Journal of Universal Computer Science, vol. 19, no. 3 (2013), 428-456
submitted: 31/3/12, accepted: 28/1/13, appeared: 1/2/13 © *J.UCS*

A Comparison of Five Programming Languages in a Graph Clustering Scenario

Martin Stein
(Karlsruhe Institute of Technology, Germany
mail@martinstein.net)

Andreas Geyer-Schulz
(Karlsruhe Institute of Technology, Germany
andreas.geyer-schulz@kit.edu)

Abstract The recent rise of social networks fuels the demand for efficient social web services, whose performance strongly benefits from the availability of fast graph clustering algorithms. Choosing a programming language heavily affects multiple aspects in this domain, such as runtime performance, code size, maintainability and tool support. Thus, an impartial comparison can provide valuable insights that are also useful for software development in general. This article investigates the languages C++, Java, C#, F# and Python (as well as its close variant Cython) in a controlled scenario: In each language, a graph clustering task is implemented and executed. The paper introduces the problem to be solved and gives an overview over the different characteristics of the languages. After a detailed description of the testing environment, we report runtime, memory and code size results and discuss them with respect to the characteristics mentioned before. The findings indicate C++ as the fastest language for the challenge at hand, but they also show that Java, C# and F# come close under some circumstances. Furthermore, it becomes clear that the amount of code can be significantly reduced with modern languages like Python or F#.

Key Words: Benchmark, Programming Languages, Language Performance, Graph Clustering, Modularity

Category: D.3, G.2.2, C.4

1 Introduction

The discussion of different programming languages often leads to heavy arguments based on strong opinions. Objective comparisons are hard to find and individual views are shaped by past experiences and personal investment in languages. An impartial evaluation of languages implies that each one should be investigated under conditions as similar as possible. Managing such a setup requires serious effort. Not surprisingly, recent studies are rare.

With the rise of the internet, the ongoing introduction of information technology in more areas of life and increasing amounts of data, processing on larger scales gains more and more importance. In the last decade numerous social networking sites have appeared and experienced considerable growth of usage. Hence, the analysis of social graph data gets more relevant and graph clustering plays a central role in this regard. Moreover, the internet as a fast-paced environment encourages the quick development and rollout of new methods.

In light of this, the choice of programming language has a significant impact. It affects the runtime performance and memory consumption of computations and thus indirectly the energy consumption in operations. It sets the frame for how much code needs to be written and shapes the level of complexity and abstraction. Consequently, it influences programmer productivity and development costs. Maintainability, a major aspect in the software industry, depends on the language used. The work flow before rollout also needs to be adapted: Can the code be executed directly in an interpreter? Or is a potentially time-consuming compile-and-link step necessary? The tool support and ecosystem vary between languages, for instance the availability and usefulness of integrated development environments (IDEs), debuggers or profilers. A language with a strong open source culture usually offers more freely usable libraries, whereas another one might allow for better business support and integration.

The goal of this article is to examine five programming languages (C++, Java, C#, F#, Python and its variant Cython) in a computationally intensive scenario and determine runtime performance, memory consumption and code size. Therefore, we specified a task that the first author implemented in each of the languages. The concrete development effort is intentionally not included in the investigation to allow the programmer to make up for differing levels of proficiency in the given languages. However, in software econometrics code size is seen as the variable with the largest impact on development effort [Boehm 1981]. The challenge centers around graph clustering algorithms, but in order to have a broader comparison, it also includes frequently encountered programming aspects like input parsing, a strategy pattern and the use of resizable lists and hash tables. The chosen languages differ in several ways: statically or dynamically typed, compiled or interpreted, object-oriented or functional. The article is not intended as an absolute statement about the languages above, but it provides an informative data point in a domain where recent, objective publications are scarce.

2 Related Work

There are few contemporary scientific publications dealing with programming language comparisons. A study by Lutz Prechelt in 2000 compared seven languages (C, C++, Java, Perl, Python, Rexx, Tcl) in various aspects [Prechelt 2000]. The author investigated runtime performance, memory requirements, code length, programming effort and reliability of 80 versions of the same program, written by 74 different authors.

Cesarini et al. compared implementations of IMAP client libraries in five languages (Erlang, C#, Java, Python, Ruby) [Cesarini et al. 2008]. The libraries differ in functionality, so the comparison includes *functionality of primitives* as a metric besides lines of code, memory requirements and execution time. In [Nyström et al. 2008], a C++/CORBA telecom application was re-engineered in Erlang and the two implementations were subsequently analyzed comparatively. The rewrite in Erlang led to a reduction in code size to less than a third of the original 15K lines of C++ code, while improving the system's robustness. Both publications focus on Erlang and support its validity in a telecommunications scenario.

A comparison of different software platforms for web development is presented by Prechelt [Prechelt 2011]. In the course of a publicly held 30 hour contest, 9 professional developer teams were given the task of implementing a web application based on the same specification. The languages Java, Perl and PHP were each used by three of the participating teams. The resulting programs were evaluated both on external, user-relevant and internal, developer-related criteria. Notably among the study's findings, the solution quality varied most between the Java teams and least within the PHP group. In 2011, a follow-up competition was held with 16 participating teams using five different languages (Java, Perl, PHP, Ruby, JavaScript). Again, the contestants implemented a web application and several characteristics of the delivered solutions were evaluated. The findings in [Stärk et al. 2012] show a high productivity of the Ruby teams, whereas the productivity within the Java and PHP groups was less uniform.

Simula Research Laboratory performed a private, controlled experiment [Anda et al. 2009]. After receiving bids from 35 companies, four of them were contracted to develop the same system. However, the study focuses on variability and reproducibility of software engineering in general. Java was a core requirement, so this study does not compare different languages. Though not a peer-reviewed publication, the website *The Computer Language Benchmarks Game* [ben 2011] presents implementations of 12 mini-benchmarks in roughly 24 programming languages, measuring runtime, memory requirements and code size for each. The *SciViews Benchmark* [sci 2012] evaluates the runtime of scientific computing software (Matlab, R, Octave etc.) for a set of standard problems. Its last update was in 2003.

Steve McConnell's *Code Complete* [McConnell 2004] shows the ratio of source code statements in higher-level languages to the equivalent number of statements in C. Relative execution times for some of the code samples in the book are presented, too. However, the statement ratio data is partly adapted from the above-mentioned work in [Prechelt 2000]. On the subject of controlled experiments in software engineering, which this paper can be considered part of, there is a general survey by Sjøberg et al. [Sjoeberg et al. 2005]. Their work investigates 103 experiments and provides a numerical summary of involved categories, participants and tasks performed. Their results confirm that comparisons of programming languages are only tackled by a small percentage of publications.

3 Programming Languages and Concepts

Besides obvious syntactical differences, programming languages can be distinguished based on several properties. Each one is relevant for the comparison at hand and described below.

3.1 Execution Strategy

From a machine point of view, source code can be executed in two manners: *compiled* or *interpreted*. A compiler translates the original code into machine code which is subsequently executed by the processor. The procedure usually starts with a front end which parses the written code, checks for syntax correctness, and generates an intermediate representation. The back end then translates that representation into assembly or machine code. In between, compilers often carry out optimizations, for instance inlining of functions, omitting unreachable code or pre-calculating constant values. C++, developed by Bjarne Stroustrup as an extension to the C language [Stroustrup 1993], belongs to this category.

An interpreter, on the other hand, executes code step-by-step. Each statement is parsed and translated into machine code on the fly which negatively impacts the runtime. This also holds when the original source code is transformed into an intermediate bytecode representation first. In that case, instead of operating on the source code directly, it is the bytecode that has to be interpreted. This is what happens in the Python reference implementation, referred to as *CPython*, which features a bytecode interpreter written in C.

C#, F# and Java also translate source code into an intermediate bytecode representation. However, their runtime environments additionally perform *Just-in-time* (JIT) compilation which means bytecode is compiled into specialized machine code at load time or during execution. In the latter case, the environment usually identifies frequent ("hot") code paths and generates optimized machine code for those parts. Thus, execution speed improves while the program is running. Java's reference virtual machine, the *HotSpot* JVM provided by Oracle Corporation, offers JIT-compilation of this kind [Paleczny et al. 2001]. C# and F#, running under Microsoft's *Common Language Runtime* (CLR), employ a similar procedure [Meijer and Gough 2000].

Cython is a Python extension which allows adding C-like type annotations and calling C functions. These code parts are then compiled to C code which in turn can be processed with a standard C-compiler. Thus, it is possible to speed up critical, slow Python code paths by replacing them with compiled Cython modules [Wilbers et al. 2009].

3.2 Type System

Cardelli distinguishes between typed and untyped languages [Cardelli 2004]. Typed languages perform *static* type checking whereas untyped languages can only use *dynamic* type checking. Another aspect of typing is the necessity or absence of explicit type declarations: We distinguish between *explicitly* and *implicitly* typed languages.

Static typing is used in C++, Java, F# and C# [Ecma International 2001]. However, in version 4.0 C# introduces a `dynamic` keyword that can be used to bypass static checks on variables. Python is dynamically checked. Cython is of a hybrid nature: Code sections can be enhanced with C-like type declarations and thus become statically typed. In dynamically checked languages, the safety of operations is verified at runtime which usually increases execution time. Static typing can be exploited for efficient machine code generation which improves runtime performance. Dynamic checking allows more flexibility, but on the downside, some errors are only caught during program execution.

Of the languages considered in this paper, explicit type declarations are required for C++, Java and C#. F# is partially implicitly typed. For instance, the statement `let value = 5` enables the compiler to infer that `value` is of type integer. Python is dynamically checked without explicit type declarations and Cython is, again, in between. The absence of explicit type declarations leads to shorter code.

3.3 Programming Paradigm

Programming languages provide different concepts for the programmer to structure the code. Currently, the most widely used paradigm is *object-oriented programming* (OOP). It emphasizes the aspects of encapsulation, inheritance and polymorphism. An object encapsulates state as private data and the means to transform it via a public interface. It inherits behavior from parent definitions and polymorphism enables the interchangeable usage of different types, as long as they conform to the same interface. C++, Java and C# are considered as OOP languages.

Functional programming has its origin in mathematics. The basic building block for defining behavior is the concept of mathematical functions. They can be assigned to variables (*first-class functions*) and have the role of input or return parameters of other functions (*higher-order functions*). In *pure* functional languages, a function call is without side effects and its result only depends on its input. Mutable state is completely disallowed and recursion replaces traditional looping constructs. In practice, however, the term "functional programming" is loosely defined and the extent to which languages cover the aspects above varies greatly.

None of the languages in the comparison are regarded as pure functional languages (like Haskell). Still, F# is generally seen as a functional language, even though it also offers support for OOP and allows mutable state. Python and Cython cannot clearly be placed in either the object-oriented or the functional category, since they contain several aspects from both.

3.4 Generics

In statically typed languages, there is often the need to write code that can deal with multiple types and is nonetheless type-safe. This applies especially to container classes like linked lists, resizable lists or hash tables. It should be possible to use one linked list for `int` and another for `double` values without duplicating the code for the list implementation. Programming languages with generic types offer the possibility to write functions or classes with general type parameters, like a hash table mapping keys K to values V. These code parts can then be used with concrete type specializations, for example a hash table mapping strings to integer values.

In the case of C++ and its *templates*, the type-specialized version is generated at compile-time. Furthermore, templates enable the powerful but also challenging technique of template metaprogramming which allows code specializations that are considered Turing complete [Veldhuizen 2003]. C# and F# both run in the CLR (Common Language Runtime) environment, where the specialized code is generated at runtime. Since generics in the CLR are restricted to type parameters, metaprogramming as in C++ is not possible. Java, on the other hand, does not generate specialized code versions and performs *type erasure* instead: The compiler replaces generic types with raw types that wrap primitive types like integers or doubles in an object. Consequently, Java's generics offer type-safety at compile-time but incur additional boxing and space overhead for primitives at runtime. Python is dynamically typed, hence the concept of statically type-safe generics does not apply. Table 1 sums up the above characterizations.

	Execution strategy (3.1)	Type system (3.2)	Main paradigm (3.3)	Generics (3.4)
C++	compiled	static	OOP	compile-time
Java	JIT-compiled	static	OOP	type-erasure
C#	JIT-compiled	static	OOP	runtime
F#	JIT-compiled	static	functional	runtime
Python	interpreted	dynamic	both	n/a
Cython	hybrid	hybrid	both	compile-time

Table 1: Overview of programming language characteristics

4 The Setup of the Comparison

The goal of this paper is to deliver a comparison of programming languages for the task of graph clustering, which is highly relevant for real-world social networking services. This leads to several decisions which are outlined below.

4.1 Scope and Complexity

Micro-benchmarks like the calculation of Fibonacci numbers are often used for quick language comparisons, but they are not indicative of real-world usage: First, they evaluate only a single or very few aspects of each language. Second, the code in these benchmarks is often heavily tuned towards runtime performance without considering common language idioms and best practices from software engineering. Third, due to their small problem scope, they completely ignore a fundamental challenge of software development: dealing with complexity in growing code bases.

The closer the benchmark scope is to a given scale, the more indicative it is for applications of that size. Ideally, we would have chosen a task that demands at least several thousand lines of code. However, it is a major undertaking to deliver solutions of that extent for five programming languages and, therefore, out of the scope of this paper. Instead, we investigate a problem that is manageable and yet representative for larger applications.

4.2 The Programming Domain

The comparison task is defined in the field of *graph clustering*. A graph (in some disciplines also referred to as *network*) can represent a multitude of models. Social networks are a prime example: Each person is a *vertex* (or *node*) and the friendship between two persons corresponds to a connecting edge. Sales data on online shopping sites also lends itself well to a network representation: The articles are nodes and the copresence of two items in the same purchase forms an edge. The Web with its documents and links is represented by a graph, but also more unusual data like the interaction between proteins in a chemical reaction.

A frequent challenge is to identify *clusters*: groups of nodes that are more densely connected within than to the rest of the graph. Finding such a partition is also referred to as *community detection*. The results can help to identify circles of friends in social networks or groups of complementary products to recommend to a customer. Fig. 1 shows an example of a graph with 15 nodes partitioned into 3 groups.

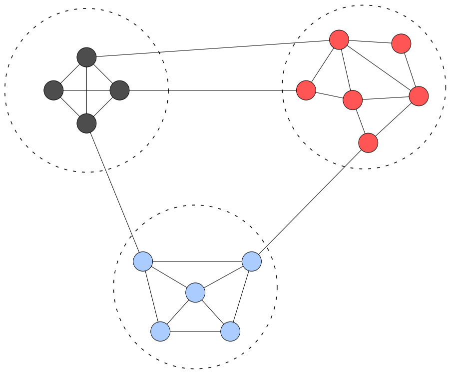

Figure 1: A graph containing three clusters

4.3 Modularity

Intuitively, a good clustering contains a high number of intra-cluster edges while keeping the weight of connections between clusters low. However, simply assigning all vertices to the same partition satisfies this requirement without providing any information. The *modularity* measure Q proposed by Newman et al. [Newman and Girvan 2004] improves upon this idea. Let $\zeta = \{C_1, C_2, \ldots, C_k\}$ be a a non-overlapping clustering of a graph into a set of k nonempty communities, e_{ij} the fraction of edges between nodes in partition C_i and C_j and a_i the fraction of edges that are connected to vertices of C_i. Then, Q is defined as follows:

$$Q = \sum_i (e_{ii} - a_i^2) \tag{1}$$

The measure compares the fraction of edge weights within clusters $\sum_i e_{ii}$ to the expected fraction of edge weights $\sum_i a_i^2$. The higher the difference, the more the chosen partitioning positively differs from what would be expected in a random graph with the same edge distribution. The trivial clustering mentioned above – all nodes in one partition – yields a value of $Q = 0.0$, whereas the theoretical maximum is $Q = 1.0$. In practice, values of $Q > 0.3$ indicate significant community structure.

4.4 Task Description

Algorithm 1: High-level specification of the clustering task

Input: Undirected graph in Pajek format
Output: List of clusters

```
graph ← new Graph                                  /* adjacency list */
for line in section *edges do
    graph.add_edges_from(line)

algorithm ← select_algo(command-line)              /* strategy pattern */
joinpath ← algorithm(graph)                        /* runtime measured */

max_Q, index ← get_max_join(joinpath)
clusters ← join_up_to(joinpath, max_Q, index)
for cluster in clusters do
    print(cluster)
```

Based on this scenario, the task is to read a graph stored in a textual file in the *Pajek*-format [Nooy et al. 2004] and represent it in memory. An adjacency matrix would require $O(n^2)$ space for n nodes and is, therefore, unsuitable for large and sparse graphs (relevant datasets usually have a density far below 0.01, see Table 2). Hence, weighted adjacency lists were chosen as a more efficient memory representation [Cormen et al. 2009]. Then, depending on the choice of command-line arguments, one of two clustering algorithms (a greedy and a randomized greedy algorithm explained in the next paragraph) must be executed. Finally, the generated partitions and the corresponding cluster quality will be stored in a result file. The procedure is described in Alg. 1.

The core of the comparison are two clustering algorithms: Newman's *greedy* procedure [Newman 2004] and a *randomized greedy* approach [Ovelgönne et al. 2010]. Both methods are agglomerative: At first, each node is assigned to its own cluster. Then, the clusters are joined iteratively, which results in a join path. For each step, the value of Q is calculated. Along this path, the clustering with the highest modularity is chosen as the result. In the pure greedy version, in each step the change of modularity ΔQ is determined for all possible joins and the one with the highest ΔQ is selected. On sparse graphs, the algorithm's running time is $O(n^2)$ for n nodes. In the randomized greedy method, instead of analyzing all potential joins, a small number of clusters k is randomly selected in each step. Only the joins between those communities and their neighbors are calculated and the best one is chosen. For constant values of k ($k \leq 2$ is recommended in [Ovelgönne et al. 2010] and has been used in the DIMACS competition [Ovelgönne and Geyer-Schulz 2013]) the runtime decreases to $O(n \log n)$ on average in sparse networks. Since each run is significantly faster, several runs l are performed in order to select the one with the highest Q. The randomized selection usually leads to a more balanced growth of clusters [Ovelgönne and Geyer-Schulz 2012]. Therefore, it is not as susceptible to local maxima as the greedy algorithm and gives considerably better results in most cases. Alg. 2 shows the algorithm in more detail. For a more extensive description of the pure greedy version we refer to its original publication ([Newman 2004]).

The data structure for representing the clusters and merging them plays a crucial role in the procedure. It needs to store sparse edge weights between all clusters and must support efficient joins between partitions. A normal matrix cannot be used because of its $O(n^2)$ space requirements. An adjacency list is inadequate since the summation of edge weigths during the merges would require iterating over all neighboring entries for the column-wise join. Therefore, the clusters will be represented as an array of hash tables that map integer keys to double values. Each hash table in the array corresponds to one cluster. The integer keys specify the neighboring cluster indexes and the double values hold the cumulative edge weight between these communities. Sparsity is achieved by storing only key-value pairs for connected clusters, where $e_{ij} > 0$ and omitting key-value pairs where $e_{ij} = 0$. The clustering will be performed on multiple graphs, the largest of which consists of more than 300.000 nodes and 1.000.000 edges. Hence, the given problem covers several aspects:

1. Deal with file I/O and parse a custom data format.

Algorithm 2: Randomized greedy modularity clustering

Input: *graph* - in adjacency list representation
Output: *joinpath*

```
e ← new Matrix(graph)                         /* array of hash tables */
a ← new Array(graph.size)
for row i in e do
    a_i ← row_sum(e_i)
for i in 1 to size(e) do
    max_ΔQ ← −∞
    selected ← k_random_clusters(e)
    for cluster c in selected do
        for neighbors n of c do
            ΔQ ← 2(e_cn − a_c a_n)
            if ΔQ > max_ΔQ then
                max_ΔQ ← ΔQ
                best join ← (c, n)
    perform_join(best join)
    joinpath ← joinpath + best join
return joinpath
```

2. Handle potentially large datasets.
3. Implement a *strategy pattern* (the clustering method is chosen at runtime).
4. Use a collection/container library (resizable lists for the adjacency list representation and hash tables for the clustering).

5 Evaluation

In order to compare the different languages in the setup described above, the task needed to be implemented and then measured on several datasets using clearly defined metrics. This section covers the relevant aspects and discusses the results gathered from the experiments.

5.1 Comparability

Before the evaluation, the actual code needs to be written in a way that ensures high comparability of the results. Software metrics like program size and runtime performance are influenced by several factors, for instance domain experience, availability of software tools or time constraints during development [Boehm et al. 2000, Kennedy et al. 2004]. Individual programmer capability affects these measures in particular [Boehm et al. 2000], resulting in differences of up to an order of magnitude [Sackman et al. 1968]. Hence, assigning the task in the different languages to different persons would be problematic. Instead, the first author, who co-authored the original randomized greedy algorithm, wrote the code for all versions. He had previous knowledge in all of the languages and two years of industry experience each in both Java and Python. He had the least amount of experience in C++. This was compensated by the implementation history of the original randomized Greedy algorithm which required a Java implementation for the *WeKnowIt* European Union research project (done by the first author) and a competitive C++ implementation for scientific publication [Ovelgönne and Geyer-Schulz 2010] (ported from the Java version by Michael Ovelgönne). For this paper, the C++ version was rewritten from scratch based on one year's experience with both previous implementations. That same experience was applied in the writing of all other implementations.

Since the task is algorithm-centric, this is especially important: Without deep domain knowledge of the underlying steps and the algorithmic complexity it would be difficult to produce comparable code. With all implementations done by the first author, we ensure the same level of algorithmic background knowledge across all languages.

Another aspect that positively impacts comparability is the strict specification of the task itself: The input format of the data is clearly defined, the two algorithms are described in detail in their corresponding publications and the output format has to be identical. Furthermore, the agglomerative clustering approach in combination with the size of the test networks dictated the exact memory representation with arrays of hash tables (see section 4.4).

Finally, data structures like resizable arrays and hash tables are central to the comparison. Each of the languages already offers the required container libraries, for instance the *STL* (Standard Template Library) in C++ or the *System.Collections* namespace in C#. These libraries are heavily in use in real life scenarios, so using them in the comparison increases the relevance of the setup. Furthermore, their usage further reduces variation between the implementations because it shifts functionality into the language libraries.

The programming author spent extra-time on researching common idioms and best practices to ensure a high code quality and comparability of the implementations in all languages. Consequently, we did not place any time constraints on the coding task and did not evaluate the time to write the code, since that would penalize research and improvements in code quality. The development time for the implementations in the various languages was about four months in total. Design variants were constrained by the decision to use standard libraries for data structures and the precise specification of the algorithm. However, improvements in one language were transferred to the other implementations when possible.

5.2 Metrics

Programming languages significantly differ in their expressiveness. Whereas one language might solve a problem with a terse one-liner, another might require a large amount of code. Program size is and has been widely considered as the main cost driver of software engineering projects [Boehm 1981, Boehm et al. 2000, Albrecht and Gaffney 1983]. Furthermore, a strong correlation between program size and program complexity has been established [Jay et al. 2009]. Source lines of code (SLOC) is often used in this regard, but its meaning must be stated exactly in order to deliver meaningful numbers. SLOC can be given as physical lines of code, which usually refers to a count of all lines in a program. However, the definitions vary in whether comment lines should be included or ignored. Logical lines of code specify the number of statements in the code, but there is no consensus on what counts as a statement. A general framework for a more detailed definition is available from the Software Engineering Institute [Park 1992], but an exact specification for all of the languages in the comparison is not available.

We will use a metric which is often named *effective lines of code* (eLOC). It measures the number of lines excluding blank lines, comment lines and lines with braces, brackets or parentheses only. Hence, eLOC is language independent and does not unnecessarily penalize good coding style involving comments or blank lines. It is especially suitable for this comparison since differences in brace placement (K&R style preferred in Java, Allmann style in C#) or the absence of braces as block delimiters (Python, F#) do not affect the results.

Type inference as in F# or dynamic typing in Python lead to shorter code, but the size reduction is not automatically reflected in smaller eLOC numbers. Both the Java statement

```
Map<String, Integer> wordCount = new HashMap<String, Integer>();
```

and the corresponding Python statement

```
word_count = {}
```

result in 1 eLOC in spite of the strong differences. Therefore, we also define the measure *effective bytes* (eBytes) as the size of the source code in bytes after excluding comments and collapsing consecutive white spaces to a single one. Like eLOC, the eBytes metric is robust with respect to commenting and different brace or indentation styles. It is, however, affected by length differences in function, class or variable names which is why equally descriptive names were used in the code of every language.

Besides code size, the performance of the solutions is of interest. In a real-world scenario, the graph data might be stored in a database, as textual file or the network might already be present in memory. Measuring the time to read and build up the graph, therefore, makes little sense. The critical part of the task is the core clustering algorithm, so we determine its running time as wall clock time and total CPU time. Since the clustering is CPU-bound, it is to be expected that both measures give similar results.

The memory consumption is also highly relevant for the comparison, especially given the differences in generics between the languages. Thus, we measure the peak memory usage for the complete task. The value was determined by spawning a separate process for the execution and querying the process API of the operating system for the required memory.

5.3 Datasets

In order to get results across a wide range of data, we have employed 6 real-world datasets of increasing order of magnitude. While the Pajek file format used in the comparison handles both directed and undirected networks, the applied clustering algorithms are targeted towards simple, undirected graphs. Therefore, directed networks were symmetrized and multiple edges and loops were removed where necessary.

The smallest dataset is the *Karate* graph: It consists of 34 nodes, the members of a karate club, whose friendship ties are represented as 78 edges [Zachary 1977]. The network has often been used as a basic indicator for the quality of clustering algorithms. The second dataset models 115 US college *football* teams as vertices and the games between them as edges [Girvan and Newman 2002]. Multiple connections between teams were collapsed to single ones, resulting in 613 edges total.

The next dataset depicts the *email* interchanges between 1133 persons at the University Rovira i Virgili and has 5451 undirected edges [Guimerà et al. 2003]. Boguñá et al. analyzed the web of trust of users of the *PGP* algorithm in its state of July 2001 [Boguñá et al. 2004]. An undirected link between two users is formed when they have mutually signed their keys. The released dataset is the largest connected component of this graph. After clearing multiple connections between vertices, it comprises 10680 nodes and 24316 edges.

Newman published a dataset based on the collaboration of authors that posted preprints to the Condensed Matter-archive at arXiv.org [Newman 2001]. Scientists coauthoring a publication are represented as nodes linked by an undirected edge. The largest component of this graph (from now on referred to as *condmat*) encompasses 27519 vertices and 116181 edges. The final dataset consists of a set of web pages within the domain `nd.edu` and the links between them, which corresponds to a directed network [Albert et al. 1999]. Loops and multiple connections were removed before transforming the remaining edges to undirected ones, which leaves 325729 vertices and 1090108 edges (referred to as *www* below). Table 2 shows the summarized information for the datasets.

5.4 Testing Environment

The experiments were performed on an Intel Core™2 Duo P8600 with 2.40 GHz and 4 GB RAM running Windows 7 with service pack 1. The power settings for the processor

	Vertices	Edges	Density	Reference
karate	34	78	0.13904	[Zachary 1977]
football	115	613	0.09352	[Girvan and Newman 2002]
email	1133	5451	0.00850	[Guimerà et al. 2003]
pgp	10680	24316	0.00043	[Boguñá et al. 2004]
condmat	27519	116181	0.00031	[Newman 2001]
www	325729	1090108	0.00002	[Albert et al. 1999]

Table 2: Dataset overview

were set to highest performance mode. Non-essential background tasks were terminated and the machine was disconnected from LAN/WLAN. All language executables were targeted at 32-bit mode with optimizations turned on.

The C++ code was compiled with GCC 4.5.2 based on MinGW, using the compiler options `-O3` and `-fomit-frame-pointer`. The Java code ran on Oracle's HotSpot JVM with version number 1.6.0_25. Command line options were `-server` (because of more advanced optimization techniques in the server VM) and `-Xmx1550m`. The C# code, based on language version 4.0, was compiled under the .NET environment 4.0.30319.225 with `/optimize+`. The F# executable was generated with the Microsoft F# 2.0 compiler at version 4.0.30319.1, using the same .NET environment as C#. Tail recursion elimination was enabled (on) and `/optimize+` was turned on. The Python interpreter for the comparison was the standard CPython release 2.7.1. For Cython 0.13, the same compiler as C++ (GCC 4.5.2) was used.

In the .NET environment, the garbage collection (GC) can operate concurrently on another thread or it can be set to run single-threadedly. The official documentation from Microsoft states that multi-threaded mode improves latency in user interactions but decreases performance [con 2013]. Our tests with both options indicate that this does not hold for small graphs. However, on larger datasets single-threaded operation is mostly superior regarding runtime and memory requirements as shown in Tables A.5 and A.6. Since the comparison concentrates on performance and not UI latency, the C# and F# code was configured for single-threaded mode with `<gcConcurrent enabled="false"/>`.

The Java virtual machine offers multiple GC options [hot 2013]. We are interested in those focused on performance, which leaves `-XX:+UseSerialGC`, `-XX:+UseParallelGC` and `-XX:+UseParallelOldGC`. The first one specifies single-threaded operation whereas the latter two enable parallel collection. The execution of the clustering task for all options made clear that there is no single best choice. The runtime results in Table A.7 and Table A.8 show that the fastest option depends on the network size and the algorithm. However, single-threaded operation requires significantly less memory on the larger datasets (see the measurements in Table A.9 and

Table A.10). Hence, the further discussion refers to the option `-XX:+UseSerialGC`. This also improves the comparability of the results, since it means that all languages were tested in single-threaded mode.

5.5 Runtime Results

The randomized greedy clustering was executed and measured 100 times for every language and dataset. The pure greedy algorithm is deterministic as long as the hashing scheme for the sparse representation is deterministic. It is also orders of magnitude slower on larger networks due to its $O(n^2)$ growth. Therefore, this algorithm was executed 50 times for the five smaller graphs and 3 times for the large www network. For Python and Cython, the greedy experiments on the largest graph were omitted due to time constraints (a single run can be expected to take about 20 hours).

Generally, wall clock and CPU times were very close to each other, which was to be expected given the low background load of the system. However, the resolution for measuring CPU times for a given process in Windows 7 is only 15.6ms [tim 2010]. The wall clock measurements feature a higher resolution of 1ms or better. Hence, we report wall clock times for the two smallest datasets (karate and football), where timings below 15.6ms occurred. For the other networks (email, pgp, condmat and www) the results refer to CPU times. The arithmetic means for the randomized greedy runtimes are shown in Table A.1, the pure greedy ones in Table A.2.

The C++ implementation emerges as the fastest one for both algorithms and on all datasets. Figure 2 displays its runtimes on a logarithmic scale over the size of the six clustered networks. On the small karate and football graphs, the pure greedy algorithm finishes more quickly. With increasing graph size, it gets clear that the randomized greedy version grows more slowly, as we would expect from its $O(n \log n)$ asymptotic growth rate. The difference reaches a factor of 10^3 on the www network (10.6s for randomized greedy versus 11.6×10^3s for greedy).

The runtimes of the other languages in relation to C++ as baseline are presented in Figure 3 and Figure 4, over an x-axis with the number of edges of the corresponding graph. A closer look reveals several facts: First, it takes time before the JIT-compilation for Java, C# and F# pays off. On very small datasets which only require short computations, the overhead for the virtual machines surpasses the benefits gained from JIT-compilation. Python and Cython outperform Java, C# and F# on the karate and football networks. The larger the graphs and the longer the calculations, the more significant the speed-ups from JIT-compilation: For the randomized clustering on the www dataset, both C# and Java catch up to C++ to within a factor of 2.

Second, the ratio between Python, Cython and C++ stays in a relatively narrow interval within one algorithm. For randomized greedy, Python is 6.4–10.0 times slower than C++ on all networks; Cython 5.4–8.8 times. The hybrid code of the Cython implementation takes 7%–21% less time than the Python approach. There are reports of Cython coming close to C++ performance under certain conditions, usually involving

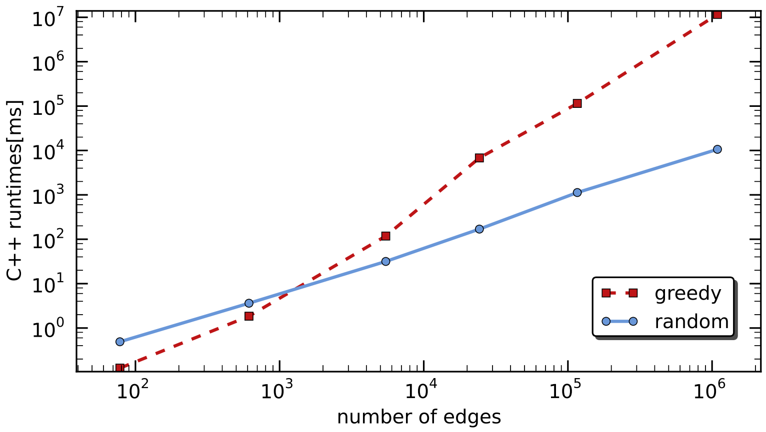

Figure 2: Runtimes[ms] for C++, arithmetic means

mostly numerical computations. However, the tested clustering algorithms heavily deal with hash data structures, which explains why Cython only yields a small benefit over Python here.

Third, in the randomized version F# is consistently 1.6–2.1 times slower than C#. In the pure greedy method, F# gets closer to the numbers of C# and even outperforms it on the pgp and www dataset. On all but the smallest graphs, Java lags behind C#, but the gap narrows towards larger networks. Interestingly, both .NET implementations perform remarkably well on the condmat dataset and even on par with C++ in the pure greedy test.

Fourth, the differences between the randomized and the pure greedy results indicate where JIT-compilation and Cython's hybrid approach are especially effective. The randomized algorithm only evaluates few joins in each iteration and then executes the best one. The two central parts of the algorithm – calculating possible merges and performing them – are alternated quickly. The greedy procedure, on the other hand, determines the modularity change ΔQ for all possible joins and only then executes the merge. Hence, the core of the algorithm is dominated by the repeated calculation of ΔQ values. JIT-compilation usually focuses particularly on optimizing hot, frequently taken code paths. It is therefore well suited to the dominating inner loop of the greedy algorithm. Consequently, C#, F# and Java come closer to C++ performance. Cython's improvement over Python is also more significant: 25%–33% runtime reduction compared to the 7%–21% in the randomized greedy method.

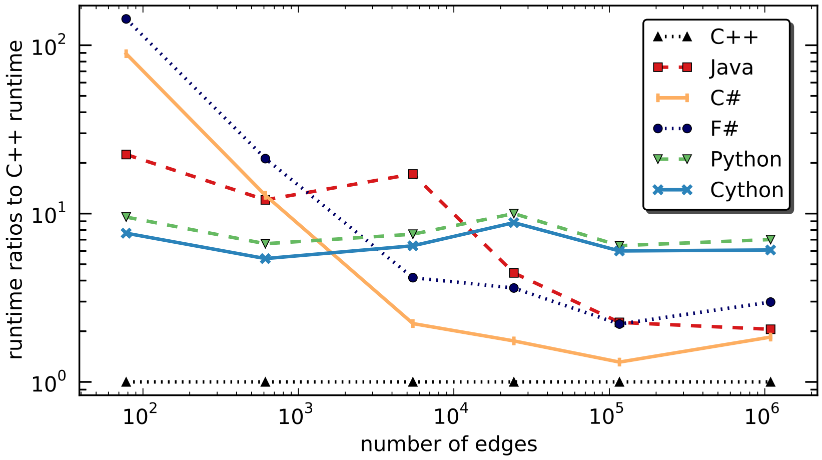

Figure 3: Runtime ratios to C++ times for Randomized Greedy

5.6 Memory Results

For both clustering algorithms, the peak memory consumption for the task was measured. The resulting arithmetic mean values after 100 randomized greedy runs are shown in Table A.3, those after 50 greedy runs in Table A.4. The standard deviations in the pure greedy approach are lower due to its deterministic behavior but otherwise it yields similar findings, so it is omitted in the discussion below. The ratios of the randomized greedy memory requirements in relation to the C++ values are displayed in Figure 5, with x denoting the graph size. The results for the www network are of special interest, since they demonstrate how well each language copes with large datasets.

With 203.9MB (208842KB), C++ takes the least amount of RAM on the www network. Python requires 1.80 times as much, followed by C# with a factor of 2.00. F# uses slightly more memory than C# and notably, Cython requires 1.24 times as much as Python. The latter can be explained with Cython's hybrid nature: Some data needs to be present both in Python and in compiled space, which evidently raises the RAM consumption.

Java is least memory efficient and uses 3.33 times as much as C++ on the www dataset. Its type-erasure generics play a role in this regard: Primitive values like integers or doubles are internally stored as objects in its generic classes. The memory consumption rises accordingly. In order to further investigate this aspect, we implemented a version of the randomized greedy algorithm that uses specialized hash tables from the high performance *Colt* library (developed at CERN, version 1.2.0) instead of the standard generic `HashMap`. Colt's `OpenIntDoubleHashMap` explicitely stores

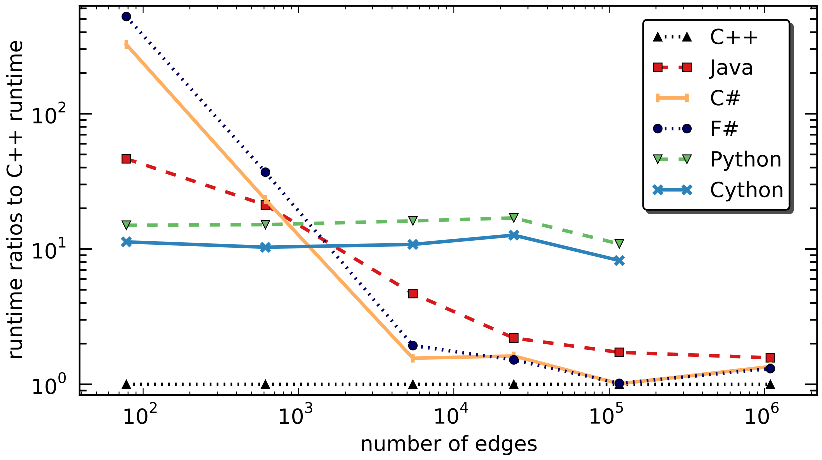

Figure 4: Runtime ratios to C++ times for Newman's Greedy

primitive integer and double values instead of objects. After 100 test runs, there was no noteworthy difference in the memory consumption on the five smaller datasets compared to the generic Java version. However, on the large www graph, the memory usage fell from 695.92MB in the generic approach down to 528.40MB in the Colt version, which clearly demonstrates the higher requirements of Java's generic data structure.

Furthermore, the requirements of the other languages are low for small datasets and only increase with growing network sizes. Java's virtual machine, on the other hand, occupies 90.69MB for the karate graph and more than 104MB for the football, email, pgp and condmat datasets. Thus, for small problem sizes, the memory consumption of the JVM itself far exceeds that of the data representation. This is consistent with the results of the Colt version, where improvements were only noticeable on the largest dataset. The trivial "Hello World!" program can be used as further evidence: It used 24.67MB of RAM on the test machine in `-server` mode, even when configured for low memory with `-Xms2m -Xmx2m`.

5.7 Code Size Results

For all implementations, the code size metrics were determined for the complete source code written by the programmer. Unit tests and linked code from libraries like the C++ *Standard Template Library* (STL) were not included. The results are presented in Table 3 and visualized in Figure 6. The order between the languages is almost the exact opposite of the runtime performance order: The faster the implementation, the bigger its code base. Python has the smallest code size of 237 eLOC. C++ at the other end

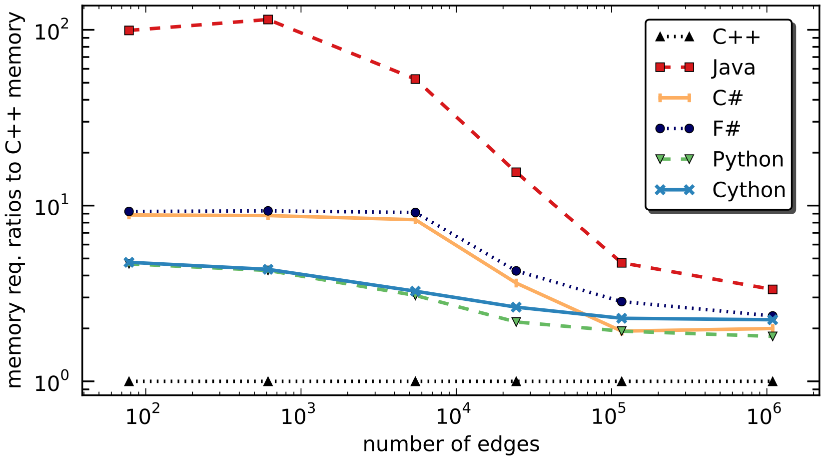

Figure 5: Memory requirement ratios to C++ memory for Randomized Greedy

uses 2.5 times as many lines. Cython speeds up Python, but requires marginally more code. F# and its functional paradigm enable a code size of 275 eLOC – 27% less than C# – at the cost of some performance. However, there is one exception: Among the JIT-compiled languages, C# and F# are on average faster than Java even though their code size is smaller.

	C++	Java	C#	F#	Python	Cython
eLOC	587	453	376	275	237	243
eBytes	21964	15732	14230	10655	8426	8655
$\frac{eBytes}{eLOC}$	37.4	34.7	37.8	38.7	35.6	35.6

Table 3: Code sizes for each implementation

The ratio of bytes per line of code is relatively similar for all compared languages. The Java implementation stands out with only 34.7 bytes per line. F# has the densest code base with 38.7 characters for every line. Python and Cython are at the lower end with 35.6 eBytes per eLOC, which can be explained with the general lack of type declarations due to dynamic typing.

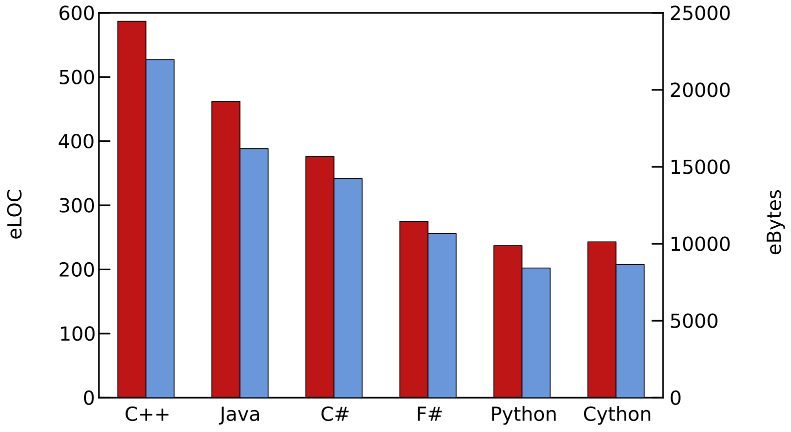

Figure 6: Code sizes in eLOC(left) and eBytes(right)

5.8 Threats to Validity

Having all the code written by the same person does not guarantee identical quality in all the implementations. Inevitably, the level of experience in the investigated languages varies. However, the implementations are centered around a computationally intensive task where a deep understanding of the underlying algorithms is paramount. With the implementations all done by the first author – who co-authored one of the two algorithms – we reduce the risk of variations in the core algorithms compared to implementations by different authors. Furthermore, extensive studies by Boehm et al. show that programming productivity is influenced far more by programmer capability than language experience [Boehm et al. 2000]. Hence, the single developer approach can be assumed to yield better comparability than assigning the task to different persons.

Another point of criticism might be raised against the general programming capability of the first author. The randomized greedy algorithm (in a C++ implementation of Michael Ovelgönne) was part of the winning submission of the 10th DIMACS implementation challenge for the modularity clustering task [Ovelgönne and Geyer-Schulz 2013] (compared against the solvers of eight other international research groups). In an internal benchmark of our research group the C++ implementation used in this paper compared favorably with the award-winning implementation. With this version as baseline, Fig. 4 shows that the performance of the versions in Java, C#, and F# converges towards the performance of the C++ implementation, which suggests that their implementation is of comparable quality. The Python and Cython performance relative to C++ is almost constant over the range of the tested networks. The slower runtime of both versions is consistent with what one would expect given their interpreted/hybrid execution strategy and with other reports both online and in literature (see section 2).

5.9 Other Observations

The section above reports the numerical results of the comparison. Two other aspects not reflected in the previous data should also be mentioned: First, the code size for Cython is hardly larger than for Python, but the increase in project complexity is considerable: The implementation additionally depends on a C-compiler and instead of executing code directly, an intermediate build-/compile-step is required. Second, both algorithms heavily use hash tables in order to deal with sparse cluster data. C++, Java, C# and F# allow fine tuning of the hash tables by specifying the initial capacity. Adjusting this value significantly improved runtimes and memory consumption for C++ and particularly for Java, where runtimes increased by a factor of 1.4 for badly chosen initial capacities. The final versions measured in the experiments use initial capacities of 4 both for C++ and Java. These values were obtained via bisection search on the www dataset with the randomized greedy algorithm.

Computationally, the comparison task is characterized by its handling of large, sparsely represented data. The use of hash tables and resizable lists plays a crucial role and their contents are subject to frequent changes due to the repeated joins between clusters. Especially on large datasets like the www network the time spent in garbage collection is significant. Accordingly, we can expect results along the same lines for programs with matching characteristics: heavy computation related to large, sparse datasets respresented in similar data structures. Purely numerical calculations like the computation of π will likely yield different results whereas algorithms that operate on graph structures, trees or linked lists are probably closer to the comparison at hand.

6 Future Work and Conclusion

This article presents a comparison of five programming languages, based on the scenario of graph clustering. The task involves input parsing, a strategy pattern, algorithmic computation and the use of container classes like resizable lists and hash tables. The code in each language was written by the first author, thus eliminating side-effects caused by differences in programmer aptitude. The goal was to provide results of interest for the IT-industry, where maintainability plays an important role. Hence, the code was written according to best practices, conventions for each language and readability aspects.

After tests on several datasets of different size, C++ proved to be the fastest and most memory efficient language. However, its implementation had the largest code size, too. Python, an interpreted language, enabled the shortest implementation, but it was almost an order of magnitude slower than C++. Statically typed, JIT-compiled languages like

Java, C# and F# came closer to matching the speed of C++ with smaller code sizes. C# and F# stand out positively since their runtimes were on par with C++ on one of the larger datasets. Future comparisons of programming languages lend themselves well to extensions in multiple directions, which are not part of this paper:

1. Programming productivity is highly relevant for the domain of software engineering. How long does it take a programmer to implement a given task in a language? How much can a software engineer achieve in a given time frame and language? Unfortunately, investigating these questions in a valid setting is hard. Differences in programmer capability severely impact the results. Varying levels of proficiency in multiple languages also distort the outcome. Involving a larger number of programmers, like the article by Prechelt [Prechelt 2000], can alleviate some of the concerns.

2. With increasing proliferation of multi-core CPUs, concurrent processing gains more and more importance. While some languages might be well suited to high-performance single-threaded programming, other languages might provide better facilities for multi-core operation. Future comparisons should extend the focus towards concurrent and parallel computation.

3. The World Wide Web or current social networking sites with tens of millions of users correspond to ultra large graphs that are beyond the memory capacity of a single machine. Problems of these dimensions can be tackled with distributed computing. An analysis of the usability and performance of languages in such a context could give valuable insights.

References

[tim 2010] "Timers, Timer Resolution, and Development of Efficient Code", Technical Report, Microsoft Corporation (2010), http://www.microsoft.com/whdc/system/pnppwr/powermgmt/Timer-Resolution.mspx [accessed 10 June 2011].

[ben 2011] "Computer Language Benchmarks Game", http://shootout.alioth.debian.org/ (2011), [accessed 31 May 2011].

[sci 2012] "SciViews Benchmark", http://www.sciviews.org/benchmark/ (2012), [accessed 30 Dec 2012].

[con 2013] "How to: Disable Concurrent Garbage Collection", http://msdn.microsoft.com/en-us/library/at1stbec[accessed 19 Feb 2013].

[hot 2013] "Java SE 6 HotSpot[tm] Virtual Machine Garbage Collection Tuning", http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html (2013), [accessed 19 Feb 2013].

[Albert et al. 1999] Albert, R., Jeong, H. and Barabási, A.-L., "Internet: Diameter of the World-Wide Web", Nature, 401, 6749, (1999), 130–131.

[Albrecht and Gaffney 1983] Albrecht, A. and Gaffney, J., J.E., "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation", IEEE Transactions on Software Engineering, SE-9, 6, (1983), 639 – 648, ISSN 0098-5589, doi: 10.1109/TSE.1983.235271.

[Anda et al. 2009] Anda, B., Sjoberg, D. and Mockus, A., "Variability and Reproducibility in Software Engineering: A Study of Four Companies that Developed the Same System", Software Engineering, IEEE Transactions on, 35, 3, (2009), 407 –429, ISSN 0098-5589, doi: 10.1109/TSE.2008.89.

[Boehm 1981] Boehm, B. W., Software Engineering Economics, Prentice Hall, Englewood Cliffs, NJ (1981).

[Boehm et al. 2000] Boehm, B. W., Clark, Horowitz, Brown, Reifer, Chulani, Madachy, R. and Steece, B., Software Cost Estimation with Cocomo II, Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition (2000), ISBN 0130266922.

[Boguñá et al. 2004] Boguñá, M., Pastor-Satorras, R., Díaz-Guilera, A. and Arenas, A., "Models of social networks based on social distance attachment", Phys. Rev. E, 70, 5, (2004), 056122, doi:10.1103/PhysRevE.70.056122.

[Cardelli 2004] Cardelli, L., "Type Systems", in A. B. Tucker (editor), The Computer Science and Engineering Handbook, chapter 97, CRC Press (2004).

[Cesarini et al. 2008] Cesarini, F., Pappalardo, V. and Santoro, C., "A comparative evaluation of imperative and functional implementations of the imap protocol", in Proceedings of the 7th ACM SIGPLAN workshop on ERLANG, ERLANG '08, ACM, New York, NY, USA (2008), ISBN 978-1-60558-065-4, 29–40, doi:10.1145/1411273.1411279.

[Cormen et al. 2009] Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C., Introduction to Algorithms, The MIT Press, Cambridge, Massachusetts, 3 edition (2009), 589–593.

[Ecma International 2001] Ecma International, ECMA-334: C# Language Specification, Ecma (European Association for Standardizing Information and Communication Systems), 1 edition (2001).

[Girvan and Newman 2002] Girvan, M. and Newman, M. E. J., "Community structure in social and biological networks", Proceedings of the National Academy of Sciences, 99, 12, (2002), 7821–7826, doi:10.1073/pnas.122653799.

[Guimerà et al. 2003] Guimerà, R., Danon, L., Díaz-Guilera, A., Giralt, F. and Arenas, A., "Self-similar community structure in a network of human interactions", Phys. Rev. E, 68, 6, (2003), 065103, doi:10.1103/PhysRevE.68.065103.

[Jay et al. 2009] Jay, G., Hale, J. E., Smith, R. K., Hale, D. P., Kraft, N. A. and Ward, C., "Cyclomatic Complexity and Lines of Code: Empirical Evidence of a Stable Linear Relationship", JSEA, 2, 3, (2009), 137–143.

[Kennedy et al. 2004] Kennedy, K., Koelbel, C. and Schreiber, R., "Defining and Measuring the Productivity of Programming Languages", International Journal of High Performance Computing Applications, 18, (2004), 441–448, ISSN 1094-3420, doi: 10.1177/1094342004048537.

[McConnell 2004] McConnell, S., Code Complete: A Practical Handbook of Software Construction, Microsoft Press, Redmond, WA, USA, 2nd edition (2004), ISBN 0735619670.

[Meijer and Gough 2000] Meijer, E. and Gough, J., "Technical Overview of the Common Language Runtime", Technical Report (2000), http://research.microsoft.com/en-us/um/people/emeijer/Papers/CLR.pdf [accessed 25 March 2012].

[Newman 2001] Newman, M. E. J., "The structure of scientific collaboration networks", Proceedings of the National Academy of Sciences, 98, 2, (2001), 404–409, doi: 10.1073/pnas.98.2.404.

[Newman 2004] Newman, M. E. J., "Fast algorithm for detecting community structure in networks", Phys. Rev. E, 69, 6, (2004), 066133, doi:10.1103/PhysRevE.69.066133.

[Newman and Girvan 2004] Newman, M. E. J. and Girvan, M., "Finding and evaluating community structure in networks", Phys. Rev. E, 69, 2, (2004), 026113, doi: 10.1103/PhysRevE.69.026113.

[Nooy et al. 2004] Nooy, W. d., Mrvar, A. and Batagelj, V., Exploratory Social Network Analysis with Pajek, Cambridge University Press, New York, NY, USA (2004), ISBN 0521602629.

[Nyström et al. 2008] Nyström, J. H., Trinder, P. W. and King, D. J., "High-level distribution for the rapid production of robust telecoms software: comparing C++ and ERLANG", Concurrency and Computation: Practice and Experience, 20, 8, (2008), 941–968, ISSN 1532-0634, doi:10.1002/cpe.1223.

[Ovelgönne and Geyer-Schulz 2010] Ovelgönne, M. and Geyer-Schulz, A., "Cluster Cores and Modularity Maximization", in W. Fan, W. Hsu, G. I. Webb, B. Liu, C. Zhang, D. Gunopulos and X. Wu (editors), ICDMW '10. 10th IEEE International Conference on Data Mining Workshops (Sydney, Australia), IEEE Computer Society, IEEE Computer Society, Los Alamitos (2010), 1204 – 1213.

[Ovelgönne and Geyer-Schulz 2012] Ovelgönne, M. and Geyer-Schulz, A., "A Comparison of Agglomerative Hierarchical Algorithms for Modularity Clustering", in Proceedings of the 34th Conference of the German Classification Society, Studies in Classification, Data Analysis, and Knowledge Organization, Springer, Heidelberg (2012), 225 – 232.

[Ovelgönne and Geyer-Schulz 2013] Ovelgönne, M. and Geyer-Schulz, A., "An Ensemble Learning Strategy for Graph Clustering", in D. A. Bader, H. Meyerhenke, P. Sanders and D. Wagner (editors), Graph Partitioning and Graph Clustering, Contemporary Mathematics, volume 588, American Mathematical Society, Providence (2013), 187–205.

[Ovelgönne et al. 2010] Ovelgönne, M., Geyer-Schulz, A. and Stein, M., "Randomized Greedy Modularity Optimization for Group Detection in Huge Social Networks", in SNA-KDD'2010: Proceedings of the 4th Workshop on Social Network Mining and Analysis, ACM, New York, NY, USA (2010).

[Paleczny et al. 2001] Paleczny, M., Vick, C. and Click, C., "The Java Hotspot™Server Compiler", in Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1, JVM'01, USENIX Association, Berkeley, CA, USA (2001), 1–1.

[Park 1992] Park, R. E., "Software Size Measurement: A Framework for Counting Source Statements", Technical Report CMU/SEI-92-TR-020, ESC-TR-92-020, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213 (1992).

[Prechelt 2000] Prechelt, L., "An empirical comparison of seven programming languages", Computer, 33, 10, (2000), 23 –29, ISSN 0018-9162, doi:10.1109/2.876288.

[Prechelt 2011] Prechelt, L., "Plat_Forms: A Web Development Platform Comparison by an Exploratory Experiment Searching for Emergent Platform Properties", IEEE Transactions on Software Engineering, 37, 1, (2011), 95–108, ISSN 0098-5589, doi:10.1109/TSE.2010.22.

[Sackman et al. 1968] Sackman, H., Erikson, W. J. and Grant, E. E., "Exploratory experimental studies comparing online and offline programming performance", Commun. ACM, 11, (1968), 3–11, ISSN 0001-0782, doi:10.1145/362851.362858.

[Sjoeberg et al. 2005] Sjoeberg, D. I., Hannay, J. E., Hansen, O., Kampenes, V. B., Karahasanovic, A., Liborg, N.-K. and Rekdal, A. C., "A Survey of Controlled Experiments in Software Engineering", IEEE Transactions on Software Engineering, 31, 9, (2005), 733–753, ISSN 0098-5589, doi:10.1109/TSE.2005.97.

[Stärk et al. 2012] Stärk, U., Prechelt, L. and Jolevski, I., "Plat_Forms 2011: finding emergent properties of web application development platforms", in Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement, ESEM '12, ACM, New York, NY, USA (2012), ISBN 978-1-4503-1056-7, 119–128, doi: 10.1145/2372251.2372273.

[Stroustrup 1993] Stroustrup, B., "A History of C++: 1979-1991", in The second ACM SIGPLAN conference on History of programming languages, HOPL-II, ACM, New York, NY, USA (1993), ISBN 0-89791-570-4, 271–297, doi:10.1145/154766.155375.

[Veldhuizen 2003] Veldhuizen, T. L., "C++ Templates are Turing Complete", Technical Report (2003), http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.3670 [accessed 10 June 2011].

[Wilbers et al. 2009] Wilbers, I., Langtangen, H. P. and Ødegård, Å., "Using Cython to Speed up Numerical Python Programs", in B. Skallerud and H. I. Andersson (editors), Proceedings of MekIT'09, NTNU, Tapir (2009), ISBN 978-82-519-2421-4, 495–512.

[Zachary 1977] Zachary, W., "An information flow model for conflict and fission in small groups", Journal of Anthropological Research, 33, (1977), 452–473.

A Appendix

The tables in the appendix are listed in the following order:

- Runtime results in all languages: Table A.1 and A.2
- Memory results in all languages: Table A.3 and Table A.4
- .NET results for different garbage collection options: A.5 and A.6
- Java runtime results for different garbage collection options: A.7 and A.8
- Java memory results for different garbage collection options: A.9 and A.10

		karate [ms]	football [ms]	email [ms]	pgp [ms]	condmat [s]	www [s]
C++	∅	0.49	3.62	31.60	169.36	1.13	10.61
	s	0.01	0.07	2.95	5.21	0.02	0.04
Java	∅	11.00	43.65	542.98	752.37	2.55	21.81
	s	5.24	4.41	101.40	65.08	0.10	0.47
C#	∅	43.65	46.47	70.20	296.71	1.48	19.60
	s	1.05	1.06	8.15	11.08	0.03	0.27
F#	∅	70.27	76.76	131.51	611.99	2.49	31.64
	s	1.44	1.73	9.47	10.37	0.04	0.44
Python	∅	4.67	23.96	238.37	1,694.01	7.27	74.42
	s	0.10	0.53	8.02	13.32	0.04	0.36
Cython	∅	3.75	19.56	203.42	1,496.21	6.76	64.51
	s	0.07	0.39	5.39	38.79	0.11	0.98

Table A.1: Runtime arithmetic means ∅ and sample standard deviations s for Randomized Greedy; 100 runs each

		karate [ms]	football [ms]	email [ms]	pgp [s]	condmat [s]	www [10^3s]
C++	⌀	0.12	1.85	118.00	6.79	114.79	11.61
	s	0.00	0.02	7.42	0.03	0.83	0.17
Java	⌀	5.80	39.14	553.08	14.94	197.83	18.25
	s	0.16	0.55	98.29	0.13	1.80	0.18
C#	⌀	40.58	42.88	184.08	11.01	115.96	15.71
	s	0.89	1.13	10.45	0.57	14.55	2.70
F#	⌀	65.28	68.30	228.07	10.30	116.77	15.17
	s	1.10	1.32	9.91	0.07	0.97	0.03
Python	⌀	1.87	27.98	1,904.77	115.26	1,251.68	n/a
	s	0.03	0.57	13.09	0.25	4.97	
Cython	⌀	1.41	19.06	1,276.40	86.01	944.08	n/a
	s	0.03	0.55	9.29	0.40	2.64	

Table A.2: Runtime arithmetic means ⌀ and sample standard deviations s for Newman's Greedy; 50 runs each except www dataset (3 runs). Values shown as 0.00 are too small to display with two decimal places.

		karate	football	email	pgp	condmat	www
C++	⌀	0.91	1.02	1.99	6.79	22.26	208.84
	s	0.00	0.00	0.02	0.06	0.17	0.50
Java	⌀	90.69	117.19	104.45	105.03	105.14	695.92
	s	0.00	11.17	2.96	2.40	2.86	30.23
C#	⌀	8.09	8.98	16.55	24.60	42.97	417.14
	s	0.04	0.04	0.06	0.06	0.10	27.36
F#	⌀	8.46	9.55	18.19	28.86	63.27	492.01
	s	0.04	0.06	0.04	0.05	3.68	50.91
Python	⌀	4.26	4.38	6.14	14.77	42.96	376.25
	s	0.00	0.00	0.03	0.11	0.14	0.50
Cython	⌀	4.35	4.44	6.49	17.94	50.84	467.86
	s	0.00	0.00	0.03	0.09	0.17	0.77

Table A.3: Arithmetic means ⌀ and sample standard deviations s of peak memory requirements[MB] for Randomized Greedy; 100 runs each. Values shown as 0.00 are too small to display with two decimal places.

		karate	football	email	pgp	condmat	www
C++	∅	0.92	1.01	1.75	5.43	18.14	166.88
	s	0.00	0.00	0.00	0.00	0.00	0.00
Java	∅	90.70	92.00	100.79	101.04	100.37	671.67
	s	0.00	0.01	1.78	1.12	1.44	5.65
C#	∅	8.02	8.32	10.00	20.10	38.87	273.99
	s	0.07	0.04	0.06	0.04	0.04	0.12
F#	∅	8.42	8.85	13.94	21.32	51.56	417.40
	s	0.06	0.05	0.07	0.05	0.06	0.15
Python	∅	4.24	4.37	6.06	14.52	41.91	n/a
	s	0.01	0.00	0.00	0.00	0.07	
Cython	∅	4.32	4.44	6.12	14.44	40.72	n/a
	s	0.00	0.00	0.00	0.00	0.00	

Table A.4: Arithmetic means ∅ and sample standard deviations s of peak memory requirements[MB] for Newman's Greedy; 50 runs each except www dataset (3 runs). Values shown as 0.00 are too small to display with two decimal places.

		karate [ms]	football [ms]	email [ms]	pgp [ms]	condmat [s]	www [s]
C#single	∅	43.65	46.47	**70.20**	**296.71**	**1.48**	**19.60**
	s	1.05	1.06	8.15	11.08	0.03	0.27
C#conc.	∅	**42.41**	**45.22**	71.92	299.99	1.66	19.84
	s	0.73	0.50	8.56	9.88	0.05	0.26
F#single	∅	70.27	76.76	131.51	**611.99**	**2.49**	31.64
	s	1.44	1.73	9.47	10.37	0.04	0.44
F#conc.	∅	**69.74**	**75.53**	**125.58**	624.00	2.68	**31.47**
	s	1.55	1.63	8.11	15.68	0.05	0.45

Table A.5: Runtime arithmetic means ∅ and sample standard deviations s for .NET Randomized Greedy with different GC settings; 100 runs each. The best performance for C# and F# each is highlighted in bold.

		karate	football	email	pgp	condmat	www
C#single	⌀	8.09	8.98	16.55	24.60	**42.97**	**417.14**
	s	0.04	0.04	0.06	0.06	0.10	27.36
C#conc.	⌀	**8.06**	**8.95**	**15.74**	**22.55**	57.68	472.34
	s	0.06	0.05	0.05	0.06	1.72	30.80
F#single	⌀	8.46	9.55	18.19	**28.86**	**63.27**	**492.01**
	s	0.04	0.06	0.04	0.05	3.68	50.91
F#conc.	⌀	**8.44**	**9.53**	**16.14**	35.16	69.01	549.23
	s	0.03	0.04	0.06	0.06	5.48	36.48

Table A.6: Arithmetic means ⌀ and sample standard deviations s of peak memory requirements[MB] for .NET Randomized Greedy; 100 runs each. The lowest requirement for C# and F# each is highlighted in bold.

		karate [ms]	football [ms]	email [ms]	pgp [ms]	condmat [s]	www [s]
Serial	⌀	11.00	43.65	**542.98**	752.37	2.55	**21.81**
	s	5.24	4.41	101.40	65.08	0.10	0.47
Parallel	⌀	**9.71**	**39.04**	582.58	753.06	**1.87**	21.85
	s	0.30	0.68	122.75	59.84	0.09	2.11
ParallelOld	⌀	10.14	39.44	574.17	**752.01**	1.90	28.45
	s	0.30	1.41	112.15	66.05	0.11	2.11

Table A.7: Runtime arithmetic means ⌀ and sample standard deviations s for Java Randomized Greedy with different GC settings; 100 runs each. The best performance is highlighted in bold.

		karate [ms]	football [ms]	email [ms]	pgp [s]	condmat [s]	www [10^3 s]
Serial	∅	5.80	39.14	553.08	**14.94**	197.83	**18.25**
	s	0.16	0.55	98.29	0.13	1.80	0.18
Parallel	∅	**5.28**	36.84	536.52	17.28	**193.62**	19.02
	s	0.13	0.31	92.37	0.30	3.30	0.20
ParallelOld	∅	5.33	**36.83**	**509.42**	17.44	194.17	19.44
	s	0.14	0.29	92.04	0.17	1.68	0.59

Table A.8: Runtime arithmetic means ∅ and sample standard deviations s for Java Newman's Greedy with different GC settings; 100 runs each. The best performance is highlighted in bold.

		karate	football	email	pgp	condmat	www
Serial	∅	**90.69**	**117.19**	**104.45**	**105.03**	**105.14**	**695.92**
	s	0.00	11.17	2.96	2.40	2.86	30.23
Parallel	∅	90.72	118.24	114.28	152.60	368.64	989.89
	s	0.00	8.66	8.46	1.92	1.43	39.37
ParallelOld	∅	158.91	189.38	180.39	220.67	437.22	992.79
	s	0.02	8.43	8.31	2.00	1.31	62.95

Table A.9: Arithmetic means ∅ and sample standard deviations s of peak memory requirements[MB] for Java Randomized Greedy; 100 runs each. The lowest requirement is highlighted in bold.

		karate	football	email	pgp	condmat	www
Serial	∅	**90.70**	**92.00**	100.79	**101.04**	**100.37**	**671.67**
	s	0.00	0.01	1.78	1.12	1.44	5.65
Parallel	∅	90.71	92.06	**100.63**	549.02	603.39	929.69
	s	0.00	0.02	1.78	52.33	0.19	1.49
ParallelOld	∅	158.90	160.54	169.83	625.06	672.14	992.81
	s	0.01	0.02	1.19	48.95	0.20	14.35

Table A.10: Arithmetic means ∅ and sample standard deviations s of peak memory requirements[MB] for Java Newman's Greedy; 50 runs each. The lowest requirement is highlighted in bold.