Quantitative Epistemology & the Structure of Logical Knowledge

Michael Barany

Submitted 9 April, 2009

1 Logical Knowledge

The Pythagorean theorem states that in any right triangle the legs of length a and b and hypotenuse of length c are related by the equation

$$a^2 + b^2 = c^2. \tag{1}$$

Equation 1 represents an element of mathematical knowledge. More generally, the Pythagorean theorem is an element of logical knowledge. But the equation itself does not make this so.

The Pythagorean theorem is an element of logical knowledge because of the way it is established and the way it is used. It is established through a step-by-step logical derivation from a collection of foundational principles. It is used in combination with other established elements of knowledge in a particular way to establish further elements of logical knowledge. Logical knowledge is recognizable less by the contents of its assertions than by the logical way in which they are connected.

How, then, is the Pythagorean theorem connected to other statements? In Euclid's *Elements*, it appears as the 47^{th} proposition of Book I, after a long sequence of definitions, postulates, common notions, and other propositions. Euclid does not hold a monopoly on proofs of the theorem, however. Loomis (1968) compiles over 250 pages of proofs and demonstrations of the theorem. When taught to school children, the theorem appears after explanations of such things as right triangles, side lengths, and square numbers. In more

advanced mathematical settings, it is often stated without proof, and is used as a foundational idea with which to establish other statements.

In this essay, we shall seek an account of logical knowledge which applies to all these different settings for the Pythagorean theorem, and, indeed, to all works which might be called logical.[1] In so doing, we shall treat logical knowledge as a system, and ask what features of the system are properties of the system itself, as opposed to the isolated meanings and referents of the statements therein. Roughly speaking, we shall enquire after *structure* rather than *content*.[2]

A starting point for our analysis of systems of logical knowledge comes from the field of proof theory.[3] Accounts typically begin with a logical system divided into individual statements or elements.[4] How a work is so divided can depend on one's analytic framework and goals. Statements can be logical sentences, theorems, proofs, or other units of an argument. At a minimum, statements should contain some indicative assertion.[5] Proof theorists then study the relationships between these statements. Such relationships take a variety of forms. They could involve inference, explanation, definition, dependency, or some other form of logical connection. Many include logical principles such as implication and negation in their structural analyses.[6]

Structural accounts of logical knowledge are relevant to many considerations. In mathematics, they give a means of understanding and comparing different proof arguments. One might ask, for instance, how different proofs of the Pythagorean theorem using different techniques are nonetheless related. Conceptually, they give a means of analysing the relationships between different abstract concepts and conclusions.[7] Thus, one can relate points, angles, lines, triangles, and the theorems about them in a single network of knowledge and understanding. Pedagogically, one might ask which

[1] Importantly, we do not make the logicist claim that such works are *reducible* to logical principles, only that they carry a logical structure.

[2] On this distinction, see Suppes, 1969, 373.

[3] On proof theory, see Hunter, 1971, 7–8; Avigad and Reck, 2001; Paggiolesi, 2009.

[4] cf Carnap, 1937, 1–2; Tarski, 1956, 30. We shall prefer the term *statements*. [5] Lepore, 2000, 6. Our use of 'statement' is slightly broader than Lepore's. [6] cf Tarski, 1956, 30–37.

[7] Field, 1984, 512, makes these relationships a central feature of mathematical knowledge. Van Bendegem and Van Kerkhove, 2009, frame these relational concepts in terms of argumentation.

presentational approaches are simplest, or which concepts are most central to an area of study.[8] Structural analyses also play an important role in both the philosophy and practice of proof mechanization, proof languages, and the foundations of mathematics.[9]

The account in this essay will focus on two closely tied logical relations. The first is that of *dependency*. Dependency can take many forms in different logical frameworks. One might be primarily interested in the mathematical claims needed to establish the Pythagorean theorem, or the logical principles used in its proof. One might also be interested in the linguistic, semantic, or conceptual prerequisites for understanding the theorem or its proof.[10] Our account will not distinguish between these different forms of dependency, and may be applied to whichever suits one's particular analytic goals. A logical work is logical, in part, because of the network of logical interdependencies among its statements. These dependencies are a direct consequence of the structurally inferential character of logical knowledge.[11] However one parses a logical work, one finds that certain statements depend on other logically anterior statements.

The second logical relationship we shall consider is that of *citation*. The presentational process of citation connects the statements of a logical work to their dependencies. Citations are the *explicit* connections which show the *implicit* relations of dependency permeating a logical work. We say a statement A cites another statement B if B is cited in order to justify A. As before, different analytic contexts may call for different notions of dependency and citation.

In this essay, we understand the structure of logical knowledge in terms of its networks of citations and dependencies. Each element of logical knowledge fits into a *logical work*—a collection of statements, each themselves elements of logical knowledge. To each statement we associate a collection of other statements upon which it depends and another collection of statements which it cites.

The citation and dependency relations between statements form struc-

[8] cf studies of definitional systems in Kieffer, 2007, 49–52, 59–63; Friedman and Flagg, 1990; and Kieffer, et al. 2008.

[9] See Azzouni, 2009; Bridges and Reeves, 1999; Eder, 1992; Friedman, 2005; Kino, et al., 1970; Longo, 2003; MacKenzie, 2001; Pelc, 2009; Rav, 1999, 2007.

[10] On semantic and logical relations, see Tarski, 1956, 401–420. [11] Audi, 1998, 157.

tures of interconnectivity within the logical work. These structures can be analysed quantitatively using techniques from discrete mathematics. A case study in *quantitative epistemology*, this essay will present several ways of assigning a numerical measure of complexity to a statement in a logical work.

Quantitative epistemology, as it appears here, consists of the study of knowledge using numerical techniques.[12] Such techniques enable one to supplement intuitive or impressionistic analyses by re-framing qualitative problems in quantitative terms.

Consider the task of assessing the complexity of an idea. Upon reading a logical work, different readers may have a range of initial impressions on the complexities of its various claims and arguments. Different readers may also agree on certain intuitive principles: foundational statements are not very complex; terminal statements are more so; some statements are harder to understand than others. But these readers may be ill-equipped to say much more or to compare (or even identify) structural features of the work in detail. The measures in this work offer a numerical framework for discussing and comparing complexities which both sheds light on qualitative questions and considerations and opens up new avenues of inquiry and assessment.

A quantitative epistemological analysis, in order to be meaningful and useful, must accomplish three tasks. First, it must render knowledge in a form which can be analysed quantitatively. Second, it must develop means of quantification according to our epistemological intuitions. Third, it must use mathematical techniques to study these quantifications, and thereby add to our understanding of the systems of knowledge under consideration. We shall here attempt all three tasks.

The main body of this essay aims to provide an intuitive philosophical framing for the structure of logical knowledge. Appendix A presents this analysis in mathematical terms, including formal proofs of our results. Footnotes to numbered definitions, notations, lemmas, theorems, remarks, notes, and examples within this essay tie its informal assertions to our formal exposition. Appendix B gives `java` source code which automates the computations we describe. Outputs from these programs will help illustrate quantitative

[12]The phrase 'quantitative epistemology' is sometimes used to describe the epistemology of the quantitative sciences or numerical knowledge. In this essay, 'quantitative' describes our epistemological approach, not its object of study.

phenomena in this essay.

2 Logical Ordering

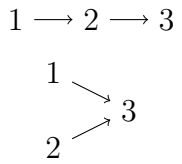
Section 1 began our first task for quantitative epistemology. We saw a logical work as a sequence of interrelated statements connected by dependencies and citations.[13] This view extracts the 'language free' structural features entailed by this collection of statements and relations and sets aside the referential content of the work.[14]

Our quantitative analysis shall be based on the observation that a logical work, so defined, contains two orderings. There is the *narrative order* in which statements are presented, and this coexists with a *logical partial order* from the work's dependencies. If statement A depends on statement B, then A logically follows B. Whenever statements depend on one another, they take this logical ordering.[15]

For example, figure 1(a) contains a work divided into three statements.[16] The first two statements of the work stand alone, with neither citations nor

1. Socrates is a man.
2. Man is mortal.
3. By 1 and 2, Socrates is mortal.

(a)



(b)

Figure 1: A logical sequence of statements about Socrates.

dependencies. Such independent foundational statements from which more complex statements are built are called *atoms*.[17] The atomic statements upon which a statement depends form its *logical support*.[18] The third statement in figure 1(a) depends on the first two, and cites them both directly. Figure 1(b) depicts the two orders for this work. In the narrative order, the three statements appear in their order of presentation. In the logical order, the first two statements are independent of each other, and both are logically

[13]Definition 1. [14]Suppe, 1974, 222; Suppe, 1989, 4. [15]Definitions 8, 9; Lemma 1.
[16]From Russell, 1919, 197–200; 1937, 7. [17]Definition 16. [18]Definition 17, Lemma 4.

prior to the third.

Figure 2 shows two more of the six possible arrangements of the statements from figure 1(a). Arrangement 2(a) is still logical, but 2(b) is not. In

1. Man is mortal.	1. By 2 and 3, Socrates is mortal.
2. Socrates is a man.	2. Socrates is a man.
3. By 1 and 2, Socrates is mortal.	3. Man is mortal.
(a)	(b)

Figure 2: Two more arrangements of our three statements about Socrates.

2(b), the first statement logically follows from the other two, but appears narratively prior to them. In a logical work, the narrative order *preserves* the logical order.[19] That is, if a statement is logically prior to another, it should also appear before that statement in the narrative.[20]

Our intuition also suggests that logical works should satisfy a criterion of *dependency transitivity*.[21] That is, if A depends on B, and B depends on C, then A must depend on C as well. For instance, if 'Socrates will die' depends on his being mortal, and this in turn depends on his being a man, then 'Socrates will die' also depends on his being a man, at least in the context of that argument.[22]

There is no analogous 'citation transitivity' requirement. Importantly, each statement need not cite every single statement upon which it depends. Instead, one may always determine a statement's dependencies by following its citations, its citations' citations, and so forth. This process is called *citation chasing*.[23] If a work's citations reveal all of its dependencies in this way, that work is *thorough*. Conversely, a work is *concise* if citation chasing does not yield any erroneous non-dependencies.

Well written works are both concise and thorough.[24] A well written work's citation information gives a complete and accurate portrayal of its

[19]Definition 5 (logical ordering criterion), Remark 4, Example 7.

[20]In ordinary exposition, one sometimes states conclusions before justifying them. For logical knowledge, however, one can always array the logical antecedents of a statement in front of the statement itself. Definition 6. In particular, there can be no circular reasoning. Remark 6.

[21]Definition 5, Remark 5. [22]There may be other ways to derive Socrates's death, but these would appear as separate arguments.

[23]Remark 8, Definition 4. [24]Definition 7.

dependency information through citation chasing.[25] Thus, the algorithms in appendix B ask only for citation data, and automatically compute dependencies.

From the ordering properties just described, we can diagrammatically represent the structure of a logical work without requiring the work itself to be at hand. These diagrams, rather than their associated works, form the basis for our structural analysis.

There are two diagrams associated to each logical work.[26] The first is the Hasse (dependency) diagram of the logical partial ordering. Here, if A is logically prior to B then A appears below B. If A depends on B, and there are no statements C such that A depends on C and C depends on B, then A and B are connected by a line (called an *edge*) in the diagram. The second diagram is the citation diagram, and looks like the Hasse diagram, but with two statements connected by an edge whenever one cites the other. In a well written work, the dependency diagram will be a copy of the citation diagram with edges corresponding to citations which are not needed for citation chasing removed.

In the example from figure 1, both diagrams are the same (figure 3). Figure 4 shows a simple example where the dependency and citation diagrams

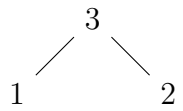


Figure 3: The dependency and citation diagram for figure 1.

differ.[27] In 4(a), the first two statements are atomic. Dependencies ($d(s)$) and citations ($c(s)$) for the other statements are in 4(d). Statement 5 is not connected to statements 1 and 2 in the dependency diagram, however, because statement 3 comes between statement 5 and statements 1 and 2 in the logical order for this work. It is for this reason that, although statements in well written works always have at least as many dependencies as citations,

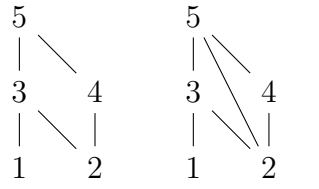
[25] If it does this without redundancy, it is called *minimally concise*. Definition 12, but see Remark 14.

[26] Remark 11.

[27] See also proof 9 of Example 15. In both cases, the works contain citations which are not strictly necessary to establish their dependencies, so are not minimally concise.

1. Man is mortal.
2. Socrates is a man.
3. By 1 and 2, Socrates is mortal.
4. By 2, Socrates is human.
5. Socrates is a mortal (by 3) human (by 4) male (by 2).

(a)



(b) (c)

s	$d(s)$	$c(s)$
3	1, 2	1, 2
4	2	2
5	1, 2, 3, 4	2, 3, 4

(d)

Figure 4: A work (a) with different dependency (b) and citation (c) diagrams and data (d).

those works' citation diagrams generally have more (and never have fewer) edges than their dependency diagrams.

3 Numerical Complexity

Sections 1 and 2 concern the first task of quantitative epistemology from page 4, namely, to present knowledge in a quantifiable form. This section will consider the second and third tasks with respect to the specific problem of quantifying complexity.

We start with our informal intuitions: a statement which depends on many statements should be more complex than a statement depending on only a few; the level of interdependence of those statements upon which a statement depends should also affect that statement's complexity; statements should be more complex than those upon which they depend, for it is impossible to grasp them without first understanding their antecedents.

The proof theoretic literature on proof complexity, particularly as it relates to computation, focuses on the scaling properties of different proof structures.[28] It is less concerned, however, with our focus on evaluating the relative complexity of smaller works and with studying intuitive notions of

[28]Avigad and Reck, 2001, 37. See also Cook and Rechkow, 1979; Lahiri, et al. 2007; Urquhart, 1995.

what makes proofs more or less complex.

Kieffer gives two structural measures of complexity.[29] One measure, which I call the statement's girth $g(s)$, counts the number of statements upon which the statement depends, along with the statement itself.[30] The second, which I term the statement's rank $r(s)$, records how tall the statement's dependency diagram is.[31] This corresponds to the length of the longest chain of implications starting from the statement in question.[32] A statement's rank is also related to 'how far' one must go in chasing its citations.[33]

From Stanley, we can regard the number of possible narrative orderings of a work as a measure of complexity.[34] This measure, denoted $e(s)$, quickly becomes unwieldy to compute by hand, but its computation can be easily automated for relatively small works.[35]

Kieffer's measures give a rapidly computable and intuitive sense of complexity, and combine to show different aspects of a statement's network of dependencies. Girth measures complexity according to the idea that the number, not the kind, of statements depended upon by s determines its complexity. Rank corresponds to the intuition that a statement is not much more complex than its most complex antecedent, regardless of how many dependencies it has or how they are interrelated.

We might hope for a measure which better reflects the interdependencies amongst a statement's dependencies. Stanley's measure agrees with our intuition at one extreme, assigning a very low complexity to chains. At the other extreme, however, it conflicts with our intuition that works with very little interdependence amongst statements should have low complexities.

In addition, we might want complexity measures to take account of not just dependencies but also the interaction of citations with those dependencies. Graph theory supplies some measures of graph complexity, such as spanning tree number, but these do not account for the crucial ordering properties of logical works. There do not appear to be combinatorial or proof-theoretical measures in the literature which adequately capture the phenomenon of citational complexity. We will measure citational complexity by applying the measures derived from dependency diagrams to citation diagrams. This solution accords with our intuition that a structure's complexity

[29]Note 24. [30]Definition 18. [31]Definition 15. [32]Note 20. A *chain* has the form 'A implies B, B implies C,...'. Example 28. [33]Lemma 3. [34]Notes 13, 25. [35]Remark 19.

has primarily to do with its internal interconnectivity.

To focus our intuitions, consider how one encounters a statement. Typically, one does not synoptically see the entire network of assertions required to support the statement.[36] Instead, one understands the statement in the context of those statements immediately connected to it—either those it cites or those dependencies which are most immediately pertinent. We shall thus derive measures of complexity from the complexities of a statement's immediate predecessors in the dependency and complexity diagrams. A statement's complexity is then recursively aggregated from the complexities of those statements upon which it depends.

We start by assigning a complexity of 1 to all atomic statements. Depending on one's intuitions, there are two basic ways for numerical complexity to aggregate from statement to statement. If the statement in question depends on two prior statements, their complexities contribute to that statement's complexity either additively (arithmetically)[37] or multiplicatively (geometrically).[38] Additive aggregation corresponds to the intuition that understanding a new statement is not much harder than understanding each of the statements upon which it depends, or that a conclusion automatically follows from a complete set of premises. Multiplicative aggregation corresponds to the intuition that the more complex a statement is, the harder it is to incorporate it with other statements in order to understand something new, or that more complex premises are harder to synthesize into a conclusion.

Figure 5 shows the diagrams from figures 3 and 4 with statement numbers replaced by complexity measures. Comparing 5(a) to 5(b) shows little difference between arithmetic and geometric measures for very simple statements. The numbers diverge rapidly, however, as statements and their antecedents become increasingly complex.

We call the complexity measures just considered $ac_d(s)$, $ac_c(s)$, $gc_d(s)$, and $gc_c(s)$. The first two are arithmetic complexities, the second two are geometric complexities, and the subscripts indicate whether the complexity is calculated according to the work's dependency (d) or citation (c) diagram. Although it is not itself a measure of complexity, we shall also compute

[36] cf Wittgenstein, 1956, II: 1–13 on surveyability and Coleman, 2009. Avigad, 2008, 2 discusses step-by-step versus global understanding.

[37] Definitions 20, 21. [38] Definitions 22, 23.

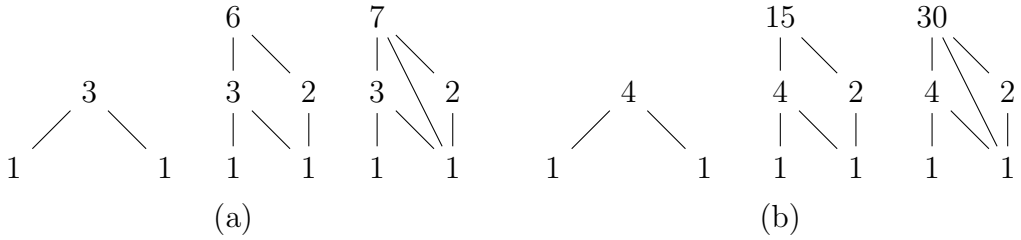


Figure 5: Diagrams showing the arithmetic (a) and geometric (b) aggregation of complexity.

the centrality $z(s)$ of a statement. A statement's centrality is one plus the number of statements which depend on it.[39]

4 Comparing Complexities

For our second task for quantitative epistemology from page 4, we have applied our intuitions to the framework developed from the first task to establish a range of complexity measures. As an example from the third task, let us compare the complexities of statements with similar girths. In other words, we shall ask how different structures for works with the same number of statements affect the complexities found in those works. Figure 6 shows the aggregation of arithmetic dependency complexity for all possible works of size 4, with geometric complexity given in parentheses where it differs from arithmetic complexity.[40]

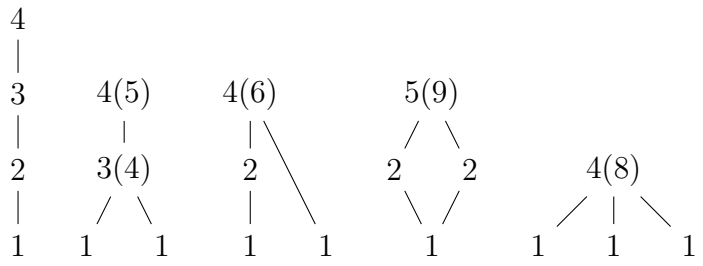


Figure 6: Dependency diagrams for statements of girth four.

[39]Definition 19. The 1 is added for conceptual symmetry with our intuitive notion of girth: if all dependencies of a work were 'flipped' then girth would become centrality and vice versa.

[40]Some of these structures arise in Example 15.

Why does the fourth diagram reach greater complexities than the others? One answer comes from studying a related question: how complex can a statement of a given girth be?[41] For citation complexity, if $g(s) = n$ then $ac_c(s)$ is at most 2^{n-1}.[42] Such complexities arise when each statement cites every statement appearing narratively before it (figure 7).

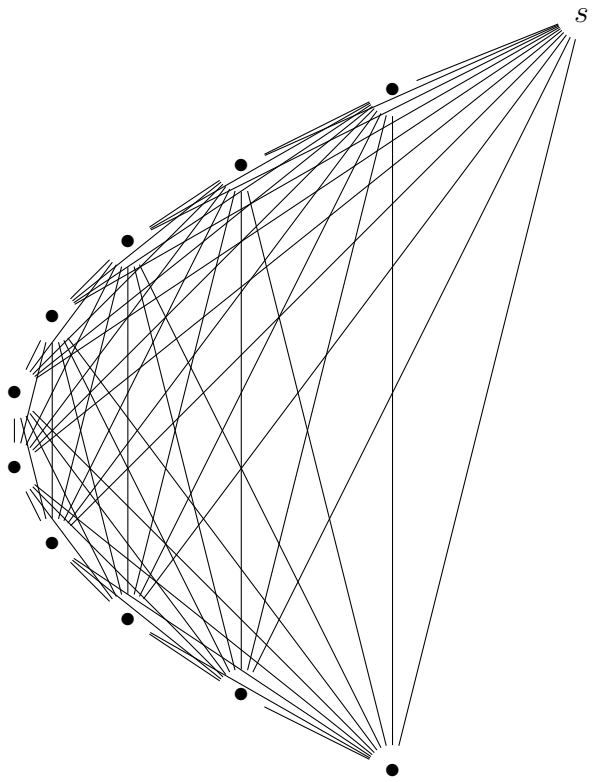


Figure 7: The citation diagram for a work maximising $ac_c(s)$.

For dependency complexity, unlike for citation complexity, adding dependencies to statements in a work does not necessarily make the terminal statement more complex. In fact, figure 7 gives the *minimal* dependency complexity for a statement of the same girth. In order to maximise dependency complexity, the statements in a logical work must be highly interconnected, but not so connected that they all start to form one long chain. The

[41] Avigad, 2008, 17 shows the importance of this question to automated proofs involving inequalities.

[42] Lemma 6.

ideal balance is achieved by works with one of the structures in figure 8,[43] for which the maximum possible dependency complexity can be computed directly.[44] Such works are roughly of the form: 'A, B, and C show D, E, and F; which show G, H, and I; which show. . . .' Our proof begins by maximising arithmetic complexity without changing the rank of any of the statements,[45] then fixes only the rank of the terminal statement,[46] and finally allows the rank of the terminal statement to vary.[47]

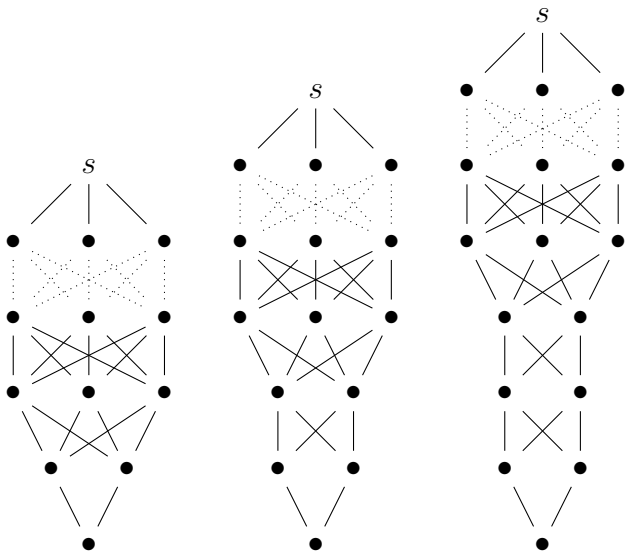


Figure 8: Works maximising $ac_d(s)$.

The third task of quantitative epistemology can also add empirical results to theoretical ones. We shall now use our complexity measures to compare the structures of two logical works. Figures 9 and 10 depict diagrams of, respectively, statements from Euclid's *Elements* and Russell and Whitehead's *Principia Mathematica*, with citation data taken from the respective translator's or authors' annotations.[48] Edges which are in the citation diagram

[43]Theorem 15. [44]Corollary 16. [45]Theorem 8.

[46]Lemmas 9, 11; Corollaries 10, 12; Theorem 14. [47]Example 35; Theorem 15.

[48]One may easily infer embedded citations which were not made explicit in the annotations. We use just the annotations for consistency and convenience. In Russell and Whitehead's case, we treat abbreviations such as 'The principle [1.3] will be referred to as "Add." ' as separate statements (100).

but not the dependency diagram are dotted. Both propositions have girth 20, but their dependencies are arranged with very different structures.

In the Euclidean proposition's diagram, we observe a large collection of atomic statements. These are combined and built into a fairly simple chain of statements rapidly ascending in rank up to proposition I.15, forming a diagram which is both taller and wider. For Russell and Whitehead's proposition, there is a small collection of atoms, and the number of statements at each rank decreases only gradually. Dependencies are regularly distributed about the statements, as are, to a slightly lesser extent, the citations.

How are these differences manifested quantitatively? Table 1 gives the computer-generated complexity data for our propositions.[49] In both the arithmetic and geometric cases, Russell and Whitehead's proposition has a higher dependency complexity and Euclid's has a higher citation complexity.

s	$r(s)$	$ac_d(s)$	$ac_c(s)$	$gc_d(s)$	$gc_c(s)$
I.15	9	20	118	1710	$7.26 \cdot 10^{18}$
2.17	6	28	49	49476	$4.14 \cdot 10^{9}$

Table 1: Complexity data for Euclid's proposition I.15 and Russell and Whitehead's proposition 2.17.

The higher dependency complexity of Russell and Whitehead's proposition reflects its more even distribution of statements across different ranks. For Euclid's proposition, higher citation complexities come not from less economy in citations (both diagrams have 33), but from the tendency of extra citations to come from more central statements.

In a sense, Euclid's proposition is more citationally redundant than Russell and Whitehead's.[50] That is, it contains more citations which are not necessary to merely establish the relations of dependency among the proposition's dependencies, and those extra citations are arranged to produce greater citational complexities. Of course, there is far more to a logical demonstration than merely establishing which statements depend upon which others. Measurements of structural complexity show only one dimension of comprehensibility or quality of presentation.[51] Nonetheless, our brief empirical study has unearthed certain structural differences between the works of Euclid and

[49]For their dependencies' complexity data, see Examples 29 and 30. [50]Definition 24.

[51]See Kieffer, 2007, 60.

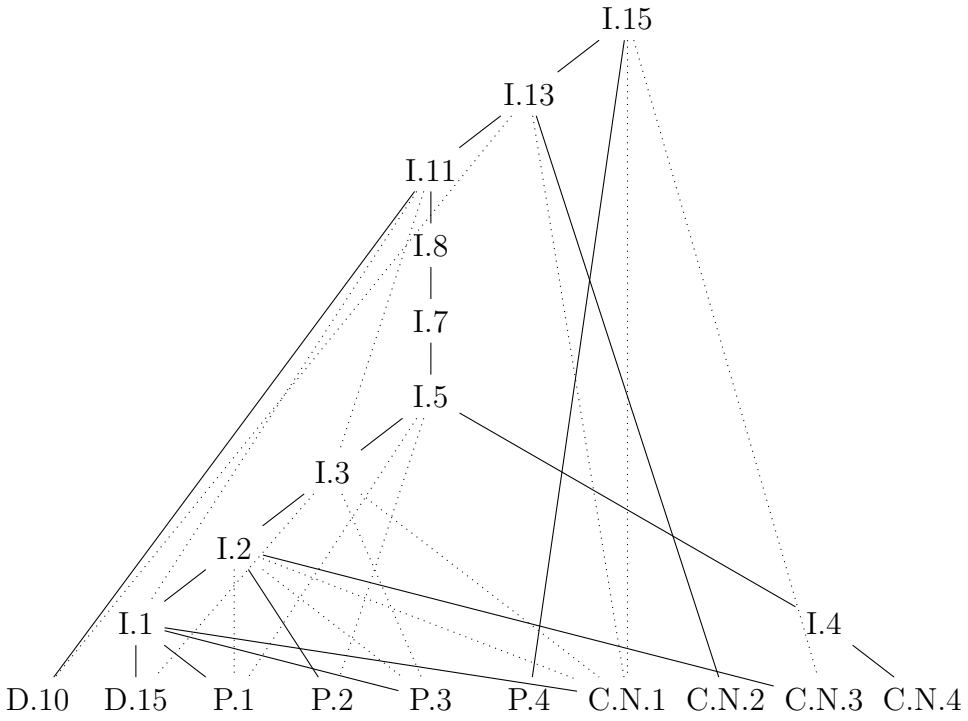


Figure 9: Euclid's proposition I.15 and its dependencies.

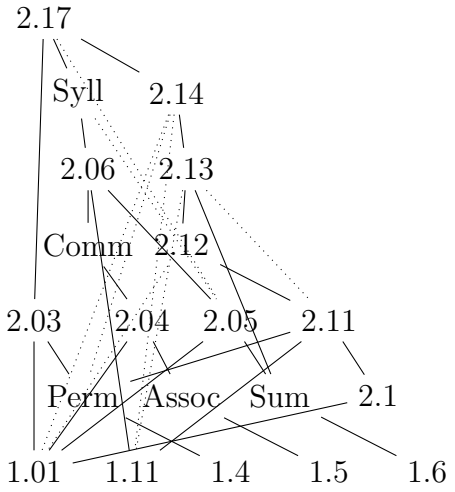


Figure 10: Russell and Whitehead's proposition 2.17 and its dependencies.

Russell and Whitehead which would not necessarily have been unearthed without our quantitative approach.

5 Conclusion

Beginning with the question of how logical knowledge is structured, we developed a range of quantitative indications of structural complexity and studied them in both extremal and empirical situations. To do so, it was necessary to see logical knowledge in a quantifiable way. This involved presenting a logical work as a collection of interconnected statements and studying the structure of its connections.

The recursive understanding of a statement's complexity gleaned from studying the logical ordering of a work reflects the local process of understanding and validating logical knowledge on the basis of a statement's immediate logical and citational antecedents. Such an understanding is quite flexible, and section A.6 of appendix A suggests one of many possible adaptations of the approach for a more nuanced study of complexity.

The framework from this essay is both pliable and powerful. It is pliable in its adaptability to a wide spectrum of analytic contexts which might arise in the study of the structure of logical works. Such an adaptability comes through abstracting from two fundamental structural and presentational features of logical works: namely, dependency and citation.

Our framework is powerful in its double capacity to facilitate new means of understanding through both theoretical and empirical channels. Our main theoretical result from appendix section A.5 is meaningful in qualitative terms, but could neither have been conceived nor proven without our quantitative approach. In an empirical context, our quantitative approach helped make visible some structural differences between two canonical works in the foundations of mathematics. A quantitative orientation enables one to seek empirical laws in addition to theoretical ones[52] while simultaneously supplementing the possibilities for meaningful case-by-case comparisons.

The scope of this essay has been small, but its aims far-reaching. We have elucidated a programme of quantitative epistemology through which, it is hoped, one might better understand the underlying structural features

[52]cf Kieffer et al., 2008, 14, on maximum quantifier depth.

permeating logical knowledge, as such. Our preliminary results indicate that such quantitative techniques may well be applied to a wide range of both theoretical and empirical problems in epistemology. Most big extant questions for epistemology may not have numerical answers, but this essay would suggest they might have numerical solutions.

A Formal Combinatorial Development

A.1 Logical Works

Definition 1. A *work* W is a set (S, d, c) with the following components:

- S is a finite ordered set (intuitively, a sequence of *statements* with a *narrative ordering*). Denote the set of subsets of S by 2^S.
- $d : S \to 2^S$ is a *dependency function* (intuitively, sending a statement to those upon which it depends).
- $c : S \to 2^S$ is a *citation function* (intuitively, mapping a statement to those it cites directly).

Notation 2. S can be generically written as $\{s_1, s_2, \ldots\} = \{s_i\}$ with statements indexed by their position in the narrative ordering.

Note 1. Definition 1 generalises the formulations of Friedman and Flagg (1990, 29–34) and Kieffer (2007, 49–52, 59–63) for *definition directed acyclic graphs* and *knowledge maps*.

Definition 3. We can define dual notions for the functions d and c:

- $d^* : S \to 2^S$ with $d^*(s) = \{t : s \in d(t)\}$, and
- $c^* : S \to 2^S$ with $c^*(s) = \{t : s \in c(t)\}$.

These are the sets of statements which depend upon or cite s.

Note 2. These dual notions will be used to indicate structural features of logical works in section A.4. Tarski (1956) bases his analysis on a notion comparable to d^*, rather than d.

Definition 4. For a subset $T \subseteq S$, let $d(T) = \cup_{s \in T} d(s)$. Write $d^1(T) = d(T)$ and $d^i(T) = d(d^{i-1}(T))$ for $i \geq 2$. Define $c^i(T)$ analogously.

Definition 5. W is *logical* if it satisfies the conditions of

- (dependency transitivity) $\forall s \in S, i > 0$, $d^i(s) \subseteq d(s)$, and
- (logical ordering) if $i \leq j$ then $s_j \notin d(s_i)$.

Note 3. The third axiom of Tarski's (1956) system of metamathematics is analogous to dependency transitivity, but for d^* instead of d (31).

Remark 4. Negating the logical ordering condition gives an equivalent formulation: $s_i \in d(s_j)$ implies $i < j$.

Remark 5. Dependency transitivity says that if s depends on s' then s also depends on everything on which s' depends. Logical ordering requires that new statements depend only on those which have already been established.

Remark 6. Logical ordering forbids circular statements, namely $s \in d(s)$.

Example 7. Libraries of formal mathematics such as the Mizar Mathematical Library maintain databases of mathematical results proved under particular formal systems, aggregating smaller articles into large meta-works. These libraries are managed in way which depends on the logical ordering criterion of definition 5. In the case of the Mizar Library, the narrative ordering is based on the order of article submission. The logical ordering properties of the articles guarantee that they will fit into a logically valid larger system when they are narratively concatenated in the project. (MPTP, online.)

Definition 6. W is *logically viable* if there exists an ordering on the elements of S such that W is logical.

Definition 7. A logical work W is *thorough* if for all $s \in S$ we have $d(s) \subseteq \cup_{i>0} c^i(s)$. It is *concise* if $\cup_{i>0} c^i(s) \subseteq d(s)$. It is *well written* if it is both thorough and concise. That is, if $d(s) = \cup_{i>0} c^i(s)$.

Remark 8. $\cup_{i>0} c^i(s)$ contains all the statements reached if one starts at s and adds all the statements cited by s, followed by all the statements cited by those statements, and so forth. It is thus a formalisation of the idea of 'citation chasing', where one follows the citations of a statement to the statements cited by each cited statement, and so on. In a thorough work,

citation chasing produces all of the dependencies of a statement, whereas a concise work does not contain statements which cite other statements upon which they do not depend. In fact, by the dependency transitivity criterion of definition 5 the definition of a concise work is equivalent to simply requiring $c(s) \subseteq d(s)$ for all $s \in S$.

A.2 Logical Ordering

For the remainder of this appendix, works will be assumed logical. In this section, we identify a partial ordering structure in the system of dependencies described in the preceding section.

Definition 8. Define the order relation $<_d$ on S by $s <_d t$ if $s \in d(t)$.

Definition 9. Let $s \leq_d t$ in S if $s = t$ or $s <_d t$.

Note 9. By remark 6, $s <_d t$ implies $s \neq t$.

Note 10. I will use Stanley's (1997) terminology for partially ordered sets, or *posets* (96–149).

Lemma 1. *$\leq_d$ gives a partial ordering on S.*

Proof. Reflexivity ($s \leq_d s$) is by definition. Antisymmetry ($s \leq_d t$ and $t \leq_d s$ implies $s = t$) follows from the logical ordering condition on W. Specifically, $s \leq_d t$ implies that either $s = t$ or $s \in d(t)$. In the latter case, s must appear before t in the narrative order on S, and so we cannot have $t \in d(s)$, and thus $t \not\leq_d s$. Transitivity ($r \leq s$ and $s \leq t$ implies $r \leq t$) follows from dependency transitivity if $r \neq s \neq t$, for $r \in d(s) \subseteq d(d(t)) \subseteq d(t)$ as $s \in d(t)$. Otherwise, transitivity holds trivially. □

Remark 11. In addition to drawing the Hasse diagram for the statements in a work, we can visualise the relationships between its elements by drawing a *citation graph* with edges going from each statement to those statements which it cites. We can also allow multiple edges if a statement is cited multiple times by the same statement. For the below, however, I will assume the citation graph is simple. For multi-edge versions, one can refer to the weighted variants in section A.6 with the weight equal to the number of edges. Using definition 15 and the narrative order on S it is possible to draw the

citation graph in a canonical way, so that statements only cite those below them in the graph and two statements on the same level of the graph appear in narrative order, left to right.

Notation 10. Write S_d for the set S with the ordering $\leq_d$ and S_n for S with its narrative ordering. We will preserve the notation S for the underlying set and $\{s_i\}$ will be used to indicate elements of S as before, regardless of ordering.

Note 12. $S_n \cong \mathbf{n}$, where $\mathbf{n}$ is the set $[n] = \{1, 2, \ldots, n\}$ with its numerical ordering and $n = |S|$. They are related by the bijection $s_i \leftrightarrow i$.

Definition 11. Define $\sigma : S_d \to \mathbf{n}$ by $\sigma(s_i) = i$.

Note 13. A *linear extension* is an order preserving bijection into a totally ordered set. One such natural linear extension comes from the map σ between the dependency and narrative orders for the work, as we show below. In general, linear extensions of S_d correspond to narrative orderings of W such that W remains logical. The number of such extensions is denoted $e(S_d)$.

Stanley (1997) calls $e(P)$ "probably the single most useful number for measuring the 'complexity' of [a poset] P" (110). We shall consider $e(S)$ among other complexity measures in section A.4.

Lemma 2. *σ is an order preserving bijection, and hence a linear extension of S_d.*

Proof. σ is trivially a bijection between the ground set S of S_d and the ground set $[n]$ of $\mathbf{n}$ by note 12. If $s_i \leq s_j$ then either $s_i = s_j$ or $s_i \in d(s_j)$. In the latter case, $i < j$ by remark 4. Hence, $s_i \leq_d s_j$ implies $i \leq j$. □

Definition 12. A work is *minimally concise* if for each $s \in S_d$ the set $c(s)$ consists of those statements covered by s, namely those $t \in S$ such that $t <_d s$ and there does not exist an r satisfying $t <_d r <_d s$. (The sense of minimality is given in definition 24.)

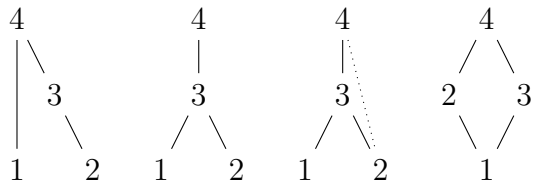
Remark 14. While it is generally possible for a logical work to be written concisely, it might not be possible for it to be minimally concise. Particularly with elementary constructions or definitions, one may need to invoke both that statement and a statement depending on it in a new statement. For

example, a definition of continuous functions on the real line might depend on the notion of a countable sequence of real numbers, and real numbers might in turn be constructed in a way depending on the notion of countable sequences.

Example 15. Compare the following 4-statement proofs from Lemmon (1993, 14–15, 21). Lemmon gives the statements (with parenthetical numbers) in the middle column, with citations and rules of inference to the right and the assumptions upon which the statement rests (compare this to definition 17 of a *logical support*; in general, these assumptions are contained in the logical support—unless the statement itself is an assumption—but may not make up the entire support) to the left. The rules of inference are Assumption (A), Double Negation (DN), Modus Tollendo Tollens (MTT), Conditional Proof (CP), &-Introduction (&I), and &-Elimination (&E).

7. $P \to -Q, Q \vdash -P$	**8.** $-P \to Q, -Q \vdash P$
1 (1) $P \to -Q$ A	1 (1) $-P \to Q$ A
2 (2) Q A	2 (2) $-Q$ A
2 (3) $--Q$ 2 DN	1,2 (3) $--P$ 1,2 MTT
1,2 (4) $-P$ 1,3 MTT	1,2 (4) P 3 DN
9. $P \to Q \vdash -Q \to -P$	**17.** $P\&Q \vdash Q\&P$
1 (1) $P \to Q$ A	1 (1) $P\&Q$ A
2 (2) $-Q$ A	1 (2) P 1 &E
1,2 (3) $-P$ 1,2 MTT	1 (3) Q 1 &E
1 (4) $-Q \to -P$ 2,3 CP	1 (4) $Q\&P$ 3,2 &I

The citations for each proof are the statements referenced in the right column. The statements labeled assumptions have neither dependencies nor citations. In each proof, (4) depends on all of the preceding statements, but does so by a different underlying structure in each case. For all but (4) in each proof, a statement's citations are identical to its dependencies. Here are the respective dependency diagrams. In proof 9, there is an extra dotted edge which is in the citation diagram but not in the dependency diagram.

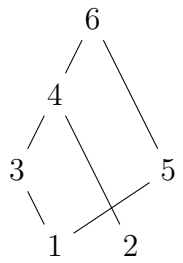


7. **8.** **9.** **17.**

Example 16. Here is a logical proof from Bergmann, et al. (2004, 160–161).

1. Whales breathe by lungs and are warm-blooded.	Assumption
2. If whales breathe by lungs, then whales are not fish.	Assumption
3. Whales breathe by lungs.	From 1
4. Whales are not fish.	From 2 and 3
5. Whales are warm-blooded	From 1
6. Whales are warm-blooded and whales are not fish	From 4 and 5

Statements 1 and 2 are atomic, and statements 3 and 5 depend upon and cite only statement 1. Statement 4 cites and depends on statements 2 and 3, and also depends on statement 1 because statement 3 does. Statement 6 cites statements 4 and 5, and depends upon all of the preceding statements. The dependency and citation diagrams for this proof are the same.



A.3 Subworks

In order to study the relationship between a work and its constituent parts, we shall need a notion of a subwork.

Definition 13. A *subwork* I of W consists of a of a subsequence of statements $S_I \subseteq S$ and the functions $d_I : S_I \to 2^S$ and $c_I : S_I \to 2^S$ with $d_I = d|_{S_I}$ and $c_I = c|_{S_I}$ corresponding with the dependency and citation functions for W on the restricted domain S_I, such that if $t \in S_I$ and $s \in d(t)$ or $s \in c(t)$ then $s \in S_I$. Thus the range of d_I and c_I for a subwork is just 2^{S_I}.

Note 17. Definition 13 is analogous to Carnap's (1937) definition of a sublanguage.

Notation 14. Where the usage is unambiguous, I shall drop the subscripts and write $I = (S, d, c)$ for a subwork $I \subseteq W$.

Remark 18. Subworks I of W correspond to order ideals of S_d. By dependency transitivity, the principal order ideals of S_d are simply the sets $s \cup d(s)$. Call these I_s. If W is concise, then the principal order ideals of S_d give the smallest subworks containing their generating statements. If W is not concise, then it is possible that an order ideal of S_d will not give rise to a subwork, as the condition on c_I might exclude subposets which are otherwise closed under the dependency ordering.

Remark 19. The posets $(S_I)_d$ corresponding to subworks I of a concise work W form a distributive lattice $J(S)$ ordered by inclusion on the sets S_I. Linear extensions of S_d correspond to maximal chains in $J(S)$. This allows us to determine $e(S)$ recursively according to the formula

$$e(S_I) = \sum_{S_{I'}} e(S_{I'}) \tag{2}$$

where the $S_{I'}$ are those order ideals covered by S_I in $J(S)$ (Stanley 1997, 106, 110–112).

Definition 15. Define the *dependency rank* (or simply *rank*) $r(s)$ of a statement $s \in S$ recursively as $r(s) = 0$ if $d(s) = \emptyset$ and $d(s) = 1 + \max_{t \in d(s)} r(t)$ otherwise.

Note 20. $r(s)$ is equivalent to Kieffer's (2007) definition of *DAG depth* as the maximal size of a chain of dependencies starting with s (50). A statement's rank is an easily-computable first indication of its complexity.

Lemma 3. *A statement's rank is related to dependency and citation chasing (see remark 8) by*

$$\min\{i > 0 : d^i(s) = \emptyset\} = r(s) + 1 \tag{3}$$

Proof. We use induction on $r(s)$. By definition 15, $d^1(s) = \emptyset$ for $r(s) = 0$. Now suppose that we have estabilshed equation 3 for $r(s) = n$.

If $r(s) = n + 1$ then by definition 15 we have

$$\max_{t \in d(s)} r(t) = n.$$

Thus, for all $t \in d(s)$ the induction hypothesis implies $d^{n+1}(t) = \emptyset$, and hence

$$d^{n+2}(s) = \cup_{t \in d(s)} d^{n+1}(t) = \emptyset.$$

Moreover, if t_0 is one such t of maximal rank, then $d^i(t_0) \neq \emptyset$ for $i \leq n$ by the induction hypothesis. We thus have

$$d^{i+1}(s) \supseteq d^i(t_0) \neq \emptyset$$

for $i \leq n$, and so $n+2$ is the smallest i such that $d^i(s) = \emptyset$. □

Remark 21. In a well written logical work, equation 3 also holds with citations replacing dependencies.

Definition 16. An *atomic statement* s is one satisfying $r(s) = 0$.

Note 22. Such statements can also be called prime statements. See Negri and von Plato, 2001, 1–2.

Remark 23. By the logical ordering criterion of definition 5 we know $d(s_1) = \emptyset$, so $r(s_1) = 0$ and W must have at least one atomic statement. Rank gives a natural rank function on S_d, with $s <_d t$ implying $r(s) < r(t)$ by definition 8. It is not true in general, however, that S_d is graded.

Definition 17. For $s \in S$, the *logical support* $\mathrm{Supp}(s)$ is the set $\{t \in d(s) : r(t) = 0\}$.

Lemma 4. *If* $s \leq_d t$ *then* $\mathrm{Supp}(s) \subseteq \mathrm{Supp}(t)$.

Proof. If $s = t$, then $\mathrm{Supp}(s) = \mathrm{Supp}(t)$. Otherwise, $s \in d(t)$, so if $r \in \mathrm{Supp}(s)$ then $r \in d(s)$ and hence $r \in d^2(t) \subseteq d(t)$, implying $r \in \mathrm{Supp}(t)$. As this holds for all $r \in \mathrm{Supp}(s)$, we have $\mathrm{Supp}(s) \subseteq \mathrm{Supp}(t)$. □

A.4 Complexity Measures

Note 24. Kieffer (2007) gives two complexity measures and applies them to definitions from two topology textbooks (49–52). They are equivalent to the rank $r(s)$ of a statement and the size $1 + |d(s)|$ of the statement's principal order ideal.

Note 25. The number of linear extensions $e(S)$ is given by Stanley (1997) as a complexity measure for posets. We thus have a statement's linear extension complexity, $e(s)$, given by the number of narrative orderings for which the subwork generated by a statement's principal order ideal remains logical with the same dependency ordering.

Definition 18. Let the *girth* of a statement be defined by $g(s) = |d(s)| + 1$ and the *normalised girth* by $\hat{g}(s) = g(s)/|S|$.

Definition 19. Define the *centrality* of a statement by $z(s) = |d^*(s)| + 1$ and the *normalised centrality* by $\hat{z}(s) = z(s)/|S|$.

Definition 20. Define the *arithmetic dependency complexity* of a statement recursively by $ac_d(\emptyset) = 0$ and

$$ac_d(s) = 1 + \sum_t ac_d(t),$$

where the sum is over all t covered by s in S_d (see definition 12).

Definition 21. Define the *arithmetic citation complexity* of a statement recursively by $ac_c(\emptyset) = 0$ and

$$ac_c(s) = 1 + \sum_{t \in c(s)} ac_c(t).$$

Remark 26. Rather than count maximal chains in the poset of order ideals for $e(s)$, arithmetic complexity counts maximal chains in the poset or citation graph itself.

Definition 22. Define the *geometric dependency complexity* of a statement recursively by $gc_d(\emptyset) = 0$ and

$$gc_d(s) = \prod_t (1 + gc_d(t)),$$

where the product is over all t covered by s in S_d.

Definition 23. Define the *geometric citation complexity* of a statement recursively by $gc_c(\emptyset) = 0$ and

$$gc_c(s) = \prod_{t \in c(s)} (1 + gc_c(t)).$$

Remark 27. Geometric complexity is intended to model the scaling features of $e(s)$, but growing more with connections instead of disconnections. Thus, many closely interlinked statements will have higher geometric complexities,

whereas a system of the same size dominated by scarcely related concepts will have a higher number of linear extensions.

Definition 24. Define *arithmetic citation redundancy* by

$$cr_a(s) = ac_c(s)/ac_d(s)$$

and *geometric citation redundancy* by

$$cr_g(s) = gc_c(s)/gc_d(s).$$

This is the sense in which minimally concise works are *minimal*.

Example 28. Two general classes of minimally concise works help to illustrate the different emphases of the measures. A *chain* is a work with

$$s_1 <_d s_2 <_d s_3 <_d \cdots <_d s_i <_d \cdots <_d s_n$$

and a *fan* is a work with $n-1$ atoms and one root statement which depends on all of them. Their complexity data are given in the following table.

s	$r(s)$	$g(s)$	$\hat{g}(s)$	$z(s)$	$\hat{z}(s)$	$e(s)$	$ac_d(s)$	$gc_d(s)$
chain:								
s_i	$i-1$	i	$\frac{i}{n}$	$n-i+1$	$\frac{n-i+1}{n}$	1	i	i
fan:								
atom	0	1	$\frac{1}{n}$	2	$\frac{2}{n}$	1	1	1
root	1	n	1	1	$\frac{1}{n}$	$(n-1)!$	n	2^{n-1}

Example 29. Here are computer-generated complexity data for Euclid's proposition I.15 and its dependencies.

s	r	g	z	ac_d	ac_c	gc_d	gc_c
D.10	0	1	4	1	1	1	1
D.15	0	1	10	1	1	1	1
P.1	0	1	10	1	1	1	1
P.2	0	1	9	1	1	1	1
P.3	0	1	10	1	1	1	1
P.4	0	1	2	1	1	1	1
C.N.1	0	1	10	1	1	1	1

s	r	g	z	ac_d	ac_c	gc_d	gc_c
C.N.2	0	1	3	1	1	1	1
C.N.3	0	1	9	1	1	1	1
C.N.4	0	1	8	1	1	1	1
I.1	1	5	9	5	7	16	64
I.2	2	8	8	8	14	68	4160
I.3	3	9	7	9	18	69	33288
I.4	1	2	7	2	2	2	2
I.5	4	12	6	12	23	210	399468
I.7	5	13	5	13	24	211	399469
I.8	6	14	4	14	25	212	399470
I.11	7	16	3	16	52	426	$1.73 \cdot 10^{12}$
I.13	8	18	2	18	57	854	$2.77 \cdot 10^{13}$
I.15	9	20	1	20	118	1710	$7.26 \cdot 10^{18}$

Example 30. Here are computer-generated complexity data for Russell and Whitehead's proposition 2.17 and its dependencies.

s	r	g	z	ac_d	ac_c	gc_d	gc_c
1.01	0	1	13	1	1	1	1
1.11	0	1	8	1	1	1	1
1.4	0	1	8	1	1	1	1
Perm	1	2	7	2	2	2	2
1.5	0	1	7	1	1	1	1
Assoc	1	2	6	2	2	2	2
16	0	1	8	1	1	1	1
Sum	1	2	7	2	2	2	2
2.03	2	4	2	4	4	6	6
2.04	2	4	5	4	4	6	6
Comm	3	5	4	5	5	7	7
2.05	2	4	4	4	4	6	6
2.06	4	10	3	11	11	112	112
Syll	5	11	2	12	16	113	791
2.1	1	2	6	2	2	2	2
2.11	2	6	5	6	6	18	18
2.12	3	7	4	7	8	19	38

s	r	g	z	ac_d	ac_c	gc_d	gc_c
2.13	4	10	3	10	19	60	8892
2.14	5	11	2	11	24	61	106716
2.17	6	20	1	28	49	49476	$4.14 \cdot 10^9$

A.5 Complexity Bounds

Lemma 5. *The arithmetic complexity of a statement s is at least 1 plus the total number of edges in the dependency diagram or citation graph for the subwork I_s.*

Proof. We use induction on the rank of s. As a base case, the lemma holds for atomic statements ($r(s) = 0$), as they have no edges and arithmetic complexities of 1, by remark 23 and definitions 20 and 21. Now suppose that the results holds for $r(s) < n$.

If $r(s) = n$ then its arithmetic complexity is 1 plus the sum of arithmetic complexities for statements sharing an edge with s of lower rank. For each statement t sharing an edge with s, there are at most 1 plus the number of edges below t which are also below s: namely, those edges below t and the edge from t to s. Thus, the defining sum for the arithmetic complexity of s (definition 20 or 21) gives

$$\begin{aligned} ac(s) = 1 + \sum_t ac(t) &\geq 1 + \sum_t 1 + |\{\text{edges below } t\}| \\ &\geq 1 + |\{\text{edges below } s\}|, \end{aligned}$$

as desired. □

Lemma 6. $ac_c(s_n) \leq 2^{n-1}$.

Proof. We use induction on n. If $n = 1$ then $c(s_1) = \emptyset$ by remark 23, so $ac_c(s_1) = 1 = 2^{1-1}$. Now suppose that we have established $ac_c(s_i) \leq 2^{i-1}$ for $1 \leq i \leq n$.

Applying 5 to well written works, we have $c(s_{n+1}) \subseteq \{s_1, \ldots, s_n\}$, so

$$\begin{aligned} ac_c(s_{n+1}) &= 1 + \sum_{s \in c(s_{n+1})} ac_c(s) \\ &\leq 1 + \sum_{i=1}^{n} ac_c(s_n) \end{aligned}$$

$$\leq 1 + \sum_{i=1}^{n} 2^{i-1}$$
$$= 1 + \sum_{k=0}^{n-1} 2^k = 1 + (2^n - 1) = 2^n = 2^{n+1-1}.$$

□

Remark 31. The bound in lemma 6 is the best possible, as it is possible for a well written (although not minimally concise) work to have every statement cite each statement before it.

Theorem 7. $cr_a(s_n) \leq 2^{n-1}/n$.

Proof. By remark 31, the bound from lemma 6 is achieved by a work forming an n-element chain. By example 28, we have $ac_d(s_n) = n$, so for this maximal work we have $cr_a(s_n) = 2^{n-1}/n$. To show that this is indeed the largest redundancy achievable, we must rule out works with $ac_d(s_n) < n$ for $n > 1$. If $ac_d(s_n) = n - i$, then by the pigeonhole principle and lemma 5 $|d(s_n)|$ would then be at most $n - i - 1$, since there must be at least one edge in the graph for each dependency of s_n. It follows from the proof of 6 that $ac_c(s_n) \leq 2^{n-i-1}$. Note that here $0 \leq i \leq n - 1$, since $ac_d(s_n) \geq 1$.

We shall next need one brief arithmetic fact, that $n/(n-i) \leq 2^i$ for $0 \leq i \leq n - 1$. For $i = 0$, we have $n/(n-i) = 1 = 2^0$. For $1 \leq i \leq n/2$, $n/(n-i) \leq 2 = 2^1 \leq 2^i$. This exhausts all cases for $n \leq 2$. Now, for $n > 3$ and $n/2 < i < n - 1$ we have

$$\frac{n}{n-i} \leq n \leq 2^{n/2} < 2^i.$$

The final case to check is $n = 3$ and $i = 2$, in which case $n/(n-i) = 3 < 4 = 2^i$.

Hence, $cr_a(s_n) \leq 2^{n-i-1}/(n-i) \leq 2^{n-1}/n$, as desired. □

Remark 32. Achieving a comparably sharp bound on $ac_d(s_n)$ to the one on $ac_c(s_n)$ in lemma 6 requires substantially more work. We break the problem into parts by maximising $ac_d(s_n)$ subject to certain conditions, working toward the general case.

Definition 25. The *rank sequence* of a work is the sequence $r_0, r_1, \ldots$ with r_i equal to the number of statements in the work of rank i.

Theorem 8. *For fixed $r(s_n)$ and fixed rank sequence $r_0, r_1, \ldots, r_k$ of I_{s_n}, we have*

$$ac_d(s_n) \leq \sum_{i=0}^{r(s_n)} \prod_{j=i}^{r(s_n)} r_j. \tag{4}$$

Proof. We have $r_k = 1$ since only s_n can be of rank k in I_{s_n}.

Consider the work $\hat{I}_{s_n}$ where $d(t) = \{s : r(s) < r(t)\}$. The cover relations in the dependency order for this work are every pair s and t with $r(t) = r(s) + 1$. We can show by induction on $r(t)$ that in this work we have

$$ac_d(t) = \frac{1}{r_{r(t)}} \sum_{i=0}^{r(t)} \prod_{j=i}^{r(t)} r_j. \tag{5}$$

First, as the dependencies of all statements of the same rank are the same, we have $ac_d(s) = ac_d(t)$ if $r(s) = r(t)$.

Now, for $r(t) = 0$, we have $ac_d(t) = 1$ by definition. The right side of equation 5 becomes $\frac{1}{r_0} r_0 = 1$. Next, fixing t, assume that equation 5 holds for all s with $r(s) < r(t)$.

By definition, using our observation about the cover relations for $\hat{I}_{s_n}$,

$$\begin{aligned}
ac_d(t) &= 1 + \sum_{r(s)=r(t)-1} ac_d(s) \\
&= 1 + \sum_{r(s)=r(t)-1} \frac{1}{r_{r(s)}} \sum_{i=0}^{r(s)} \prod_{j=i}^{r(s)} r_j \\
&= 1 + \frac{r_{r(s)}}{r_{r(s)}} \sum_{i=0}^{r(t)-1} \prod_{j=i}^{r(t)-1} r_j \\
&= \frac{1}{r_{r(t)}} \left(r_{r(t)} + r_{r(t)} \sum_{i=0}^{r(t)-1} \prod_{j=i}^{r(t)-1} r_j \right) \\
&= \frac{1}{r_{r(t)}} \left(r_{r(t)} + \sum_{i=0}^{r(t)-1} \prod_{j=i}^{r(t)} r_j \right) \\
&= \frac{1}{r_{r(t)}} \sum_{i=0}^{r(t)} \prod_{j=i}^{r(t)} r_j,
\end{aligned}$$

as desired.

It immediately follows that for $\hat{I}_{s_n}$ we have $ac_d(s_n)$ at our claimed maximal value from equation 4.

Now, starting with an arbitrary work I_{s_n} with the appropriate rank sequence, we claim that adding statements of lower rank to the dependencies of a statement s (and modifying other statements' dependencies to respect dependency transitivity) does not affect the work's rank sequence and does not decrease $ac_d(t)$ for any t in the work. This is a slightly stronger set of claims than is strictly needed to establish the result, for which we need only that there is some way of adding dependencies (including possibly all at once) to I_{s_n} in order to reach $\hat{I}_{s_n}$ while respecting rank sequence and not decreasing ac_d. The former claim follows immediately from definition 15. Specifically, the rank of s does not change because it still has no dependencies of rank greater than $r(s) - 1$. The rank of statements t with $r(t) \leq r(s)$ does not change, because s is not in $d(t)$, and so $d(t)$ does not need to be modified to respect dependency transitivity. The same is true for any $r(t) > r(s)$ where $s \notin d(t)$. Lastly, if $s \in d(t)$, then the added dependency must also be added to $d(t)$. This does not, however, change the rank of t, as the new statement has a smaller rank than s, which in turn has rank at most $r(t) - 1$.

For the second claim, consider the addition's effect on the cover relations for I_{s_n} when we add u to $d(s)$ for $r(u) < r(s)$. As $u \notin d(s)$ before the addition, dependency transitivity implies that s will cover u after the addition. The addition adds u to $d(s)$, and also $d(t)$ for all $t \geq_d s$. Thus, s is the only statement which could possibly cover something after the addition which it hadn't covered before the addition. The only statements which might no longer be covering a statement after the addition which they had covered before the addition are those t such that $s \in d(t)$ and t covered u before the addition. Now, $ac_d(s)$ increases by $ac_d(u)$ after the addition. This increase gets passed on to all statements $t \geq_d s$. This is because the summand $ac_d(s)$ will appear at least once when expanding the sum in the recursive definition of ac_d until one reaches an atomic statement or s. If t had previously been covering u, then its complexity will decrease by precisely $ac_d(u)$ as a result of losing that cover relation, but will increase by at least $ac_d(u)$ because $t \geq_d s$. No other arithmetic dependency complexities are affected by the addition.

We can thus add to the dependencies of every statement all those statements of lower rank without affecting the rank sequence and without de-

creasing any arithmetic dependency complexities. At the conclusion of this process, we will have $\hat{I}_{s_n}$, which therefore has dependency complexities which are maximal among all works with our given rank sequence. □

Remark 33. Having maximised $ac_d(s_n)$ for a fixed rank sequence, we now continue to fix $r(s_n)$, but allow the rank sequence to change. Note that the sum $r_0 + r_1 + \cdots + r_{r(s_n)}$ is at most n, as there are at most n statements in I_{s_n} by the logical ordering criterion and the rank function partitions the statements. Without loss of generality, we will assume $r_0+r_1+\cdots+r_{r(s_n)} = n$, as our (smaller) bounds for smaller values of n would otherwise apply. In the below, we will also assume $n > 1$ and $r(s_n) \geq 1$. If $n = 1$ or $r(s_n) = 0$ then the only possible value for $ac_d(s_n)$ is 1. Incidentally, if $r(s_n) = 1$ then the maximum value of $ac_d(s_n)$ is n, as in the fan work from example 28.

Notation 26. For every statement s and work I_s with some rank sequence, call $\hat{ac}_d(s)$ the maximum arithmetic dependency complexity of a statement t such that the rank sequence of I_t is the same as that of I_s. $\hat{ac}_d(s)$ is given by the right hand side of equation 4.

Lemma 9. *If I_s has rank sequence $r_0, r_1, \ldots, r_m$ and $I_{s'}$ has the same rank sequence, but with r_i, r_{i+1} replaced by $r_i - 1, r_{i+1} + 1$ for some $1 \leq i \leq m-2$ (since r_m must equal 1), then $\hat{ac}_d(s) \leq \hat{ac}_d(s')$ if and only if $r_i > r_{i+1}$. If $i = 0$, the conclusion holds except for the case $r_i = r_{i+1}$, in which case $\hat{ac}_d(s) = \hat{ac}_d(s')$, and this is the unique case where the complexities are equal.*

Proof. This lemma claims that if we take a statement of rank i in $\hat{I}_s$, move it up one rank, and adjust dependencies to form $\hat{I}_t$, then we will have increased the complexity of the top statement if and only if the move served to even out the sizes of the ranks (with a preference for the higher rank).

The proof requires simply that we compare the expressions for $\hat{ac}_d(s)$ and $\hat{ac}_d(s')$ from equation 4. We need only compare terms involving r_i and r_{i+1}. Let $r'_i = r_i - 1$, $r'_{i+1} = r_{i+1} + 1$, and $r'_j = r_j$ if j is neither i nor $i + 1$. The relationship between each of the following lines is that of 'if and only if'. We have

$$\hat{ac}_d(s) \leq \hat{ac}_d(s')$$
$$\sum_{k=0}^{m} \prod_{j=k}^{m} r_j \leq \sum_{k=0}^{m} \prod_{j=k}^{m} r'_j$$

$$\sum_{k=0}^{i+1}\prod_{j=k}^{m} r_j \leq \sum_{k=0}^{i+1}\prod_{j=k}^{m} r'_j$$

$$\left(\prod_{j=i+2}^{m} r_j\right)\sum_{k=0}^{i+1}\prod_{j=k}^{i+1} r_j \leq \left(\prod_{j=i+2}^{m} r'_j\right)\sum_{k=0}^{i+1}\prod_{j=k}^{i+1} r'_j$$

$$\sum_{k=0}^{i+1}\prod_{j=k}^{i+1} r_j \leq \sum_{k=0}^{i+1}\prod_{j=k}^{i+1} r'_j$$

$$r_{i+1} + \sum_{k=0}^{i}\prod_{j=k}^{i+1} r_j \leq r_{i+1} + 1 + \sum_{k=0}^{i}\prod_{j=k}^{i+1} r'_j$$

$$(r_i r_{i+1})\left(1 + \sum_{k=0}^{i-1}\prod_{j=k}^{i-1} r_j\right) \leq 1 + (r_i - 1)(r_{i+1} + 1)\left(1 + \sum_{k=0}^{i-1}\prod_{j=k}^{i-1} r'_j\right),$$

where in the last step we cancelled summands r_{i+1} and factored out $r_i r_{i+1}$ (respectively $r'_i r'_{i+1}$) from each side. Now, the parenthetical expressions on both sides of the last equation are equal. Call their value R, and note R is an integer whose value is at least 1. It is precisely 1 when $i = 0$, and is strictly greater than 1 when $i > 0$, as $r_0 > 0$. We have

$$\begin{aligned} r_i r_{i+1} R &\leq 1 + (r_i - 1)(r_{i+1} + 1)R \\ r_i r_{i+1} R &\leq 1 + (r_i r_{i+1} + r_i - r_{i+1} - 1)R \\ 0 &\leq 1 - R + r_i R - r_{i+1} R \\ r_{i+1} &\leq r_i - \frac{R-1}{R} \end{aligned}$$

As $0 < (R-1)/R < 1$ for $i > 0$ and $R - 1 = 0$ for $i = 0$, this establishes the desired inequality. Replacing the '$\leq$' signs with '$=$' signs in the above gives equality if and only if $i = 0$ and $r_i = r_{i+1}$. □

Corollary 10. *Under the hypotheses of lemma 9, but instead replacing r_i and r_{i+1} in $\hat{I}_t$ with $r_i + 1$ and $r_{i+1} - 1$ in $\hat{I}_{t'}$, we have $\hat{ac}_d(t) \leq \hat{ac}_d(t')$ if and only if $r_i \leq r_{i+1} - 2$.*

Proof. Apply lemma 9 with $\hat{I}_s = \hat{I}_{t'}$, so $\hat{I}_{s'} = \hat{I}_t$. For $i > 0$, we have $\hat{ac}_d(t') < \hat{ac}_d(t)$ if and only if $r_i + 1 > r_{i+1} - 1$. Negating this result gives $\hat{ac}_d(t') \geq \hat{ac}_d(t)$ if and only if $r_i + 1 \leq r_{i+1} - 1$, as was to be shown. If $i = 0$, the same argument holds for $r_i < r_{i+1} - 2$. If $r_i = r_{i+1} - 2$ then we have $\hat{ac}_d(t) = \hat{ac}_d(t')$, and

this is the unique case of equality. □

Remark 34. Lemma 9 and corollary 10 allow us to transform an arbitrary rank sequence (excluding the top rank) into a monotonically increasing one (by the lemma) where the steps are at most 1 between consecutive ranks (by the corollary), while not decreasing the arithmetic complexity of the top statement. We shall need one more specially tailored lemma in the spirit of these results in order to establish a sharp bound on $ac_d(s_n)$ for fixed $r(s_n)$.

Lemma 11. *For $0 < i < i' - 1$ and I_s with rank sequence satisfying $r_j \geq 2$ for $i < j < i'$, let $I_{s'}$ have the same rank sequence but with r_i and $r_{i'}$ replaced with, respectively, $r_i + 1$ and $r_{i'} - 1$. Then $\hat{ac}_d(s) \leq \hat{ac}_d(s')$ if and only if $r_i < r_{i'} - 1$. If $i = 0$, the second inequality becomes $r_i < r_{i'} - 2$. In either case, $\hat{ac}_d(s) \neq \hat{ac}_d(s')$.*

Proof. We proceed as in the proof of lemma 9. Compare the expressions for $\hat{ac}_d(s)$ and $\hat{ac}_d(s')$ from equation 4. As before, we need only compare terms involving r_i or $r_{i'}$. Let $r'_i = r_i + 1$, $r'_{i'} = r_{i'} - 1$, and $r'_j = r_j$ if j is neither i nor i'. The relationship between each of the following lines is that of 'if and only if'. We have

$$\begin{gathered}
\hat{ac}_d(s) \leq \hat{ac}_d(s') \\
\sum_{k=0}^{m} \prod_{j=k}^{m} r_j \leq \sum_{k=0}^{m} \prod_{j=k}^{m} r'_j \\
\sum_{k=0}^{i'} \prod_{j=k}^{i'} r_j \leq \sum_{k=0}^{i'} \prod_{j=k}^{i'} r'_j \\
\sum_{k=0}^{i} \prod_{j=k}^{i'} r_j + \sum_{k=i+1}^{i'} \prod_{j=k}^{i'} r_j \leq \sum_{k=0}^{i} \prod_{j=k}^{i'} r'_j + \sum_{k=i+1}^{i'} \prod_{j=k}^{i'} r'_j.
\end{gathered}$$

Now let

$$R_1 = \prod_{j=i+1}^{i'-1} r_j \left(1 + \sum_{k=0}^{i-1} \prod_{j=k}^{i-1} r_j\right) = \prod_{j=i+1}^{i'-1} r'_j \left(1 + \sum_{k=0}^{i-1} \prod_{j=k}^{i-1} r'_j\right)$$

and

$$R_2 = 1 + \sum_{k=i+1}^{i'-1} \prod_{j=k}^{i'-1} r_j = 1 + \sum_{k=i+1}^{i'-1} \prod_{j=k}^{i'-1} r'_j.$$

Continuing the chain of 'if and only if's from above, we have

$$\begin{aligned} r_i r_{i'} R_1 + r_{i'} R_2 &\leq r'_i r'_{i'} R_1 + r'_{i'} R_2 \\ r_i r_{i'} R_1 + r_{i'} R_2 &\leq (r_i + 1)(r_{i'} - 1) R_1 + (r_{i'} - 1) R_2 \\ r_i &\leq r_{i'} - 1 - \frac{R_2}{R_1}. \end{aligned} \tag{6}$$

Now,

$$\frac{R_2}{R_1} = \frac{1 + \sum_{k=i+1}^{i'-1} \prod_{j=k}^{i'-1} r_j}{\prod_{j=i+1}^{i'-1} r_j \left(1 + \sum_{k=0}^{i-1} \prod_{j=k}^{i-1} r_j\right)} = \frac{1 + \sum_{k=i+1}^{i'-1} \prod_{j=i+1}^{i'-1} r_j^{-1}}{1 + \sum_{k=0}^{i-1} \prod_{j=k}^{i-1} r_j},$$

and, since each of the r_j in the numerator is at least 2, we can compare the numerator to the geometric series $\sum_{k \geq 0} 2^{-k}$ to find that its value is strictly between 1 and 2. As the denominator is integer-valued, along with every other quantity in (6), changing '$\leq$' signs to '$=$' signs in the above gives that $\hat{ac}_d(s) \neq \hat{ac}_d(s')$. Now, when $i = 0$ the denominator is exactly 1. Otherwise, it is at least 2. Thus, the last inequality of (6) holds if and only if $r_i < r_{i'} - 1$ for $i > 0$ and $r_i < r_{i'} - 2$ otherwise. □

We shall only need a corollary to lemma 11 for $i = 0$.

Corollary 12. *Under the hypotheses of lemma 11 with $i = 0$ and $r_0 > 1$, but instead replacing $r_0, r_{i'}$ in $\hat{I}_t$ with $r_0 - 1, r_{i'} + 1$ in $\hat{I}_{t'}$, we have $\hat{ac}_d(t) \leq \hat{ac}_d(t')$ if and only if $r_0 \geq r_{i'}$.*

Proof. Apply lemma 9 with $\hat{I}_s = \hat{I}_{t'}$, so $\hat{I}_{s'} = \hat{I}_t$. For $i > 0$, we have $\hat{ac}_d(t') < \hat{ac}_d(t)$ if and only if $r_0 - 1 < r_{i'} + 1 - 2$, i.e. $r_0 < r_{i'}$. Negating this result gives $\hat{ac}_d(t') \geq \hat{ac}_d(t)$ if and only if $r_i \geq r_{i'}$, as was to be shown. □

Definition 27. Call a rank sequence *level* if it satisfies $r_i \leq r_j \leq r_i + 1$ for $i \leq j$. Call it *almost level* if $r_1, r_2, \ldots$ is a level subsequence and one of the following holds:

1. $r_0 = r_1 - 1$.
2. $r_0 = r_1 = r_2 - 1$. (Or, if there is no r_2, simply $r_1 = r_2$.)
3. There is an initial sequence of at least two ranks equal to 1. (Note that then the only other possible value for r_j is 2.)

Additionally, a single rank is vacuously almost level.

Lemma 13. *A number $n > 0$ has a unique almost level partition into m parts of monotonically increasing size for each $1 \leq m \leq n$.*

Proof. We can specify the unique level partition of n into m parts. Simply start each part with size $\lfloor \frac{n}{m} \rfloor$, and increment the sizes of the parts by 1, starting with the last part, until their total is n. If we write n uniquely as $qm + r$, there will be $m - r$ parts of size q and r parts of size $q + 1$.

If $m \leq 2$ the unique level partition is also the unique almost level partition. This is trivially so for $m = 1$. For $m = 2$ and n even, either the two parts are equal, satisfying condition 2, or they differ by at least two. If n is odd, the two parts satisfy condition 1 and must be of different size.

Now consider $m > 2$. Let us list the almost level sequences of length m and their sums using some suggestively chosen parameters. If the sequence is almost level by the third criterion, then it consists of an initial sequence of at least two ones, followed by r twos for some $0 \leq r \leq m - 2$ in order for all but the first part to be level. Possible sums are of the form $m + r$ for $0 \leq r \leq m - 2$. If the sequence follows the second criterion (excluding the sequence $1, 1, 2, 2, \ldots, 2$ which satisfies both the second and third criteria) then it must take the form $q, q, q + 1, q + 1, \ldots, q + 1$ for some $q \geq 2$. The possible sums are $qm + (m - 2)$. If the sequence follows the first criterion, then it starts with parts of size $q - 1$ and q for some $q \geq 2$, and continues with parts of size q until it reaches a terminating sequence of r parts of size $q + 1$, with $0 \leq r \leq m - 2$. The possible sums are $qm + r - 1$. We might rewrite this as $qm + r$ with $-1 \leq r \leq m - 3$.

Now with $n = qm + r$ as before, our suggestively named parameters help us to see that the unique $q \geq 1$ and $0 \leq r \leq m - 1$ give n a sum of a unique one of the three forms above, with the case $r = m - 1$ corresponding in the above to $r = -1$ and q one larger than it is in the equation $n = qm + r$. □

Theorem 14. *For fixed $r(s_n) = m$, $ac_d(s_n)$ is maximised by an almost level rank sequence for $r_0, \ldots, r_{m-1}$ partitioning $n - 1$.*

Proof. Starting at r_{m-1}, we create level rank subsequences by comparing adjacent ranks and incrementing or decrementing as necessary. The method of comparison is similar to the familiar 'bubble sort' algorithm.

By lemmas 9 and 11, corollary 10 and 12, if we start with an arbitrary rank sequence and use only exchanges of the form $r_i, r_{i'} \to r_i - 1, r_{i'} + 1$ and $r_i, r_{i'} \to r_i + 1, r_{i'} - 1$ for appropriate values of $i, i', r_i, r_{i'}$, then we will have a progression of rank sequences giving rise to non-decreasing (and in most cases increasing) maximum arithmetic dependency complexity values.

We use induction to generate level subsequences of the form $r_i, \ldots, r_{m-1}$ with $i > 0$, and then extend to $i = 0$ in a special final step.

Base Case. The rank subsequence consisting of just r_{m-1} is vacuously level.

Induction Hypothesis. We assume that the rank subsequence $r_{i+1}, \ldots, r_{m-1}$ can be made level for a fixed $i > 0$ without decreasing maximum arithmetic dependency complexity.

Inductive Step. The inductive step involves taking the rank subsequence $r_{i+1}, \ldots, r_{m-1}$, assumed level by the induction hypothesis, appending r_i to the beginning, and then re-levelling the new sequence. If $r_i = r_{i+1}$, then the new sequence is already level. If $r_i > r_{i+1}$ then apply lemma 9 to increment r_{i+1} and decrement r_i, and then apply the induction hypothesis to re-level $r_{i+1}, \ldots, r_{m-1}$. The value of r_{i+1} after re-levelling will either be returned to where it started or left greater by 1. Repeat this process until $r_i \leq r_{i+1}$. Again, we are done if $r_i = r_{i+1}$. Otherwise, if $r_i < r_{i+1} - 1$ apply corollary 10 to increment r_i and decrement r_{i+1}, and re-level by the induction hypothesis. As before, r_{i+1} will either be returned to its original value or left decremented by 1. Repeat this process until either $r_i = r_{i+1}$ (in which case we're done) or $r_i = r_{i+1} - 1$. In the latter case, we're done if $r_{i+1} = r_{i+2} = \cdots = r_{m-1}$. Otherwise, there is some smallest $i + 2 \leq k \leq m - 1$ such that $r_i = r_k - 2$. Moreover, since $r_i \geq 1$ we have $r_j \geq 2$ for $j > i$ in this situation. Applying lemma 11 allows us to increment r_i and decrement r_k, leaving the resulting subsequence $r_i, \ldots, r_{m-1}$ level.

Final Step. Appending r_0 to the beginning of the level sequence $r_1, \ldots, r_{m-1}$, we can apply the procedure of the inductive step until either $r_0 = r_1$ or $r_0 = r_1 - 1$. In the latter case, our sequence is almost level and we are done. In the former case, if $r_0 = r_1 = 1$ then we also have an almost level sequence. Otherwise, if $r_1 = \cdots = r_{m-1} > 1$ then apply corollary 12 to decrement r_0 and increment r_{m-1} to create an almost level sequence. Otherwise, find the least $2 \leq k \leq m - 1$ such that $r_0 = r_k - 1$. If $k = 2$, the sequence is almost

level and we are done. If $k > 2$ then apply corollary 12 to decrement r_0 and increment r_{k-1} to create an almost level sequence. □

Example 35. We can use a computer to generate the maximum arithmetic complexity for different combinations of n and $r(s_n)$. Here are the results for $2 \leq n \leq 12$.

$n \backslash r(s_n)$	1	2	3	4	5	6	7	8	9	10	11
2	2										
3	3	3									
4	4	5	4								
5	5	7	7	5							
6	6	10	11	9	6						
7	7	13	16	15	11	7					
8	8	17	22	23	19	13	8				
9	9	21	31	34	31	23	15	9			
10	10	26	41	49	47	39	27	17	10		
11	11	31	53	67	70	63	47	31	19	11	
12	12	37	69	94	103	95	79	55	35	21	12

Theorem 15. *For $n \geq 2$, $ac_d(s_n)$ is maximised when $r(s_n) = \lceil \frac{n-1}{3} \rceil + 1$, and the maximal values for a fixed rank increase or decrease with rank up to or, respectively, after this rank.*

Proof. The theorem holds by inspection for all cases in example 35, so we may assume rank sequences of sufficient length for the below to make sense. We compare posets in the optimal almost level form from theorem 14 to posets of shorter or longer rank sequences involving the same number of statements. Arithmetic complexity is defined recursively from values for statements of smaller rank, so we shall only have to look at the beginnings of the rank sequences we are comparing if they are eventually identical.

Case 1. If $\frac{n-1}{r_{s_n}} < 2$ then in all but one case its optimal rank sequence is of the form $1, 1, r_2, r_3, \ldots$, with the r_i a level sequence of 1s and 2s. We compare this to the sequence $2, r_2, r_3, \ldots$. In both cases, the complexity of each element of the rank corresponding to r_2 is 3, and so these two rank sequences give rise to identical complexities for s_n. As the latter rank sequence is not the optimal one for a rank sequence of its length, we have that decreasing

the rank of s_n by 1 gives a higher maximal complexity. We shall deal with the rank sequence $1, 2, 2, \ldots$ with case 3, below.

Case 2. If $\frac{n-1}{r_{s_n}} \geq 3$ then the optimal rank sequence will begin with either q or $q-1$ for $q = \lfloor \frac{n-1}{r_{s_n}} \rfloor \geq 3$. Compare this rank sequence to the sequence beginning with $q-1, 1$, respectively $q-2, 1$ and continuing as with the former sequence. In both cases, the complexity of each statement of the next rank in the sequence will be $q+1$, respectively q. As the latter rank sequence is not the optimal one, we have that increasing the rank of s_n by 1 gives a higher maximal complexity. This argument also applies to the rank sequences $2, 3, 3, 3, \ldots$ and $2, 2, 3, 3, 3, \ldots$, which occur for $\frac{n-1}{r_{s_n}}$ just less than 3.

Case 3. Finally, we consider rank sequences beginning $1, 2, \ldots$. This covers the stray instance from case 1 and the case where $2 \leq \frac{n-1}{r_{s_n}} < 3$ (excluding the two extra instances considered in case 2). Now, when $r(s_n)$ is the claimed optimal rank from the theorem, the rank sequence begins $1, 2, 3, \ldots$, $1, 2, 2, 3, \ldots$, or $1, 2, 2, 2, 3, \ldots$, followed in each case by a number of ranks of size three. In all other cases (for n sufficiently large), the rank sequence begins $1, 2, 2, 2, 2$ and $r(s_n)$ is greater than our claimed optimal rank. Using equation 5 we compare our given rank sequence to one beginning instead with $1, 2, 3, 3$. In the former case, statements of rank 5 have complexity 47, where in the latter the corresponding statements (now with rank 4) have complexity 49. Thus, the latter gives a larger $ac_d(s_n)$, and we can conclude that we can increase the complexity of s_n by decreasing its rank by 1. □

Remark 36. For $n > 5$, the increase or decrease in maximal complexity by rank from theorem 15 is strict.

Corollary 16. *Letting* $m = \lceil \frac{n-1}{3} \rceil + 1$*, we have*

$$ac_d(s_n) \leq \frac{1}{2}\left(3^{m-4}(2c+1) - 1\right)$$

for $n \geq 8$ *and*

$$c = \begin{cases} 47 & \text{if } n-1 \pmod 3 \equiv 1 \\ 69 & \text{if } n-1 \pmod 3 \equiv 2 \\ 99 & \text{if } n-1 \pmod 3 \equiv 0 \end{cases}.$$

Proof. This follows from equation 4 applied to the rank sequences from the proof of theorem 15. The table in example 35 gives bounds on $ac_d(s_n)$ for $n < 8$. □

Remark 37. The bound in Corollary 16 is the best possible, as it corresponds to the complexity for the terminal statements of almost-level works of the optimal rank from Theorem 15.

Remark 38. Comparable bounds to those of lemma 6 and corollary 16 can be obtained for geometric complexities as well. In fact, the logarithm of the geometric complexity grows in a very similar way to the arithmetic complexity for a statement. The difference between the recursions for log-geometric complexity and arithmetic complexity is most pronounced when the complexities involved are small, and the effects of the small complexities on the recursion can be significant. I leave the precise bounding of geometric complexity to future work.

A.6 Further Directions

One shortcoming of this approach is its difficulty in handling works which contain multiple distinct proofs of the same theorem. Such works can still be analysed semantically but it is not obvious how to represent their logical relationship.[53] We might also want to represent some citation or dependency cover relations as 'more difficult' than others. It is possible that many concepts above can be adapted to 'weighted' citation or dependency, where each dependency and citation relation is also assigned a weight. Then multiple proofs can be cited with weights adding to 1. This also allows statements which make multiple citations of another statement to be joined to that statement with weights greater than 1. The above analysis then becomes equivalent to the weighted analysis with all the weights set to 1. Some possible formulations for well written works are given below. If a work is not well written, replace the criterion $t \leq_d s$ with $t \in c(s)$ for the citation measures.

Notation 28. Call the edge between two statements s and t in the citation graph or Hasse diagram for a work e_{st}, and denote by $w(st)$ the *weight* of that edge. If there is no edge between s and t then $w(st) = 0$.

We can now give weighted versions of the complexity measures from above.

[53] See Kieffer, 2007, 63 for one approach. See also Dawson, 2006, and Avigad, 2006, for discussions of proof differentiation and comparison.

Definition 29. Define the *weighted arithmetic complexity* of a statement (in either the dependency poset or citation graph) recursively by $wac(\emptyset) = 0$ and

$$wac(s) = 1 + \sum_{t <_d s} w(st) wac(t).$$

Definition 30. Define the *weighted geometric complexity* of a statement (in either the dependency poset or citation graph) recursively by $wgc(\emptyset) = 0$ and

$$wgc(s) = \prod_{t <_d s} (1 + wgc(t))^{w(st)}.$$

B Source Code

B.1 Usage

The `java` programs below are designed to be run from the command line. To run the program using just the default work and parameters, type `java LDP` into the command line.[54] The program will then print to the console LaTeX source code for a table of the default work's complexity data and for its dependency and citation diagrams. The table data is meant to be put in a `tabular` LaTeX environment, and the diagrams can be processed using the `pgf` package in the `tikzpicture` environment.

The program can take up to three inputs in the command line. It parses the first input as the name of a text file containing citation data for a work to be analysed. The second input is parsed as an integer upper bound on the number of lines in the file from the first input. The third input is a boolean parameter (`true` or `false`) indicating whether the program should compute the computationally-intensive linear extension data. The default for this last parameter is `false`. A full input would look like

```
user:dir$ java LDP data.txt 200 true
```

This input processes the data in the first 200 lines of the file `data.txt` and includes linear extensions when computing complexity data.

Examples of data input files are in sections B.11 and B.12. The program accepts two types of newline-separated inputs in these data files. The first is a type declaration of the form `Type,Statement,st` which sets standard types

[54] This requires a properly installed Java Runtime Environment (JRE), such as can

to be used in the work. Statements can then be assigned this type using either its full name (`Statement`) or its shorthand (`st`). The second type of input is a statement declaration of the form `st,Name,Label,cite1,cite2`. The first entry of the input is either the name or shorthand of the statement's type. If that type has not been previously established, then the entry is processed to include a type declaration using the entry for both the name and shorthand. The second entry is the name by which the program and the `pgf` LaTeX package refer to the statement, and the third entry gives the label used in the program's outputs. Finally, an arbitrary number of comma-separated citations can be given by listing the names of previous statements.

B.2 Organisation

The main class is `LDP`, for *logical dependency poset*. It holds one logical work, and keeps track of the types of statements in that work. Works are defined in the `Work` class, which holds a searchable ordered queue of statements and contains method for generating the program's outputs. Statements are made by the `Statement` class. Each statement has a type (from the `Type` class), a name, a label for printed outputs, queues and a collection of complexity data. Complexity data for the statement are generated as the statement is added to its work based on the queue of citations given at the start. `WrappedInt` is a simple wrapper class used in generating complexity data. `Q2` implements a double-linked queue using the `Binode` class for nodes. Finally, the `MaxACDByRank` class contains separate tools for calculating the arithmetic dependency complexities of extremal works.

B.3 LDP.java

```
/*
 * LDP is the main class for the analysis of
 * logical work complexity. It contains basic
 * file reading and command parsing functions
 * for taking inputs from a file specified
 * either upon running the program or entered
 * into the program itself as a default option.
 * The main function creates a new work from
 * the input file and prints the complexity
 * data and LaTeX source for citation and
```

be found at http://java.sun.com/javase/downloads/index.jsp. For basic usage we assume that any data and `java` class files are located in the console's current directory.

```
 * dependency diagrams.
 */

// input/output import declarations
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

public class LDP
{
  private Q2 types = new Q2(); // types of statements
     defined
  private Work work = new Work(); // the work in
     question
  private static final String DEFAULT_FILE = "Euclid.
     txt";
  private static final int DEFAULT_NUM_LINES = 1000;
  private static final boolean DEFAULT_LE = false;

  public static void main(String[] args)
  {
    // set the file and an upper bound on its number of
        lines to the defaults
    String file = LDP.DEFAULT_FILE;
    int numlines = LDP.DEFAULT_NUM_LINES;
    boolean LEs = DEFAULT_LE; // default is not to
       compute linear extensions
    // if a file name is given in the command line, use
        that instead of the default
    if(args.length > 0) file = args[0];
    // check if a number of lines is also specified
    if(args.length > 1)
    {
      try {numlines = Integer.parseInt(args[1]);}
      catch(Exception e)
      {
        System.out.println("Exception when reading
           number of lines from command line: " + e);
      }
    }
```

```
    // a third argument in the command line will
       indicate whether to compute linear extensions
    if(args.length > 2)
    {
      try {LEs = Boolean.parseBoolean(args[2]);}
      catch(Exception e)
      {
        System.out.println("Exception when reading LEs
           from command line: " + e);
      }
    }
    // create a new work by reading and parsing the
       input file
    String[] cmds = LDP.readfile(file,numlines);
    LDP ldp = new LDP();
    ldp.parseCommands(cmds);
    // print the required outputs, which are generated
       from within the work itself
    System.out.println(ldp.getWork().complexityData(LEs
       ));
    System.out.println(ldp.getWork().laTeXDeps());
    System.out.println(ldp.getWork().laTeXCites());
  }

/*
*  Want to take commands of the form
*
* Type,Lemma,L
* Type,Definition ---> decides that the shorthand is
   the first letter (this is handled in the Type
   constructor)
* Type,Statement,S
* Definition,name,label
* D,name,label
* S,namo,label,name ---> everything coming after the
   name is something it cites
* S,S2,label,name
* L,L1,label,S2,namo
*
*/
```

```
// This is a simple accessor method which returns the
    work under consideration
public Work getWork()
{
  return this.work;
}

// This function parses a sequence of input commands
   given as an array of Strings
public void parseCommands(String [] commands)
{
  for(int i=0 ; i < commands.length ; i++)
  {
    String [] command = commands[i].split(",");  //
       split command i by commas
    if(command.length > 0) // ignore empty commands
    {
      if(command[0].equals("Type"))
      { // the first kind of command specifies a
         statement type
        if(command.length > 2) this.types.joinQ(new
           Type(command[1],command[2]));    // ignore
           nameless types and commands beyond the
           first three
        else if(command.length > 1) this.types.joinQ(
           new Type(command[1])); // the types
           constructor automatically gives a
           shorthand
      }
      else
      { // if the command doesn't specify a statement
         type, assume it is specifying a statement
        Type currentType = null;
        boolean found = false; // see if the command
           gives a statement of a known type
        for(Binode current = this.types.getFront(); !
           found && current != null && current.
           getData() != null; current = current.
           getNext())
        {
          currentType = (Type) current.getData();
```

```
            if(command[0].equals(currentType.getName())
                || command[0].equals(currentType.
                getShorthand())) found = true;
          }
          if (!found) // adds a new type if it's not
              one already on the list; otherwise
              currentType will be the correct type from
              the list
          {
            currentType = new Type(command[0]);
            this.types.joinQ(currentType);
          }
          // having given the statement a type, we
              collect any citations from the command
          Q2 cites = new Q2();
          for(int j=3; j < command.length; j++)
              cites.joinQ(this.work.findStatement(
                  command[j]));
          if(command.length > 2)
          { // finally, add the statement to the work
            this.work.addStatement(command[1],command
                [2], currentType, cites);
          }

        }
      }
    }
  }

//   here is a simple filereader function I wrote for
    an intro computer programming course in 2007
  public static String [] readfile(String nameoffile,
      int numlines)
  {
    // returns an array of Strings corresponding to the
        lines of the file;
    // numlines is the maximum number of lines to read
    // first, declare the name of our FileReader and
        BufferedFile Reader
    // and initialize to something silly.
    FileReader file = null;
```

```
BufferedReader bfile = null; // Same here; we
    declare these out of the try catch statement so
    that we can make the try-catch short and use
    them outside of the try-catch.

// try to find the file
boolean filefound = false;
try{
   file = new FileReader(nameoffile);
   filefound = true;  // let us know you found the
      file
}
catch(FileNotFoundException fnfe){
   System.out.println("I couldn't find your file.\n"
       + fnfe +"\nInstead I will use a default
      string.");
   // filefound will remain false if the file wasn't
       found
}
//  we do one of two things, depending on whether
    the file was found
if(filefound)
{
   bfile = new BufferedReader(file); //make a
      buffered reader
   // make a big array of strings, which will be
      shrunken for the output
   String [] strarray = new String [numlines];
   String thisline;
   //we have to do a try-catch here because java isn
      't totally sure our
   //buffered reader was set up correctly
   try {
     thisline = bfile.readLine();
     //will return the first line of the file, or
        null if file is empty
   } catch (IOException e) {
     thisline = null; //if there's an exception, we
        might as well pretend
     // that we couldn't read anything
```

```
    System.out.println("There was a problem with
        the buffered file reader: "+e);
}

int linecount = 0; //this will tell us what line
    we're on
while(thisline != null && linecount < strarray.
    length)
{
    // while we're still getting interesting data
        from the file reader and we're not
        overflowing
    strarray[linecount++] = thisline; //copy our
        line into the array and increment the number
         of lines found
    try{
      thisline = bfile.readLine(); // get the next
          line, see above for try-catch explanation
    } catch (IOException e) {
      thisline = null;
      System.out.println("There was a problem with
          the buffered file reader");
    }
}
// we're done reading the file, so we close it,
    that is, if we were able to open it in the
    first place
try {
    bfile.close(); // try closing it
} catch (IOException e) { // or else don't
    System.out.println("Problem closing buffered
        reader");
}
try {
    file.close(); // try closing it
} catch (IOException e) { // or else don't
    System.out.println("Problem closing file reader
        ");
}
// next, make an array which only contains the
    interesting lines
```

```
      String [] nicearray = new String [linecount];
      // give the addresses of the strings from the big
          array to the nice array
      // if those strings are the interesting ones
      for(int i = 0; i < nicearray.length; i++)
        nicearray[i] = strarray[i];
      return nicearray; // return the nice array
    }
    // if the file wasn't found, do something silly but
        inoffensive
    String [] sillyarray = new String [0];  //an array
       of length zero! how silly!
    // I guess we haven't read anything interesting!
    return sillyarray;
  }
}
```

B.4 MaxACDByRank.java

```
/*
 * This class generates a table of maximal arithmetic
 * complexity values for s_n for various values of
 * n and r(s_n).
 * The maximum value of n can be given in the main
 * function or in the command line.
 */

public class MaxACDByRank
{
  public static void main(String[] args)
  {
    int max = 20;
    if(args.length > 0)
    {
      try {max = Integer.parseInt(args[0]);}
      catch(Exception e)
      {
        System.out.println("Exception when reading n
            from command line: " + e);
      }
    }
```

```
   printMaxACDByRank(max);
}

// prints an output suitable for a LaTeX table
// giving maximum arithmetic complexities with
// rows indexed by n and columns indexed by r(s_n)
public static void printMaxACDByRank(int MAX)
{
   System.out.println("n is the index of s\nm is the
       rank of s");
   System.out.print("n\\m &\t");
   for(int m=1; m<= MAX; m++)
   {
      System.out.print(""+m+" &\t");
   }
   System.out.println("\\\\");
   for(int n=1; n <= MAX; n++)
   {
      System.out.print(""+n+" &\t");
      for(int m=1; m < n; m++)
      {
         int [] parts = MaxACDByRank.part(n-1,m);
         System.out.print("" + MaxACDByRank.acd(parts) +
             " &\t");
      }


      System.out.println("\\\\");
   }
}

// this function, used in debugging, prints the ideal
// rank sequence for values of n up to a specified
   maximum
// and a rank up to a specified maximum
public static void printparts(int NMAX, int MMAX)
{
   for(int n=1; n <= NMAX ; n++)
      for(int m=1; m <= MMAX; m++)
      {
         System.out.print("" + n + "," + m + ":");
```

```
        int [] parts = part(n,m);
        for(int i=0; i< parts.length; i++)
          System.out.print("" + parts[i] + ",");
        System.out.println(";");
      }
  }

  // given a rank sequence, this function computes the
  // maximum arithmetic complexity using the formula
  // from the essay
  private static int acd(int [] parts)
  {
    int total = 1;
    for(int i=0; i < parts.length; i++) total = total *
        parts[i] + 1;
    return total;
  }

  // puts n into m parts which are almost level
  // according to the criteria from the essay
  private static int [] part(int n,int m)
  {
    if(m > 0 && n >= m)
    {
      int [] parts = new int[m];
      for(int i = 0; i<m; i++) parts[i] = (int) n/m;
      for(int i = 0; i < n % m; i++) parts[m-i-1]++; //
          after this point the sequence will be level
      // if n mod m = m-1 or m-2, then level is also
         almost level
      if(parts[0] > 1 && n % m < m - 2 )
      {
        parts[0]--;
        parts[m - (n % m) - 1]++;
      }
      return parts;
    }
    // else
    return new int[0];
  }
}
```

B.5 Binode.java

```
// this is a basic 2-pointer node for use in our double
   -linked queue Q2.
// It consists of an object, called data,
// and the adresses of two other nodes,
// named prev(ious) and next (or pointing to null nodes
   ).
// Its pointer structure is like this:
//  prev <------ this ------> next
//                 |--> data

public class Binode {

  private Object data;
  private Binode prev,next;

  //Constructors
  public Binode(Object data,Binode prev,Binode next)
  {
    this.data=data;
    this.prev=prev;
    this.next=next;
  }
  public Binode(Object data)
  {
    this(data,null,null);
  }
  public Binode()
  {
    this(null,null,null);
  }

  //Accessors
  public Binode getPrev()
  {
    return prev;
  }
  public Binode getNext()
  {
    return next;
```

```
  }
  public Object getData()
  {
    return data;
  }
  public boolean noData()
  {
    return data == null;
  }
  //Mutators
  public void setData(Object data)
  {
    this.data = data;
  }
  public void setPrev(Binode prev)
  {
    this.prev = prev;
  }
  public void setNext(Binode next)
  {
    this.next = next;
  }
}
```

B.6 Q2.java

```
// A Q2 is a double-linked queue, with pointers like
   this:
//         |-> d           |-> d           |-> d
//    <----*0----> <----*1----> <-------*2------>
//         ^-- front                       ^-- back
// the Q2 is served from the front and entries are
   added to the back
// the name comes from Q= queue, 2= doubly-linked
// this is a somewhat bulky implementation

public class Q2 {

  private Binode front, back;
  private int length;
```

```
// constructor
public Q2()
{
  this.front = null;
  this.back = null;
  this.length = 0;
}

// accessors
public boolean isEmpty()
{
  return length == 0;
}
public boolean isFull()
{
  return false; // never full, because pointer-based!
}
public int getLength()
{
  return this.length;
}
public Binode getFront()
{
  return this.front;
}
public Binode getBack()
{
  return this.back;
}
// searchable Q, so can get object at position i;
   indexing starts at 0
public Object getPos(int i)
{
  if(!isEmpty())
  {
    Binode pos = this.getNode(i);
    if(pos != null)
      return pos.getData();
    return null;
  }
  return null;
```

```
}
// searchable Q, so can get node at position i;
   indexing starts at 0
public Binode getNode(int i)
{
  if(!isEmpty())
  {
    Binode pos = this.front; // start pointing to the
        front, which is position 0
    if(i>=0 && i < this.length)
    {
      // if i=0, we're in the right place;
      // if i=1, we do one getNext and are in the
          right place, etc.
      // will return null if out of bounds
      for(int j = 0; j<i ;j++)
        pos = pos.getNext();
      return pos;
    }
  }
  return null;
}

//mutators
public void joinQ(Object thing)
{
  Binode temp = new Binode(thing); // create a node
      called temp for the new thing
  if(isEmpty())
  {
    this.front = temp;
    this.back = temp;
  }
  else // not empty
  {
    temp.setPrev(this.back); // insert temp after the
        back
    this.back.setNext(temp); // so temp is back's new
        next
    this.back = temp; // and temp is the new back
  }
```

```
  this.length++; // increment length of Q
}

public void cutQ(Object thing)
{ // joinQ, but joining to the front of the Q instead
    of the back;
  // this lets the Q function like a stack if we need
      it to
  Binode temp = new Binode(thing); // create a node
     called temp for the new thing
  if(isEmpty())
  {
    this.front = temp;
    this.back = temp;
  }
  else // not empty
  {
    temp.setNext(this.front); // insert temp before
       the front
    this.front.setPrev(temp); // so temp is front's
       new prev
    this.front = temp; // and temp is the new front
  }
  this.length++; // increment length of Q
}
// given a Binode from the Q, this connects up its
   prev and next and decrements the Q length
public void excise(Binode b)
{// This will do funny things if b isn't in the Q,
  // and I haven't forced this to do a check the b
  // is in fact in the Q, so use with caution.
  // if b is in the front, only need to deal with its
      next;
  //if b is in the back, only need to deal with its
     front.
  if(b.getNext() != null) b.getNext().setPrev(b.
     getPrev());
  else this.back = b.getPrev(); // if b is in the
     back
  if(b.getPrev() != null) b.getPrev().setNext(b.
     getNext());
```

```
    else this.front = b.getNext(); // if b is in the
        front
    this.length--;
  }
  // given a Binode which has been excised, this
     reinserts it and increments the Q length
  public void reinsert(Binode b)
  {// This will do funny things if b wasn't in the Q,
    // and I haven't forced this to do a check the b
    // was in fact in the Q, so use with caution.
    // if b was in the front, only need to deal with
       its next;
    // if b was in the back, only need to deal with its
        front.
    if(b.getNext() != null) b.getNext().setPrev(b);
    else this.back = b; // if b was in the back
    if(b.getPrev() != null) b.getPrev().setNext(b);
    else this.front = b; // if b was in the front
    this.length++;
  }
  // returns object at front of Q, removing it from the
     Q, if the Q is non-empty; otherwise, returns null
  public Object leaveQ()
  {
    if(!isEmpty())
    {
      Object temp = this.front.getData();
      this.front = this.front.getNext(); // front may
         not get garbage collected yet if length = 1,
         but it will be when someone else joins the Q
      if(this.front != null) this.front.setPrev(null);
         // the new front, if it is a node, is still
         pointing to the old front.  maybe it's time to
          let go.
      this.length--;
      return temp;
    }
    //else
    return null;
  }
```

```
  // returns a copy of the current Q, but DOES NOT COPY
      THE OBJECTS IN THE Q, JUST THE Q ITSELF!
  public Q2 copy()
  {
    Q2 copy = new Q2();
    for(Binode temp = this.front; temp != null ; temp =
        temp.getNext())
      copy.joinQ(temp.getData());
    return copy;
  }
  // empties out the Q
  public void clearQ()
  {
    this.front = null;
    this.back = null;
    this.length = 0;
  }
}
```

B.7 Statement.java

```
// This class gives statement objects.
// A statement has a name, a type (e.g. definition,
   theorem, lemma,...)
// and also has a set of citation relations from which
// other sorts of relations are inferred.
// This implementation is designed to be used with
// 'well written' inputs, and will not work properly
// if statements are input out of order or if the
   citation
// data do not fully and correctly specify the
   dependencies.

public class Statement
{
  private int ID; // this is a statement id which
     should typically be set to its place in the
     narrative order
  private String name,label;
  private Type type;
```

```
private Q2 cites;  // this is a queue of statements
   cited by the current one.  I will assume all
   dependencies can be found by citation chasing
private Q2 depcovs; // queue of dependency cover
   relations, determined from citations
private Q2 deps; // queue of dependencies
private int rank;
private int girth, centrality = 1; // to compute
   girth, use IDs and citation chasing to build d and
    increment the centralities of new depended-upon
   statements as you go
private long acc, acd, gcc, gcd; // arithmetic and
   geometric dependency and citation complexities
private double cra, crg;

// constructors
public Statement(int place, String n, String l, Type
   t, Q2 c)
{
  this.ID = place;
  this.name = n;
  this.label = l;
  this.type = t;
  this.cites = c; // c must be a queue of statements
  this.rank = this.findRank();
  this.depcovs = new Q2();
  dependencyCovers(); // find the dependency cover
     relations from citation and rank data and store
     them in deps
  this.acc = this.findACC();
  this.acd = this.findACD();
  this.gcc = this.findGCC();
  this.gcd = this.findGCD();
  this.cra = this.acc / this.acd;
  this.crg = this.gcc / this.gcd;
  this.deps = new Q2();
  dependencies(); // find the dependencies of the
     statement by citation chasing
  this.girth = this.deps.getLength() + 1;
}
```

```
// accessors
public int getID()
{
  return this.ID;
}
public String getName()
{
  return this.name;
}
public String getLabel()
{
  return this.label;
}
public Type getType()
{
  return this.type;
}
public Q2 getCites()
{
  return this.cites;
}
public Q2 getDepCovs()
{
  return this.depcovs;
}
public Q2 getDeps()
{
  return this.deps;
}
public int getRank()
{
  return this.rank;
}
public int getGirth()
{
  return this.girth;
}
public int getCentrality()
{
  return this.centrality;
}
```

```
public long getACC()
{
  return this.acc;
}
public long getACD()
{
  return this.acd;
}
public long getGCC()
{
  return this.gcc;
}
public long getGCD()
{
  return this.gcd;
}
public double getCRA()
{
  return this.cra;
}
public double getCRG()
{
  return this.crg;
}

// other methods
public void incrementCentrality()
{
  this.centrality++;
}
// this function finds the rank of the statement by
   comparing the ranks of those it cites
private int findRank()
{
  int max = -1;
  if(this.cites != null)
  { //go down the list of everything the statement
     cites and find the largest rank
    for(Binode current = this.cites.getFront() ;
       current != null && current.getData() != null ;
        current = current.getNext())
```

```
    {
      int r = ((Statement) current.getData()).getRank
          ();
      if(r > max) max = r;
    }
  }
  return max+1;
}
// this function determines which of the cited
   statements are covered by the current statement
// in the dependency poset
private void dependencyCovers()
{
  if(this.depcovs != null && this.cites != null)
  {
    for(Binode current = this.cites.getFront() ;
       current != null && current.getData() != null ;
        current = current.getNext())
    {// go down list of cites and find everything of
       rank one less than the our statement's rank
      // if the current statement's rank is one less
         than our statement's rank, then it's
         definitely covered by the current statement
      boolean possiblyCovered = true;
      if(((Statement) current.getData()).getRank() ==
          this.rank - 1)
      {
        // do nothing; possibly covered remains true
           and we get to skip the stuff in the 'else'
            section below
      }
      else // if not, it may still be covered, but we
          have to see if there are any dependencies
         between our statement and the current one
      { // this can be easily done because if a
         statement is not covered by our statement
         then it will be in the dependencies of
         something else cited by our statement

        for(Binode present = this.cites.getFront();
           possiblyCovered && present != null &&
```

```
            present.getData() != null; present =
            present.getNext())
          { // the current statement is still possibly
            covered by our statement if it is not in
            the dependencies of something else in the
            citation queue
            possiblyCovered = ! ((Statement) present.
              getData()).inDeps((Statement) current.
              getData());
          }
        }
        // if we haven't ruled out the current
          statement being covered by our statement,
          then we just do one final check that we
          haven't already added the statement to our
          list (have to do this in case of multiple
          citations of the same thing)
        if(possiblyCovered && ! this.inDepCovs((
          Statement) current.getData())) this.depcovs.
          joinQ(current.getData());
      }
    }
  }
  // finds arithmetic citation complexity, assuming all
     cited statements have complexities already
  private long findACC()
  {
    long total = 0;
    if(this.cites != null)
    { //go down the list of everything the statement
       cites and add its acc to the total
      for(Binode current = this.cites.getFront() ;
         current != null && current.getData() != null ;
          current = current.getNext())
      {
        total += ((Statement) current.getData()).getACC
          ();
      }
    }
    return total + 1;
  }
```

```
//   finds arithmetic dependency complexity, assuming
   all cited statements have complexities already
  private long findACD()
  {
    long total = 0;
    if(this.depcovs != null)
    { //go down the list of everything the statement
       covers and add its acd to the total
      for(Binode current = this.depcovs.getFront() ;
         current != null && current.getData() != null ;
          current = current.getNext())
      {
        total += ((Statement) current.getData()).getACD
           ();
      }
    }
    return total + 1;
  }
//   finds geometric citation complexity, assuming all
   cited statements have complexities already
  private long findGCC()
  {
    long total = 1;
    if(this.cites != null)
    { //go down the list of everything the statement
       cites and multiply its gcc to the total
      for(Binode current = this.cites.getFront() ;
         current != null && current.getData() != null ;
          current = current.getNext())
      {
        total *= 1 + ((Statement) current.getData()).
           getGCC();
      }
    }
    return total;
  }
//   finds geometric dependency complexity, assuming
   all cited statements have complexities already
  private long findGCD()
  {
    long total = 1;
```

```
    if(this.depcovs != null)
    { //go down the list of everything the statement
       covers and multiply its gcd to the total
      for(Binode current = this.depcovs.getFront() ;
         current != null && current.getData() != null ;
          current = current.getNext())
      {
        total *= 1 + ((Statement) current.getData()).
           getGCD();
      }
    }
    return total;
  }
//   builds a queue of the current statement's
   dependencies
  private void dependencies()
  { // this does citation chasing with comparison to
     make sure we're only adding new things
    if(this.deps != null && this.depcovs != null)
    {
      for(Binode current = this.depcovs.getFront() ;
         current != null && current.getData() != null ;
          current = current.getNext())
      { // everything this covers is certainly a
         dependency
        this.deps.joinQ((Statement) current.getData());
        ((Statement) current.getData()).
           incrementCentrality();
        for(Binode present = ((Statement) current.
           getData()).getDeps().getFront() ; present !=
            null && present.getData() != null ; present
            = present.getNext())
        { // here we're going through the dependencies
           of each statement covered by our current
           statement
          if(! this.inDeps((Statement) present.getData
             ()) )// if it's not already in the
             dependency queue...
          {
            this.deps.joinQ(present.getData()); // add
               it to the queue
```

```
            ((Statement) present.getData()).
               incrementCentrality(); // the statement
               we just added is depended upon by
               something, so add to its centrality
          }
        }
      }

    }
  }
//   returns true if the statement is in the
   dependencies list, and false otherwise
  public boolean inDeps(Statement s)
  {
    return this.inStatementQ(s, this.deps);
  }
//   returns true if the statement is in the dependency
    covers list, and false otherwise
  private boolean inDepCovs(Statement s)
  {
    return this.inStatementQ(s, this.depcovs);
  }
//   returns true if statement s is in queue q
  private boolean inStatementQ(Statement s, Q2 q)
  {
    if(q != null)
      for(Binode current = q.getFront() ; current !=
         null && current.getData() != null; current =
         current.getNext())
      {
        if(((Statement) current.getData()).getID() == s
           .getID()) return true;
      }
    return false;
  }

  private boolean supportedByQ(Statement s, Q2 q)
  {
    // returns true if every dependency of s is
       contained in q
    // returns true if s has no dependencies
```

```
  boolean supp = true;
  for(Binode current = s.getDeps().getFront(); supp
     && current != null && current.getData() != null;
      current = current.getNext())
  { // this will stop looping as soon as a statement
     is found in d(s) which is not in q, or when it
     runs out of statements to try
    supp = inStatementQ((Statement)current.getData(),
       q);
  }
  return supp;
}
// computes the number of linear extensions by
   generating all viable permutations of statements
public int linearExtensions()
{
  WrappedInt total = new WrappedInt(0);
  // we know that our present statement will be at
     the top of any linear extensions
  // so we must determine the number of order
     preserving permutations of the statement's
     dependencies
  Q2 ins = new Q2(); // in the linear extension
  Q2 outs = this.deps.copy(); // to be added; start
     with a copy of the dependencies of the statement
  this.lE(total, ins, outs);
  return total.value();
}
// this is the recursive part of the linearExtensions
    function
private void lE(WrappedInt total,Q2 ins, Q2 outs)
{
  if(outs.isEmpty())
  { // if this method is passed a pair of ins and
     outs which is already a linear extension,
     increment the total and be done
    total.increment();
    // For debugging, print the linear extension:
    //String record = this.name + ":";
    //for(Binode current = ins.getFront(); current !=
        null && current.getData() != null; current =
```

```
            current.getNext())
        //{
        //  record += ((Statement) current.getData()).
            getName() + ",";
        //}
        //System.out.println(record);
      }
      else
      {
        for(Binode current = outs.getFront(); current !=
            null && current.getData() != null ; current =
            current.getNext())
        {
          // we must take care to pull statements out of
              outs and put them into ins in such a way
              that we can return them to their original
              state without damaging anything
          // first, check that the statement in current
              is a viable addition to ins, if not, do
              nothing and the recursion stops
          if(supportedByQ((Statement) current.getData(),
              ins))
          {
            ins.cutQ((Statement) current.getData()); //
                add current to the linear extension
            outs.excise(current); // take current out of
                the outs queue, but hold on to its address
                 so it can be reinserted after all is said
                 and done
            lE(total,ins,outs);
            outs.reinsert(current);
            ins.leaveQ(); // lE is designed to leave both
                 queues the way it found them, so a simple
                 reinsertion into outs and having the
                object we just added to ins removed will
                restore things back to how we found them
          }
        }
      }
    }
  }
```

B.8 Type.java

```
public class Type
{
  private String name; // name of type, e.g. 'Lemma'
  private String shorthand; // short name, e.g. 'L'

  //constructors
  public Type(String n, String s)
  {
    this.name = n;
    this.shorthand = s;
  }

  public Type(String n)
  {
    this.name = n;
    if(n.length() > 0) this.shorthand = "" + n.charAt
       (0);
    else this.shorthand = "";
  }

  //accessors
  public String getName()
  {
    return this.name;
  }

  public String getShorthand()
  {
    return this.shorthand;
  }
  // no mutators,... types are pretty boring
}
```

B.9 Work.java

```
// This class implements a well written mathematical
   work

public class Work
{
```

```
  private Q2 statements; // a queue of statements

  // constructors
  public Work(Q2 st)
  {
    this.statements = st;
  }
  public Work()
  {
    this(new Q2());
  }

  // accessors
  public Q2 getStatements()
  {
    return this.statements;
  }
  public int getSize()
  {
    return this.statements.getLength();
  }
  public Statement findStatement(String name)
  { // this will stop at the first time it returns
     something, so there are a lot of implicit if-else'
     s here
    for(Binode current = this.statements.getFront();
       current != null && current.getData() != null;
       current = current.getNext())
    {
      if(((Statement) current.getData()).getName().
         equals(name)) return (Statement) current.
         getData();
    }
    return this.addStatement(name,name,new Type("
       Statement"),new Q2());
  }
//   the boolean LE tells whether to compute the
   computationally intensive linear extensions
  public String complexityData(boolean LE)
  { // rough version
    String output = "";
```

```
  if(LE) output += "s &r &g &z &acd &acc &gcd &
     gcc &e\\\\\n";
  else output += "s &r &g &z &acd &acc &gcd &
     gcc \\\\\n";
  for(Binode current = this.statements.getFront();
     current != null && current.getData() != null;
     current = current.getNext())
  {
    Statement s = (Statement) current.getData();
    if(LE)  output += s.getLabel() + " &" + s.
       getRank() + " &" + s.getGirth() + " &" + s.
       getCentrality() + " &" + s.getACD() + " &" +
        s.getACC() + " &" + s.getGCD() + " &"+ s.
       getGCC() + " &" + s.linearExtensions() + "
       \\\\\n";
    else output += s.getLabel() + " &" + s.getRank()
        + " &" + s.getGirth() + " &" + s.
       getCentrality() + " &" + s.getACD() + " &" +
        s.getACC() + " &" + s.getGCD() + " &"+ s.
       getGCC()  + "\\\\\n";
    // below are some commands from debugging for
       checking the statements' data
  //  output += s.getType().getName() + "," + s.
     getType().getShorthand() + ";";
  //  for(Binode i = s.getDepCovs().getFront(); i !=
     null && i.getData() != null; i = i.getNext())
  //  {
  //    output += ((Statement) i.getData()).getName()
     + ",";
  //  }
  //  output += "\n";
  }
  return output;
}

public String complexityData()
{ // default is to compute linear extensions
  return this.complexityData(true);
}

public String laTeXDeps()
```

```
{ // gives a tikzpicture output of the dependency
   Hasse diagram
  String output = "\\path\n";
  int numinrow = 1; // go until we have a rank with
     no statements in it
  for(int rank = 0; numinrow > 0 ; rank++)
  {
    numinrow = 0;
    for(Binode current = this.statements.getFront();
       current != null && current.getData() != null;
       current = current.getNext())
    {
      if(((Statement) current.getData()).getRank() ==
         rank)
      {
        if(rank > 0 || numinrow > 0) output += "--";
        else output += "(0,0";
        if(numinrow > 0) output += "+(" + numinrow +
           ",0";
        if(rank > 0 && numinrow == 0) output +="
           ++(0,1";
        output += ") node (" +  ((Statement) current.
           getData()).getName() + ") {" + ((Statement
           ) current.getData()).getLabel() + "} ";
        for(Binode present = ((Statement) current.
           getData()).getDepCovs().getFront();
           present != null && present.getData() !=
           null; present = present.getNext())
        {
          output += "edge (" + ((Statement) present.
             getData()).getName() + ") ";
        }
        output +="\n";
        numinrow++;
      }
    }
  }
  output += ";";
  return output;
}
```

```
public String laTeXCites()
{ // gives a tikzpicture output of the citation graph
  String output = "\\path\n";
  int numinrow = 1; // go until we have a rank with
     no statements in it
  for(int rank = 0; numinrow > 0 ; rank++)
  {
    numinrow = 0;
    for(Binode current = this.statements.getFront();
       current != null && current.getData() != null;
       current = current.getNext())
    {
      if(((Statement) current.getData()).getRank() ==
         rank)
      {
        if(rank > 0 || numinrow > 0) output += "-- ";
        else output += "(0,0";
        if(numinrow > 0) output += "+(" + numinrow +
           ",0";
        if(rank > 0 && numinrow == 0) output +="
           ++(0,1";
        output += ") node (" +  ((Statement) current.
           getData()).getName() + ") {" + ((Statement
           ) current.getData()).getLabel() + "} ";
        for(Binode present = ((Statement) current.
           getData()).getCites().getFront(); present
           != null && present.getData() != null;
           present = present.getNext())
        {
          output += "edge (" + ((Statement) present.
             getData()).getName() + ") ";
        }
        output +="\n";
        numinrow++;
      }
    }
  }
  output += ";";
  return output;
}
```

```
  // mutators

  public Statement addStatement(Statement s)
  {
    this.statements.joinQ(s);
    return s;
  }

  public Statement addStatement(String name, String
     label, Type type, Q2 cites) // cites should be a
     queue of statements
  {
    return this.addStatement(new Statement(this.getSize
       (),name,label,type,cites));
  }
}
```

B.10 WrappedInt.java

```
// a wrapper class for an integer with an increment
   command
public class WrappedInt
{
  private int val; // the integer value of the
     WrappedInt
  // constructors
  public WrappedInt(int v)
  {
    this.val = v;
  }
  public WrappedInt()
  {
    this(0);
  }
  // accessor
  public int value()
  {
    return this.val;
  }
  //mutator
  public void increment()
```

```
  {
    this.val++;
  }
}
```

B.11 RussellWhitehead.txt

```
Type,Definition,Df
Df,101,1.01
Type,Primitive Proposition,Pp
Pp,11,1.1
Pp,111,1.11
Pp,12,1.2
Type,Abbreviation,Ab
Ab,Taut,Taut,12
Pp,13,1.3
Ab,Add,Add,13
Pp,14,1.4
Ab,Perm,Perm,14
Pp,15,1.5
Ab,Assoc,Assoc,15
Pp,16,1.6
Ab,Sum,Sum,16
Pp,17,1.7
Pp,171,1.71
Pp,172,1.72
Type,Proposition,P
P,201,2.01,Taut,101
P,202,2.02,Add,101
P,203,2.03,Perm,101
P,204,2.04,Assoc,101
Ab,Comm,Comm,204
P,205,2.05,Sum,101
P,206,2.06,Comm,205,111
Ab,Syll,Syll,205,206
P,207,2.07,13
P,208,2.08,205,Taut,111,207,111
P,21,2.1,101
P,211,2.11,Perm,21,111
P,212,2.12,211,101
P,213,2.13,Sum,212,111,211,111
P,214,2.14,Perm,213,111,101
```

```
P,215,2.15,205,212,111,203,205,214,111,205,111,111,205,111,111
P,216,2.16,212,205,203,Syll,
P,217,2.17,203,214,205,Syll
Ab,Transp,Transp,215,216,217
P,218,2.18,212,205,201,Syll,214,Syll
P,22,2.2,Add,Perm,Syll
P,221,2.21,22
P,224,2.24,221,Comm
P,225,2.25,21,Assoc
P,226,2.26,225
P,227,2.27,226
P,23,2.3,Perm,Sum
```

B.12 Euclid.txt

```
Def,D10,D.10
Def,D15,D.15
Post,P1,P.1
Post,P2,P.2
Post,P3,P.3
Post,P4,P.4
CN,CN1,C.N.1
CN,CN2,C.N.2
CN,CN3,C.N.3
CN,CN4,C.N.4
Prop,I1,I.1,P3,P3,P1,D15,D15,CN1,
Prop,I2,I.2,P1,I1,P2,P3,P3,CN3,CN1
Prop,I3,I.3,I2,P3,D15,CN1
Prop,I4,I.4,CN4
Prop,I5,I.5,P2,I3,P1,I4
Prop,I7,I.7,I5
Prop,I8,I.8,I7
Prop,I11,I.11,I3,I1,I8,D10
Prop,I13,I.13,D10,I11,CN2,CN2,CN1
Prop,I15,I.15,I13,I13,P4,CN1,CN3
```

Sources

Audi, Robert (1998) *Epistemology: A contemporary introduction to the theory of knowledge.* London: Routledge.

Avigad, Jeremy (2006) 'Mathematical method and proof,' *Synthese* 153: 105–159.

— (2008) 'Understanding proofs' in Paolo Mancosu (ed.) *The Philosophy of Mathematical Practice.* Oxford: Oxford UP, 317-353.

Avigad, Jeremy and Reck, Erich H. (2001) ' "Clarifying the nature of the infinite:" the development of metamathematics and proof theory' *Carnegie Mellon Technical Report* CMU-PHIL-120. http://www.andrew.cmu.edu/user/avigad/ (accessed April, 2009).

Azzouni, Jodi (2009) 'Why Do Informal Proofs Conform to Formal Norms?' *Foundations of Science* 14:9–26.

Bergmann, Merrie; Moor, James; Nelson, Jack (2004) *The Logic Book,* 4^{th} *edn.* McGraw Hill.

Carnap, Rudolf (1937) *The Logical Syntax of Language.* Amethe Smeaton (trans.) London: Kegan Paul, Trench, Trubner & Co. Ltd.

Coleman, Edwin (2009) 'The Surveyability of Long Proofs' *Foundations of Science* 14:27–43.

Cook, Stephen A. and Reckhow, Robert A. (1979) 'The Relative Efficiency of Propositional Proof Systems,' *The Journal of Symbolic Logic* 44(1):36–50.

Dawson, John W. Jr. (2006) 'Why Do Mathematicians Re-prove Theorems?' *Philosophia Mathematica* 14:269–286.

Eder, Elmar (1992) *Relative complexities of first order calculi.* Wiesbaden: Vieweg.

Field, Hartry (1984) 'Is Mathematical Knowledge Just Logical Knowledge?' *The Philosophical Review* 93(4):509–552.

Friedman, Harvey M. (2005) 'Proofless text' http://www.andrew.cmu.edu/user/avigad/Papers/mkm/ (accessed April, 2009).

Friedman, H. and Flagg, R.C. (1990) 'A Framework for Measuring the Complexity of Mathematical Concepts,' *Advances in Applied Mathematics* 11:1–34.

Heath, Sir Thomas L. (1956) *The Thirteen Books of Euclid's Elements, translated from the text of Heiberg, with introduction and commentary. Second edition, revised with additions. Volume I: Introduction and Books I, II.* New York: Dover.

Hempel, Carl G. (1965) *Aspects of Scientific Explanation and Other Essays in the Philosophy of Science.* London: The Free Press (Macmillan).

Hunter, Geoffrey (1971) *Metalogic: An Introduction to the Metatheory of Standard First Order Logic.* Berkeley: U of California P.

Kieffer, Steve (2007) 'A Language for Mathematical Knowledge Management.' http://www.andrew.cmu.edu/user/avigad/Papers/mkm/ (accessed April, 2009).

Kieffer, Steven; Avigad, Jeremy; Friedman, Harvey (2008) 'A language for mathematical knowledge management.' Preprint (arXiv:0805.1386v2) http://www.andrew.cmu.edu/user/avigad/Papers/mkm/ (accessed April, 2009).

Kino, A.; Myhill, J.; Vesley, R. E., eds. (1970) *Intuitionism and Proof Theory: Proceedings of the Summer Conference at Buffalo N.Y. 1968.* Amsterdam: North-Holland Publishing Co.

Kitcher, Philip (1983) *The Nature of Mathematical Knowledge.* Oxford: Oxford UP.

Lahiri, Shuvendu K.; Ball, Thomas; Cook, Byron (2007) 'Predicate Abstraction via Symbolic Decision Procedures,' *Logical Methods in Computer Science* 3(2:1):1–20.

Lemmon, E. J. (1993) *Beginning Logic.* London: Chapman & Hall.

Lepore, Ernest (2000) *Meaning and Argument: An Introduction to Logic through Language.* Oxford: Blackwell.

Longo, Giuseppe (2003) 'Proofs and Programs,' *Synthese* 134:85–117.

Loomis, Elisha S. (1968) *The Pythagorean proposition; its demonstrations analyzed and classified, and bibliography of sources for data of the four kinds of proofs.* Washington: National Council of Teachers of Mathematics.

MacKenzie, Donald (2001) *Mechanizing Proof: Computing, Risk, and Trust.* Cambridge, MA: The MIT P.

Mizar Problems for Theorem Proving (MPTP) http://www.cs.miami.edu/~tptp/MPTPChallenge/ (accessed April, 2009).

Negri, Sara and von Plato, Jan (2001) *Structural Proof Theory.* Cambridge: Cambridge UP.

Paggiolesi, Francesca (2009) 'Introducing... Proof Theory,' *The Reasoner* 3(4):8–9.

Pelc, Andrzej (2009) 'Why Do We Believe Theorems?' *Philosophia Mathematica* 17:84–94.

Rav, Yehuda (1999) 'Why do we prove theorems?' *Philosophia Mathematica* 7:5–41.

— (2007) 'A Critique of a Formalist-Mechanist Version of the Justification of Arguments in Mathematicians' Proof Practices,' *Philosophia Mathematica* 15:291–320.

Russell, Bertrand (1919) *Introduction to Mathematical Philosophy.* London: George Allen and Unwin Ltd.

— (1937) *The Principles of Mathematics,* 2^{nd} *edn.* London: George Allen and Unwin Ltd.

Stanley, Richard P. (1997) *Enumerative Combinatorics: Volume I.* Cambridge: Cambridge UP.

Suppe, Frederick, ed. (1974) *The Structure of Scientific Theories.* Urbana: U Illinois P.

Suppe, Frederick (1989) *The Semantic Conception of Theories and Scientific Realism.* Urbana: U Illinois P.

Suppes, Patrick (1969) *Studies in the Methodology and Foundations of Science: Selected Papers from 1951 to 1969.* Dordrecht-Holland: D. Reidel Pub. Co.

Tarski, Alfred (1956) *Logic, Semantics, Metamathematics: Papers from 1923 to 1938.* J. H. Woodger (trans.) Oxford: Clarendon P.

Tavanec, P. V., ed., Blakeley, T. J., trans., (1970) *Problems of the Logic of Scientific Knowledge.* Dordrecht-Holland: D. Reidel Pub. Co.

Urquhart, Alasdair (1995) 'The Complexity of Propositional Proofs,' *The Bulletin of Symbolic Logic* 1(4):425–467.

Van Bendegem, Jean Paul and Van Kerkhove, Bart (2009) 'Mathematical Arguments in Context' *Foundations of Science* 14:45–57.

Whitehead, Alfred North and Russell, Bertrand (1910) *Principia Mathematica.* Cambridge: Cambridge UP. Accessed online at http://name.umdl.umich.edu/AAT3201.0001.001.

Wittgenstein, Ludwig (1956) *Remarks on the Foundations of Mathematics.* G. E. M. Anscombe (trans.) Oxford: Basil Blackwell.