4 Introduction to Software Complexity

At the end of the previous chapter, five categories of complexity factors that would serve as a starting point to deterministically evaluate the complexity of proceduralized tasks were identified. As In this chapter, software complexity measures will be explained as a theoretical basis for quantifying the complexity of proceduralized tasks. In this regard, it may be necessary to start this chapter by examining why software complexity must be considered in order to quantify the complexity of proceduralized tasks.

4.1 Software Complexity

We live in a very convenient time, and our lives are made easier by various kinds of computer technologies. For example, (1) we can buy a book from an online bookstore managed by powerful mainframes as well as sophisticated software, (2) we can produce merchandise using a fully automated machine controlled by well-structured software, and (3) we can even operate on a patient using a robot that is manipulated by precise software. However, in order to enjoy these conveniences we must secure reliable software that is able to perform all the required functions we want. For this reason, allied industries have been spending a tremendous amount of money and other resources to develop reliable software.

From this standpoint, one of the canonical approaches is to manage the complexity of software, because it directly affects software maintainability. Carver (1987) pointed out that the maintainability of software is a kind of quantitative measure that makes it possible to evaluate how easy it is to understand given software. Similarly, Gibson and Senn (1989) stated that *maintainability is defined as the ease with which systems can be understood and modified* (p. 348). Although there are other definitions about maintainability, it is evident that maintenance is one of the crucial aspects determining the reliability of software, because it contains all kinds of software engineering activities required after the implementation of software. Carver (1987) summarized these activities as follows:

> These distinct categories of maintenance can be identified: (1) corrective maintenance, (2) adaptive maintenance, and (3) perfective maintenance. Corrective maintenance is the diagnosis and correction of latent software errors. It is required when errors undiscovered during testing and debugging are found. Since a correct program is rare, latent errors are common. The errors may vary in impact from trivial to critical. In any case, the code must

be modified to correct the error. Adaptive maintenance is maintenance due to changes in the external environment of a program. New generations of hardware and later releases of software are among causes of adaptive maintenance. Perfective maintenance is maintenance intended to enhance the system to meet the changing needs of the user. It includes modifications of existing functions, inclusion of general enhancements, and modifications for improved system performance (p. 299).

Therefore, if a new error is introduced in the course of performing software maintenance activities, the increase in maintenance costs is unavoidable (Cant et al. 1995; Carver 1987; Gibson and Senn 1989; Hops and Sherif 1995; Lew et al. 1988; Soi 1985). A more serious problem is that the possibility of undesired consequences will increase in proportion to the increase of the possibility of software malfunctions. As a result, since the early 1970s, diverse research projects on software complexity have been conducted in order to quantitatively control as well as predict the complexity of software, because it has been revealed that maintenance personnel are apt to show impaired performance when they have to deal with complicated software (Curtis et al. 1979; Davis and LeBlanc 1988; Kafura and Reddy 1987; McCabe and Butler 1989; Rombach 1987).

It is worth emphasizing that one of the major purposes of quantifying the complexity of software is to evaluate its understandability. For example, Gibson and Senn (1989) stated that *the more complex system is, the more difficult it is to understand, and therefore to maintain* (p. 347). Similarly, Carver (1987) pointed out that "Ease of understanding decreases as program complexity increases. Since complexity is a measure of the effort to comprehend, to maintain and to test software, the level of complexity of a program affects the maintainability of a program (p. 299)."

Moreover, Davis and LeBlanc (1988) articulated that "Available evidence and the opinion of many experts strongly suggest that programmers do not understand programs on a character by character basis. Rather they assimilate groups of statements which have a common function (p. 1366)."

This means that a theoretical framework quantifying the complexity of software can be used for quantifying the complexity of proceduralized tasks because (1) software complexity mainly deals with the level of understandability of software and (2) understandability in software complexity focuses not on reading comprehension (i.e., WR, CR and CMP) but on task comprehension, which affects the performance of tasks to be done by qualified operators (i.e., TP). Actually, this is not a new idea, because other researchers have already tried to apply software complexity measures to evaluating the complexity of supervisory control tasks (Murray and Liu 1994) and vice versa (Darcy et al. 2005). Therefore, it is very helpful to scrutinize the applicability of software complexity measures to quantifying the complexity of proceduralized tasks.

4.2 Software Complexity Measure

Many kinds of unique measures that are capable of quantifying the complexity of software from diverse viewpoints have been suggested for several decades. However, without loss of generality, software complexity measures fall into one of the following four categories: (1) those based on the size of the software, (2) those based on the data structure of the software, (3) those based on the control structure of the software, and (4) a combination of the first three measures (Carver 1987; Coskun and Grabowski 2001; Davis and LeBlanc 1988; Fenton and Neil 1999; Gonzalez 1995; Hops and Sherif 1995; Huang and Lai 1998; Khoshgoftaar et al. 1997; Lakshmanan et al. 1991; Soi, 1985).

First, one of the representative measures belonging to the first category is the line of code (LOC). This measure is very clear and straightforward because it is strongly expected that the longer the software source code, the greater the complexity of the software. Another typical measure is Halstead's E measure, which considers the frequencies of occurrence of operators as well as operands included in source code. Figure 4.1 illustrates how to quantify the value of Halstead's E measure with respect to an arbitrary source code.

Source code	Operator	Frequency	Operand	Frequency
IF (A = 0) THEN	;	2	A	3
A = B;	=	3	B	1
ELSE	( )	1	C	1
A = C;	IF	1		
	THEN	1		
	ELSE	1		

$$E = \frac{\eta_1 N_2 (N_1 + N_2)}{2\eta_2} \log_2(\eta_1 + \eta_2) = 221.9$$

η_1 = Number of unique operators = 6

η_2 = Number of unique operands = 3

N_1 = Total number of operators = 9

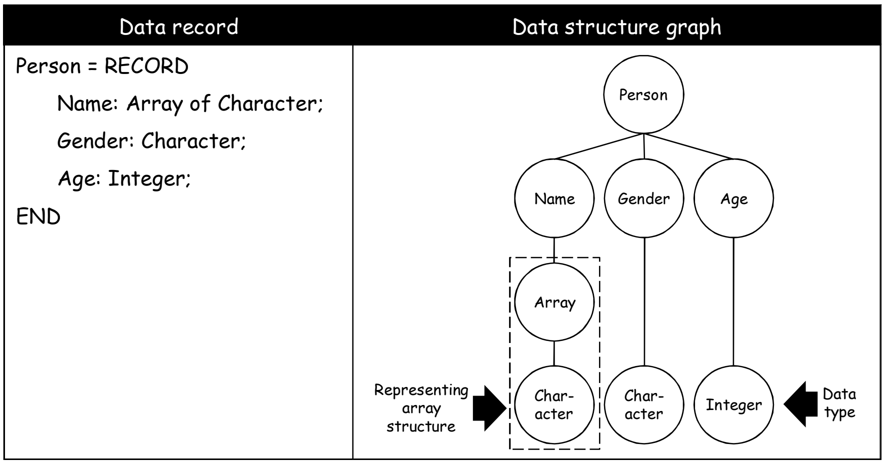
N_2 = Total number of operands = 5

Fig. 4.1 Quantifying the value of Halstead's E measure (Park et al. 2001, © Elsevier)

Second, the complexity of software can be quantified from the point of view of a data structure. Regarding this, it would be interesting to quote Wirth (1985):

> Yet, it is abundantly clear that a systematic and scientific approach to program construction primarily has a bearing in the case of large, complex programs which involve complicated sets of data. Hence, a methodology of programming is also bound to include all aspects of data structuring. Programs, after all, are concrete formulations of abstract algorithms based on particular representations and structures of data (p. 7).

This strongly suggests that complicated software requires complicated data structures as well as huge amounts of data. Accordingly, the complexity of data structures should be a good measure for quantifying the complexity of software. For this reason, many kinds of complexity measures that are able to deal with the complexity of data structures have been suggested. One of the typical measures is the depth of a data structure graph (Gonzalez 1995). Here, data structure graph means a graph that consists of nodes and arcs, where nodes denote data entities and arcs represent the relationship between nodes. For example, the hierarchical level of an arbitrary data structure shown in Fig. 4.2 is three due to the existence of a linear array (refer to the area surrounded by dotted lines).



Data record

```
Person = RECORD
    Name: Array of Character;
    Gender: Character;
    Age: Integer;
END
```

Fig. 4.2 Example of a data structure graph

Third, much work has been done considering the effect of a control flow graph on the complexity of software. Here, control flow graph (also called a program control graph) means a directed graph that has a unique entry and exit node, which is very similar to the flowchart of software (Baker 1978; Lakshmanan et al. 1991). In a control flow graph, each node denotes a block in source code that performs a specific function, and each arc represents a branch taken between nodes (Ramamurthy and Melton 1988). Therefore, it is very straightforward to expect that the complexity of software will be proportional to the complexity of the control flow graph.

One of the canonical measures belonging to this category is McCabe's cyclomatic complexity (v), which can be calculated by $v = e - n + 2p$. Here, e, n, and p denote the number of edges (i.e., arcs), the number of nodes, and the number of connected components included in an arbitrary control flow graph, respectively. More simply, it was found that v is equal to the number of decision nodes plus one (McCabe and Butler; 1989). Therefore, from the point of view of McCabe's cyclomatic complexity, the complexity of two control flow graphs shown in Fig. 4.3 is identical, because they have two decision nodes.

Lastly, it is possible to measure the complexity of software by combining two or more complexity measures that belong to the aforementioned categories. For example, Ramamurthy and Melton (1988) and Curtis et al. (1979) suggested novel measures based on the integration of Halstead's E measure with McCabe's cyclomatic complexity. In addition, Bail and Zelkowitz (1988) and Oviedo (1980) suggested software complexity measures by simultaneously considering the control flow graph and the data structure graph of software.

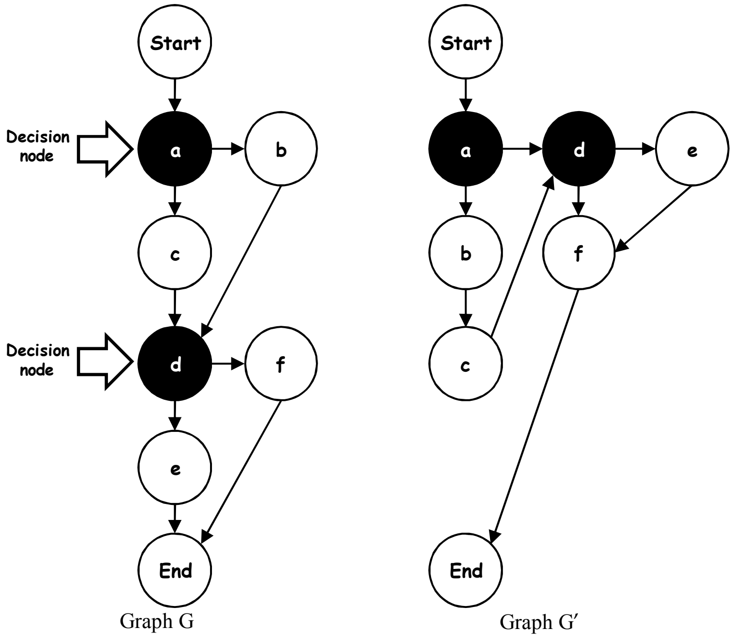


Fig. 4.3 Two control flow graphs with the same McCabe's cyclomatic complexity

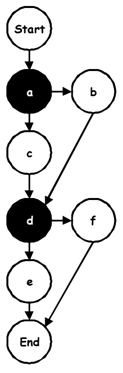
Here, it is important to point out that there is another complexity measure that belongs to this category. That is, instead of combining several complexity measures that quantify the complexity of software using *different* methods, a new measure can be developed based on the integration of submeasures quantifying the complexity of software with an *identical* method. A typical example is a measure based on the concept of graph entropies because, as illustrated in Figs. 4.2 and 4.3, many graphic representation techniques have been used to analyze the characteristics of software.

4.3 The Concept of Graph Entropies

Traditionally, the entropy concept has been widely adopted in various research areas because it is very useful for expressing the degree of complexity (Shannon 1948). For this reason, including a series of works done by Mowshowitz (Mowshowitz 1968a-d), many researchers have expended considerable effort to quantify the complexity of software using the concept of graph entropies (Davis and LeBlanc 1988; Huang and Lai 1998; Gonzalez 1995; Lew et al.1988). For example, let us consider the definition of the first-order and the second-order entropy suggested by Davis and LeBlanc (1988).

In order to quantify the first order entropy, the classes of nodes in a control flow graph should be identified based on their in- and out-degree as they appear. If there are nodes that share the same in- and out-degree, then they are regarded as nodes belonging to an equivalent class. In this regard, Fig. 4.4 depicts how to quantify the first-order entropy of two arbitrary graphs shown in Fig. 4.3.

Graph



Class	In-degree	Out-degree	Node
I	0	1	{Start}
II	1	1	{b, c, e, f}
III	1	2	{a }
IV	2	0	{End}
V	2	2	{d}

A_i =Number of nodes belonging to the ith distinctive class

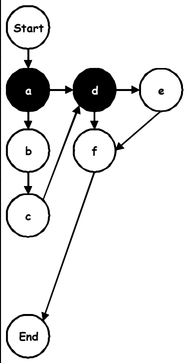
N =Total number of nodes in a graph

h =Number of distinctive classes

p_i = Estimated probability of the ith distinctive class = A_i/N

The first-order entropy of graph G = $H_1(G)$ $= -\sum_{i=1}^{h} p_i \log_2 p_i$

$$= -\left\{\frac{1}{8}\log_2(\frac{1}{8}) + \frac{4}{8}\log_2(\frac{4}{8}) + \frac{1}{8}\log_2(\frac{1}{8}) + \frac{1}{8}\log_2(\frac{1}{8}) + \frac{1}{8}\log_2(\frac{1}{8})\right\} = 2.000$$



Class	In-degree	Out-degree	Node
I	0	1	{Start}
II	1	0	{End}
III	1	1	{b, c, e}
IV	1	2	{a}
V	2	1	{f}
VI	2	2	{d}

The first-order entropy of graph G' = $H_1(G')$

$$= -\left\{\frac{1}{8}\log_2(\frac{1}{8}) + \frac{1}{8}\log_2(\frac{1}{8}) + \frac{3}{8}\log_2(\frac{3}{8}) + \frac{1}{8}\log_2(\frac{1}{8}) + \frac{1}{8}\log_2(\frac{1}{8}) + \frac{1}{8}\log_2(\frac{1}{8})\right\}$$

$$= 2.406$$

Fig. 4.4 The first order entropy of two arbitrary control flow graphs

In Fig. 4.4, it is apparent that all the nodes included in a graph G fall into the following classes: {Start}, {b, c, e, f}, {a}, {End}, and {d}. Accordingly, the number of distinctive classes denoted by h is five. In addition, the probability of each class is 1/8, 4/8, 1/8, 1/8, and 1/8, respectively. In this way, h and the probability of the associated classes can be calculated with respect to the graph G′. As a result, the first-order entropy of graphs G and G′ is 2.000 and 2.406, respectively.

From the point of view of the logical entanglement of control flow graphs, the value of the first-order entropy is very interesting, because the logic structure of graph G′ seems to be more complicated than that of graph G. Intuitively, this result is meaningful, because a control flow graph that consists of many equivalent nodes will tend to have a lower first-order-entropy value. In other words, if there is a kind of regularity in a control flow graph, it is expected that the value of the first-order entropy will be reduced because of the repetition of similar execution patterns, which results in an increase of the number of nodes belonging to identical node classes. In contrast, the value of the first-order entropy will increase due to irregular execution patterns, because the number of distinctive classes that are necessary to express the irregularity of execution patterns will increase. This means that the effect of logical entanglement on the complexity of software can be quantified by the first-order entropy.

Similarly, the second-order entropy can be calculated except for the class identification scheme. That is, nodes are considered to be equivalent if they share identical neighbors within one arc distance. The intention of this classification scheme is to express the amount of information that is needed to describe each node position, since the comprehension of a control flow graph becomes difficult with respect to the increase in the number of distinctive classes. For example, let us consider Table 4.1, which shows the distinctive classes of two control flow graphs G and G′, which are necessary to calculate the values of the second-order entropy.

Table 4.1 Distinctive classes of two control flow graphs

Graph G		Class	Graph G′	
Node	Neighbor node		Node	Neighbor node
{Start}	{a}	I	{Start}	{a}
{a}	{Start, b, c}	II	{a}	{Start, b, d}
{b, c}	{a, d}	III	{b}	{a, c}
{d}	{b, c, e, f}	IV	{c}	{b, d}
{e, f}	{d, End}	V	{d}	{a, c, e, f}
{End}	{e, f}	VI	{e}	{d, f}
–	–	VII	{f}	{d, e, End}
–	–	VIII	{End}	{f}

Based on the results of node class identifications summarized in Table 4.1, the values of the second-order entropy of two graphs can be calculated as below.

The second-order entropy of graph G = $H_2(G)$

$$= -\sum_{i=1}^{6} p_i \log_2 p_i$$

$$= -\left\{\frac{1}{8}\log_2(\frac{1}{8}) + \frac{1}{8}\log_2(\frac{1}{8}) + \frac{2}{8}\log_2(\frac{2}{8}) + \frac{1}{8}\log_2(\frac{1}{8}) + \frac{2}{8}\log_2(\frac{2}{8}) + \frac{1}{8}\log_2(\frac{1}{8})\right\} = 2.500$$

The second-order entropy of graph G′ = $H_2(G')$

$$= -\sum_{i=1}^{8} p_i \log_2 p_i$$

$$= -\left\{\begin{matrix} \frac{1}{8}\log_2(\frac{1}{8}) + \frac{1}{8}\log_2(\frac{1}{8}) + \frac{1}{8}\log_2(\frac{1}{8}) + \frac{1}{8}\log_2(\frac{1}{8}) \\ + \frac{1}{8}\log_2(\frac{1}{8}) + \frac{1}{8}\log_2(\frac{1}{8}) + \frac{1}{8}\log_2(\frac{1}{8}) + \frac{1}{8}\log_2(\frac{1}{8}) \end{matrix}\right\} = 3.000$$

As can be seen from the above results, the value of the second-order entropy will increase in proportion to the increase in the number of nodes because the meaning of each node position becomes more unique. This means that more effort is required to understand the contents of software that consists of many nodes. Therefore, the second-order entropy of a control flow graph can be used to measure the effect of size on the complexity of software.

Here, it should be noted that the second-order entropy can be used to quantity the amount of information pertaining to a data structure graph. In other words, if the second-order entropy of an arbitrary graph implies the amount of information needed to understand its contents, the second-order entropy of a data structure graph can be used to measure the effect of the data structure on the complexity of the software. Therefore, based on the concept of graph entropies, it is possible to define a novel measure of software complexity. For example, Lew et al. (1988), Gonzalez (1995), and Huang and Lai (1998) proposed a novel measure by integrating several complexity measures quantified by the concept of graph entropies.

4.4 Selecting Appropriate Measures

At the end of Sect. 4.1, it was pointed out that software complexity measures could be used for quantifying the complexity of proceduralized tasks. The easiest way to do this is to use the associated software complexity measure that is able to evaluate one of the task complexity factors. For example, let us look at Table 4.2, which compares five kinds of task complexity factors with the associated software complexity measures.

Table 4.2 Comparing task complexity factors with the associated software complexity measures

Task complexity factor	Software complexity measure based on	Example
Amount of information	Data structure of software	The hierarchical level of a data structure graph
Number of actions	Size of software	Halstead's E measure
Logical entanglement	Control structure of software	McCabe's cyclomatic complexity
Amount of domain knowledge	–	–
Level of engineering decision	–	–

Table 4.2 suggests that we should be able to use Halstead's E measure to quantify the effect of the number of actions on the complexity of proceduralized tasks, because this related to the size of software, which would be directly comparable to the size of proceduralized tasks (i.e., number of actions to be conducted by qualified operators). Similarly, McCabe's cyclomatic complexity, which evaluates the logical entanglement of the control structure of software, would be a good alterna-

tive to quantify the effect of logical entanglement on the complexity of proceduralized tasks.

Unfortunately, there are three critical problems in this approach. First, there is no corresponding software complexity measure that is capable of evaluating the effect of the amount of domain knowledge on the complexity of proceduralized tasks. Likewise, there is no appropriate software complexity measure regarding the level of engineering decision.

Second, even if corresponding software complexity measures were available, some of them would likely have limited application to the quantification of the complexity of proceduralized tasks. For example, let us recall the value of the first-order entropy of two arbitrary graphs G and G′ (Fig. 4.3). From the point of view of McCabe's cyclomatic complexity, these two graphs have the same value. However, it is intuitively evident that the control structure of graph G′ is more complicated than that of graph G. This means that there are times when McCabe's cyclomatic complexity is not appropriate for quantifying the effect of logical entanglement on the complexity of proceduralized tasks. In addition, the result of a previous study has revealed that Halstead's E measure has a limitation in application to the complexity of proceduralized tasks (Park et al. 2001).

This limitation engenders the third problem, which is related to integrating the effects of five kinds of task complexity factors. It is very natural to assume that the overall complexity of proceduralized tasks should be determined based on the integration of partial contributions originating from five kinds of task complexity factors. Unfortunately, this is not a valid idea. Let us assume that we quantified the effects of the number of actions and logical entanglement on the complexity of proceduralized tasks by using Halstead's E measure and McCabe'c cyclomatic complexity, respectively. Nevertheless, combining the value of Halstead's E measure with that of McCabe's cyclomatic complexity is less meaningful because, as mentioned earlier, there are times when these measures give inappropriate results about the complexity of proceduralized tasks. In addition, the integration of heterogeneous measures would become another source of difficulty in quantifying the complexity of proceduralized tasks.

For the above reasons, a better way to quantify the complexity of proceduralized tasks seems to use the concept of graph entropies. That is, if we construct a series of graphs that are able to represent the nature of five kinds of task complexity factors, the contribution of each factor can be quantified by either the first-order entropy or the second-order entropy. In addition, since the technical basis of graph entropies is homogeneous to some extent (i.e., the entropy value of an arbitrary graph can be calculated by a set of probabilities obtained from the definition of a node classification scheme), it is expected that one should be able to integrate the contributions of five kinds of task complexity factors into a single and meaningful value.

References

Bail WG, Zelkowitz MV (1998) Program complexity using hierarchical computers. Comput Lang 13(3/4):109–123
Baker TP (1978) Natural properties of flowchart step-counting measures. J Comput Syst Sci 16:1–22
Cant SN, Jeffery DR, Henderson-Sellers B (1995) A conceptual model of cognitive complexity of elements of the programming process. Inf Softw Technol 37(7):351–362
Carver DL (1987) Producing maintainable software. Comput Ind Eng 12(4):299–305
Coskun E, Grabowski M (2001) An interdisciplinary model of complexity in embedded intelligent real-time systems. Inf Softw Technol 43:527–537
Curtis B, Sheppard SB, Milliman P, Borst MA, Love T (1979) Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe Metrics. IEEE Trans Softw Eng 5(2):96–104
Darcy DP, Kemerer CF, Slaughter SA, Tomayko JE (2005) The structural complexity of software: An experimental test. IEEE Trans Softw Eng 31(11):982–995
Davis JS, LeBlanc RJ (1988) A study of the applicability of complexity measures. IEEE Trans Softw Eng 14(9):1366–1372
Fenton NE, Neil M (1999) Software metrics: successes, failures and new directions. J Syst Softw 47:149–157
Gibson VR, Senn JA (1989) System structure and software maintenance performance. Commun ACM 32(3):347–358
Gonzalez RR (1995) A unified metric of software complexity: measuring productivity, quality and value. J Syst Softw 29:17–37
Hops JM, Sherif JS (1995) Development and application of composite complexity models and a relative complexity metric in a software maintenance environment. J Syst Softw 31:157–169
Huang SJ, Lai R (1998) On measuring the complexity of an Estelle specification. J Syst Softw 40:165–181
Kafura D, Reddy GR (1987) The use of software complexity metrics in software maintenance. IEEE Trans Softw Eng 13(3):335–343
Khoshgoftaar TM, Allen EB, Lanning DL (1997) An information theory-based approach to quantifying the contribution of a software metric. J Syst Softw 36:103–113
Lakshmanan KB, Jayaprakash S, Sinha PK (1991) Properties of control-flow complexity measures. IEEE Trans Softw Eng 17(12):1289–1295
Lew KS, Dillon TS, Forward KE (1988) Software complexity and its impact on software reliability. IEEE Trans Softw Eng 14(11):1645–1655
McCabe TJ, Butler CW (1989) Design complexity measurement and testing. Commun ACM 32(12):1415–1425
Mowshowitz A (1968a) Entropy and the complexity of graphs: I. An index of the relative complexity of a graph. Bull Math Biophys 30:175–204
Mowshowitz A(1968b) Entropy and the complexity of graphs: II. The information content of digraphs and infinite graphs. Bull Math Biophys 30:225–240
Mowshowitz A (1968c) Entropy and the complexity of graphs: III. Graphs with prescribed information content. Bull Math Biophys 30:387–414
Mowshowitz A (1968d) Entropy and the complexity of graphs: IV. Entropy measures and graphical structure. Bull Math Biophys 30:533–546
Murray J, Liu Y (1994) A software engineering approach to assessing complexity in network supervision tasks. In: Proceedings of the IEEE International Conference on Human, Information and Technology, San Antonio, TX, 1:25–29

Oviedo EI (1980) Control flow, data flow and program complexity. In: Proceedings on IEEE COMPSAC, Chicago, pp.146–152,

Park J, Jung W, Ha J (2001) Development of the step complexity measure for emergency operating procedures using entropy concepts. Reliabil Eng Syst Saf 71:115–130

Ramamurthy B, Melton A (1988) A synthesis of software science measures and the cyclomatic number. IEEE Trans Softw Eng 14(8):1116–1121

Rombach HD (1987) A controlled experiment on the impact of software structure on maintainability. IEEE Trans Softw Eng 13(3):344–354

Shannon CE (1948) A mathematical theory of communication. Bell Syst Tech J 27:379-423/623–656

Soi IM (1985) Software complexity: an aid to software maintainability. Microelectron Reliabil 25(2):223–228

Wirth N (1985) Algorithms and Data Structures. Prentice Hall, Englewood Cliffs, NJ