Available online at www.sciencedirect.com
SCIENCE DIRECT®
PERGAMON
Computers & Operations Research 30 (2003) 1121–1134
computers & operations research
www.elsevier.com/locate/dsw

Using software complexity measures to analyze algorithms—an experiment with the shortest-paths algorithms

Jukka K. Nurminen*

Nokia Research Center, P.O.Box 407, FIN-00045 Nokia Group, Finland

Received 1 February 2001; received in revised form 1 November 2001; accepted 1 February 2002

Abstract

In this paper, we apply different software complexity measures to a set of shortest-path algorithms. Our intention is to study what kind of new information about the algorithms the complexity measures (lines-of-code, Halstead's volume, and cyclomatic number) are able to give, to study which software complexity measures are the most useful ones in algorithm comparison, and to analyze when the software complexity comparisons are appropriate. The experiment indicates that the software complexity measures give a new dimension to empirical algorithm comparison. The results explicitly show the trade-off between speed and implementation complexity: a faster algorithm typically requires a more complex implementation. Different complexity measures correlate strongly. Therefore, even the simple lines-of-code measure gives useful results. As the software complexity measures are easy to calculate and since they give useful information, the study suggests that such measures should be included in empirical algorithm comparisons. Unfortunately, for meaningful results, all the algorithms have to be developed in the same fashion which makes the comparison of independent implementations difficult.

Scope and purpose

For practical use an algorithm has to be fast and accurate as well as easy to implement, test, and maintain. In this work, we investigate whether software complexity measures could make the implementation aspects more explicit and allow algorithm comparisons also in this dimension. We calculate lines-of-code, Halstead's volume, and cyclomatic number measures for different shortest-path algorithms. We investigate if such measures are applicable to algorithm comparison and study what can be learned from algorithms when they are also compared in implementation dimension.

The main purpose of the work is to understand if it is possible to explicitly measure the implementation complexity of an algorithm. Having such measures available would help the practitioner to choose the algorithm that best matches the need. The optimal algorithm for a given task would have adequate performance with minimal implementation complexity. For algorithm research implementation, complexity measures would offer

*Tel.: +358-7180-36855; fax: +358-7180-36229.
E-mail address: jukka.k.nurminen@nokia.com (J.K. Nurminen).

0305-0548/03/$ - see front matter
PII: S0305-0548(02)00060-6

a new analysis dimension and make easy implementation a parallel goal with algorithm performance.

Keywords: Algorithm implementation; Software complexity measures; Shortest-path algorithms

1. Introduction

Algorithms are frequently assessed by the execution time and by the accuracy or optimality of the results. For practical use, a third important aspect is the implementation complexity. An algorithm which is complex to implement requires skilled developers, longer implementation time, and has a higher risk of implementation errors. Moreover, complicated algorithms tend to be highly specialized and they do not necessarily work well when the problem changes [1].

Algorithms can be studied theoretically or empirically. Theoretical analysis allows mathematical proofs of the execution times of algorithms but can typically be used for worst-case analysis only. Empirical analysis is often necessary to study how an algorithm behaves with "typical" input, see e.g. [2] or [3]. Empirical analysis tends to focus on the execution time and optimality.

Ball and Magazine [4] listed criteria for the comparison of heuristic algorithms that in addition to execution time included ease of implementation, flexibility, and simplicity. Likewise, in our study of the evolution of routing algorithms [1], we noted that it would be useful if some measure of the implementation complexity would be available.

Unfortunately, the details of the implementation complexity are hardly ever reported in empirical algorithm comparisons. This can be due to several reasons. Firstly, implementation aspects are "soft" and there is no specific measure that would accurately characterize them. Secondly, it can be argued that the implementation complexity is implicitly visible in the algorithm description and thus available to the interested readers. Finally, it can be that the importance of implementation complexity is not fully understood by researchers who do not have to worry about issues like time-to-market, robustness of implementation and maintainability.

In this study, we experiment with software complexity measures [5,6] to see if they could supplement the speed and optimality attributes and provide useful insight into algorithm implementation. Software complexity affects the testability, maintainability, defect rate, and evolution potential. Several studies, e.g. [7–10], have found that the software complexity measures can be successfully used to estimate the maintenance effort of software.

The established use of software complexity measures is, as part of the implementation process, to find critical program modules and to make predictions about the maintenance effort. Contrary to that, our idea is to use software complexity measures proactively to compare alternative algorithms and, in this way, to assist in the selection of the most appropriate algorithm for a given task.

In particular, we try to answer the following questions:

- do implementation complexity measures provide useful information about algorithms,
- when and under what conditions can the measures be used to compare algorithms, and
- which of the measures would be most suitable for algorithm comparison.

The rest of the paper is structured as follows. Section 2 reviews the main measures used in this study. Section 3 presents the material we have used in this study and describes our setting for the experiment. Section 4 gives the results, which are discussed in Section 5. In Section 6, we present some conclusion and ideas for future work.

2. Software complexity measures

There are a large number of implementation complexity measures available. In this study, we have concentrated on three established measures: lines-of-code, Halstead's volume, and cyclomatic number. Since there are slight variations in how different tools implement the measures, the following definitions define how the tool CMT++, which we have used in this study, calculates the measures [11].

2.1. Lines-of-code (LOCpro)

Lines-of-code are the most traditional measures used to quantify software complexity. They are simple, easy to count, and very easy to understand. They do not, however, take into account the intelligence content and the layout of the code. In this study, we use a LOCpro measure, which counts the number of program lines (declarations, definitions, directives, and code).

2.2. Volume (V)

Halstead's volume V [5] describes the size of the implementation of an algorithm. Computation of V is based on the number of operations performed and operands handled in the algorithm. Therefore, V is less sensitive to code layout than the lines-of-code measures. You could roughly think of V as the number of bits the code portion could be represented with when comments are stripped away, possible code layout variations are stripped away and the short/long identifier naming issue has been filtered away.

2.3. Cyclomatic number V(G)

Cyclomatic number $V(G)$ [6] describes the complexity of the control flow of the program. For a single function, $V(G)$ is one less than the number of conditional branching points in the function. $V(G)$ is also increased for each 'and' and 'or' logical operator met in condition expressions. The greater the cyclomatic number is the more execution paths there are through the function, and the harder it is to understand.

Note that cyclomatic number is insensitive to complexity of data structures, data flows, and module interfaces.

3. Experiment with the shortest-path algorithms

For the experiment, we used the shortest-path algorithms that were developed and empirically compared in [12]. We focused only on the comparison of different algorithms (Sections 6–9 in [12])

Table 1
Summary of studied algorithms

	Brief description	Worst-case complexity
BFP	FIFO order selection (Bellman–Ford–Moore algorithm) with parent-checking	$O(nm)$
DIKBD	Minimum label selection using double buckets	$O(m + n(2 + C^{1/2}))$[a]
DIKH	Minimum label selection using k-ary heaps	$O(m \log n)$[a]
GOR	Topological order selection for general graphs	$O(nm)$
GOR1	GOR with scans during topological sort	$O(nm), O(m + n)$ with acyclic networks
PAPE	Selection using a double-ended queue (Pape-Levit algorithm)	$O(n2^n)$
THRESH	Threshold selection	$O(nm)$[a]
TWO-Q	Two queue selection (Pallottino's algorithm)	$O(n^2m)$

[a]Complexity with nonnegative length function.
C, biggest absolute value of an arc length; n, number of nodes; m, number of arcs.

and ignored the experiments with the Dijkstra variations. We used the data of only the largest problem in each table. In most cases, there were no significant differences between the relative behavior of the algorithms with the different problem sizes. The code for these algorithms is available in the SPLIB that can be found at http://www.intertrust.com/star/goldberg/soft.html.

Table 1 describes briefly the algorithms and the worst-case complexities of their running times. The algorithms are analyzed in detail in [12]. All the studied algorithms are based on the labeling method. In graph $G = (V, E)$ for every node v, the method maintains its distance label $d(v)$, parent $\pi(v)$, and status $S(v) \in \{\text{unreached, labeled, scanned}\}$. Initially, for every node $v, d(v) = \infty, \pi(v) = \text{nil}$, and $S(v) = \text{unreached}$. The method starts by setting for the source node $s, d(s) = 0$ and $S(s) = \text{labeled}$, and applies the following SCAN operation to the labeled nodes until none exists, in which case, the method terminates.

```
Procedure SCAN(v)
   for all (v, w) ∈ E do
      if d(v) + distance(v, w) < d(w) then
         d(w) ← d(v) + distance(v, w);
         S(w) ← labeled;
         π (w) ← v;
      end if.
      S(v) ← scanned;
   end for.
end procedure.
```

Different algorithms use different strategies to select the labeled nodes to be scanned next. Bellman–Ford–Moore (BFP) algorithm uses an FIFO queue. Dijkstra's algorithm selects a labeled node with the minimum distance label as the next node to be scanned. The variants of Dijkstra's algorithms

differ by how the nodes with the minimum distance labels are found. DIKH uses a heap while DIKBD uses a double bucket implementation.

Pape–Levit (PAPE) algorithm and Pallottino's algorithm (TWO-Q) divide the labeled nodes into two sets. High-priority set (S_1) contains nodes which have been scanned at least once and low-priority set (S_2) contains the rest of the labeled nodes. The next node to be scanned is always selected from S_1. If S_1 is empty the node is selected from S_2. Pape–Levit algorithm maintains S_1 as an LIFO stack and S_2 as an FIFO queue. Pallottino's algorithm maintains both $S_1$and S_2 as FIFO queues.

The threshold (THRESH) algorithm also divides the labeled nodes into two sets, S_1 and S_2, which are maintained as FIFO queues. The set selection is done by comparing the distance label of a node v with a threshold parameter t, which is set to a weighted average of the minimum and average distance labels of the nodes in S_2. An iteration starts when S_1 is empty and the algorithm moves all the nodes with distance labels less than t from S_2 to S_1. During an iteration, nodes that become labeled are added to S_2.

The Goldberg–Radzik algorithms (GOR and GOR1) also use two sets, S_1 and S_2. At the beginning of each iteration, the algorithm computes S_1 by adding the reachable nodes from S_2 and applying topological sort to order S_1 using the parent–child links. GOR implements the topological sort using depth-first search. GOR1 combines the depth-first search with the shortest-path scanning (the if-statement within the above SCAN procedure).

Table 2 gives an overview of the test problems. As described in detail in [12] the problems are from three different families. Rectangular grid networks (Grid-*) consist of points in the plane. Points with a fixed x-coordinate form a layer which is a doubly connected cycle. The problems in this family differ by the shape of the rectangular grid and by the lengths of the arcs connecting the layers. The second problem family (Rand-*) is constructed by creating a hamiltonian cycle and then adding arcs with distinct random end points. The final problem family (Asyc-*) consists of acyclic networks. The nodes are numbered from 1 to n, and there is a path of arcs $(i, i+1)$, $1 \leqslant i < n$. In addition to these path arcs, additional arcs exist between random nodes leading from lower to the higher numbered nodes.

Based on the running times, we ranked the algorithms so that the fastest one was given a ranking of 1, the second fastest 2, and so on. We then calculated the average ranking over the different problem types, and ranked the algorithms based on their average ranking. The rationale behind this idea was to base the ranking on good general-purpose behavior—not on exceptionally good behavior on some particular problem type.

All the algorithms were not able to solve all the problems. For those algorithms, which were not able to solve a problem, we gave a ranking which equaled the number of algorithms in the comparison.

As the code of [12] was properly structured, one file contained exactly one algorithm. Executor code (*_t.c and *_run.c) was ignored since it only contained input and output code.

Additionally, in order to compare different, independent implementations we used our own shortest path algorithm implementations for a network planning tool, NPS/10 [13] and the Dijkstra implementation from Generic Graph Component Library (GGCL). [14], that is available in http://www.lsc.nd.edu/research/ggcl/. In these implementations, the algorithms were not as cleanly isolated as in [12]. They contained extra code for debugging, for interfacing with other programs, and for the reuse of common data structures. To study their effect on algorithm complexity, we

Table 2
Summary of test problems

Problem	Brief description	Nodes	Arcs	Arc lengths
Grid-SSquare	Square grid	1048577	3145728	uniformly distributed [0, 10000]
Grid-SSquare-S	Square grid with artificial source	1048578	4194305	uniformly distributed [0, 10000]
Grid-SWide	Wide grid	524289	1572864	uniformly distributed [0, 10000]
Grid-SLong	Long grid	524289	1572864	uniformly distributed [0, 10000]
Grid-PHard	Hard problem with nonnegative arc lengths	262145	2095424	intralayer: small, nonnegative interlayer: large, nonnegative
Grid-NHard	Hard problem with mixed arc lengths	262145	2095424	intralayer: small, nonnegative interlayer: large, nonpositive
Rand-4	Random hamiltonian graph with density 4	1048576	4194304	cycle arcs: 1 intercycle arcs:uniformly distributed [0, 10000]
Rand-1:4	Random hamiltonian graph with density 25%	4096	4194304	cycle arcs: 1 intercycle arcs:uniformly distributed [0, 10000]
Rand-Len	Random hamiltonian graph with large length range	131072	524288	cycle arcs: 1 intercycle arcs:uniformly distributed [0, 1000000]
Rand-P	Random hamiltonian graph with potential transformations	131072	524288	cycle arcs: 1 intercycle arcs: negative and positive values
Acyc-Pos	Acyclic graph with nonnegative arc lengths	131072	2097152	path arcs: 1 random arcs: uniformly distributed [0, 10000]
Acyc-Neg	Acyclic graph with negative arc lengths	131072	2097152	path arcs: 1 random arcs: uniformly distributed [− 10000, 0]
Acyc-P2N	Acyclic graph with variable fraction of negative arcs	16384	262144	path arcs: uniformly distributed, nonpositive random arcs: uniformly distributed, nonpositive

Table 3
Shortest-path running times

	BFP	GOR	GOR1	DIKH	DIKBD	PAPE	TWO-Q	THRESH
Grid-SSquare	231.33	7.18	42.5	16.28	8.9	4.33	4.48	7.02
Grid-SSquare-S	351.96	20.39	36.74	53.86	12.78			19.46
Grid-SWide	6	6.15	7.31	23.68	7.94	4.5	4.68	8.16
Grid-SLong		3.03	18.25	3.85	3.68	1.73	1.82	2.48
Grid-PHard		18.82	19.25	6.86	4.23			
Grid-NHard		29.38	19.34					
Rand-4	235.63	194.31	143.54	63.27	21.09	257.31	302.76	186.23
Rand-1:4	8.37	13.45	11.41	1.95	2.02	16.18	16.51	9.21
Rand-Len	35.09	26.27	11.99	6.23	2.23	41.64	43.64	27.25
Rand-P	23.18	19.2	15.65	147.31	46.32	25.01	28	58.88
Acyc-Pos	42.86	54.96	7.24	9.67	5.38	46.43	48.66	23.39
Acyc-Neg		8.65	8.51					
Acyc-P2N	313.34	0.43	0.43		1078.63			939.29

calculated the measures separately for the whole implementation and for a limited subset which contained only the core algorithmic parts.

For NPS/10, we calculated a separate measure for the whole shortest-path algorithm and heap code, while another measure after the debugging output and extra functions for interfacing with other parts of the software had been removed.

Similar to GGCL, we calculated the measures separately for all the needed code and for only the essential parts (dijkstra.h, best_first_search.h, mutable_queue.h, mutable_heap.h, and array_binary_tree.h).

For the calculation of the complexity measures, we used CMT++, which is a static analysis/metric tool for C and C++ code [11]. We executed the analysis using the file summary option (-s) which gave a figure for each file in the implementation.

4. Results

4.1. Shortest-path algorithms

We have collected the running times, from [12] in Table 3 (the largest problem instances only).

Table 4 contains the ranking of the algorithms based on their running times. When a result was not available, the ranking of 8 (the number of algorithms) was used.

The average ranking in Table 5 summarizes the main results. Average ranking is calculated using the rankings of Table 4. A further ranking based on the average ranking is derived and it is used as a basis for the rest of the study. Volume (V), cyclomatic number ($V(G)$), and lines-of-code (LOCpro) measures were calculated for each algorithm and the algorithms were also ranked with these measures.

Figs. 1–3 plot the complexity measures for different algorithms. The algorithms on the x-axis have been sorted based on their average ranking (the fastest algorithm on the left).

Table 4
Ranking based on execution time

	BFP	GOR	GOR1	DIKH	DIKBD	PAPE	TWO-Q	THRESH
Grid-SSquare	8	4	7	6	5	1	2	3
Grid-SSquare-S	6	3	4	5	1	8	8	2
Grid-SWide	3	4	5	8	6	1	2	7
Grid-SLong	8	4	7	6	5	1	2	3
Grid-PHard	8	3	4	2	1	8	8	8
Grid-NHard	8	2	1	8	8	8	8	8
Rand-4	6	5	3	2	1	7	8	4
Rand-1:4	3	6	5	1	2	7	8	4
Rand-Len	6	4	3	2	1	7	8	5
Rand-P	3	2	1	8	6	4	5	7
Acyc-Pos	5	8	2	3	1	6	7	4
Acyc-Neg	8	2	1	8	8	8	8	8
Acyc-P2N	3	1	1	8	5	8	8	4

Table 5
Ranking of shortest-path algorithms by different measures

	BFP	GOR	GOR1	DIKH	DIKBD	PAPE	TWO-Q	THRESH
Average ranking	5.77	3.69	3.38	5.15	3.85	5.69	6.31	5.15
Ranking by average ranking	7	2	1	4	3	6	8	4
V	1829	4116	4456	4034	8005	2196	2478	4070
Ranking by V	1	6	7	4	8	2	3	5
$V(G)$	8	22	25	15	35	9	12	17
Ranking by $V(G)$	1	6	7	4	8	2	3	5
LOCpro	78	147	160	140	242	87	100	149
Ranking by LOCpro	1	5	7	4	8	2	3	6

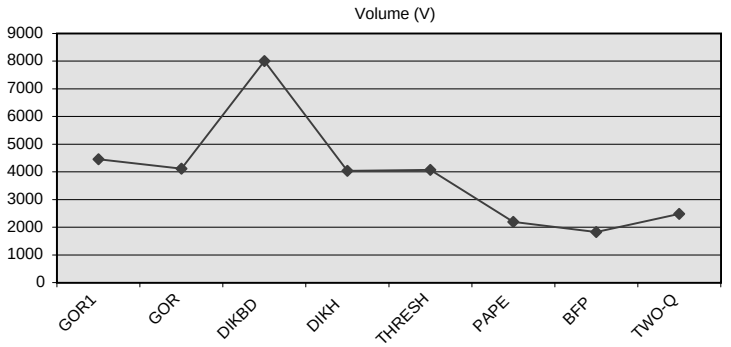

Fig. 1. Volume (V) of shortest-path algorithms.

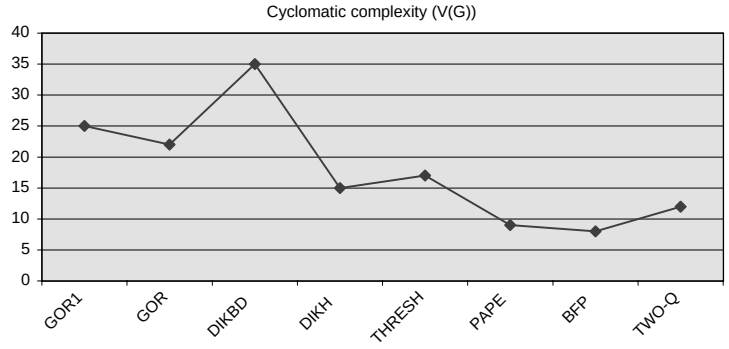

Fig. 2. Cyclomatic complexity ($V(G)$) of shortest-path algorithms.

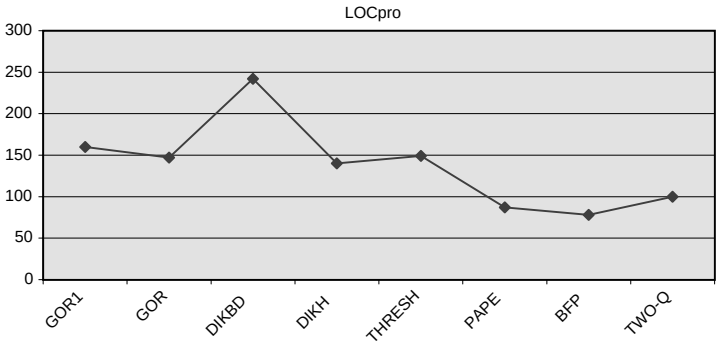

Fig. 3. Lines of code metrics of shortest-path algorithms.

Table 6
Correlation between speed ranking and complexity measures

	V	$V(G)$	LOCpro
Correlation	−0.706	−0.842	−0.799
Confidence	0.811	0.890	0.863

Table 6 calculates the correlation between the ranking based on the algorithm speed and the different complexity measures. The confidence line is the probability that the hypothesis that the measure depends on the ranking is valid.

Table 7 contains the correlations between different measures.

Table 8 compares implementation from different sources. The NPS/10 and GGCL lines show the measures for the whole implementation, the ‘Alg only’ line shows the measures only for the core algorithm.

Table 7
Correlation of different metrics

	V	$V(G)$	LOCpro
V	1.0000	0.9560	0.9950
$V(G)$		1.0000	0.9678
LOCpro			1.0000

Table 8
Comparison of different implementations of the same algorithm

	V	$V(G)$	LOCpro
NPS10	23700	55	438
Alg only	14507	30	301
GGCL	126983	223	3208
Alg only	17593	21	401
DIKH	4034	15	140

Table 9
Function metrics compared to recommended limits

File	Measured object	V	$V(G)$	LOCpro
bfp.c	bf( )	**1894**	8	**78**
dikbd.c	dikbd( )	**8005**	**35**	**242**
dikh.c	Init_heap( )	271	1	10
dikh.c	Heap_decrease_key( )	322	3	16
dikh.c	Insert_to_heap( )	118	1	7
dikh.c	Extract_min( )	917	8	34
dikh.c	dikh( )	**1145**	6	**46**
gor.c	gor( )	**4116**	**22**	**147**
pape.c	pape( )	**2196**	9	**87**
thresh.c	thresh( )	**4070**	**17**	**149**
two_q.c	two_q( )	**2478**	12	**100**
Recommended limit		1000	15	40

Table 9 shows the measures of each function. The items in bold exceeded the recommended limits.

5. Discussion

5.1. Complexities of the shortest-path algorithms

As shown in Table 5, there are considerable differences between the implementation complexities of the different algorithms. Depending on the used measure the most complex algorithm (DIKBD)

is 3–4 times as complex as the simplest one (BFP). The average complexity was about twice the complexity of the simplest algorithm.

These are significant differences, in particular, if the complexity measures, as experimental studies [7,10]. suggest, correlate strongly with debugging time, implementation time, number of errors and future maintenance effort. The benefit of a better algorithm has to be significant to justify a 2–4 times longer development time, higher bug probability, and higher maintanance cost in the future. In this setting, the simplier algorithms are very attractive especially when considering the shortage of skilled developers and the increasing pressure to reduce the time-to-market.

Since all the analyzed algorithms are based on the labeling method the complexity differences are caused by the use of advanced data structures and ingenious ordering of nodes to be scanned. By inspection, the implementation of the algorithms in the study looks very clear and clean suggesting that the complexity is a result of the complexity of the algorithm, not a result of a poor implementation.

Another interesting observation is that the results show that implementing algorithms is difficult. The data in Table 9 compare the complexities of the key functions to the recommended guidelines. All shortest-path functions exceed the recommended limits when lines-of-code and Halstead's volume measures were used. With the cyclomatic number measure this happened in 50% of the algorithms.

The results indicate that high implementation complexity is typical to the shortest-path algorithms. Our practical experience of algorithm implementation as part of application development corresponds with this finding. It seems likely that this observation also applies to other numeric and combinatorial algorithms. The practical implication of this is that particular attention should be paid to the correctness and testing of the algorithmic parts of the software.

Software reuse and the use of program libraries reduce the importance of implementation complexity. When the algorithms are already implemented, the user of the library does not have to worry about their implementation effort. What is important in this case is the effort to convert the problem to such a form that an existing software module is able to solve it. This conversion can be quite complex. It is sometimes reasonable to reimplement the algorithm to avoid a complex conversion from one presentation to another.

5.2. Algorithm efficiency vs. implementation complexity

Since the shortest-path problem can be solved to an optimum in polynomial time we can focus our analyses on two attributes: speed and implementation complexity. With NP-hard problems and heuristic solution procedures, even a third dimension, optimality, would have to be included in the comparison since the deviation from strict optimum varies with different heuristics and algorithms.

Figs. 1–3 plot the complexity as a function of the algorithm. Since the algorithms on the x-axis are sorted by their speed ranking, the chart effectively shows the dependency between the algorithm speed and implementation complexity. The decreasing curve shows that typically a faster algorithm requires a more complex implementation than a slower one (Dikbd is a major and TWO-Q a minor deviation from this). As can be seen from Table 6, the correlation between the speed ranking and the complexity measures is quite strong.

This result is intuitively appealing indicating that a gain in one dimension (increased speed) requires a loss in another dimension (more complex implementation). For instance, in terms of

speed, GOR1 is a better algorithm than DIKH. If only speed dimension is considered then there would not be much point to use DIKH at all. However, when the software complexity dimension is considered the situation changes. If a fast and simple implementation is required, then DIKH might be a better choice than GOR1. Selecting the algorithm to implement becomes explicitly a multicriteria optimization problem where there are multiple optimal choices depending on the importance of speed vs. simple implementation.

According to our experience, the implementation complexity dimension is very important in practice. The speed has to exceed a threshold to be adequate but after that extra speed is not a major benefit. Simpler implementation, on the other hand, is always useful by reducing the development time and effort, risk, probability of bugs, and maintenance effort.

With this data the DIKBD algorithm looks like a poor algorithm. It is inferior to another algorithm in both dimensions. The performance is worse than with DIKH while the implementation is more complex. As we use the average ranking as our criteria, this conclusion is a bit too strong. For certain problem types DIKBD is very fast. But if we are looking for a good general-purpose algorithm, DIKBD does not look like a good algorithm with this analysis.

A general conclusion is that for good algorithms the speed-complexity function should be a decreasing curve. An increase in the curve would indicate that the algorithm is both slower and more complex to implement compared to some other algorithm.

An interesting issue to speculate is why the curves in Figs. 1–3 are actually descending. Can it be that even if the implementation complexity is seldom explicitly measured, it, however, implicitly affects the choice of good algorithms. Developers are implicitly able to judge if the extra implementation effort of a more complex algorithm is useful. As a result the "natural selection" process eliminates those algorithms with poorer speed and complexity.

The analysis results naturally contain various sources of errors. The "typical" input used in the speed analysis may be quite far from the type of input that is needed in a particular task. Some algorithms are very good at certain specific problems, while others are perform reasonably well with a wide variety of different problem types. The implementation complexity measurement tries to cover in a single figure many different aspects: implementation effort, maintenance effort, probability of implementation errors, needed developer competence, etc. Naturally, the multitude of different aspects cannot be captured in a single figure.

5.3. Which measure to use

Table 7 confirms the common finding, e.g. [8,15], that the different measures have a strong correlation. This suggests that for algorithm comparison, the simple line-of-code measure would be equally useful as the more complicated performance measures.

However, the trouble with all the measurements is that it is possible to bias the results to favor a certain alternative. As noted in [2], a danger in empirical comparison of algorithms is that by careful implementation an algorithm can outperform its less well-implemented competitors. The same problem applies to the implementation complexity measurements. Especially, the line-of-code measure is very easily affected by using a different coding style e.g. by changing the code layout or variable naming. The other measures are probably less easy to mislead.

5.4. When are the comparisons feasible

Table 8 shows that the measures vary a lot when different implementations of the same algorithm are compared. It indicates that it is difficult to compare independent implementations with these measures. The selected programming style (procedural, algorithm only in DIKH; special purpose template library in NPS/10; general-purpose template library in GGCL) affects the complexity measures a lot. In the case of GGCL, the extra complexity coming mainly from the reuse of data structures and subalgorithm results in the complexity that is 7–10 times higher than the core algorithm. Also, in the case of NPS/10, there is 50% overhead coming from the necessary routines for data access and for debugging support.

It seems that a comparison is reasonable only when the algorithms have been implemented in exactly the same style, which typically can happen only when the algorithms have been implemented by the same group of developers. This is unfortunate since it does not allow a cross-comparison of the software complexity figures of different studies. Furthermore, it is not possible to state any universal implementation complexity values for different algorithms. Only relative comparisons are possible. To a lesser extent, the same argument also applies to the algorithm speed comparisons.

6. Conclusions

It can be argued that the implementation complexity is not a property of an algorithm. Complexity changes from implementation to implementation and depends on how carefully the algorithm is implemented. Therefore, it does not tell us how good the algorithms are per se, but about how good their implementations are. However, in practice, the algorithms cannot be strictly separated from their implementations. Any practical application using a certain algorithm must have it implemented. Since the theoretical analysis of algorithms is not adequate, it is common practice to empirically compare algorithms by running them with typical input and measuring the speed and possibly some other relevant performance attributes. If speed, which is dependent on the implementation, can be used to compare algorithms it is hard to see why implementation complexity would be any different.

We feel that using implementation complexity in empirical algorithm studies has at least three major benefits:

- People tend to focus on those aspects that are measured. As the algorithm studies currently focus on speed and, if appropriate, optimality, there is little incentive to study how a simpler algorithm would solve the same problem. Implementation complexity measurements would also bring into light those cases where a highly complex algorithm is able to provide only a slightly better performance.
- Simple implementation is one of the important practical issues of software. Measurement of implementation complexity might focus the algorithm research to aspects that are more relevant to the practitioner.
- Software complexity measures might help practitioners to choose, out of a large number of alternatives, the algorithms that best match their needs. Understanding the trade-off between implementation and performance would give a firmer basis to decision-making.

This study suggests that the implementation complexity measures offer a new insight into the comparison of algorithms. As it is easy to calculate the implementation complexity measures, it would be useful to include them in empirical algorithm comparisons.

Unfortunately, the implementation complexity measures did not provide any useful results when comparing different implementations of the same algorithm. It would be better if some measure would allow the comparisons even in this dimension. Other issues for further research are to apply the complexity measures to another group of algorithms, to clarify which complexity measure is the most suitable for algorithm comparison, and to investigate how well the complexity measures estimate algorithm implementation, maintenance effort and evolution potential.

Acknowledgements

I would like to thank Nokia Networks for their support and funding as well as Testwell Oy for the use of their CMT++ tool for the analysis. Thanks are also due to the anonymous referees for their comments and suggestions.

References

[1] Akkanen J, Nurminen JK. Case-study of the evolution of routing algorithms in a network planning tool. Journal of Systems and Software 2000;58:181–98.
[2] Sedgewick R. Algorithms in C++. Reading, MA: Addison-Wesley, 1995.
[3] Ahuja RK, Magnanti TL, Orlin JB. Network flows: theory, algorithms, and applications. Englewood Cliffs, NJ: Prentice-Hall, 1993.
[4] Ball M, Magazine M. The design and analysis of heuristics. Networks 1981;11:215–9.
[5] Halstead MH. Elements of software science. New York: Elsevier North-Holland, 1977.
[6] McCabe TJ. Complexity measure. IEEE Transactions on Software Engineering 1976;SE-5(4):308–20.
[7] Davis JS, LeBlanc RJ. A study of the applicability of complexity measures. IEEE Transactions on Software Engineering 1988;14(9):1366–72.
[8] Gill GK, Kemerer CF. Cyclomatic complexity density and software maintenance productivity. IEEE Transactions on Software Engineering 1991;17(2):1284–8.
[9] Hall GA, Munson JC. Software evolution: code delta and code churn. Journal of Systems and Software 2000;2000(54):111–8.
[10] Welker KD, Oman PW, Atkinson GG. Development and application of an automated source code maintainability index. Journal of Software Maintenance: Research and Practice 1997;9(3):127–59.
[11] CMT++. Computer software. Testwell, 1999.
[12] Cherkassky BV, Goldberg AV, Radzik T. Shortest paths algorithms: theory and experimental evaluation. Mathematical Programming 1996;73:129–74.
[13] NPS/10. Computer software. Nokia Networks, 1999.
[14] Lumsdaine A, Lee L-Q, Siek JG. Generic graph component Library. Computer software, 2000.
[15] Shepperd M. A critique of cyclomatic complexity as a software metric. Software Engineering Journal 1988;3(2):30–6.

Jukka K. Nurminen received the M.Sc and Tech.Lic. degrees from the Helsinki University of Technology. In 1986, he joined the Nokia Research Center, where he is working as a principal scientist on the area of network modeling and planning tools. His current interests are communication network modeling and algorithms as well as the model development and implementation processes.