

EXPLORING ALGORITHMS FOR EFFECTIVE APPLICATIONS OF THE GRAPH THEORY

Vladimir V. Riabov

Rivier University

420 S. Main Street ▪ Nashua, NH 03060-5086, USA ▪ E-mail: vriabov@rivier.edu

Mathematical models and computing tools are explored for applications of the graph theory in informatics, networking technologies, complexity code analyses, and dynamics of biological and social systems. Various concepts are examined to evaluate route, covering and decomposition problems (minimum spanning tree, shortest path, travelling salesman problem, maximum cliques, and edge coloring).

1. Introduction

The graph theory [1], traditionally covered in *Discrete Mathematics* and *Algorithms* courses [2, 3], introduces students to the modeling of various types of relations between entities in technical, physical, chemical, biological, and social systems and their complex dynamics. Graph representations are widely used in computer science (analyses of the flow of computation, data communication in networks, complexity code analysis, data-mining strategies, etc.); mathematics (geometry, topology); physics, chemistry, and biology (studies of molecular structures; habitation and migration patterns, the spread of diseases, etc.); sociology (social network analysis, prestige measurement); linguistics (syntax, grammar, compositional and lexical semantics), and in other disciplines.

The objective of this paper is to review mathematical aspects of various algorithms that involve graphs, and explore practical applications with computing tools. The algorithms and codes were examined by undergraduate and graduate students in *Discrete Mathematics*, *Algorithms*, *Computer Networks*, *Software Engineering*, and other courses.

2. Exploring the Minimum Spanning Tree Algorithm

Various route problems (e.g., *minimum spanning tree* (MCT), *shortest path*, and *travelling salesman problems*) were studied using graphs. The purpose of the first case study was to analyze subgraphs of an undirected, weighted-edge graph representing a network (used in telecommunications, transportation, water supply, and electrical grids) and to identify the subgraph that is a tree (known as a *minimum [weight] spanning tree*) and connects all the vertices together. Using the method of induction and contradiction, it can be proved [3] that, if each edge has a distinct weight, then there will be only one, unique minimum spanning tree. The first MCT algorithm for developing an efficient electrical grid of Moravia was offered by Czech scientist Otakar Borůvka in 1926 [4]. Nowadays, Prim's algorithm and Kruskal's algorithm [3, 5] are commonly used. All three are *greedy algorithms* [3] that run in polynomial time. Estimations of running time for the codes that implemented Prim's and Kruskal's algorithms confirm this finding.

Prim's minimum spanning tree algorithm [3] was applied for solving the problem 8.6b from the textbook [6, p. 416] used by students in the *Algorithms* course. The original weighted graph [6] is shown in Fig. 1. The task is to build the minimum spanning tree starting at the vertex *H* and continuing until the tree spans all the vertices in the graph. The pseudocode of the Prim's algorithm is discussed in [3, pp. 634-636] in detail.

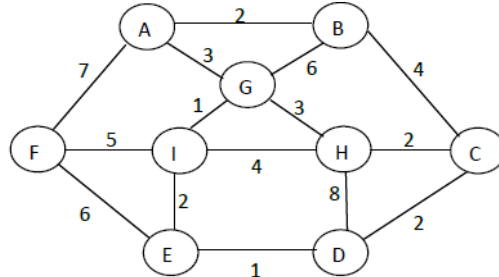


Figure 1. The original weighted graph [6, p. 396].

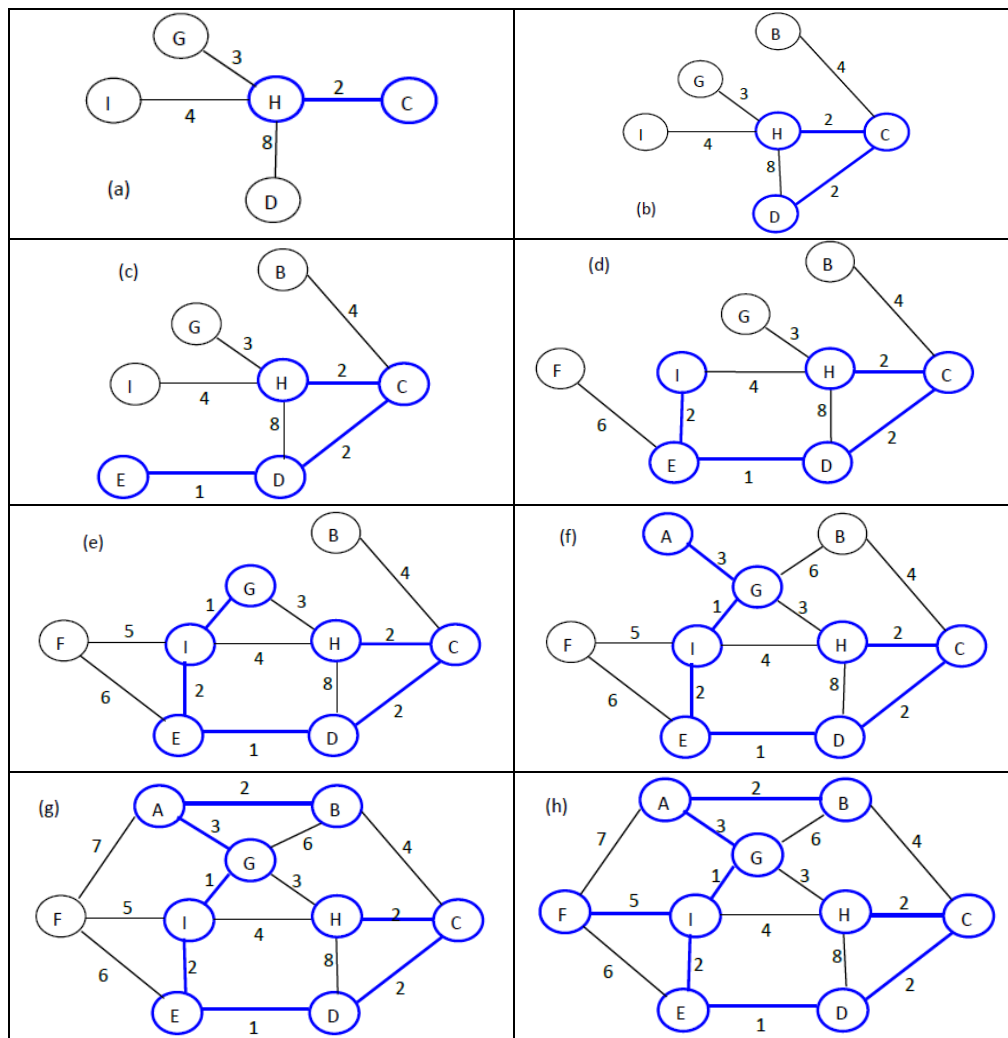
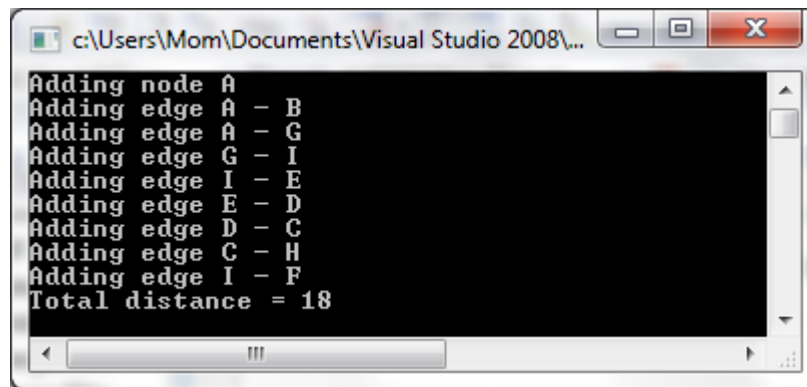


Figure 2. The execution of Prim's algorithm on the graph (Fig. 1) starting at vertex *H*.

The steps of implementation of Prim's algorithm [3, 6] on the original graph from Fig. 1 (with the starting vertex H) are shown in Fig. 2 (a-h). The final minimum spanning tree is shown in blue in Fig. 2h with the total weight of 18.

Using Visual Studio™ 2008 under Windows-7™ OS, Mary Slocum, a student in the *Algorithms* course created the C++ program (see *Appendix*) that implements the Prim's minimum spanning tree algorithm described in [6, pp. 388-399, 422]. The program builds a data structure to record the minimum spanning tree found. The separate procedure generates the output that includes the graph, the set of edges in the tree, along with their weights, and the total weight of the tree. Using this C++ program, the Prim's algorithm was applied for building the minimal spanning trees from various graphs, including the weighted graph shown in Fig. 1.



```
c:\Users\Mom\Documents\Visual Studio 2008\...
Adding node A
Adding edge A - B
Adding edge A - G
Adding edge G - I
Adding edge I - E
Adding edge E - D
Adding edge D - C
Adding edge C - H
Adding edge I - F
Total distance = 18
```

Figure 3. Results of running the C++ code (see *Appendix*) for the graph shown in Fig. 1.

The sequence of edges that were selected from the original weighted graph (see Fig. 1), starting from the vertex *A*, and added to the minimum spanning tree is shown in Fig. 3. The total weight of the tree is 18 and remains the same for any other choice of the starting vertex (e.g., see the case of selecting the vertex *H* analyzed in this section).

3. Solving the Single-Source Shortest Path Problems

The second case study addresses the *shortest path problem* of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized [3]. One of the most important algorithms for solving the single-source shortest path problems, Dijkstra's algorithm [3, 6, 7], is studied in detail. Analyzing directed graphs with nonnegative weights, it has been found that different algorithm implementations have different complexity time (T) characteristics, which depend on numbers of vertices (V) and edges (E): 1) implementation with list is estimated as $T = O(V^2)$ [7]; 2) implementation with modified binary heap is estimated as $T = O((E+V)\log V)$ [3]; and 3) implementation with Fibonacci heap is estimated as $T = O(E+V\log V)$ [8]. Fast specialized algorithms [9] for modern applications of shortest path algorithms allow automatically find directions between physical locations, such as driving directions on web mapping websites like MapQuest™ or Google Maps™.

As an example (see Problem 8.15c from [6, p. 419]), Dijkstra's shortest-path algorithm [7] was executed for the graph shown in Fig. 4a, starting from the vertex A . The solution (see Fig. 4) was found by Kai Wang, a student in the *Algorithms* class. The blue arrows (see Figs. 4a-f) clearly indicate which edges become part of the tree and in what order.

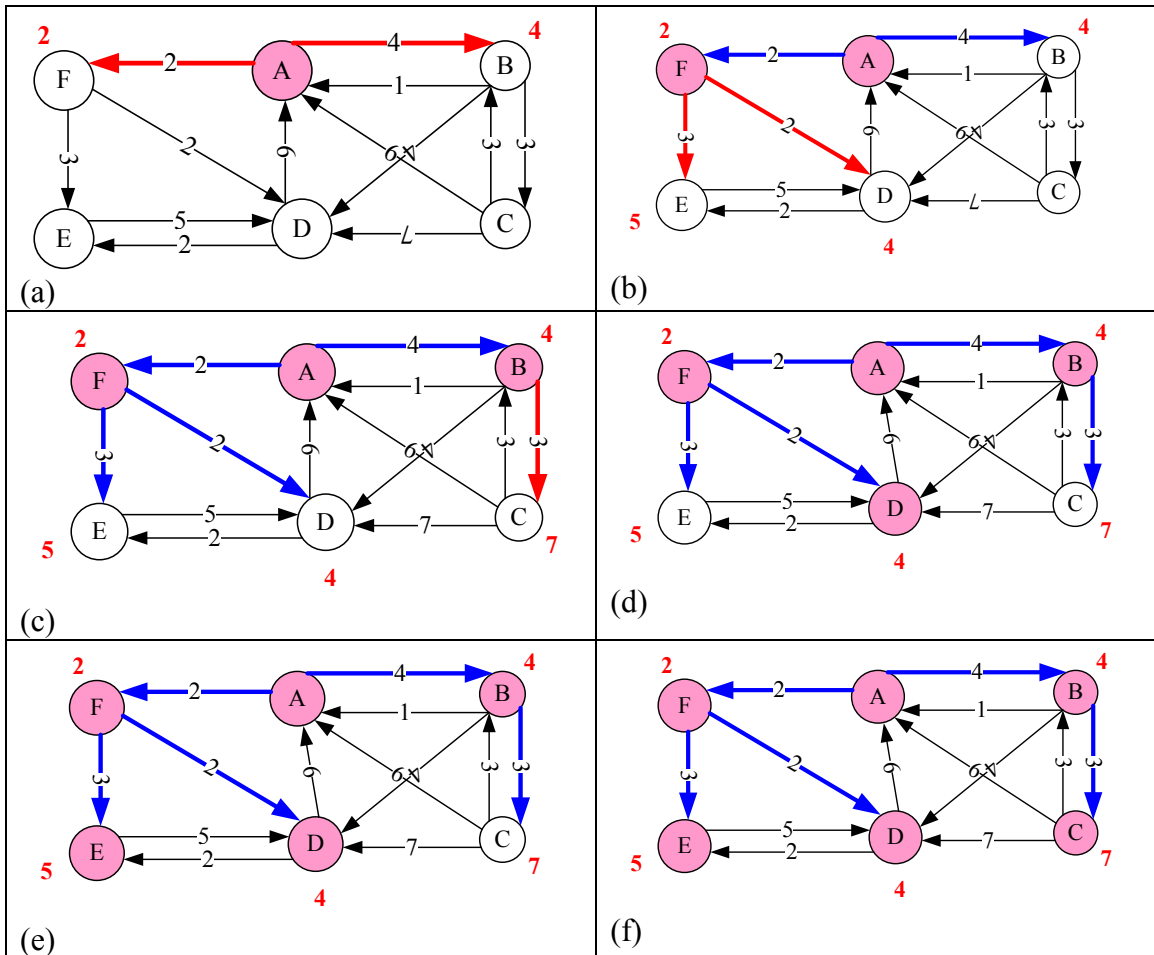


Figure 4. The steps (a-f) of Dijkstra's shortest-path algorithm for the given digraph [6, p. 419], starting from the vertex $s = A$.

Table 1. Progression of the selected edges in the Dijkstra's shortest-path algorithm.

Steps	A	B	C	D	E	F	From (Cost)
1	0	∞	∞	∞	∞	∞	A(0)
2		4	∞	∞	∞	2	F(2)
3		4	∞	4	5		B(4)
4			7	4	5		D(4)
5			7		5		E(5)
6			7				C(7)

The Table 1 represents the progression of selecting edges in the Dijkstra's shortest-path algorithm. The selected edges of the final graph and the order in which they were selected

could be easily found from the Table 1: {AF, AB, FD, FE, BC}. The graph without the unused edges is shown below in Fig. 5 below. The pseudocode for the Dijkstra's shortest-path algorithm implementation is given in [3, p. 658].

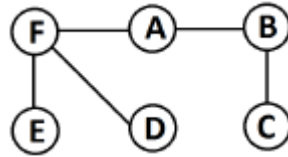


Figure 5. The final graph that represent Dijkstra's shortest-path algorithm (see Table 1).

4. Exploring the Travelling Salesman Problem

The third case study explores the *travelling salesman problem* (TSP) of finding the shortest path that goes through every vertex exactly once, and returns to the start. Unlike the shortest path problem, which can be solved in polynomial time in graphs without negative cycles [10], the travelling salesman problem is NP-complete [11] and could not be efficiently solvable. It is likely that the worst-case running time for any algorithm for the TSP increases exponentially with the number of nodes ("cities"). Since 1930, this problem is intensively studied with applications in combinatorial optimization, logistics, planning, manufacturing microchips, DNA sequencing, etc. As the first attempt, students started exploring the "*brute force search*" method of the direct solution trying all permutations. The running time for this approach lies within a polynomial factor of $O(n!)$; therefore, this solution becomes impractical even for about 20 "cities". Following the principles of dynamic programming [3], the Held-Karp algorithm [12] was developed that solves the problem in time $O(n^2 2^n)$. In team projects, students made overview of modern approaches for solving the TSP problem including implementations of *branch-and-bound* and *cut generation* methods for graphs with up to 85,900 "cities" [13].

5. Studying the Programming Code Complexity

In the last case study, the structured testing methodology [14, 15] and graph-based metrics (cyclomatic complexity, essential complexity, module design complexity, system design complexity, and system integration complexity) were reviewed by students and applied for studying the C-code complexity and estimating the number of possible errors and required unit and integration tests for the Carrier Networks Support system [16]. Comparing different code releases, it is found that the reduction of the code complexity leads to significant reduction of errors and maintainability efforts.

5.1 Software Metrics Overview

The McCabe metrics [14, 15] are based on graph theory and mathematically rigorous analyses of the code structure, which explicitly identify high-risk areas. For each module (a subroutine with a single entry point and a single exit point), an annotated code listing and flowgraph is generated as shown in Fig. 6. The flowgraph is an architectural diagram of a software module's logic.

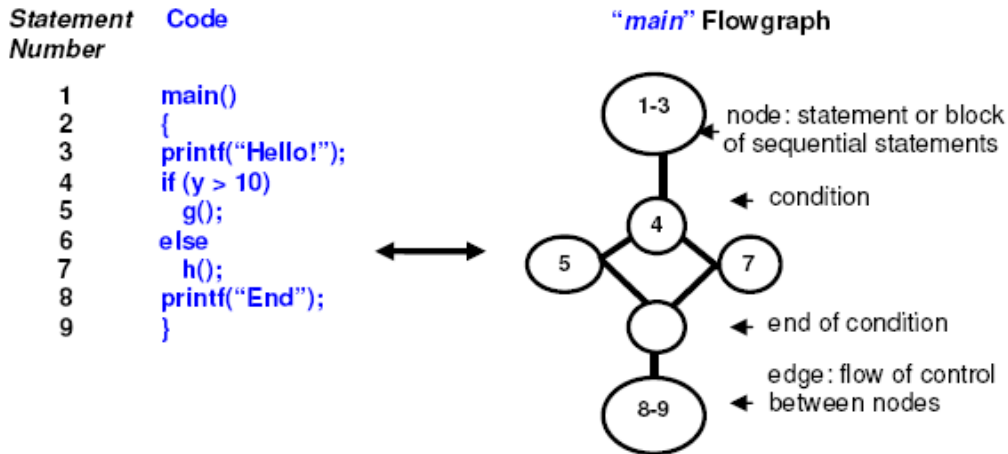


Figure 6. The annotated source listing and the related flowgraph.

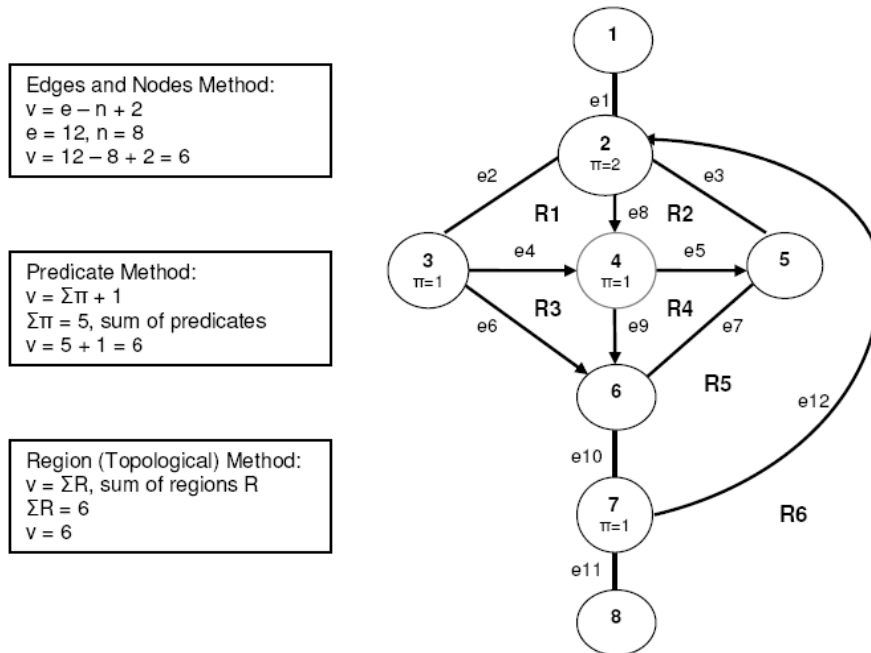


Figure 7. Three methods of evaluating the cyclomatic complexity of the sample graph.

Cyclomatic complexity, v , is a measure of the complexity of a module's decision structure [14]. It is the minimum number of independent paths that should be tested to reasonably guard against errors. A high cyclomatic complexity indicates that the code may be of low quality and difficult to test and maintain. The results of Miller's experiments [17] suggest that modules approach zero defects when v is within 7 ± 2 . Therefore, the threshold of v -metric is chosen as 10. A node is the smallest unit of code in a program. Edges on a flowgraph represent the transfer of control from one node to another [14]. A module flowgraph with e edges and n nodes has the cyclomatic complexity $v = e - n + 2$ that is the number of topologically independent regions of the graph [14, 15] (see Fig. 7).

Essential complexity, ev , is a measure of unstructuredness [14], the degree to which a module contains unstructured logical constructs [15], which decrease the quality of the code and increase the effort required to maintain the code and break it into separate modules. When a number of unstructured constructs is high (ev is high), modularization and maintenance is difficult. These modules should be recommended for redesigning.

Module design complexity, iv , is a measure of its decision structure as it relates to calls to other modules. This quantifies test efforts of a module with respect to integration with other modules. Software with high values of iv tends to have a high degree of control coupling, which makes it difficult to isolate, maintain, and reuse software components.

System design complexity, S_0 , measures the amount of interaction between modules in a program [15]. The S_0 metric is calculated as the sum of the module design complexities of all modules in a program. It reveals the complexity of the module calls in a program and measures the effort required for bottom-up integration testing. Integration complexity, S_I , measures the number of integration tests necessary to guard against errors [15]. It is the number of linearly independent sub-trees in a program. A sub-tree is a sequence of calls and returns from a module to its descendant modules. The S_I metric quantifies the integration testing effort. It is calculated by using a simple formula, $S_I = S_0 - N + 1$, where N is the number of modules in the program. Modules with no decision logic do not contribute to S_I . This fact isolates system complexity from its total size.

The McCabe IQ tool produces Halstead metrics [18] for selected languages. Supported by numerous industry studies [15], the B-metric of Halstead represents the estimated number of errors in the program.

5.2 Results of the Project Code Analysis

The structured testing methodology [15] and McCabe's IQ tools were used in the C code analyses of different internetworking systems. As an example, the Support Carrier Networks system [16] is studied. It provides both services of conventional Layer 2 switches and the routing and control services of Layer 3 devices. The code (3400 modules, about 300,000 lines written in C language) was examined with the McCabe metrics. Nine protocol-based sub-trees of the code (for BGP, DVMRP, FR, ISIS, IP, MOSPF, OSPF2, PIM, and PPP protocols) were analyzed. This code analysis identified areas with potential errors and modules to be reviewed. This experience significantly improved the code implementation practices of our students.

It was found that 38% of the code modules have the cyclomatic complexity more than 10. About 48% of modules have the essential cyclomatic complexity more than 4. Only two parts of the code (for FR and ISIS protocols) have low v and ev metrics. Totally 1147 modules (34%) are unreliable and unmaintainable. Among 3400 modules considered, 1447 modules (42%) are fully structured with $ev = 1$, and 500 modules (15%) are completely unstructured with $ev = v$. 1066 code modules (31%) have the module design complexity more than 5. Only four protocol-based branches of the code (FR, ISIS, IP,

and PPP) have low *iv*-metrics. BGP, MOSPF, and PIM protocol implementations have the worst characteristics (42% of modules require more than 7 integration tests per module). The system design complexity (S_0) is 19417, which is a top estimation of the number of unit tests that are required to fully test the program. The system integration complexity (S_I) is 16026, which is a top estimation of the number of integration tests.

The study of Halstead metrics [18] indicates that the code potentially contains 2920 errors; 203 code modules (6%) have the number of delivered bugs $B > 3$ per module. Only five parts of the code (for FR, ISIS, IP, OSPF2, and PPP protocols) have relatively low error metrics ($B \ll 1$). In other branches (BGP, DVMRP, MOSPF, and PIM), $B > 1$.

5.3 Comparison of Two Customer Software Releases: Redesign Efforts

Based on this analysis of the code, we recommended 271 modules of the old Release 1.2 for redesigning by the software development team. As a result, 16 old modules were deleted and 7 new modules were added for issuing the new Release 1.3. Analyzing the deleted modules, we found that 7 deleted modules were unreliable ($v > 10$) and 6 deleted modules were unmaintainable ($ev > 4$). Also, 19% of the deleted code was both unreliable and unmaintainable. All seven newly added modules were reliable and maintainable.

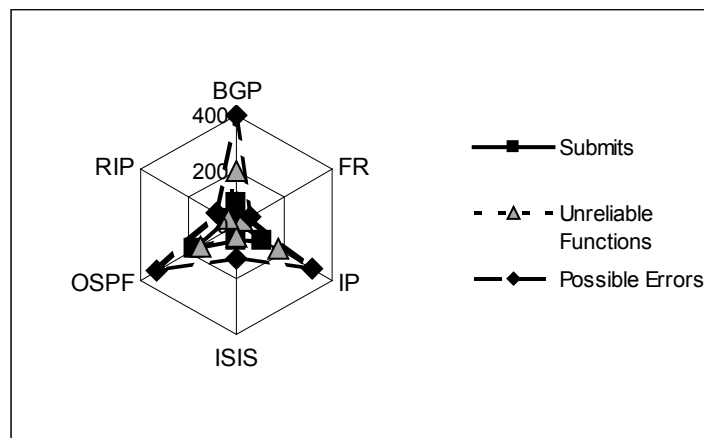


Figure 8. Correlation between the number of error submits, number of unreliable functions ($v > 10$), and the number of possible errors for six protocols.

After redesigning, the code cyclomatic complexity was reduced by 115 units. 70 old modules (41% of the code) were improved, and only 12 modules (7% of the code) become worse. This analysis demonstrates a robustness of the structured testing methodology and mutual successful efforts of design and test engineers, which allow improving the quality of customer releases. Studying the relationship between software defect corrections and cyclomatic complexity [16], we found a great correlation between the numbers of possible errors, unreliable functions (with $v > 10$), and the error submits from the Code Releases (see Fig. 8).

6. Concluding Remarks on Students' Involvement

After brief in-class discussions of the case studies, each student continued working on a selected case analyzing algorithms, creating computer codes (in C/C++ or Java), running them at various parameters, comparing numerical results with known data, and presenting the findings to classmates. In the course evaluations, students stated that they became deeply engaged in course activities through examining the challenging problems related to the advanced concepts from the graph theory and its practical applications.

Acknowledgements

The author expresses gratitude to his students Lois Carvalho, Mary Slocum, and Kai Wang for participation in developing algorithms, exploring case studies, and discussions.

References

- [1] Bondy, J.A., and Murty, U.S.R. *Graph Theory*, Springer, 2008.
- [2] Ferland, K. *Discrete Mathematics*, Cengage Learning, 2008.
- [3] Cormen, T. H., et al. *Introduction to Algorithms*, 3rd edition, The MIT Press, 2009.
- [4] Nešetřil, J.; Milková, E.; Nešetřilová, H. *Discrete Mathematics*, 2001; **233** (1–3): 3–36.
- [5] Gibbons, A. *Algorithmic Graph Theory*, Cambridge University Press, 1985.
- [6] Baase, S., and Van Gelder, A. *Computer Algorithms: Introduction to Design & Analysis*, 3rd edition, Addison Wesley Logman, 2000.
- [7] Dijkstra, E. W. *Numerische Mathematik*, 1959; **1**: 269–271.
- [8] Fredman, M. L., and Tarjan, R. E. *J. Assoc. Computing Machinery*, 1987; **34** (3): 596–615.
- [9] Sanders, P. (March 23, 2009). *Fast route planning*. Google Tech Talk. [Online]
<http://www.youtube.com/watch?v=-0ErpE8tQbw>
- [10] The Shortest Path Problem, Wikipedia, 2013. [Online]
http://en.wikipedia.org/wiki/Shortest_path_problem
- [11] Karp, R. M. Reducibility among Combinatorial Problems. In R. E. Miller and J. W. Thatcher (editors). *Complexity of Computer Computations*. New York: Plenum, 1972, pp. 85–103.
- [12] Held, M.; Karp, R. M. *J. Society for Industrial and Appl. Mathem.*, 1962; **10** (1): 196–210.
- [13] Applegate, D. L., et al. *The Traveling Salesman Problem*, Princeton University Press, 2007.
- [14] McCabe, T. J. *IEEE Transactions on Software Engineering*, 1976; **2** (4), 308-320.
- [15] Watson, A. H., and McCabe, T. J. *NIST Special Publication*, No. 500-235. NIST, 1996.
- [16] Riabov, V. V. *Journal of Computing Sciences in Colleges*, 2011; **26** (6): 86-92.
- [17] Miller, G. *Psychological Review*, 1956; **63** (2), 81-97.
- [18] Halstead, M. H. *Elements of Software Science*. New York: Elsevier North Holland, 1977.

Appendix: C++ Code for Implementing Prim's Minimum Spanning Tree Algorithm

```
// UpdateDistances() keeps track of distances added to tree
void UpdateDistances(int target)
{
    for (int i = 0; i < NumNodes; ++i)
    {
        if ((Weight[target][i] != 0) && (Distances[i] > Weight[target][i]))
        {
            Distances[i] = Weight[target][i];
        }
    }
}
```

```
        ConnectedTo[i] = target;
    }
}

int main(int argc, char *argv[])
{
    CString strMsg; // Output message string

    ReadDataFromFile();

    /* Initialize Distances with 9999 */
    for (int i = 0; i < NumNodes; ++i)
        Distances[i] = 9999;

    /* Mark all nodes as NOT being in the minimum spanning tree */
    for (int i = 0; i < NumNodes; ++i)
        NodeInTree[i] = 0;

    /* Add the first node to the tree */
    strMsg.Format(_T("Adding node %c"), 0 + 'A');
    std::wcout << (LPCTSTR)strMsg << std::endl;
    NodeInTree[0] = 1;

    UpdateDistances(0); // Set to 0 initially

    int total = 0; // Initialize total distance
    for (int i = 1; i < NumNodes; i++)
    {
        /* Find the node with the smallest distance to the tree */
        int min = -1; // Initialize
        for (int j = 0; j < NumNodes; j++)
        {
            if (NodeInTree[j] == 0)
                if ((min == -1) || (Distances[min] > Distances[j]))
                    min = j; // Found minimum
        }

        /* Added node to tree */
        strMsg.Format(_T("Adding edge %c - %c"), ConnectedTo[min] +
'A', min + 'A');
        std::wcout << (LPCTSTR)strMsg << std::endl;

        NodeInTree[min] = 1; // 1 indicates that node is added to tree
        total += Distances[min];
        UpdateDistances(min);

    } // end of for loop

    cout << "Total distance = " << total << endl;

    return 0;
}
```