Speed.

To address the first question we tracked the running time of Grapher with an input of a 5 node graph. Then, we added up to 10 nodes to see how fast the output graph is generated. We repeated this process 10 times and averaged the times for each graph class. The results are shown below for trees (Figure 6, bipartite graphs (Figure 7), and chordal graphs (Figure 8). A key takeaway is that the running time grows with the input. This is due to our variable DSL that is based on the graph size. If we had a default number of nodes in the DSL the enumerator would have to try to add edges corresponding to nodes that won't exist and take the same amount of time on all inputs. Another facet of the algorithm is that although it runs in a reasonable amount of time for 15 node graphs, the running time grows exponentially. We will address easy to implement features that can improve this running time that we can Section 7 when we discuss future work. Finally, note that chordal graphs have the smallest average running time. We believe this is because chordal graphs are the least restrictive graph class that we try. The definition of a chordal graph is a graph with no induced cycle of greater than 3 vertices. Clearly, trees are a subset of this graph class. However, more experiments including more graph classes are necessary to verify this observation.

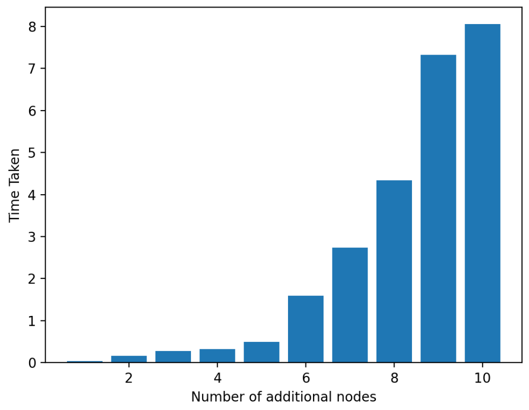

Figure 6. Average Time taken over ten trials to expand graph with additional nodes on top of a 5 node graph for a tree.

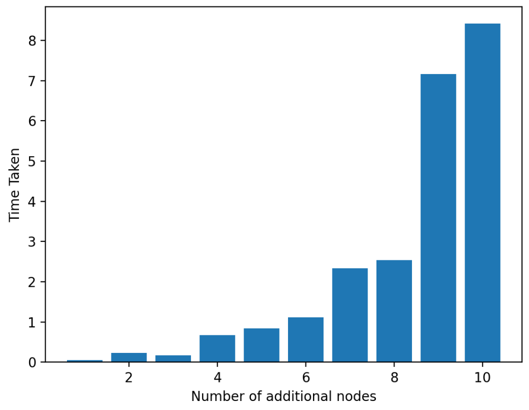

Figure 7. Average Time taken over ten trials to expand graph with additional nodes on top of a 5 node graph for a bipartite graph .

6.1.2 Quality of Graph Output.

We will now adress the second question and the quality of Grapher outputs. The output for a tree given in Figure 9 seems satisfactory for the user. However, this is expected because trees are the simplest of the graph classes we have implemented in Grapher . We can look at the larger graph classes implemented like chordal and bipartite graphs in Figure 10 and Figure 11 respectively to verify the quality of Grapher output. The bipartite graph includes a cycle and more edges than a tree. This means bipartite graphs also have the properties of

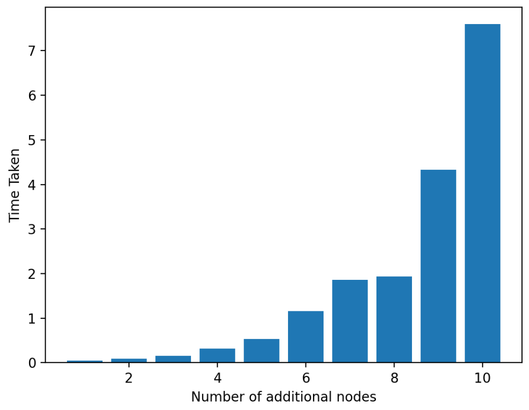

Figure 8. Average Time taken over ten trials to expand graph with additional nodes on top of a 5 node graph for a chordal graph.

a bipartite graph but are still complex. Similarly, the chordal graph contains multiple C_3's or 3 cycles. This means that the chordal graph does not simply output a tree. This output occurs because we increase the minimum depth of enumeration for more complex graphs. This increases the complexity of the graph and gives outputs that are not just trees.

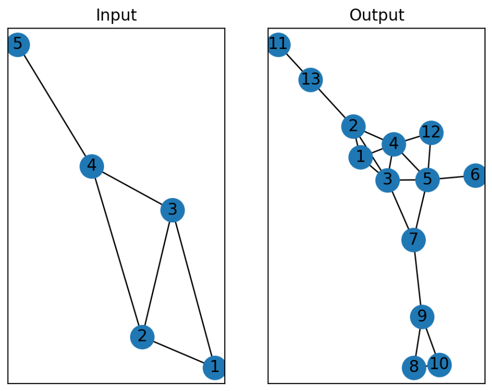

Figure 9. Example of input and output of a tree

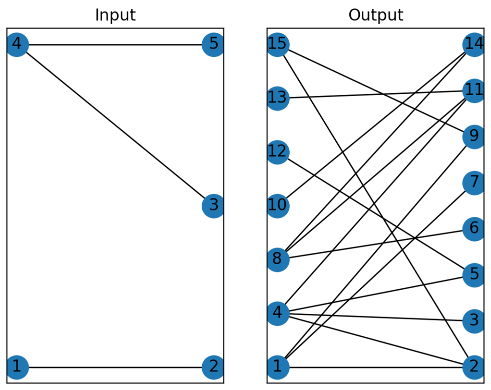

Figure 10. Example of input and output of a bipartite graph

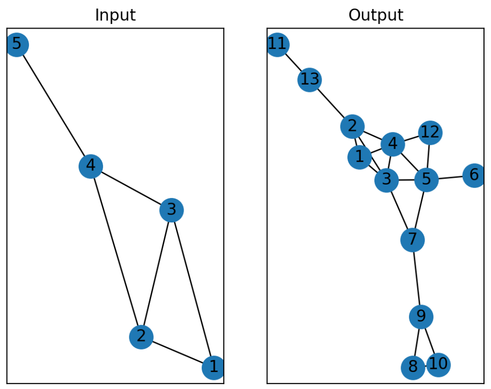

Figure 11. Example of input and output of a chordal graph

6.2 Graph Algorithm Synthesis

For graph algorithm synthesis we are confident that the programs outputted satisfy the examples based on our definition of equality. Therefore, we only need to make sure that the time taken is not excessive. To verify this fact we gave examples using random graphs that had outputs that were not satisfiable to Grapher . Therefore, we had to evaluate all programs to deduce there was no solution. The results of this expirement are shown in Figure 12. Note that the time also grows with the input because of the variable size DSL. Furthermore, we can verify that the time taken to enumerate all programs is not too long because it takes only 5 seconds to enumerate example graphs with 10 nodes.

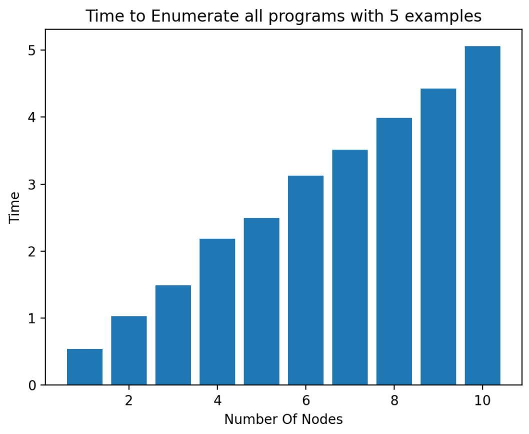

Figure 12. Time taken to evaluate all programs for graph algorithm synthesis

7 Future work

Throughout the paper we proposed many extensions to Grapher . In the future, we hope to implement these extensions. Recall that we discussed embedding graphs for a similarity measure, adding more graph classes, and allowing for approximation algorithms. We wanted to embed graphs to allow for users to not specify a graph class and still be able to have Grapher output a "similar" graph. Then, we wanted to add more graph classes that Grapher can synthesize to be more usefull. Finally, we wanted to allow for approximation algorithms to be able to add algorithms that approximate solutions to NP-Complete problems and allow for users to make mistakes in their examples. Additionally, we hope to speed up the graph extension process. We plan to add a heuristic approach for Grapher that involves adding edges to to nodes that have no edges. Furthermore, note that the order that edges are added do not matter to the final graph. We plan to make use of this fact to prune possible programs that add the same edges in a different order. These heuristics will make Grapher faster so it can synthesize larger graphs.

8 Conclusion

We have presented a tool to synthesize random graphs while maintaining user defined structure and a tool to synthesize graph algorithms. We have implemented both these tools in Grapher , an all purpose graph expansion and algorithm synthesizer. Furthermore, we have shown through multiple benchmarks that both processes are completed in a feasible amount of time and give good solutions. In future work, we hope to make grapher more robust and add the features described in Section 7.

9 Acknowledgements

We thank Yu Feng and Yanju Chen for offering and teaching CS190I at UCSB.

References

[1] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program synthesis using conflict-driven learning. In *Proc. Conference on Programming Language Design and Implementation*. 420–435.

[2] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. 2008. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference (SciPy 2008)*. 11–15.