Multi-Level Algorithms for Modularity Clustering

Andreas Noack* Randolf Rotta*

Abstract

Modularity is one of the most widely used quality measures for graph clusterings. Maximizing modularity is NP-hard, and the runtime of exact algorithms is prohibitive for large graphs. A simple and effective class of heuristics coarsens the graph by iteratively merging clusters (starting from singletons), and optionally refines the resulting clustering by iteratively moving individual vertices between clusters. Several heuristics of this type have been proposed in the literature, but little is known about their relative performance.

This paper experimentally compares existing and new coarsening- and refinement-based heuristics with respect to their effectiveness (achieved modularity) and efficiency (runtime). Concerning coarsening, it turns out that the most widely used criterion for merging clusters (modularity increase) is outperformed by other simple criteria, and that a recent algorithm by Schuetz and Caflisch is no improvement over simple greedy coarsening for these criteria. Concerning refinement, a new multi-level algorithm is shown to produce significantly better clusterings than conventional single-level algorithms. A comparison with published benchmark results and algorithm implementations shows that combinations of coarsening and multi-level refinement are competitive with the best algorithms in the literature.

1 Introduction

A *graph clustering* partitions the vertex set of a graph into disjoint subsets called *clusters*. *Modularity* was introduced by Newman and Girvan as formalization of the common requirement that the connections within graph clusters should be dense, and the connections between different graph clusters should be sparse [30]. It is by far not the only quality measure for graph clusterings [13, 35], but one of the most widely used measures, and has been successfully applied for detecting meaningful groups in a wide variety of complex systems.

The problem of finding a clustering with maximum modularity for a given graph is NP-hard [6], and even recent exact algorithms scale only to graphs with a few hundred vertices [6, 1, 40]. In practice, modularity is almost exclusively optimized with heuristic algorithms. Like modularity itself, many of these heuristics have been proposed in the physics literature.

A particularly simple heuristic is the iterative merging of cluster pairs, starting from singleton clusters, and always choosing the merge that results in the largest modularity increase. This *greedy coarsening* can be efficiently implemented and produces reasonable clusterings [27, 7], but was soon observed to be biased towards merging large clusters [9, 38]. To remove this bias and obtain clusterings with even higher modularity, researchers suggested to replace modularity increase with other prioritizing criteria for potential merges [9, 38], and to modify the purely greedy merge strategy [36]. However, the numerous proposals have not been organized into a coherent design space, and the published evaluation results are largely incomparable due to the use of different (and often small) graph collections. Therefore, Section 3 systematically describes major design alternatives for coarsening algorithms, including two new prioritizing criteria for merges, and Section 5.2 compares them experimentally.

The clusterings produced by coarsening heuristics can be improved with *refinement algorithms*, which iteratively move individual vertices between clusters [36]. An obvious solution is greedy refinement, which always chooses the vertex move resulting in the largest increase of modularity. However, moving the vertices in arbitrary order (instead of always moving the best vertex) is much faster and not necessarily less effective, and adaptations of the classic Kernighan-Lin refinement [20] are not much slower and have some capability to escape local maxima. All of these algorithms can be applied not only to the original graph, but to any level of the coarsening hierarchy, by considering each cluster of the coarsening level as a single coarse vertex. This *multi-level refinement* is extremely effective for minimum cut partitioning problems [18, 19], but has not previously been adapted to modularity clustering. Section 4 details the single-level and multi-level refinement heuristics, and Section 5.3 compares them experimentally. Because the effectiveness of (particularly multi-level) refinement may depend on the coarsening algorithm, Section 5.4 examines various combinations of coarsening and refinement heuristics.

Section 6 compares public implementations and benchmark results of modularity clustering heuristics, without a restriction to coarsening and refinement algorithms. While this is one of the most extensive comparisons in the literature, it is far from exhaustive, because implementations and sufficient experimental results have not been published for some proposed heuris-

*Brandenburg University of Technology, 03013 Cottbus, Germany; {an,rrotta}@informatik.tu-cottbus.de

tics. The main purpose is to demonstrate that particular combinations of simple coarsening and multi-level refinement algorithms are (at least) competitive with the best available heuristics, and thus the results of the previous sections are indeed practically significant.

2 Graph Clusterings and Modularity

2.1 Graph Clusterings. A *graph* (V, f) consist of a finite set V of *vertices* and a function $f : V \times V \to \mathbb{N}$ that assigns a nonnegative *edge weight* to each vertex pair. For simplicity, graphs are assumed to be undirected, i.e., $f(u, v) = f(v, u)$ for all $u, v \in V$. The *degree* $\deg(v)$ of a vertex v is the total weight $\sum_{u \in V} f(u, v)$ of its edges. The degrees and weights are naturally generalized to sets of vertices, e.g., $f(V, V) = \sum_{u \in V, v \in V} f(u, v)$. Note that $\deg(V) = f(V, V)$.

A *graph clustering* $\mathcal{C} = \{C_1, \ldots, C_k\}$ partitions the vertex set V into disjoint non-empty subsets C_i.

2.2 Modularity. Modularity is a quality measure for graph clusterings. It was originally introduced for graphs where the edge weights are either 0 or 1 [30], and was later generalized to arbitrary edge weights [26]. The *modularity* of a clustering $\mathcal{C}$ is defined as

$$Q(\mathcal{C}) \;:=\; \sum_{C \in \mathcal{C}} \left(\frac{f(C, C)}{f(V, V)} - \frac{\deg(C)^2}{\deg(V)^2} \right).$$

Intuitively, the first term is the *actual* fraction of intra-cluster edge weight. In itself, it is not a good measure of clustering quality, because it takes the maximum value 1 for the trivial clustering where one cluster contains all vertices. The second term specifies the *expected* fraction of intra-cluster edge weight in a null model where the end-vertices of $\frac{1}{2}\deg(V)$ edges are chosen at random, and the probability that an end-vertex of an edge attaches to a particular vertex v is $\frac{\deg(v)}{\deg(V)}$ [28]. In this null model, the edge weight $f(u, v)$ between each vertex pair $(u, v) \in V^2$ is binomially distributed with the expected value $\frac{\deg(u)\deg(v)}{\deg(V)}$.

It can be easily verified that merging two clusters C and D increases the modularity by

$$\Delta Q_{C,D} \;:=\; \frac{2f(C, D)}{f(V, V)} \;-\; \frac{2\deg(C)\deg(D)}{\deg(V)^2}\;,$$

and moving a vertex v from its current cluster C to another cluster D increases the modularity by

$$\begin{aligned} \Delta Q_{v \to D} \;&:=\; \frac{2f(v, D) - 2f(v, C - v)}{f(V, V)} \\ &-\; \frac{2\deg(v)\deg(D) - 2\deg(v)\deg(C - v)}{\deg(V)^2}\;. \end{aligned}$$

3 Coarsening Algorithms

Greedy coarsening algorithms iteratively merge either one cluster pair, as detailed in the first subsection, or several disjoint cluster pairs, as detailed in the second subsection, and choose the merged cluster pairs according to certain priority criteria, which are discussed in the third subsection.

3.1 Single-Step Greedy. The Single-Step Greedy algorithm starts with single-vertex clusters, and iteratively merges the cluster pair with the highest priority, until this merge would decrease the modularity.

Algorithm: Single-Step Greedy Coarsening
Input: graph, merge prioritizer
Output: clustering

initialize clustering with singleton clusters;
while prioritized pair (C, D) satisfies $\Delta Q_{C,D} > 0$ **do**
  merge clusters C and D;

Implementation and Runtime. For the calculation of the priorities (see Section 3.3) it is necessary to quickly retrieve the total edge weights between adjacent clusters. These total weights change locally with each merge and are thus stored in a dynamically coarsened graph where each cluster is represented by a single vertex. In each merge of two vertices u and v, the edge list of the vertex with fewer edges (say u) is merged into the edge list of the other vertex. Using the sorted double-linked edge lists proposed by Wakita and Tsurumi [38], this requires linear time in the list lengths. However, if some neighbor vertices of u are not neighbors of v, then one end-vertex of the edges to these neighbors changes from u to v, and the position of these edges in the neighbors' edge lists must be corrected to retain the sorting.

Let n be the number of clusters (initially the vertex count), m be the number of adjacent cluster pairs (edge count), and d be the height of the merge tree. Merging the edge lists of two clusters has linear runtime in the list lengths, and each edge participates in at most d such merges. Thus the worst-case runtime is $\mathcal{O}(dm)$ for the merges and, given that the length of each edge list is at most n, $\mathcal{O}(dmn)$ for the position corrections. (The implementation of Clauset et al. [7] has better worst-case bounds, but experimental results in Section 5 indicate that it is not more efficient in practice.)

In order to quickly find the prioritized cluster pair for the next merge, a priority queue (max-heap) over the clusters and their current best partner is used. It is updated as described in [38]. In worst case the priority queue is updated with each merged edge, taking $\mathcal{O}(dm \log n)$ runtime.

3.2 Multi-Step Greedy. To prevent extremely unbalanced cluster growth, Schuetz and Caflisch introduced Multi-Step Greedy coarsening, which iteratively merges the l *disjoint* cluster pairs with the highest priority (unless the merge decreases the modularity) [36]. Single-Step Greedy coarsening corresponds to the special case of $l = 1$ (at least conceptually, the implementation differs). To make the parameter l independent of the graph size, we specify it as percentage of the number of modularity-increasing cluster pairs, and call it *merge fraction*.[1] The impact of the merge fraction on the effectiveness of Multi-Step Greedy coarsening will be examined experimentally in Section 5.2.

Algorithm: Multi-Step Greedy Coarsening
Input: graph, merge prioritizer, merge fraction
Output: clustering

```
initialize clustering with singleton clusters;
while ∃ cluster pair (C, D) : ΔQ_{C,D} > 0 do
    l ← merge fraction × |{(C, D) : ΔQ_{C,D} > 0}|;
    mark all clusters as unmerged;
    for ⌈l⌉ most prioritized pairs (C, D) do
        if C and D are marked as unmerged then
            merge clusters C and D;
            mark clusters C and D as merged;
```

Implementation and Runtime. The same basic data structures as in Single-Step Greedy are used. To iterate over the l cluster pairs in priority order, the edges are sorted once before entering the inner loop. This requires $\mathcal{O}(m \log m)$ time in worst case. Alternative implementations optimized for very small merge fractions could use partial sorting with $\mathcal{O}(m \log l)$ (but a larger constant factor). With merge fraction α the inner loop is repeated at least n/α times. However, if only few disjoint cluster pairs exist, as in some power-law graphs, up to n iterations may be necessary.

3.3 Merge Prioritizers. A merge prioritizer assigns to each cluster pair (C, D) a real number called *merge priority*, and thereby determines the order in which the coarsening algorithms merge cluster pairs. Because the coarsening algorithms use only the order of the priorities, two prioritizers can be considered as equivalent if one can be transformed into the other by adding a constant or multiplying with a positive constant.

[1] Recently, Schuetz and Caflisch provided the empirical formula $l_{opt} := \alpha\sqrt{f(V,V)}$ for good values of l [37]. It does not outperform our formula for unweighted graphs (see Section 6), and is unsuitable for weighted graphs, because scaling all edge weights with a positive constant changes l_{opt} but not the optimal clustering.

The *Modularity Increase (MI)* $\Delta Q_{C,D}$ resulting from the merge of the clusters C and D is an obvious and widely used merge prioritizer [27, 7, 36, 41].

The *Weight Density (WD)* is defined as $\frac{f(C,D)}{\deg(C)\deg(D)}$, and is equivalent to $\frac{\Delta Q_{C,D}}{\deg(C)\deg(D)}$. Its use as merge prioritizer has not yet been proposed in the literature, although Newman and Girvan originally introduced the modularity measure to formalize the requirement of intra-cluster density and inter-cluster sparsity [30], and Reichardt and Bornholdt showed that clusterings with optimal modularity indeed fulfill this requirement [33].

The *Significance (Sig)*, another new merge prioritizer, is defined as $\frac{\Delta Q_{C,D}}{\sqrt{\deg(C)\deg(D)}}$, and is thus a natural compromise between Modularity Increase and Weight Density. A further motivation is its relation to the (im)probability of the edge weight $f(C, D)$ in the null model described in Section 2.2. Under this null model, both the expected value and the variance (at least for large $\deg(V)$) of the edge weight between C and D are $\frac{\deg(C)\deg(D)}{\deg(V)}$, and the Significance is equivalent to the number of standard deviations that separate the actual edge weight from the expected edge weight.

Danon et al. (DA) observed that the Modularity Increase $\Delta Q_{C,D}$ tends to prioritize pairs of clusters with large degrees, and proposed the merge prioritizer $\frac{\Delta Q_{C,D}}{\min(\deg(C),\deg(D))}$ to avoid this bias [9]. It equals the Significance if $\deg(C) = \deg(D)$, and is another compromise between Modularity Increase and Weight Density.

Wakita and Tsurumi found that greedy coarsening by Modularity Increase tends to merge clusters of extremely uneven sizes [38]. In order to suppress unbalanced merges, they proposed the merge prioritizer $\min\left(\frac{\text{size}(C)}{\text{size}(D)}, \frac{\text{size}(D)}{\text{size}(C)}\right)\Delta Q_{C,D}$, where $\text{size}(C)$ is either the number of vertices in C (prioritizer *HN*) or the number of other clusters to which C is connected by an edge of positive weight (prioritizer *HE*).

Other types of merge prioritizers are clearly possible. For example, vertex distances from random walks or eigenvectors of certain matrices have been successfully applied in several clustering algorithms (e.g., [31, 28, 11]). However, preliminary experiments suggest that these relatively complicated and computationally expensive prioritizers may not be more effective than the simple prioritizers in this section [34].

4 Refinement Algorithms

Refinement algorithms perform a local search by iteratively moving individual vertices to different clusters (including newly created clusters) such that the modularity increases.

The first three subsections describe simple variants of greedy refinement, and the final subsection proposes, for the first time in modularity clustering, to apply refinement on more than one level of the coarsening hierarchy. Excluded from consideration are algorithms with several tunable parameters or explicit randomness, like simulated annealing [33, 24, 23] or extremal optimization [12].

4.1 Complete Greedy. Complete Greedy refinement repeatedly performs the best vertex move, until no further modularity-increasing vertex moves are possible. Here the *best* vertex move is a move with the largest modularity increase $\Delta Q_{v \to D}$ over all vertices v and all target clusters D.

Algorithm: Complete Greedy Refinement

Input: graph, clustering
Output: clustering

repeat
  $(v, D) \leftarrow$ best vertex move;
  if $\Delta Q_{v \to D} > 0$ **then**
    move vertex v to cluster D;
until $\Delta Q_{v \to D} \leq 0$;

Implementation and Runtime. Vertices are moved in constant time using a vector mapping vertices to their current cluster. To find the best move, the modularity changes $\Delta Q_{v \to D}$ for all vertices v and clusters D adjacent to v (and a newly created cluster) need to be determined. For this purpose the algorithm iterates over the vertices. For each vertex v the summed weights $f(v, D)$ are collected in one pass over its edges by using a search tree similar to [4]. Given $f(v, D)$, the modularity change $\Delta Q_{v \to D}$ can be computed in constant time (see Section 2.2). Therefore, to find the globally best move, all n vertices and m edges are visited once, and the weight of each edge is added in a search tree of at most n entries. Assuming $\mathcal{O}(n)$ moves yields a worst-case runtime of $\mathcal{O}(nm \log n)$.

4.2 Fast Greedy. Fast Greedy refinement repeatedly iterates through all vertices and moves each vertex to its best cluster, until no improvement is found for any vertex. Finding the best move for a particular vertex is considerable cheaper than finding the globally best vertex move, as in Complete Greedy refinement; the question whether this improved efficiency comes at the cost of worse effectiveness will be addressed by an experiment in Section 5.3. Fast Greedy refinement has been previously proposed by Schuetz and Caflisch [36] and Ye et al. [41].

Algorithm: Fast Greedy Refinement

Input: graph, clustering
Output: clustering

repeat
  foreach vertex v **do**
    $D \leftarrow$ best cluster for v;
    if $\Delta Q_{v \to D} > 0$ **then**
      move vertex v to cluster D;
until no improved clustering found ;

Implementation and Runtime. The implementation is very similar to the previous algorithm. The worst-case time for one run of the inner loop is $\mathcal{O}(m \log n)$, and a few runs usually suffice.

The order in which the inner loop visits the vertices seems to have little impact on the obtained modularity in practice; in our implementation, vertices are sorted by increasing number of edges.

4.3 Adapted Kernighan-Lin. Kernighan-Lin refinement extends Complete Greedy refinement with a basic capability to escape local maxima. The algorithm was originally proposed by Kernighan and Lin for minimum cut partitioning [20], and was adapted to modularity clustering by Newman [29] (though with a limitation to two clusters). In its inner loop, the algorithm iteratively performs the best vertex move, with the restriction that each vertex is moved only once, but without the restriction that each move must increase the modularity. After all vertices have been moved, the inner loop is restarted from the best found clustering. Preliminary experiments indicated that it is much more efficient and rarely less effective to abort the inner loop when the best found clustering has not improved in the last $k := 10 \log_2 |V|$ vertex moves [34].

Algorithm: Adapted Kernighan-Lin Refinement

Input: graph, clustering
Output: clustering

repeat
  peak $\leftarrow$ clustering;
  mark all vertices as unmoved;
  while unmoved vertices exist **do**
    $(v, D) \leftarrow$ best move with v unmoved;
    move v to cluster D, mark v as moved;
    if Q(clustering) $> Q$(peak) **then**
      peak $\leftarrow$ clustering;
    if k moves since last peak **then**
      break ;
  clustering $\leftarrow$ peak;
until no improved clustering found ;

Implementation and Runtime. The implementation is largely straightforward; to improve efficiency, the current clustering is not copied unless the modularity begins to decrease. The worst case runtime is the same as for Complete Greedy, assuming that a few outer iterations suffice. In practice, Kernighan-Lin refinement takes somewhat longer, because it also performs modularity-decreasing moves.

4.4 Multi-Level Refinement. The refinement algorithms in the previous subsections easily get stuck in suboptimal clusterings because they only move individual vertices. Even Kernighan-Lin refinement is unlikely to move a medium-sized group of densely interconnected vertices to another cluster, because this would require a series of sharply modularity-decreasing vertex moves. However, the vertex group may well have been merged into a single cluster at some stage of the coarsening, and a refinement algorithm can easily reassign the group if it moves entire clusters of this coarsening level, instead of individual vertices. This is the basic idea of multi-level refinement, which has already proved to be very effective for minimum cut partitioning problems [18, 19].

The Multi-Level Clustering algorithm first executes a coarsening algorithm (for example, any algorithm from Section 3) and then, usually several times, a refinement algorithm (for example, any algorithm from the previous subsections). Intermediate results of the coarsening algorithm are recorded as *coarsening levels* whenever the number of clusters has decreased by a certain percentage, which is provided as a parameter called *reduction factor*. Each coarsening level is a graph whose vertices are the clusters at the respective state of coarsening. The refinement algorithm is applied to every coarsening level, from the coarsest level to the original graph. At the coarsest level, each vertex belongs to a separate cluster, and at the finer levels, the initial cluster membership of each vertex is copied from the corresponding vertex of the previous (coarser) level.

The conventional Single-Level refinement, which executes a refinement algorithm only on the original graph, is the special case of Multi-Level refinement with a reduction factor of 100%. While Multi-Level refinement, and more generally every decrease of the reduction factor, potentially produces better clusterings, it may be suspected to significantly increase the required runtime. However, this is not necessarily the case, because the additional coarsening levels are smaller than the original graph, and one cheap vertex move on a coarse graph can save many expensive vertex moves on a finer graph. The impact of the reduction factor on both effectiveness and efficiency is examined experimentally in Section 5.3.

Algorithm: Multi-Level Clustering
Input: graph, `coarsener`, `refiner`, reduction factor
Output: clustering

```
// coarsening phase
level[1] ← graph;
for l from 1 to ... do
    level[l+1] ← coarsener(level[l], reduction factor);
    if no clusters merged then break ;
// refinement phase
clustering ← vertices of level[l_max];
for l from l_max − 1 to 1 do
    project clustering from level[l+1] to level[l];
    clustering ←refiner(level[l], clustering);
```

Related Work. Several recent algorithms for modularity clustering are related to Multi-Level refinement, but differ in crucial respects. Djidjev's method is (despite its name) not itself a multi-level algorithm, but a divisive method built on an existing multi-level algorithm for minimum cut partitioning [10]. Blondel et al. use local search on multiple levels to coarsen graphs, but do not refine the results of the coarsening [4]. Ye et al.'s algorithm performs refinement on multiple coarsening levels, but only moves vertices of the original graph instead of coarse vertices (clusters) [41].

Implementation and Runtime. With reduction factor α at most $\log_{1/(1-\alpha)}(n)$ coarsening levels are generated. For each new level a graph homomorphism connecting it to the previous level is constructed and used to transfer weights and clusterings between levels. To decouple the Multi-Level Clustering algorithm from details of the coarsening algorithm, the coarse graph and its homomorphism is constructed from the clustering produced by the coarsening algorithm: The vertices of each cluster are connected to their cluster-vertex in $\mathcal{O}(n)$ time. For each edge the corresponding cluster-edge has to be found or added if not yet existing. This search is accelerated by processing all vertices of a cluster successively and using a search tree over the cluster-edges of the current cluster. Thus constructing and connecting all edges costs $\mathcal{O}(m \log n)$ time.

5 Experiments

This section experimentally compares the effectiveness (achieved modularity) and efficiency (runtime) of the various heuristics presented in the previous sections.

5.1 Experimental Setup. The heuristics were implemented in C++ and compiled with GCC 4.2.3. The implementations are available online at `http://www.informatik.tu-cottbus.de/~rrotta/`.

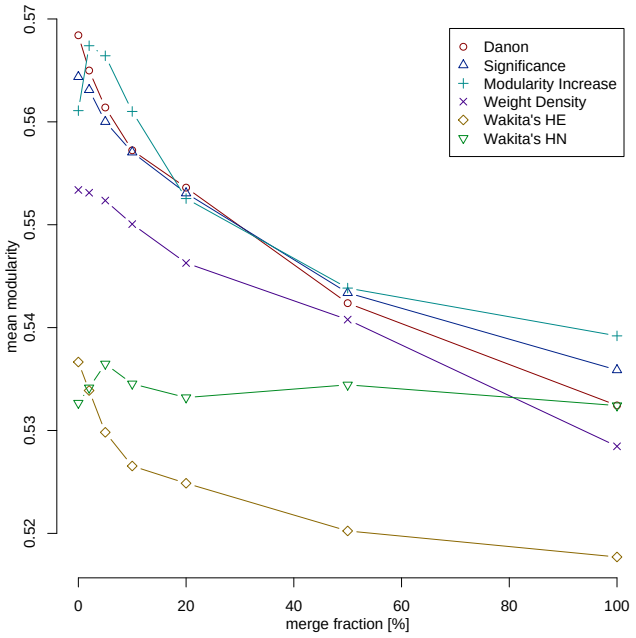

Figure 1: Modularity by merge fraction and prioritizer.

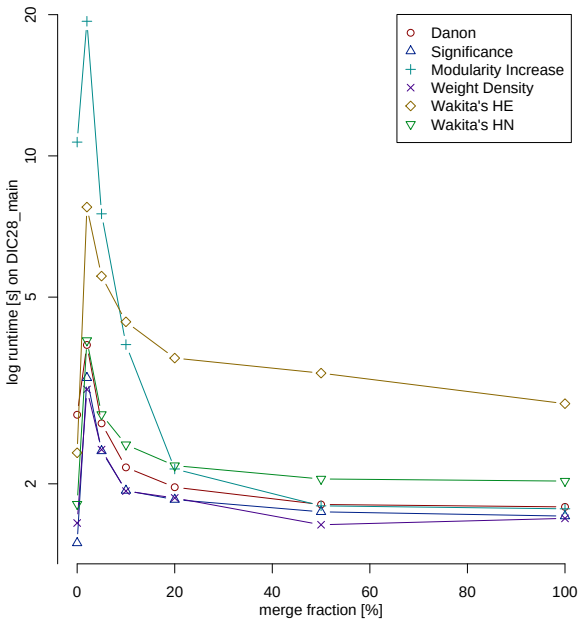

Figure 2: Runtime by merge fraction and prioritizer on the graph 'DIC28_main'.

In order to compare the effectiveness of the heuristics, the arithmetic mean of the modularity over a fixed set of graphs is measured; higher means indicate more effective algorithms. (Thus only the relative values of the means are interpreted, the absolute values are not intended to be meaningful.) Generated graphs are not used because they are structurally very limited (e.g., in their vertex degree distribution), and do not necessarily permit generalizations to graphs from real applications. Instead the graph set contains 58 real-world graphs retrieved from various resources as listed in Appendix A. The available graphs were roughly classified by their application domain and graphs of diverse size that fairly represent all major domains were selected. In addition the collection includes commonly used benchmark graphs like Zachary's karate club network [42]. The graphs range from a few to 75k vertices and 352k edges.

All runtimes were measured on a 3.00GHz Intel Pentium 4 processor with 1GB main memory. The time for reading the graph was excluded to avoid that it interferes with the aspects studied here.

5.2 Coarsening Algorithms. Figure 1 compares the effectiveness of the merge prioritizers for Single-Step Greedy coarsening (represented by a merge fraction of 0%) and Multi-Step Greedy coarsening with merge fractions of 2%, 5%, 10%, 20%, 50%, and 100%. No refinement was used. The runtime measured on the graph 'DIC28_main' is shown in Fig. 2 and is typical for larger graphs.

Concerning the merge prioritizers, Wakita's HE and HN are much less effective than the others, and not more (usually even less) efficient. The lower effectiveness is also visible in Fig. 9 for Wakita's original implementation.

Concerning the algorithms, Multi-Step Greedy is generally less effective and less efficient than the simpler Single-Step Greedy. Only for Modularity Increase, Multi-Step Greedy is faster and, for merge fractions of 2% and 5%, also slightly more effective, but still similar to Single-Step Greedy with Danon and Significance. Apparently the other merge prioritizers do not benefit from Multi-Step Greedy's tendency to balance cluster sizes because, unlike Modularity Increase, they have no strong bias towards merging large clusters.

5.3 Refinement Algorithms. Figure 3 compares the effectiveness of the refinement algorithms for Single-Level refinement (reduction factor 100%) and Multi-Level refinement with reduction factors of 5%, 10%, 20%, and 50%. As coarsener Single-Step Greedy with the Significance prioritizer was chosen, because it proved to be effective and efficient in the previous subsection. The runtime measurements on 'DIC28_main' are shown in Fig. 4 and the dependency of the runtime on the graph size is depicted in Fig. 5.

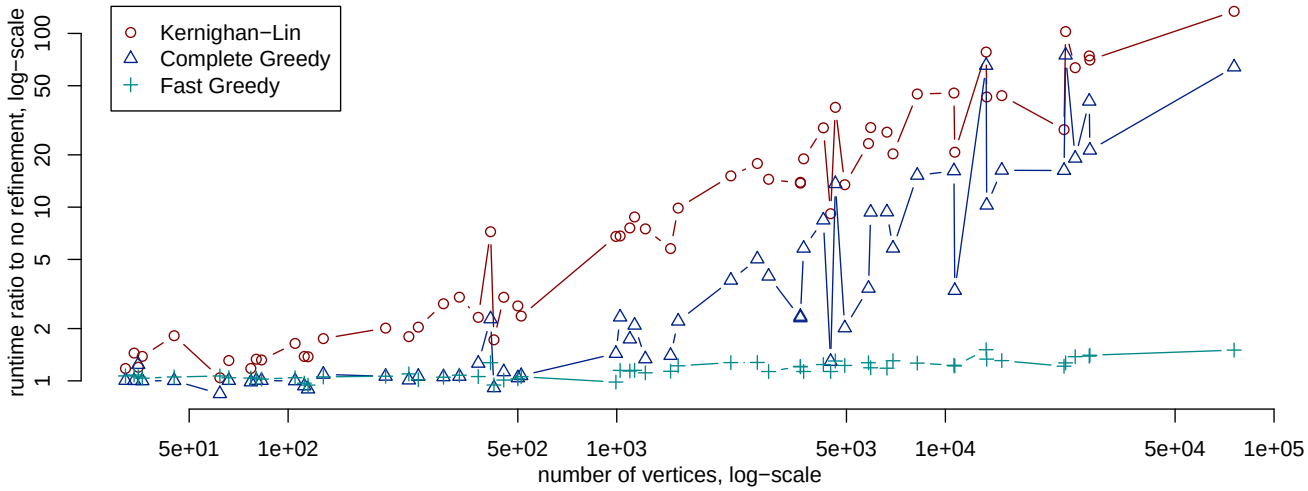

Figure 5: Runtime ratio of the refinement heuristics to raw coarsening, log-log scaled. Reduction factor is 50%.

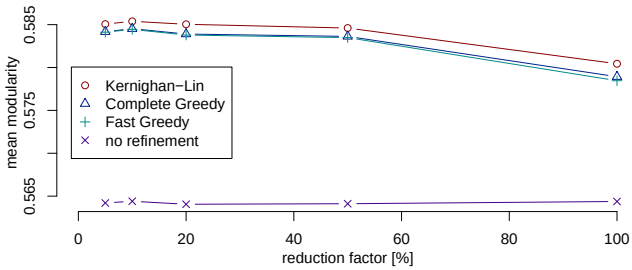

Figure 3: Modularity by reduction factor and refinement method.

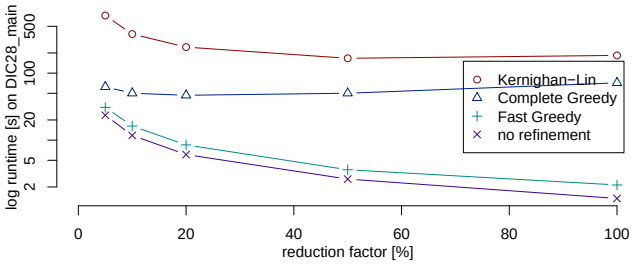

Figure 4: Runtime by reduction factor and refinement method on the graph 'DIC28_main'.

Multi-Level refinement with a reduction factor of 50% turns out to be more effective than Single-Level refinement, and similarly efficient. Reduction factors below 50% do not considerably improve the modularity, but significantly increase the runtime for Fast Greedy.

Fast Greedy refinement is about as effective as Complete Greedy, and just slightly less effective than Kernighan-Lin, but much faster. It scales well with the graph size (see Fig. 5), while Complete Greedy and Kernighan-Lin become prohibitively expensive.

5.4 Combining Coarsening and Refinement. Concerning Single-Level vs. Multi-Level refinement, Fig. 6 shows that Multi-Level refinement is consistently more effective for all merge prioritizers, and thus confirms the results for the Significance prioritizer in Fig. 3.

Concerning Single-Step vs. Multi-Step Greedy coarsening, Fig. 7 shows that for the best merge prioritizers, both are similarly effective with Multi-Level refinement, while Single-Step Greedy is more effective without refinement. Clearly, Multi-Level refinement benefits from the uniform cluster growth enforced by Multi-Step Greedy coarsening. Overall, Single-Step Greedy coarsening is still preferable because of its greater simplicity and efficiency.

Concerning the merge prioritizers, Figs. 6 and 7 show that Modularity Increase is only competitive without refinement (ignoring efficiency), and Weight Density is only competitive with Multi-Level refinement. Here Multi-Level refinement benefits from the bias of Weight Density towards balanced cluster growth, and suffers from the bias of Modularity Increase towards unbalanced cluster growth. Danon and Significance are effective with and without refinement.

5.5 Conclusions. The best algorithm found in these experiments is Single-Step Greedy coarsening with Danon or Significance as merge prioritizer combined with Multi-Level Fast Greedy refinement (or Multi-Level Kernighan-Lin, if efficiency is no concern).

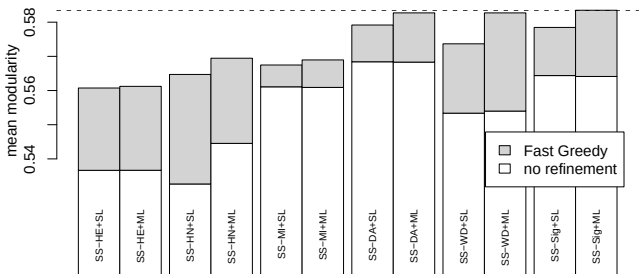

Figure 6: Mean modularity by merge prioritizer. Left bars show reduction factor 100% (Single-Level), right bars 50% (Multi-Level). Both use Single-Step Greedy.

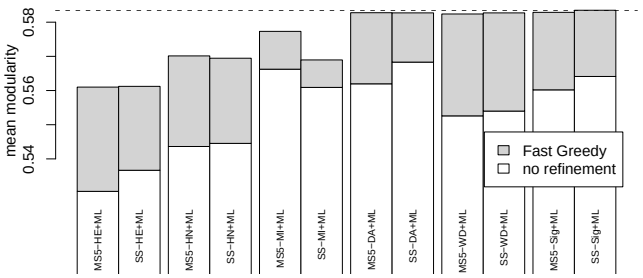

Figure 7: Mean modularity by merge prioritizer: Left bars show merge fraction 5% (Multi-Step) and right bars 0% (Single-Step). Both use reduction factor 50%.

Interestingly, Single-Step Greedy refinement outperformed the recent and more complex Multi-Step Greedy (for the best merge prioritizers), the Danon and Significance merge prioritizers clearly outperformed the much more widely used Modularity Increase (especially with refinement, and considering efficiency) and Wakita's prioritizers, and the newly proposed Multi-Level refinement consistently outperformed the popular Single-Level refinement.

6 Related Algorithms

An exhaustive review and comparison of the numerous algorithms for modularity clustering is beyond the scope of this paper; the purpose of this section is to provide evidence that our recommended heuristic – Single-Step Greedy coarsening by Significance with Multi-Level Fast Greedy refinement (*SS+ML*) – is competitive with the best existing methods.

6.1 Basic Approaches. Algorithms for modularity clustering can be categorized into the following four types: *Subdivision* heuristics try to divide the network, for example by iteratively removing edges [30] or by recursively splitting the graph using eigenvectors [29]. *Coarsening* (or agglomeration) heuristics iteratively merge clusters starting from singletons. Cluster pairs can be selected based on random walks [31, 32], increase of modularity [7, 36, 41], or other criteria [38, 9, 11]. *Local search* heuristics move vertices between clusters, with Kernighan-Lin-style and greedy search being the most prominent examples. Other approaches include Tabu Search [3], Extremal Optimization [12], and Simulated Annealing [33, 24, 23]. Finally, *mathematical programming* approaches model modularity maximization as a linear or quadratic programming problem which can be solved with existing software packages [1, 6, 40].

graph	size	subdivision	coarsening	local search	math prog	SS+ML
karate [42]	34	[29] .419	[41] .4198	[12] .4188	[1] .4197	.4197
dolphins [22]	62	[29] *.4893*	[31] *.5171*	[33] *.5285*	[40] .5285	.5276
polBooks [21]	105	[29] *.3992*	[37] *.5269*	[4] *.5204*	[1] .5272	.5269
afootball [14]	115	[39] .602	[41] .605	[4] *.6045*	[1] .6046	.6002
jazz [15]	198	[29] .442	[9] .4409	[12] .4452	[1] .445	.4446
celeg_metab [12]	453	[29] .435	[36] .450	[12] .4342	[1] .450	.4452
email [17]	1133	[29] .572	[9] .5569	[12] .5738	[1] .579	.5774
Erdos02 [16]	6927	[29] *.5969*	[32] .6817	[33] *.7094*		.7162
PGP_main [5]	11k	[29] .855	[9] .7462	[12] .8459		.8841
cmat03_main [25]	28k	[29] .723	[41] .761	[12] .6790		.8146
ND_edu [2]	325k		[7] .927	[4] .935		.9509

Table 1: Best published modularity values for four algorithm classes, compared to the modularity values for our heuristic SS+ML. Where possible, missing values were substituted with results from published implementations (shown in italics).

6.2 Published Modularity Values. Table 1 compares modularity values from various publications with the results of our heuristic SS+ML. Mathematical programming approaches consistently find better clusterings than SS+ML, though by a very small margin; however, they are computationally much more expensive and do not scale to large graphs [1, 40]. Compared to the best algorithms in the three other classes, the results of SS+ML are very competitive, and for large graphs significantly better.

6.3 Published Implementations. In order to directly compare our heuristics with existing algorithms, a range of publicly available implementations was retrieved from authors' websites and through the *igraph* library of Csárdi and Nepusz [8]. Only a subset of the graph collection could be used as some implementations cannot process graphs with weighted edges or self-edges. The employed 23 graphs range from a few to 75k vertices and are marked with *UW* in Appendix A. In some of these graphs negligible differences in edge weights and small amounts of self-edges were removed.

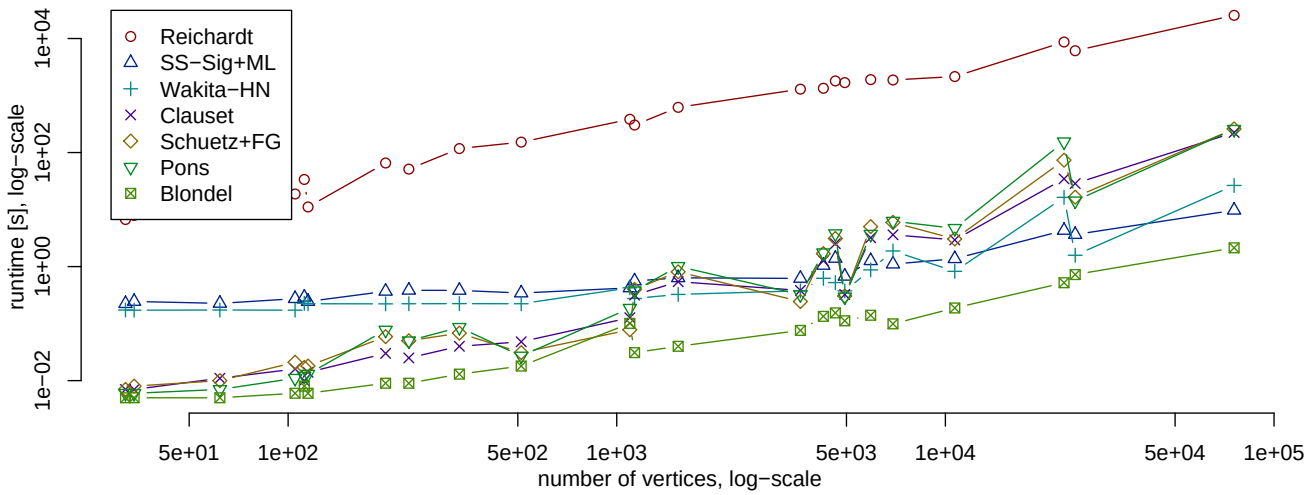

Figure 10: Runtime of the published implementations on unweighted graphs, log-log scaled.

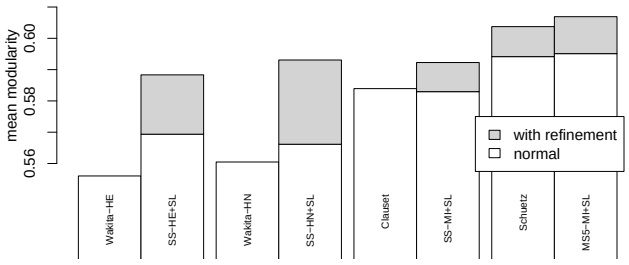

Figure 8: Mean modularities from four published implementations and our (approximate) reimplementations on unweighted graphs.

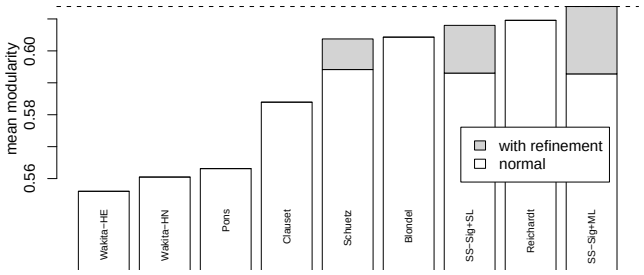

Figure 9: Mean modularities from the published implementations and our recommended heuristic *SS-Sig+ML* on unweighted graphs.

The included coarsening heuristics are the fast greedy joining of Clauset et al. [7], the algorithms of Wakita and Tsurumi [38], the recent multi-step greedy algorithm of Schuetz and Caflisch [36] (with parameter $l = 0.25\sqrt{f(V,V)/2}$, as recommended by Schuetz and Caflisch in [37]), and the algorithm of Pons and Latapy [31] based on short random walks (here of length 4). The examined local search heuristics are simulated annealing of Reichardt and Bornholdt [33] (here with at most 120 clusters) and the recent hierarchical algorithm of Blondel et al. [4].

Concerning the performance of the (approximately) reimplemented heuristics, the mean modularities from the published implementations are roughly reproduced or slightly improved by our implementations (Fig. 8) – even for Schuetz and Caflisch, where the computation of the parameter l differs (see Section 3.2). Note that refinement is not available in the implementations of Wakita and Tsurumi and of Clauset et al., and is optional in the implementation of Schuetz and Caflisch.

Concerning the performance of our recommended heuristic, Single-Step Greedy coarsening by Significance with Multi-Level Fast Greedy refinement (*SS-Sig+ML*), only Reichardt and Bornholdt's implementation produces clusterings of similarly high modularity, but it is much slower, and only Blondel et al.'s implementation is faster, but it produces worse clusterings (Figs. 9 and 10). Even the still simpler and faster variant with Single-Level refinement (*SS-Sig+SL*) produces competitive clusterings, notably in comparison with the recent algorithm of Schuetz and Caflisch which is more complex and requires parameter tuning (see Section 3.2).

7 Summary and Conclusion

Various coarsening and refinement heuristics for modularity clustering can be organized into a design space with four dimensions: merge fraction (including Single-Step and Multi-Step Greedy coarsening), merge prior-

itizer, refinement algorithm, and reduction factor (including Single-Level and Multi-Level refinement). In an experimental comparison of achieved modularities and runtimes, some widely used or rather complex design alternatives – for example, Multi-Step Greedy coarsening, merge prioritization by Modularity Increase, or Single-Level refinement – were outperformed by newly proposed or simpler alternatives – particularly Single-Step Greedy coarsening by Significance with Multi-Level Fast Greedy refinement. In a comparison with published implementations and benchmark results, this heuristic was more efficient than algorithms that achieved similar modularities, and achieved higher modularities than algorithms with similar or better efficiency.

References

[1] G. Agarwal and D. Kempe, *Modularity-maximizing graph communities via mathematical programming*, The European Physical Journal B, 66 (2008), pp. 409–418.

[2] R. Albert, H. Jeong, and A.-L. Barabási, *Diameter of the World-Wide Web*, Nature, 401 (1999), pp. 130–131.

[3] A. Arenas, A. Fernández, and S. Gómez, *Analysis of the structure of complex networks at different resolution levels*, New Journal of Physics, 10 (2008), p. 053039.

[4] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, *Fast unfolding of communities in large networks*, Journal of Statistical Mechanics: Theory and Experiment, (2008), p. P10008.

[5] M. Boguñá, R. Pastor-Satorras, A. Díaz-Guilera, and A. Arenas, *Models of social networks based on social distance attachment*, Physical Review E, 70 (2004), p. 056122.

[6] U. Brandes, D. Delling, M. Gaertler, R. Görke, M. Hoefer, Z. Nikoloski, and D. Wagner, *On modularity clustering*, IEEE Transactions on Knowledge and Data Engineering, 20 (2008), pp. 172–188.

[7] A. Clauset, M. E. J. Newman, and C. Moore, *Finding community structure in very large networks*, Physical Review E, 70 (2004), p. 066111.

[8] G. Csárdi and T. Nepusz, *The igraph software package for complex network research*, InterJournal Complex Systems, 1695 (2006).

[9] L. Danon, A. Díaz-Guilera, and A. Arenas, *Effect of size heterogeneity on community identification in complex networks*, Journal of Statistical Mechanics: Theory and Experiment, (2006), p. P11010.

[10] H. N. Djidjev, *A scalable multilevel algorithm for graph clustering and community structure detection*, in Proceedings of the 4th International Workshop on Algorithms and Models for the Web-Graph (WAW 2006), W. Aiello, A. Broder, J. Janssen, and E. Milios, eds., LNCS 4936, Springer-Verlag, 2008, pp. 117–128.

[11] L. Donetti and M. A. Muñoz, *Detecting network communities: a new systematic and efficient algorithm*, Journal of Statistical Mechanics: Theory and Experiment, (2004), p. P10012.

[12] J. Duch and A. Arenas, *Community detection in complex networks using extremal optimization*, Physical Review E, 72 (2005), p. 027104.

[13] M. Gaertler, *Clustering*, in Network Analysis: Methodological Foundations, U. Brandes and T. Erlebach, eds., LNCS 3418, Springer-Verlag, Berlin, 2005, pp. 178–215.

[14] M. Girvan and M. E. J. Newman, *Community structure in social and biological networks*, Proceedings of the National Academy of Sciences, 99 (2002), pp. 7821–7826.

[15] P. Gleiser and L. Danon, *Community structure in jazz*, Advances in Complex Systems, 6 (2003), pp. 565–573.

[16] J. Grossman, *The Erdös number project*. http://www.oakland.edu/enp/, 2007.

[17] R. Guimerà, L. Danon, A. Díaz-Guilera, F. Giralt, and A. Arenas, *Self-similar community structure in a network of human interactions*, Physical Review E, 68 (2003), p. 065103.

[18] B. Hendrickson and R. W. Leland, *A multi-level algorithm for partitioning graphs*, in Proceedings of the ACM/IEEE Supercomputing Conference (SC 1995), 1995.

[19] G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM Journal on Scientific Computing, 20 (1998), pp. 359–392.

[20] B. Kernighan and S. Lin, *An efficient heuristic procedure for partitioning graphs*, Bell System Technical Journal, 49 (1970), pp. 291–307.

[21] V. Krebs, *A network of books about recent US politics sold by the online bookseller amazon.com*. http://www.orgnet.com/, 2008.

[22] D. Lusseau, K. Schneider, O. J. Boisseau, P. Haase, E. Slooten, and S. M. Dawson, *The bottlenose dolphin community of Doubtful Sound features a large proportion of long-lasting associations*, Behavioral Ecology and Sociobiology, 54 (2003), pp. 396–405.

[23] C. P. Massen and J. P. K. Doye, *Identifying communities within energy landscapes*, Physical Review E, 71 (2005), p. 046101.

[24] A. Medus, G. Acuña, and C. O. Dorso, *Detection of community structures in networks via global optimization*, Physica A, 358 (2005), pp. 593–604.

[25] M. E. J. Newman, *The structure of scientific collaboration networks*, Proceedings of the National Academy of Sciences, 98 (2001), pp. 404–409.

[26] ———, *Analysis of weighted networks*, Physical Review E, 70 (2004), p. 056131.

[27] ———, *Fast algorithm for detecting community structure in networks*, Physical Review E, 69 (2004), p. 066133.

[28] ———, *Finding community structure in networks using the eigenvectors of matrices*, Physical Review E, 74 (2006), p. 036104.

[29] ———, *Modularity and community structure in net-*

works, Proceedings of the National Academy of Sciences, 103 (2006), pp. 8577–8582.

[30] M. E. J. Newman and M. Girvan, *Finding and evaluating community structure in networks*, Physical Review E, 69 (2004), p. 026113.

[31] P. Pons and M. Latapy, *Computing communities in large networks using random walks*, Journal of Graph Algorithms and Applications, 10 (2006), pp. 191–218.

[32] J. M. Pujol, J. Béjar, and J. Delgado, *Clustering algorithm for determining community structure in large networks*, Physical Review E, 74 (2006), p. 016107.

[33] J. Reichardt and S. Bornholdt, *Statistical mechanics of community detection*, Physical Review E, 74 (2006), p. 016110.

[34] R. Rotta, *A multi-level algorithm for modularity graph clustering*, Master's thesis, Brandenburg University of Technology, 2008.

[35] S. E. Schaeffer, *Graph clustering*, Computer Science Review, 1 (2007), pp. 27–64.

[36] P. Schuetz and A. Caflisch, *Efficient modularity optimization by multistep greedy algorithm and vertex mover refinement*, Physical Review E, 77 (2008), p. 046112.

[37] ———, *Multistep greedy algorithm identifies community structure in real-world and computer-generated networks*, Physical Review E, 78 (2008), p. 026112.

[38] K. Wakita and T. Tsurumi, *Finding community structure in mega-scale social networks*. Preprint arXiv:cs/0702048, 2007.

[39] S. White and P. Smyth, *A spectral clustering approach to finding communities in graphs*, in Proceedings of the 5th SIAM International Conference on Data Mining (SDM 2005), SIAM, 2005, pp. 274–285.

[40] G. Xu, S. Tsoka, and L. G. Papageorgiou, *Finding community structures in complex networks using mixed integer optimisation*, The European Physical Journal B, 60 (2007), pp. 231–239.

[41] Z. Ye, S. Hu, and J. Yu, *Adaptive clustering algorithm for community detection in complex networks*, Physical Review E, 78 (2008), p. 046115.

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

A The Benchmark Graph Collection

Table 2 on the next page lists graphs used for the experiments. The graphs postfixed with '_main' just contain the largest connectivity component of the original graph. All graphs from the subset 'UW' were used without edge weights and self-edges for the experiments on published implementations. For each graph the source collection is named in the last column. Web addresses to these collections are listed in Table 3. For information about the original authors please visit the respective websites.

	subset	vertices	edges	edge weight	type	source
SouthernWomen	UW	32	89	89.0	social	pajek
karate	UW	34	78	78.0	social	Newman
football		35	118	295.0	economy	pajek
morse		36	666	25448.0	similarity	ANoack
Food		45	990	11426.0	similarity	ANoack
dolphins	UW	62	159	159.0	social	pajek
WorldImport1999		66	2145	4367930.4	economy	ANoack
lesmis		77	254	820.0	social	Newman
world_trade		80	875	65761594.0	economy	pajek
A00_main		83	135	135.0	software	GraphDrawing
polBooks	UW	105	441	441.0	similarity	pajek
adjnoun	UW	112	425	425.0	linguistics	Newman
afootball	UW	115	613	616.0	social	Newman
baywet		128	2075	3459.4	biology	pajek
jazz	UW	198	2742	5484.0	social	Arenas
SmallW_main	UW	233	994	1988.0	citation	pajek
A01_main		249	635	642.0	citation	pajek
celegansneural		297	2148	8817.0	biology	Newman
USAir97	UW	332	2126	2126.0	flight	pajek
netscience_main		379	914	489.5	co-author	Newman
WorldCities_main		413	7518	16892.0	social	pajek
A03		423	578	578.0	biology	GraphDrawing
celeg_metab		453	2040	4596.0	biology	Arenas
USAir500		500	2980	453914166.0	flight	Cx-Nets
s838	UW	512	819	819.0	technology	UriAlon
Roget_main		994	3641	5059.0	linguistics	pajek
SmaGri_main		1024	4917	4922.0	citation	pajek
A96	UW	1096	1677	1691.0	software	GraphDrawing
email	UW	1133	5451	10902.0	social	Arenas
polBlogs_main		1222	16717	19089.0	citation	pajek
NDyeast_main		1458	1993	1993.0	biology	pajek
Java	UW	1538	7817	8032.0	software	GraphDrawing
Yeast_main		2224	7049	7049.0	biology	pajek
SciMet_main		2678	10369	10385.0	citation	pajek
ODLIS_main		2898	16381	18417.0	linguistics	pajek
DutchElite_main	UW	3621	4310	4311.0	economy	pajek
geom_main		3621	9461	19770.0	co-author	pajek
Kohonen_main		3704	12675	12685.0	citation	pajek
Epa_main	UW	4253	8897	8953.0	web	pajek
eva_main		4475	4654	4664.0	economy	pajek
PPI_SCerevisiae_main	UW	4626	14801	29602.0	biology	Cx-Nets
USpowerGrid	UW	4941	6594	13188.0	technology	pajek
hep-th_main		5835	13815	13674.6	citation	Newman
California_main	UW	5925	15770	15946.0	web	pajek
Zewail_main		6640	54174	54244.0	citation	pajek
Erdos02	UW	6927	11850	11850.0	co-author	pajek
Lederberg_main		8212	41436	41507.0	citation	pajek
PairsP		10617	63786	612563.0	similarity	pajek
PGP_main	UW	10680	24316	24340.0	social	Arenas
DaysAll		13308	148035	338706.0	similarity	pajek
foldoc		13356	91471	125207.0	linguistics	pajek
astro-ph_main		14845	119652	33372.3	co-author	Newman
as-22july06	UW	22963	48436	48436.0	web	Newman
eatRS		23219	305501	788876.0	linguistics	pajek
DIC28_main	UW	24831	71014	71014.0	linguistics	pajek
hep-th-new_main		27400	352059	352542.0	co-author	pajek
cmat03_main		27519	116181	60793.1	co-author	Newman
wordnet3_main	UW	75606	120472	131780.0	linguistics	pajek

Table 2: Graph collection.

source	web address
Arenas	`http://deim.urv.cat/~aarenas/data/welcome.htm`
ANoack	`http://www-sst.informatik.tu-cottbus.de/~an/GD/`
Cx-Nets	`http://cxnets.googlepages.com/`
GraphDrawing	`http://vlado.fmf.uni-lj.si/pub/networks/data/GD/GD.htm`
Newman	`http://www-personal.umich.edu/~mejn/netdata/`
pajek	`http://vlado.fmf.uni-lj.si/pub/networks/data/`
UriAlon	`http://www.weizmann.ac.il/mcb/UriAlon/`

Table 3: Graph sources.