

Generalizing cyclomatic complexity via path homology

Steve Huntsman
BAE Systems FAST Labs
Arlington, Virginia
steve.huntsman@baesystems.com

ABSTRACT

Cyclomatic complexity is an incompletely specified but mathematically principled software metric that can be usefully applied to both source and binary code. We consider the application of path homology as a more powerful analogue of cyclomatic complexity. There exist control flow graphs realizable at the assembly level with non-trivial path homology in arbitrary dimension. We exhibit several classes of examples in this vein while also experimentally demonstrating that path homology gives identical results to cyclomatic complexity for at least one detailed notion of structured control flow. Thus path homology generalizes cyclomatic complexity, and has the potential to substantially improve upon it.

CCS CONCEPTS

• **General and reference** → **Metrics**; • **Mathematics of computing** → *Algebraic topology*; *Paths and connectivity problems*.

KEYWORDS

software metric, cyclomatic complexity, path homology

1 INTRODUCTION

An archetypal software metric is the *cyclomatic complexity* of the control flow graph of a computer program [25]. Although cyclomatic complexity continues to be widely used [12], it has also been widely criticized, though there are no widely agreed-upon alternatives [1], much less any with a comparably mathematical underpinning that permits principled adaptation and generalization. Moreover, at least some of the criticism directed at cyclomatic complexity stems from ambiguity of control flow representations of source code that disappear at the level of disassembled binary code, where cyclomatic complexity can usefully guide testing and reverse engineering efforts such as fuzzing [10, 23]. Even at the source code level, cyclomatic complexity and other software metrics can contribute substantially to the identification of fault-prone or vulnerable code [2, 9, 26]. In short, cyclomatic complexity is an incomplete but principled software metric with useful applications to both source and binary code.

The recent series of papers [14–20] develops a theory of *path homology* that both generalizes the simplicial homology theory underlying cyclomatic complexity and directly applies to digraphs. Whereas the simplicial Betti numbers of a 1-complex equal the cyclomatic complexity in dimension 1 and are trivial in all higher dimensions (see §2.2), the path homology of a digraph is a richer topological invariant whose Betti numbers can take any values in every dimension. It is therefore natural to consider its application as a more powerful analogue of cyclomatic complexity.

2 HOMOLOGY

A *chain complex* over a field \mathbb{F} is a pair of sequences (indexed by a “dimension” $p \in \mathbb{N}$) of \mathbb{F} -vector spaces C_p and linear *boundary operators* $\partial_p : C_p \rightarrow C_{p-1}$ such that $\partial_{p-1} \circ \partial_p \equiv 0$. We can visualize such a structure as in Fig. 1, and write it as

$$\dots C_{p+1} \xrightarrow{\partial_{p+1}} C_p \xrightarrow{\partial_p} C_{p-1} \xrightarrow{\partial_{p-1}} \dots \xrightarrow{\partial_1} C_0 \xrightarrow{\partial_0} 0. \quad (1)$$

Writing $Z_p := \ker \partial_p$ and $B_p := \text{im } \partial_{p+1}$, the *homology* of (1) is

$$H_p := Z_p / B_p. \quad (2)$$

The *Betti numbers* are $\beta_p := \dim H_p = \dim Z_p - \dim B_p$.

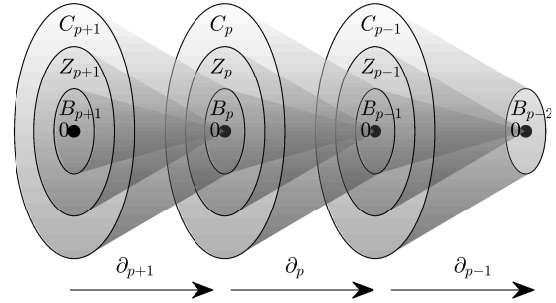


Figure 1: Schematic picture of a chain complex.

In practice, it is frequently convenient for technical reasons to (and we do) work with the *reduced* homology \tilde{H}_p . This has the minor effect $\tilde{H}_0 \oplus \mathbb{F} \cong H_0$, while $\tilde{H}_p \cong H_p$ for $p > 0$. Similarly, and using an obvious notational device, $\tilde{\beta}_p = \beta_p - \delta_{p0}$, where $\delta_{jk} = 1$ iff $j = k$ and $\delta_{jk} = 0$ otherwise.

2.1 Simplicial homology

Following [24], we now proceed to sketch the archetypal notion of *simplicial homology* that underlies cyclomatic complexity. An *abstract simplicial complex* (ASC) is a family Δ of finite subsets $\{v_0, \dots, v_p\}$ (called *simplices*) of a set V of *vertices* such that if $X \in \Delta$ and $\emptyset \neq Y \subseteq X$, then $Y \in \Delta$.¹

Given an ASC Δ , let C_p be the \mathbb{F} -vector space generated by basis elements $e_{(v_0, \dots, v_p)}$ corresponding to *oriented simplices* of *dimension* p in Δ . This essentially means that if σ is a permutation acting on (v_0, \dots, v_p) , then $e_{(v_0, \dots, v_p)} = (-1)^\sigma e_{(v_{\sigma(0)}, \dots, v_{\sigma(p)})}$.^{2 3}

¹ In other words, an ASC is a hypergraph with all sub-hyperedges.

² Thus for example $e_{(v_0, v_1, v_2)} = -e_{(v_0, v_2, v_1)}$.

³ Note that an order on V induces an order on each simplex in Δ , and in turn this induces an orientation.

The simplicial boundary operator ∂_p is now defined to be the linear map acting on basis elements as

$$\partial_p e_{(v_0, \dots, v_p)} = \sum_{j=0}^p (-1)^j e_{\nabla_j(v_0, \dots, v_p)} \quad (3)$$

where ∇_j deletes the j th entry of a tuple. It turns out that this construction yields a *bona fide* chain complex. Moreover, the simplicial Betti numbers measure the number of voids of a given dimension in a geometric realization of an ASC.⁴

For example, the boundary of a 2-simplex or “triangle” is

$$\partial_2 e_{(1,2,3)} = e_{(2,3)} - e_{(1,3)} + e_{(1,2)} = e_{(1,2)} + e_{(2,3)} + e_{(3,1)}$$

and its boundary in turn is

$$\partial_1 (e_{(1,2)} + e_{(2,3)} + e_{(3,1)}) = 0.$$

Thus the homology of the boundary of a triangle has $\beta_p = \delta_{p1}$: there is a single void in dimension 1, and none in other dimensions.

2.2 Cyclomatic complexity

The *cyclomatic number* or *cyclomatic complexity* [25] of an undirected connected graph $G = (V, E)$ is $v(G) := |E| - |V| + 1$: a result dating to Euler (for a modern treatment, see, e.g. [4]) is that this equals the dimension of the so-called cycle space of G . Meanwhile, the cyclomatic number also equals the first (simplicial) Betti number of the abstract simplicial complex whose 2-simplices correspond to edges [29], and all Betti numbers in $\dim > 1$ are identically zero.⁵

2.3 Path homology

Our sketch of path homology mostly follows [6, 18], with some small changes to notation and terminology that should be easily handled by the interested reader.

Let $D = (V, A)$ be a loopless digraph and let $p \in \mathbb{Z}_+$. The set $\mathcal{A}_p(D)$ of *allowed p -paths* is

$$\{(v_0, \dots, v_p) \in V^{p+1} : (v_{j-1}, v_j) \in A, 1 \leq j \leq p\}. \quad (4)$$

By convention, we set $\mathcal{A}_0 := V$, $V^0 \equiv \mathcal{A}_{-1} := \{0\}$ and $V^{-1} \equiv \mathcal{A}_{-2} := \emptyset$. For a field \mathbb{F} (in practice, we take $\mathbb{F} = \mathbb{R}$) and a finite set X , let $\mathbb{F}^X \cong \mathbb{F}^{|X|}$ be the free \mathbb{F} -vector space on X , with the convention $\mathbb{F}^\emptyset := \{0\}$. The *non-regular boundary operator* $\partial_{[p]} : \mathbb{F}^{V^{p+1}} \rightarrow \mathbb{F}^{V^p}$ is the linear map acting on the standard basis as

$$\partial_{[p]} e_{(v_0, \dots, v_p)} = \sum_{j=0}^p (-1)^j e_{\nabla_j(v_0, \dots, v_p)}. \quad (5)$$

A straightforward calculation shows that $\partial_{[p-1]} \circ \partial_{[p]} \equiv 0$, so $(\mathbb{F}^{V^{p+1}}, \partial_{[p]})$ is a chain complex. However, we are not concerned with the homology of this chain complex, but of a derived one.

Towards this end, set

$$\Omega_p := \left\{ \omega \in \mathbb{F}^{\mathcal{A}_p} : \partial_{[p]} \omega \in \mathbb{F}^{\mathcal{A}_{p-1}} \right\}, \quad (6)$$

$\Omega_{-1} := \mathbb{F}^{\{0\}} \cong \mathbb{F}$, and $\Omega_{-2} := \mathbb{F}^\emptyset = \{0\}$. If $\omega \in \Omega_p$, then automatically $\partial_{[p]} \omega \in \mathbb{F}^{\mathcal{A}_{p-1}}$, so $\partial_{[p-1]} \partial_{[p]} \omega = 0 \in \mathbb{F}^{\mathcal{A}_{p-2}}$. Therefore, $\partial_{[p]} \omega \in \Omega_{p-1}$. We thus get a chain complex (Ω_p, ∂_p) called the

⁴ Here, 0-dimensional voids amount to connected components.

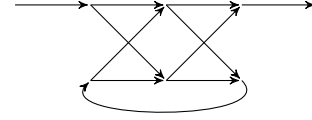
⁵ For background on simplicial homology, see [24].

(*non-regular*) *path complex* of D , where $\partial_p := \partial_{[p]}|_{\Omega_p}$.⁶ The corresponding homology is the (*non-regular*) *path homology* of D .

3 FLOW GRAPHS

Following [22], a *flow graph* is a digraph with exactly one source (i.e., a vertex with indegree 0) and exactly one target (i.e., a vertex with outdegree 0) such that there is a unique (entry) arc from the source and a unique (exit) arc to the target, and such that identifying the source of the entry arc with the target of the exit arc yields a strongly connected digraph.⁷ (We do not require the entry and exit arcs to be distinct, e.g., if the flow graph has only two vertices.)

We exhibit here a flow graph that is realizable as the assembly-level control flow graph of a program and with $\tilde{\beta}_\bullet = (0, 1, 1, 0, \dots)$:



This example was constructed via the following theorem of [13]:

THEOREM. For $L, n_1, \dots, n_L \in \mathbb{Z}_+$, define the digraph $K_{n_1, \dots, n_L}^{\rightarrow} := ([N], A_{n_1, \dots, n_L})$ where $N := \sum_{\ell=1}^L n_\ell$, $[N] := \{1, \dots, N\}$, $A_{n_1, \dots, n_L} := \bigcup_{\ell=1}^{L-1} A_{n_\ell, n_{\ell+1}}$, and

$$A_{n_\ell, n_{\ell+1}} := \left([n_\ell] + \sum_{k=1}^{\ell-1} n_k \right) \times \left([n_{\ell+1}] + \sum_{k=1}^{\ell} n_k \right).$$

Then

$$\tilde{\beta}_p (K_{n_1, \dots, n_L}^{\rightarrow}) = \delta_{p, L-1} \prod_{\ell=1}^L (n_\ell - 1). \quad \square \quad (7)$$

The proof for a variant of the flow graph above where the bottom arc is replaced by two consecutive arcs follows from the theorem above along with Theorems 5.1 and 5.7 of [18] (which hold in the non-regular context), so we shall not belabor the exact variant shown. Furthermore, adding “layers” to this example shows how we can produce valid flow graphs realizable as assembly-level control flow graphs and that have nontrivial path homology in arbitrarily high dimension.

3.1 2-flow graphs

We could exhibit nontrivial path homology in $\dim > 1$ for many other flow graphs, but we focus here on a highly restricted class of flow graphs to demonstrate that interesting behavior still occurs. The set of flow graphs in which the outdegrees of vertices are all ≤ 2 models control flow at the assembly level in typical architectures: the program counter either advances linearly through memory addresses as operations are executed, or it jumps to a new memory address based on the truth value of a Boolean predicate [7, 27].

⁶ The implied *regular path complex* amounts to enforcing a condition that prevents a directed 2-cycle from having nontrivial 1-homology. While [18] advocates using regular path homology, in our view non-regular path homology is simpler, more likely useful in applications such as the present one, and exhibits richer phenomenology.

⁷ Unlike in [22], here we disallow loops (i.e., 1-cycles) in digraphs, but the ramifications of this difference are straightforward. In particular, loops in control flow at the assembly level can be transformed into 2-cycles through the introduction of unconditional jumps to/from “distant” memory addresses in a way that does little violence to the actual binary code, and no violence at all to its semantics.

In practice, many vertices in assembly-level control flow graphs have outdegree 1, but indegree > 1 owing to the presence of inbound jumps.⁸ More generally, jumps into or out of an otherwise “structured” control flow motif that can be interpreted as an if, a while, or a repeat construct are called “unstructured” gotos in the programming literature [8, 25, 28]. These can be eliminated through control flow restructuring [3, 21, 30], so that the control flow is “structured” in the sense that it can be interpreted as arising from a combination of if, while, and repeat-type structures (though at the assembly level, every control flow operation is instantiated as a jump—i.e., a goto—of some sort, however it may be morally interpreted).

Rather than precisely define the notion of a structured flow graph *per se*, we consider the simpler, related notion of a 2-flow graph (2FG). A 2FG is a flow graph with a *source vertex* with outdegree 1, a *target vertex* with outdegree 0, a single vertex adjacent to the target with outdegree 1, and such that all other vertices have outdegree 2.

Suppose we construct all digraphs on N vertices with outdegree identically 2, except for a single vertex with outdegree 0.⁹ If such a digraph has a vertex a with nonzero outdegree and such that adding an arc to a from the unique vertex z with outdegree 0 results in a strongly connected digraph, then we call the digraph a 2FG progenitor at (a, z) . Note that a digraph can be a 2FG progenitor at (a, z) and at (a', z) for $a \neq a'$: indeed, the only way such a situation can be avoided is if a has indegree 0, in which case the strong connectivity requirement ensures that a is unique. Given a 2FG progenitor at (a, z) , we can construct a 2FG by adding a vertex s and the arc (s, a) and possibly also a vertex t and the arc (z, t) . Conversely, removing the entry arc, and possibly also the exit arc, from a 2FG yields a 2FG progenitor, so every 2FG can be constructed from a 2FG progenitor.

In Figs. 2 and 4 we show the 2FG progenitors with $\tilde{\beta}_2 > 0$ on 5 and 6 vertices, respectively; Fig. 3 shows an example of disassembled binary code whose control flow is that of the digraph on the right of Fig. 2. These examples make it clear that even in a very restrictive setting, the path homology of many different control flow graphs can be nontrivial in $\dim > 1$.

3.2 Control flow skeletons

We generated 20000 “program skeletons” each resulting from 20 uniformly random productions at uniformly random nonterminals from a context-free grammar along the lines of

$$\begin{aligned}
 S &\rightarrow S;S \\
 S &\rightarrow \text{if } b;S;\text{endif} \\
 S &\rightarrow \text{do while } b;S;\text{enddo} \\
 S &\rightarrow \text{repeat};S;\text{until } b
 \end{aligned}
 \tag{8}$$

where $;$ is shorthand for a newline. We then analyzed the resulting control flow graphs, in which every line had its own vertex. In every case, the equalities $\nu = |b| = \tilde{\beta}_1$ were satisfied, where here $|b|$

⁸ Such inbound jumps may land in the middle of what would otherwise be a basic block (i.e., a sequence of code statements without any [nondegenerate] control flow), which results in vertices with indegree and outdegree both equal to 1: in fact, a first attempt to produce Fig. 3 exhibited this behavior.

⁹ The number of such digraphs follows the sequence at <http://oeis.org/A003286>. That is, starting from $N = 3$, there are 1, 7, 66, 916, 16816, . . . such digraphs.

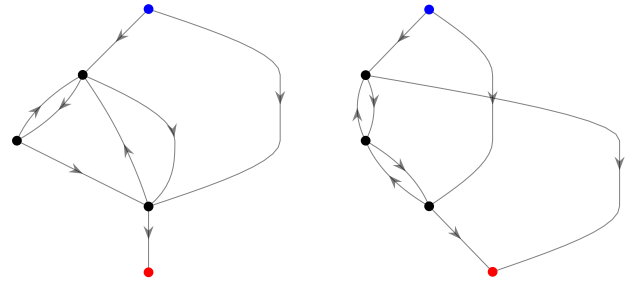


Figure 2: There are 2 5-vertex 2FG progenitors at (a, z) with $\tilde{\beta}_2 > 0$. The Betti numbers on the left and right are $(0, 0, 1, 0, \dots)$ and $(0, 1, 1, 0, \dots)$, respectively. Note that removing the target arc from the 2FG progenitor on the left yields the unique 4-vertex 2FG progenitor with $\tilde{\beta}_2 > 0$.

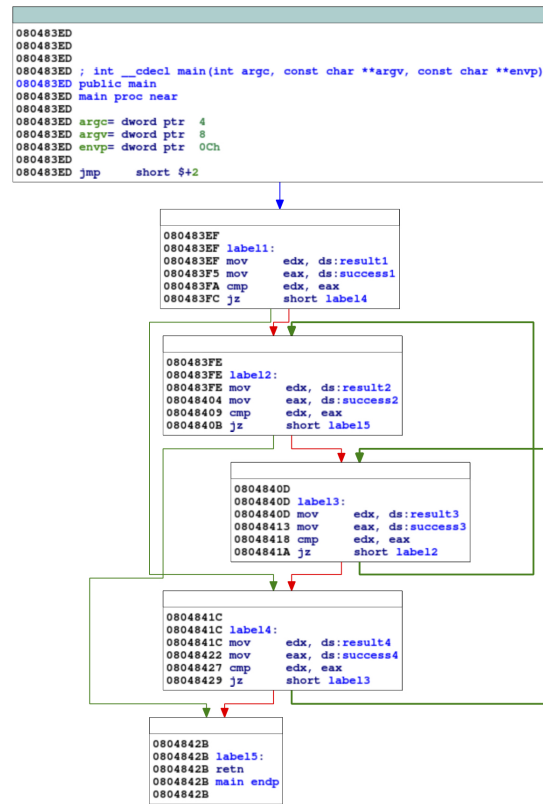


Figure 3: A control flow example with the (moral) structure of the digraph on the right in Fig. 2, shown in IDA Pro [11]. The Intel assembly instructions are directly compiled from C code (albeit using gotos and assembly statements). The common instruction motif in all of the basic blocks (i.e., binary code corresponding to vertices) except for the function exit clearly indicates how to construct binaries with control flow given by an arbitrary 2FG (progenitor).

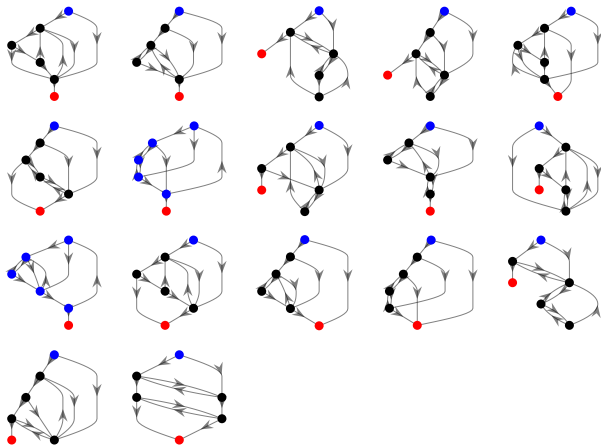


Figure 4: All 6-vertex 2FG progenitors at (a, z) with $\tilde{\beta}_2 > 0$. In each case $\tilde{\beta}_2 = 1$. From left to right and top to bottom, we have $\tilde{\beta}_1 = 0, 0, 0, 0, 1, 1, 2, 0, 0, 0, 1, 1, 1, 2, 0, 0, 0$; all other Betti numbers are zero.

indicates the number of “predicates” in the skeleton.¹⁰ Moreover, in every case we also had $\tilde{\beta}_2 = 0$. This suggests that, e.g., postprocessing, alternative conventions for control flow graph construction, or additional constructs such as case statements are necessary at the source code level in order to obtain useful invariants in $\dim > 1$.

To illustrate direct applicability to binary code, we also generated 20000 program skeletons whose control flow consisted of 16 conditional gotos to different addresses chosen uniformly at random. In this experiment, 55/20000 skeletons had $\tilde{\beta}_2 > 0$; we show the first such in the left panel of Fig. 5. In fact, this example is also more complex in other ways than the first skeleton with $\tilde{\beta}_2 = 0$, shown in the right panel of Fig. 5: e.g., the digraph drawing in the left panel of Fig. 5 has 15 arc crossings, while the digraph drawing in the right panel has only 10.¹¹

4 CONCLUSION

Path homology offers several improvements on cyclomatic complexity. With case statements, the Betti numbers can take on arbitrary values for control flow graphs at the source code level. The Betti numbers can also take on nontrivial values in arbitrary dimension for control flow graphs at the assembly level.

As a practical matter, it makes sense to analyze complicated flow graphs in a hierarchical and modular fashion using the program structure tree along the lines of [22]: this is not only scalable, but also extends the analogy of cyclomatic complexity in relation to essential complexity [25].¹²

In dimension 1 path homology appears to give the same results as cyclomatic complexity for structured control flow. While it is obviously desirable to turn evidence for this claim into proof, it is

¹⁰ The first equality here is generic, and can be proved via an easy counting argument, but the second equality is not obvious.

¹¹ Unfortunately, computing the minimal number of arc crossings is NP-hard, so we did not attempt to analyze arc crossings in any more detail.

¹² Tracking the birth and death of homology classes along these lines via tree persistence [5] is a particularly tempting prospect.

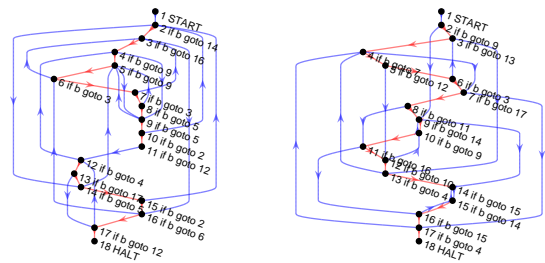


Figure 5: (L) The first of 55 (out of 20000) realizations of a control flow graph for a program skeleton generated through 16 uniformly random conditional gotos that result in $\tilde{\beta}_2 > 0$. Blue (resp., red) arcs indicate branches where a Boolean predicate (placeholder) b evaluates to \top (resp., \perp). This particular example has $\tilde{\beta}_\bullet = (0, 11, 1, 0, \dots)$. (R) The first of the remaining 19945 realizations that do not result in $\tilde{\beta}_2 > 0$. This particular example has $\tilde{\beta}_\bullet = (0, 13, 0, \dots)$.

already apparent that path homology has the potential to substantially improve upon an archetypal software metric.

ACKNOWLEDGMENTS

We thank Samir Chowdhury for many conversations regarding path homology, Greg Sadosuk for writing, compiling, and disassembling C code to produce Fig. 3, and Matvey Yutin for a separate analysis of analogues of various results from [18] in the non-regular context.

REFERENCES

- [1] S. Ajami, Y. Woodbridge, and D. G. Feitelson. 2019. Syntax, predicates, idioms—what really affects code complexity? *Empirical Soft. Eng.* 24 (2019), 287.
- [2] H. Alves, B. Fonseca, and N. Antunes. 2016. Software metrics and security vulnerabilities: dataset and exploratory study. In *IDCC*.
- [3] C. Böhm and G. Jacopini. 1966. Flow diagrams, Turing machines, and languages with only two formation rules. *Comm. ACM* 9 (1966), 366.
- [4] B. Bollobás. 2002. *Modern Graph Theory*. Springer.
- [5] E. C. Chambers and D. Letscher. 2018. Persistent homology over directed acyclic graphs. In *Research in Computational Topology*, E. Chambers, B. Fasy, and L. Ziegelmeier (Eds.). Springer.
- [6] S. Chowdhury and F. Mémoli. 2018. Persistent path homology of directed networks. In *SODA*.
- [7] K. D. Cooper and L. Torczon. 2012. *Engineering a Compiler, 2nd ed.* Morgan Kaufmann.
- [8] E. Dijkstra. 1968. Go to statement considered harmful. *Comm. ACM* 11 (1968), 147.
- [9] X. Du, B. Chen, Y. Li, J. Guo, Y. Zhou, Y. Liu, and Y. Jiang. 2019. LEOPARD: identifying vulnerable code for vulnerability assessment through program metrics. In *ICSE*.
- [10] D. Duran, D. Weston, and M. Miller. 2011. Targeted taint driven fuzzing using software metrics. In *CanSecWest*.
- [11] C. Eagle. 2011. *The IDA Pro Book: The Unofficial Guide to the World’s Most Popular Disassembler*. No Starch Press.
- [12] C. Ebert and J. Cain. 2016. Cyclomatic complexity. *IEEE Soft.* 33 (2016), 27.
- [13] T. Gebhart, S. Huntsman, S. Chowdhury, and M. Yutin. 2020. Path homologies of deep feedforward networks. In *ICMLA*.
- [14] A. Grigor’yan, R. Jimenez, Yu. Muranov, and S.-T. Yau. 2018. On the path homology theory of digraphs and Eilenberg-Steenrod axioms. *Homology Homotopy Appl.* 20 (2018), 179.
- [15] A. Grigor’yan, Yu. Muranov, V. Vershinin, and S.-T. Yau. 2018. Path homology theory of multigraphs and quivers. *Forum Math.* 30 (2018), 1319.
- [16] A. Grigor’yan, Yu. Muranov, and S.-T. Yau. 2014. Graphs associated with simplicial complexes. *Homology Homotopy Appl.* 16 (2014), 295.
- [17] A. Grigor’yan, Yu. Muranov, and S.-T. Yau. 2017. Homologies of graphs and Künneth formulas. *Comm. Anal. Geom.* 25 (2017), 969.

- [18] A. Grigor'yan, L. Yong, Yu. Muranov, and S.-T. Yau. 2012. Homologies of path complexes and digraphs. [arXiv:1207.2834](https://arxiv.org/abs/1207.2834)
- [19] A. Grigor'yan, L. Yong, Yu. Muranov, and S.-T. Yau. 2014. Homotopy theory for digraphs. *Pure Appl. Math. Quart.* 10 (2014), 619.
- [20] A. Grigor'yan, L. Yong, Yu. Muranov, and S.-T. Yau. 2015. Cohomology of digraphs and (undirected) graphs. *Asian J. Math.* 19 (2015), 887.
- [21] D. Harel. 1980. On folk theorems. *Comm. ACM* 23 (1980), 379.
- [22] S. Huntsman. 2019. The multiresolution analysis of flow graphs. In *WoLLIC*.
- [23] V. Iozzo. 2010. 0-knowledge fuzzing. In *Black Hat DC*.
- [24] D. Kozlov. 2008. *Combinatorial Algebraic Topology*. Springer.
- [25] T. J. McCabe. 1976. A complexity measure. *IEEE Trans. Soft. Eng.* SE-2 (1976), 308.
- [26] N. Medeiros, N. Ivaki, P. Costa, and M. Vieira. 2017. Software metrics as indicators of security vulnerabilities. In *ISSRE*.
- [27] S. S. Muchnick. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann.
- [28] E. Sennesh and Y. Gil. 2016. Structured gotos are (slightly) harmful. In *SAC*.
- [29] J.-P. Serre. 1980. *Trees*. Springer.
- [30] F. Zhang and E. H. D'Hollander. 2004. Using hammock graphs to structure programs. *IEEE Trans. Soft. Eng.* 30 (2004), 231.