universität
wien

Bachelorarbeit

zur Erlangung des akademischen Grades

Bachelor of Science

an der Fakultät für Informatik
der Universität Wien

Evolutionary Graph Clustering

Eingereicht bei:	Univ.-Prof. Dr. Monika Henzinger Research Group Theory and Application of Algorithms
Betreuer:	Dipl.-Math. Dipl.-Inform. Dr. Christian Schulz
Vorgelegt von:	Sonja Biedermann Matrikelnummer 01402891

Wien, den 14. September 2017

Abstract

Graph clustering is the detection of tightly connected regions in a graph. A clustering may yield structural information about the graph that is especially valuable in this data driven age.

As graph clustering is considered to be an NP-hard optimization problem, the effort of accurately computing the best solution clustering for larger graphs is intractable. On these grounds, heuristics that search for an acceptable solution are used to find candidate solutions to graph clustering problems while having limited knowledge about the problem and not exceeding time limitations.

Evolutionary algorithms are a type of metaheuristic optimization algorithm and search for solutions by applying mechanisms of evolution, and have been proven to be a good choice for NP-complete problems.

In this thesis, we take on defining and implementing an evolutionary algorithm that aims to improve on candidate solutions found by a base algorithm. We evaluate our algorithm by comparing our results to the results achieved in a graph clustering benchmark.

Zusammenfassung

Unter Graphclustering versteht man das Auffinden von besonders stark verbundenen Regionen in einem Graphen. Ein Clustering kann strukturelle Informationen über den Graphen zu Tage bringen und ist somit vor allem im heutigen hoch datenorientierten Zeitalter von großem Interesse.

Da Graphenclustering zu den NP-harten Optimierungsproblemen zählt, ist der Aufwand für eine exakte Berechnung des besten Clusterings für größere Graphen nicht vertretbar. Deshalb kommen Heuristiken zum Einsatz, die in vertretbarer Zeit und mit stark begrenztem Wissen dennoch eine annehmbar gute Lösung suchen.

Evolutionäre Algorithmen zählen zu den Metaheuristiken und lösen Probleme hinreichend durch die Entlehnung von Mechanismen aus der Evolution, und haben sich besonders bei der Lösung von NP-vollständigen Optimierungsproblemen etabliert.

In dieser Thesis beschäftigen wir uns mit der Definition und Implementierung eines Evolutionären Algorithmus, der sich der sich aufbauend auf einer initialen, von einem Basisalgorithmus generierten Lösungskandidatenmenge dem Finden einer noch besseren Lösung widmet. Wir evaluieren unseren Algorithmus indem wir die von uns erzielten Resultate mit den erzielten Resultaten aus einem Graphclustering Benchmark vergleichen.

Contents

1 Introduction

Graph clustering is the problem of detecting tightly connected regions of a graph. Depending on the task, knowledge about the structure of the graph can reveal information such as voter behavior, formation of new trends, existing terrorist groups and recruitment [37] or a natural partitioning of data records onto pages [15]. Further application areas include the study of protein interaction [33], gene expression networks [44], fraud detection [4], program optimization [24, 14] and the spread of epidemics [27]—possible applications are plentiful, as almost all systems containing interacting or coexisting entities can be modeled as a graph. Graph clustering is especially relevant in the age of Big Data.

A clustering is typically realized as a mapping of vertex IDs to cluster IDs, usually as a vector c in which every index i is a vertex ID, and c_i is the cluster ID of that vertex. Unlike with other related problems that search for a partition of vertices that satisfies an input, the number of clusters is not predefined. A clustering algorithm will generally start with an extreme clustering—such as a clustering consisting of only one cluster containing all vertices, or a clustering in which every cluster consists of only one vertex—and then apply changes to the clustering until a clustering is found that best fits the graph, which is quantified by an objective function. As such, graph clustering is an combinatorial optimization problem.

1.1 Motivation

Many heuristics optimizing a quality measure known as *modularity* have been proposed in the last decade [28, 13, 30, 29, 42, 8, 32, 35], aiming to produce better clusterings in less time. Although it has been proven that modularity maximization has no claim to any approximation guarantee [9], this approach has proven popular and capable of producing good clusterings. Graph clustering by modularity maximization is NP-hard [9], however some of the known heuristics are quite fast and known to yield reasonably good results.

A particularly interesting class of heuristics are evolutionary algorithms, which evolve a set of competing solutions in a given environment which is defined by the problem. Individual solutions are selected according to their fitness, combined and mutated until some halting

criterion is reached. By imposing a pressure that allows good individuals to reproduce, thus passing on their good genes, while the same is denied to worse individuals, better solutions emerge.

Evolutionary algorithms for graph clustering have been proposed [39, 20, 23], but we know of no evolutionary algorithm that makes use of the work presented in [8] and [32], and we did not find any benchmarking results that compare evolutionary graph clustering algorithms to other heuristic algorithms. In this thesis, we will build upon the aforementioned work and then compare it against benchmark values achieved by various top-of-the-field graph clustering algorithms.

1.2 Contribution

We propose an evolutionary graph clustering algorithm that maximizes modularity. The algorithm makes use of four recombination operators, as well as a mutation operator and an additional refinement routine. The population is built using the Louvain method [8], and recombination operators make use of routines such as maximum overlap described in [32], as well as SCLP [25] and highly balanced graph partitioning [36].

Our algorithm is competitive, and obtains results that are at least as good as the results of the benchmarking done in the context of the 10th DIMACS implementation challenge [7] in 91% of cases.

1.3 Structure of Thesis

In Chapter 2, we review basic definitions and notation used, as well as introduce the problems of graph partitioning and clustering. In Chapter 3, we survey the current state-of-the-art graph clustering methods, as well as outline some of the evolution of the corresponding algorithms. Next, we discuss our proposed algorithm in Chapter 4. Then, we evaluate our algorithm in Chapter 5, tune parameters and benchmark against other well-known graph clustering algorithms, and finally we discuss our findings and conclude this thesis in Chapter 6.

A short overview over the implementation done as part of this thesis is given in Appendix A, and a more detailed overview of the experimental data that arose during evaluation is shown in Appendix B.

2 Preliminaries

In this chapter, we introduce the core concepts of this thesis. Moreover, we introduce the problem of graph partitioning and clustering, as well as some basic definitions and notations used in this thesis.

2.1 Definitions and Notation

Graphs. A graph G is a two-tuple (V, E) with V being the set of *vertices*, and E the set of *edges*. An edge is a tuple (u, v) with $u \in V$ and $v \in V$. An edge can be *directed* or *undirected*, and it can be *weighted*. An undirected edge is written as a set $\{u, v\}$. A weighted graph is written as $G = (V, E, \omega)$, with $\omega : E \to \mathbb{R}_{>0}$. A vertex can be weighted as well; a graph containing weighted edges and weighted vertices is written as $G = (V, E, \omega, c)$ with $c : V \to \mathbb{R}_{>0}$. We will be mostly dealing with undirected and unweighted graphs in this thesis. The number of vertices in a graph is $|V|$, commonly referred to as n, the number of edges is $|E|$, commonly written as m. The *neighborhood* of a vertex v is $\Gamma(v) = \{ w \mid \{v, w\} \in E \}$. The *degree* of a vertex is the number of its neighbors, $deg(v) = |\Gamma(v)|$. A graph is called *complete* iff $\forall u, v \in V$, $\{u, v\} \in E$, i.e. for all pairs of vertices, an edge connecting them exists. The *induced subgraph* $G[V']$ of some graph G is a graph defined by a set $V' \subset V$ of vertices and all edges $E' = \{ \{u, v\} \in E \mid u, v \in V' \}$.

Cliques. A clique is a subgraph such that every two vertices u and v are *connected*, i.e. the induced subgraph is complete. A clique is a prime example of a cluster, especially if the clique is weakly connected to other parts of the graph.

Cuts. A cut of a graph (S, T) with $T = V - S$ is a partition into two parts S and T. An edge $\{v, w\}$ *crosses the cut* if $v \in S$ and $w \in T$. The set of cut edges, called the *cut-set*, is defined as $E_C = \{ \{v, w\} \in E \mid v \in T, w \in S \}$. The *cut function* is $cut(S, T) = \sum_{\{v,w\} \in E_C} \omega(\{v, w\})$, i.e. the sum of all weights of crossing edges. A cut can be generalized for k parts. The cut is typically minimized in graph partitioning and clustering.

2.2 Graph Partitioning

Graph partitioning pertains to the segregation of a graph into a given number of chunks. It is a combinatorial optimization problem, parameterized by k, the number of parts, and ε, a balance parameter, such that the relation

$$\max_i |V_i| \leq (1+\varepsilon) \left\lceil \frac{|V|}{k} \right\rceil \tag{2.1}$$

holds (which is the balance constraint), while minimizing the cut of the partition, i.e. having as few edges cross the resulting cut as possible. It is important to note that both k and ε must be known beforehand, i.e. they are input parameters that must be supplied to the algorithm. Picking a good k, if not already given by exterior circumstances, when the graph structure is unknown is a problem unto itself.

A need for partitioning arises, for example, when distributing computer processes onto a number of processors on different physical machines [29]. The communications between processes can be modeled as a graph. The goal would be that every node has roughly the same workload and to minimize the need for communication between computer nodes; that is, to also group processes that need to communicate onto the same physical machine to avoid tedious inter-machine communication. A fitting partition would be calculated using a graph partitioning algorithm on the communication graph. This assumes that all computer nodes have the same capabilities.

However, finding an algorithm for an exact solution to this problem that scales up nicely proves difficult; graph partitioning is believed to be NP-hard [18] and as such, only approximations and heuristics are possibly feasibly computable. It turns out that approximating this problem is NP-hard [11], as well, which only leaves heuristics for sufficiently large graphs. This may seem damning at first, but in truth means that it is an interesting problem since the potential for improvement is large.

2.3 Graph Clustering

A related problem is that of graph clustering. While graph partitioning works by partitioning a graph into a given number of chunks of roughly same sizes, this constraint is not to be found in graph clustering. Instead, the goal is to detect strongly connected groups of vertices within the graph, i.e. groups that have a significantly higher edge-density within themselves than towards the outside. Figure 2.1 shows an example for a clustering of a small graph. The minimum cut

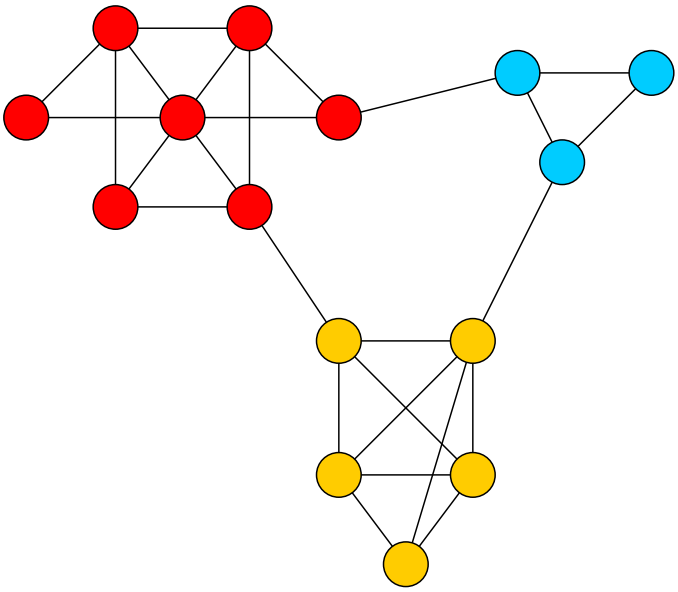


Figure 2.1: A clustering of a graph

constraint holds, but the clusters are of different sizes. Graph clustering is often referred to as community detection in the spirit of social networks.

Graph clustering is an important technique for analyzing and understanding graph structures, where graph partitioning falls short due to the imposition of a given number of parts, which may or may not—most likely not—correspond to the actual structure native to the graph [29].

Formalizing the idea that a cluster has a high intra- to inter-cluster edge ratio, we gain the definition of *coverage* [10]. The coverage of a clustering is exactly that: the ratio of the sum of all edges inside a cluster to all edges in the graph. Ideally, following our idea of strongly connected clusters with weak connections to other clusters, we would expect that the number of intra-cluster edges is very close to the number of edges in total, as the set of inter-cluster edges is expected to be small. The coverage of a clustering is thus defined as

$$cov(\mathcal{C}) = \frac{1}{|E|} \sum_{c \in \mathcal{C}} |E_c|, \tag{2.2}$$

where $\mathcal{C}$ is the clustering, i.e. the whole mapping of a vertex to some cluster. c is a cluster within this clustering, and $E_c = \{\, \{u, v\} \mid \{u, v\} \in E \wedge \mathcal{C}(u) = \mathcal{C}(v) = c \,\}$, i.e. all edges running within a cluster. The closer the value is to 1, the better the clustering—purportedly.

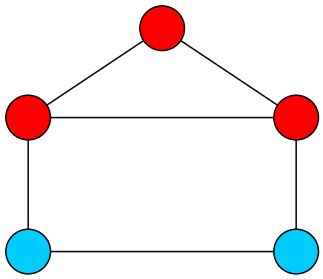


Figure 2.2: An example

While this definition nicely encapsulates our now established idea of a good clustering, it strongly prefers a trivial clustering where all vertices are in the same be-all-end-all cluster. Such a clustering would be awarded the highest quality, but is obviously not useful.

Modularity [29] punishes the trivial clustering quality-wise by comparing the coverage of the given clustering to the hypothetical coverage of applying the clustering to a graph with the same degrees, but with randomized edges. On a high level, modularity is defined as

$$mod(\mathcal{C}) = cov(\mathcal{C}) - \mathbb{E}[cov(\mathcal{C})] \tag{2.3}$$

and the expected value of $cov(\mathcal{C})$ is given as

$$\mathbb{E}[cov(\mathcal{C})] = \frac{1}{4m^2} \sum_{c \in \mathcal{C}} \left(\sum_{v \in c} deg(v) \right)^2 . \tag{2.4}$$

This adequately gets rid of the issue with trivial clusterings, which now are of quality 0. Modularity is a popular measure of quality, but is by far not the only one and is not without issues. One such issue is the resolution problem—small clusters tend to be unified with other clusters, as small clusters have few edges and thus also little contribution to modularity—this occurs even if the clusters are cliques and only connected by a single edge to the rest of the graph [16, 22], which could not be more indicative of a cluster. Modifications have been proposed as this may be undesirable if the given graph is such that this problem manifests itself, however we will be using the unmodified modularity for our purposes.

Figure 2.2 shows a small example graph for illustrating modularity. The clusters are given by the vertex colors. This clustering has a coverage of $cov(\mathcal{C}) = \frac{2}{3}$ and its first moment is $\mathbb{E}[cov(\mathcal{C})] = \frac{5}{9}$. The modularity is thus $mod(\mathcal{C}) = \frac{1}{9}$.

3 Related Work

In this chapter, we survey the state-of-the-art in graph clustering by taking a closer look at a few clustering approaches and algorithms. We study global clustering approaches, both agglomerative and divisive, followed by the local clustering approach. Then, we draw attention to a few algorithms that were used in our work, such as the Louvain method and Label Propagation. We finish by examining a few examples of the use of evolutionary algorithms in connection with graph clustering.

3.1 Global Clustering Approaches

When clustering a graph, two different viewpoints can be taken. The first one is to include the whole graph into the clustering effort, i.e. consider all edges while calculating a solution. This can be effective, but is also expensive in both time and space, so much so that just storing the needed information becomes intractable for sufficiently large graphs [34, 37].

3.1.1 Agglomerative Clustering

Agglomerative clustering pertains to an hierarchical approach where communities are continuously merged until only one community remains. This is an intuitive approach especially for human networks [29, 34], which usually have hierarchical structures, consider e.g. a company with departments and sub-departments, or a family tree. The resulting hierarchy of clusters can be represented by a dendrogram [34]. Such a dendrogram is especially valuable if information about sub-cluster structures is of interest. Figure 3.1 shows such a dendrogram along with the graph. The dashed green line marks the clustering with the highest fitness according to modularity, which is also shown on the graph on the left side.

Vertices are merged based on some similarity metric. All edges are removed from the graph and gradually added back to the graph between vertices that are most similar to each other. This approach is especially evident for networks that already contain some information pertinent to similarity, or where a similarity metric can be easily derived (e.g. clustering a network of actors, where actors that often co-star on movies can be regarded as similar [29]).

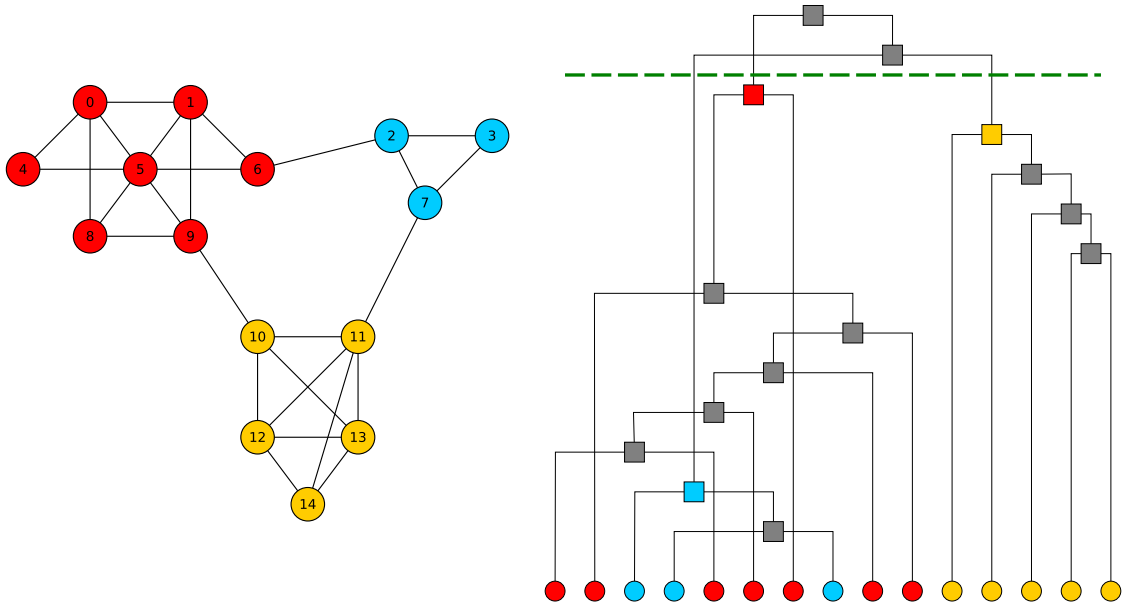


Figure 3.1: Dendrogram of an agglomerative clustering algorithm

However, it is known [31] that agglomerative clustering approaches sometimes fail to come up with good clustering configurations for networks with known structure. Another problematic characteristic is that such methods often fail to classify fringe vertices [29], i.e. vertices that are not in the core but still obviously part of the cluster, as part of the (expected) community.

Greedy Modularity Maximization

We have already seen that modularity is an apt fitness measure for a given clustering. It stands to reason that it would be possible to gradually merge clusters such that the modularity increases to gain an hierarchical clustering. Such was devised in [28] by Newman, where the author outlines a simple but fast algorithm that does precisely that.

This algorithm (outlined in Algorithm 3.1.1) works by optimizing the modularity Q over a small subset of possible divisions, as exhaustively computing all possible divisions is exceedingly costly. It starts with singleton clusters (i.e. clusters containing only one vertex) and then looks for whichever clusters would be most beneficial (i.e. would result in the highest increase in Q) to merge. It is of note that since only clusters that are directly connected to each other could result in an increase of modularity, not all combinations of clusters need to be looked at. The number of connected clusters to some cluster is in the worst case $O(m)$, and the updating of the modularity housekeeping takes at worst $O(n)$ time, which makes every iteration of complexity

Algorithm 3.1.1 General form of greedy modularity maximization

```
procedure GREEDY-MODMAX(G)
    C ← initialize all vertices to a singleton cluster
    while |C| ≠ 1 do
        for all pairs in { (c_i, c_j) | c_i, c_j ∈ C } do
            calculate ΔQ if c_i and c_j were to be merged
        end for
        c_i, c_j ← pick pair with the largest ΔQ
        MERGE(C, c_i, c_j)
    end while
end procedure
```

$O(m+n)$. Since $O(n)$ merges are necessary to arrive at the trivial clustering, this algorithm is in $O((m+n)\,n)$.

As it turns out, the run-time complexity can be lowered by using more sophisticated modularity housekeeping [13]. For this, we will need a matrix E, in which element e_{ij} counts the number of edges that span from community i to community j, diagonal elements being intra-cluster edges, and a vector a, in which element a_i counts the sum of all degrees of vertices in community i, i.e. the number of edges that are attached to any vertex in this community. Using this, modularity can be calculated as

$$Q = \sum_i (e_{ii} - a_i^2) \tag{3.1}$$

and the change in modularity upon merging community i and j can be calculated by

$$\Delta Q_{ij} = \begin{cases} \frac{1}{2m} - \frac{k_i k_j}{4m^2} & \text{if } i \text{ and } j \text{ are connected} \\ 0 & \text{otherwise} \end{cases} \tag{3.2}$$

where k_i is defined by

$$a_i = \frac{k_i}{2m}. \tag{3.3}$$

The largest ΔQ_{ij} for all connected communities i, j can then be calculated and stored in a max heap for logarithmic retrieval of the next best merging pair. When communities i and j are merged, only the ΔQ_{jk} where k is connected to either i, j, or both need to be updated. This results in a complexity of $O(md \log n)$ [13], where d is the depth of the dendrogram. This version of the algorithm is referred to as CNM, named after the authors. Another advantage

is that as soon as the maximum ΔQ_{ij} is negative, the algorithm can be aborted instead of merging until only one community remains [13, 9].

Unfortunately, while this does make the algorithm run very fast, it does not fix the problems inherent with greedy modularity maximization. As can be seen in Figure 3.1, which was generated using the naïve algorithm, large clusters tend to be merged with much smaller clusters, i.e. merges are very unbalanced, which causes some regions of the graph to be quickly clumped into one large cluster, while other regions remain untouched until late into the algorithm. This introduces a bias for bad clustering decisions [31], but also introduces a run-time overhead which tampers with the suggested scalability of CNM, as found by Wakita and Tsurumi [43].

To fix the balance issues, Ovelgönne and Geyer-Schulz [30] devised a version of this algorithm which only samples k of all clusters and then looks for the best pair to merge of which one has been k-sampled. This successfully prevents unbalanced merges [30, 31].

3.1.2 Divisive Clustering

Divisive clustering is somewhat similar to agglomerative clustering, but instead of merging communities until only one remains, the reverse approach is taken: splitting until every vertex is alone by itself in a community, i.e. the starting and ending cluster configurations are swapped in agglomerative vs. divisive clustering. This process again results in a hierarchy that can be represented by a dendrogram.

One possibility is to successively remove certain edges until no more suitable edges remain. Usually, the inverse similarity between vertices is used, i.e. edges between dissimilar vertices will be removed, but other measures are also possible and might even be advantageous, as we shall see shortly.

Algorithm of Girvan-Newman

The algorithm of Girvan-Newman (from here on GN for short) is a divisive greedy algorithm that works by removing edges. The measure by which the edges are selected, however, is not based on similarity of vertices but on a measure called *betweenness*. An edge with high betweenness shall be an edge that connects two communities [29], thus, an edge with low betweenness lies inside a community.

The original definition of betweenness [17] pertains to vertices, and is based in sociometry. A person that has a central role in social interaction networks stands *between* other people, i.e. the

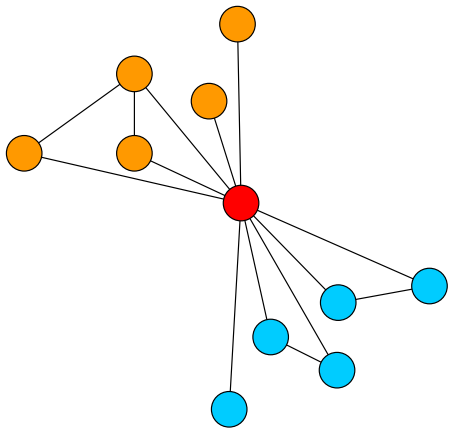


Figure 3.2: An example illustrating betweenness

corresponding vertex of the graph modeling such a situation would have high betweenness. Such choke points of social interaction are of special interest when studying personal relationships. Figure 3.2 shows an example. The red node has high betweenness.

For clustering, this definition must be generalized to edges. This general definition of betweenness is then called the *shortest path betweenness* by the authors [29] and is calculated by finding the shortest paths between all pairs of vertices and then, for each edge in the graph, counting how many of these paths contain this edge. This can be calculated in $O(mn)$ time based on breadth-first search [29]. The shortest path betweenness can also be derived using resistor networks, and random walks.

Using this betweenness measure, the basic outline of the algorithm is as given in Algorithm 3.1.2. This algorithm has a run-time of $O(m^2n)$ for dense graphs [29], which is intractable for large problem instances ($n \geq 10\,000$). Unfortunately, large problem instances are the de facto standard for clustering problems in the age of Big Data.

3.1.3 Graph Clustering by Flow Simulation

A random walk in G that visits a densely connected cluster will likely not leave that cluster until many nodes have been visited—that is the single key premise of clustering by stochastic flow simulation. This is an interesting approach since it is devoid of any procedural rules for actually finding the clusters (e.g. by looking at similarities or some such), but rather finds the clusters by applying two alternating operators onto a matrix until convergence is reached and then analyzes the resulting data, which is rather elegant.

Algorithm 3.1.2 General form of the GN algorithm

```
procedure GN(G)
    for all e ∈ E do
        calculate betweenness score of e
    end for
    while E ≠ ∅ do
        e ← edge with highest betweenness
        E ← E − e
        for all e ∈ E do
            recalculate betweenness score of e
        end for
    end while
end procedure
```

The input graph is represented as a Markov chain, which then models the random walk across this graph. A Markov chain is a memoryless stochastic process, that is, a process in which future states only depend on the present states and not on any states past. A Markov chain is characterized by its transition matrix T, where each element t_{ij} is defined as the probability to switch to state i if the current state is j, i.e. T is a *column stochastic matrix*. The states for a random walk across some graph G are its vertices, and the probabilities of state changes $v \to w$ are expressed as

$$p_{vw} = \begin{cases} deg(v)^{-1} & \text{if } w \in \Gamma(v), \\ 0 & \text{otherwise} \end{cases} \tag{3.4}$$

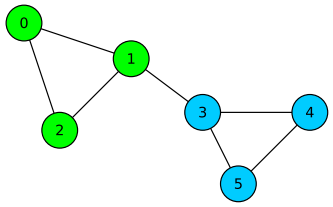
for an unweighted graph, i.e. any outgoing edge is picked uniformly by chance. The probability to be at some state w, presently being in state j, after k time units is calculated as $T^k e_j[w]$.

Markov chain clustering (also Markov clustering, or MCL) was first introduced in [42] by van Dongen. As mentioned, MCL uses two operators: expansion and inflation. Both are parameterizable by a parameter e and r, respectively.

Expansion is the exponentiation of the transition matrix by some e [42], i.e. calculation of the state probabilities after e time units. When the transition matrix is continuously multiplied by itself, the probabilities will spread out, some of them increasing while others decrease. This trend—towards or away from 0—is what makes this algorithm work, but is not pronounced enough on its own. This is where inflation comes into play. Inflation is the element-wise exponentiation (followed by a re-scaling to preserve the column sums) by some $r \geq 2$ and

Algorithm 3.1.3 Outline of the MCL algorithm

procedure MCL(T, e, r) ▷ T is a stochastic matrix with self loops already included.
  $T_k \leftarrow 0$
  $T_{k+1} \leftarrow T$
  repeat
    $T_k \leftarrow T_{k+1}$
    $T_{k+1} \leftarrow \textsc{Inflate}(\textsc{Expand}(T_k, e), r)$
  until $T_k \approx T_{k+1}$
end procedure



$$T = \begin{bmatrix} \frac{1}{3} & \frac{1}{4} & \frac{1}{3} & 0 & 0 & 0 \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{3} & \frac{1}{4} & 0 & 0 \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{3} & 0 & 0 & 0 \\ 0 & \frac{1}{4} & 0 & \frac{1}{4} & \frac{1}{3} & \frac{1}{3} \\ 0 & 0 & 0 & \frac{1}{4} & \frac{1}{3} & \frac{1}{3} \\ 0 & 0 & 0 & \frac{1}{4} & \frac{1}{3} & \frac{1}{3} \end{bmatrix}$$

Figure 3.3: A smaller example clustering with its stochastic matrix

models the contraction of a flow over the graph, which amplifies the trend towards or away from 0 [42, 2]. MCL is outlined in Algorithm 3.1.3.

As the main component at work here is the multiplication of a $n \times n$ matrix by itself, naïve MCL has a complexity of $O(n^3)$. However, since the matrix will quickly become sparse, this complexity can be lowered to $O(n^2 k)$ where k is the number of non-zero elements (called *resources* [2] in MCL lingo) with some simple pruning and sparse matrix multiplication, and even $O(nk^2)$ with additional measures [2] which are already quite involved implementation-wise. According to the author, k can be picked quite low without significantly compacting the quality of the resulting clustering [2, 41], which makes this algorithm quite competitive indeed, especially considering its simplicity. However, MCL takes great issue with directed graphs (transition matrices must be symmetric) and has been shown to sometimes bring forth degenerate clusterings.

Figure 3.3 shows a small example. The cluster structure is already quite obvious in this matrix due to the vertex numbering. Notice the self-loops that must be added in order for the algorithm to run correctly, and how the columns sum to 1. The matrix we get after letting MCL run with $e = 2$ and $r = 2$ is the following:

$$R = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix},$$

from which we can deduce that MCL found two clusters, one consisting of nodes $c_1 = \{0, 1, 2\}$ and the other one consisting of $c_2 = \{3, 4, 5\}$.

3.2 Local Clustering Approaches

We have now seen some global clustering algorithms, but all of them fail on graphs that are so large that they cannot be kept fully in memory, which grinds even the fastest of them to a halt. Local clustering algorithms work around this by only requiring information about the neighborhood of some given vertex as they "crawl" through the graph.

3.2.1 Local Search

Local search is the main proponent of local clustering and pertains to visiting vertices in a random order (akin to a random walk, but with equal probabilities for all vertices) and then evaluating for every adjacent cluster whether an increase in quality could be attained by moving the current vertex to this cluster [8, 37]. Due to this locality, local search is only ever able to find local optima, as whatever seems to be beneficial locally need not be beneficial towards reaching the global optimum. This makes using different iterated refinement strategies on top of local search essential if good results are to be achieved. Figure 3.4 shows an example, highlighting a node that is about to be moved into the green cluster.

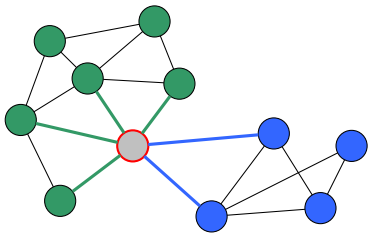
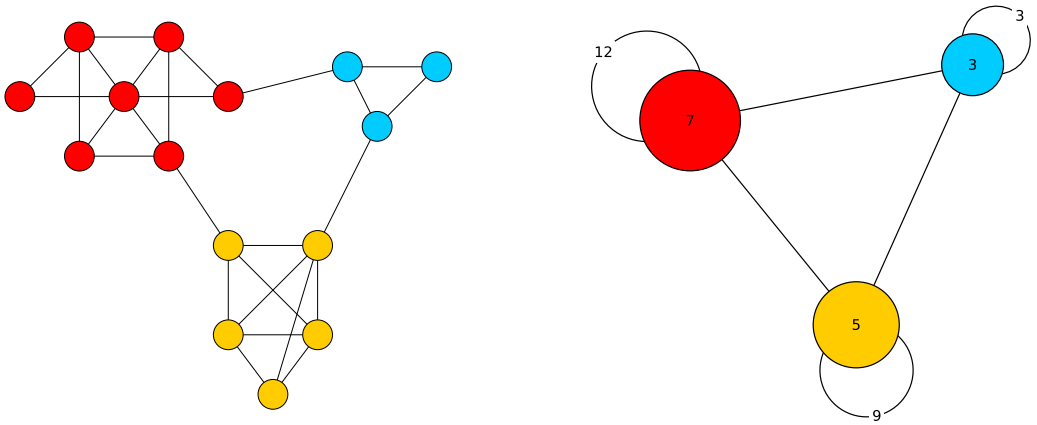


Figure 3.4: An example illustrating local search



(a) (b)

Figure 3.5: Contracted graph (b) induced by clustering (a)

3.3 Louvain Method

The Louvain method is a multilevel clustering algorithm introduced by Blondel et al [8]. The main component is local search, followed by a "zooming out" in form of a graph contraction. Figure 3.5 shows a contracted graph induced by a clustering. Note that the node weight denotes the cluster size, and edges have been collated. Intra-cluster edges have turned into self loops of a given weight. Contraction is also commonly referred to as coarsening, especially in multilevel graph partitioning schemes.

These two phases are repeated until no more changes were made during local search. The modularity is calculated with regards to node and edge weight. Due to this, many virtual nodes can be moved in just one iteration of local search on the contracted graph, which allows for much more daring changes in the cluster structure and thus potentially big jumps in quality. Finally, the clustering of the coarsest graph is projected back onto the original.

This method is especially interesting due to its speed. On most graphs, only very few passes are necessary, and the resulting clusterings are of rather good quality, as evaluated [21] by Lancichinetti and Fortunato. Another worthwhile feature is that due to the use of local search, which only ever moves single nodes (although one node may represent whole clusters of a previous iteration), the resolution-limit problem of modularity is partially circumvented [8]. All this makes the Louvain method a very attractive routine for generating initial partitions to be refined by some other, perhaps more costly method.

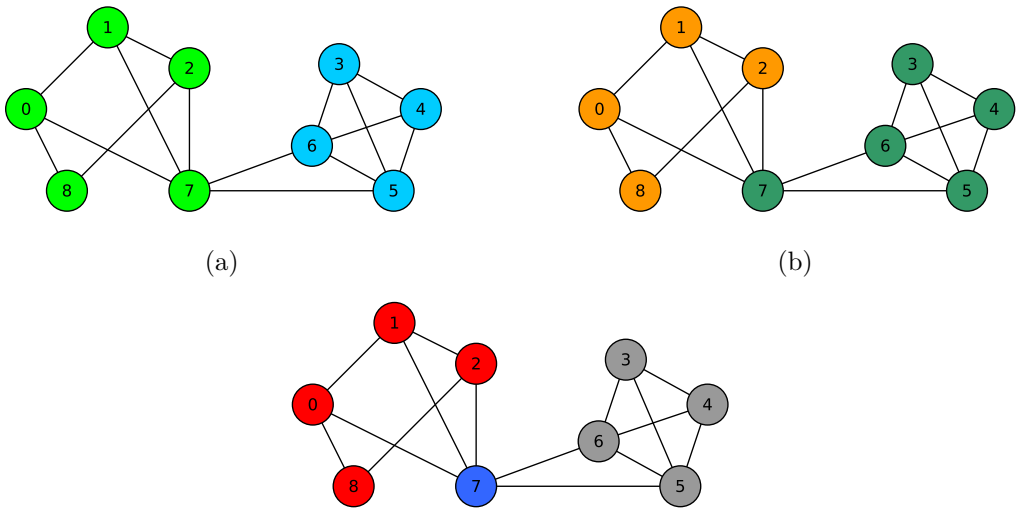


Figure 3.6: Maximum Overlap (c) of two clusterings (a) and (b)

3.4 Ensemble Clustering

Ensemble clustering is based on ensemble learning which is commonly used in classifier systems. The fundamental idea behind ensemble learning is combining an array of weak classifiers which are only marginally better than random classification, yielding a strong classifier [32]. Intuitively, if many classifiers—however weak—agree on a classification, there is likely some truth to it.

This concept can be generalized to clustering. If multiple clusterings agree that some vertices belong to the same cluster, they probably do belong in the same cluster. Conversely, groups of vertices that are not agreed upon might be worthwhile to carefully reconsider.

This principle is put to work in the Core Groups Graph Clustering scheme (CGGC) [32] introduced by Ovelgönne and Geyer-Schulz, which uses an initial non-deterministic clustering algorithm to generate a set of clusterings, and then looks at the resulting clustering to find common as well as disputed clusters. A contracted graph is induced by the combined result, which is then refined using another clustering algorithm and then projected back onto the original graph.

The combining of multiple clusterings used in this method is called *maximum overlap*. Figure 3.6 illustrates how this would be done for two clusterings. This is of course an artificial example, but it shows how the overlap might help with finding and reevaluating poor clustering choices of the past, e.g. the obviously bad classification of vertex 7 in the upper right clustering.

This property giving a shot at remedying errors makes the maximum overlap attractive for incorporation in hierarchical schemes, which are prone to mis-merge clusters, or refinement in iterative schemes. Furthermore, the overlaps of multiple clusterings can be used for detection of core clusters, since fringe vertices are harder to classify and thus will stick out in the overlap. As soon as relatively unambiguous areas of the graph—namely, the core clusters—have been classified, more time can be spent on correctly classifying the harder fringe vertices [32].

3.5 Label Propagation

Label Propagation is a linear method introduced by Raghavan et al [35]. Each vertex gets a unique label which is then iteratively propagated by looking at every neighbor of every vertex and assigning the vertex the label that is most prevalent in the vertex' neighborhood. Ties are broken uniformly by chance. This method works without any measure of quality, but also commonly results in weak clusterings [32].

To avoid oscillating labels between iterations, updated labels are considered while picking the most common label as soon as they are available, rather than relying on only the labels of the past full iteration. The vertex order is chosen randomly and the amount of labels decreases as the algorithm runs. Termination occurs when every node has the same label as the majority of its neighbors at the end of an iteration. Resulting partitions are not unique [35]. Due to its quickness and non-deterministic nature, it is apt to use this method as a diversifying step in iterative or evolutionary schemes.

The Label Propagation Algorithm (LPA) has also been used in graph partitioning [25, 40], usually with additional constraints emplaced, such as a size constraint, as done by Ovelgönne and Geyer-Schulz [25], or a balance constraint, as done by Ugander and Backstrom [40].

3.6 Evolutionary Algorithms

An evolutionary algorithm is a meta-heuristic optimization algorithm borrowing heavily from how nature optimizes life—evolution. Darwin's theory of evolution is based on two important cornerstones: competition-based selection and heritable phenotypic variations. Some phenotypic variations are more favorable in a given environment than others, which means that individuals with those variations are more likely to survive. This phenomenon is called survival of the fittest, and is also the key concept of evolutionary computing [3]. As fitter individuals are more likely to be selected for reproduction, and those favorable phenotypic variations are heritable, the

Algorithm 3.6.1 Schematic example of an evolutionary algorithm

```
procedure EA(S, p_m)
    pop ← generate random population of size S
    repeat
        a, b ← select two reasonably fit parents from pop
        o ← recombine parents a and b to produce offspring
        mutate offspring o with chance p_m
        evict an individual from population pop to make space for o
        place o into the population
    until termination condition fulfilled
    return best individual in pop
end procedure
```

population will converge in individuals that all share these traits, while the less-fit individuals eventually die. Non-ideal features disappear from the population.

Random variations—mutations—that occur with reproduction result in new phenotypic traits that may prove advantageous and will ultimately lead to greater diversity in the population. Naturally, mutations may also yield individuals that are worse off in terms of fitness, but with regards to optimization this allows candidate solutions to escape local optima, which is a crucial feature for optimization problems built upon multi-modal objectives [3].

These mechanisms provide a framework for building an optimization algorithm that is completely unconcerned with the actual parameters of the problem to be solved. The basic structure of an evolutionary algorithm is as shown in Algorithm 3.6.1.

Variations on this scheme exist, but in general, every evolutionary algorithm has a selection component, a recombination operation (or multiple to choose from), a mutation operation and some strategy to evict an individual for the newly generated one, also called survivor selection or replacement. Recombination and mutation are often referenced as variation operators [6].

Fine-tuning an evolutionary algorithm is often quite involved due to the interplay between different parameters. Parameters to tune include the population size, the mutation probability, the selection (e.g. tournament size when using tournament selection) and eviction strategy, and various other aspects of e.g. combination et cetera. Trying out all possible reasonable parameter combinations is hardly feasible. This is aggravated by the fact that one configuration may very well perform great for some range of problem instances, but under-perform for another. Tuning algorithms (e.g., in the form of another evolutionary algorithm) exist [3, Ch. 7], but are not widely adopted.

3.7 Evolutionary Algorithms for Graph Clustering

It is known that evolutionary algorithms lend themselves well to difficult problems, particularly when there are few alternatives in terms of scalable methods. As graph clustering is NP-hard and thus is often solved using heuristics containing random components, it follows that an evolutionary algorithm that somehow evolves a better clustering would be a good fit.

Tasgin and Bingol [39] introduce a genetic algorithm using modularity as a quality measure. They chose an integer encoding for representing the population clusterings, i.e. an vector c where c_i is equal to the cluster-ID of vertex i. Individuals are randomly initialized, with some bias for assigning direct neighbors the same cluster-IDs. Crossover is used as recombination operator, but is modified to take into account that clusters might be numbered differently. As for mutation, the authors chose to pick one vertex and move it to a random cluster, which is almost guaranteed to decrease the fitness. To avoid excessive worsening of the individuals, the authors employ a repair mechanism on a subset of nodes based on a newly proposed measure named *community variance*, which pertains to the ratio between the number of neighbors in different clusters to the degree of the node in question. The authors argue that iff this measure is close to zero, then the node is likely placed correctly. The method runs in $O(m)$ time.

A two-phase genetic algorithm is proposed by Kohout and Neruda [20]. The authors use a combination of two quality measures, performance [10] and modularity and argue that this leads to better results. Like Tasgin and Bingol [39], the authors chose to use an integer encoding for representing individuals. Candidates are chosen randomly using tournament selection.

The algorithm constantly flip-flops between the two phases. The first phase focuses on exploring, while the second phase refines previously found candidate solutions. The exploratory phase mutates clusterings by randomly changing cluster assignments and splitting or joining clusters, while the refinement phase tries to refine low-density clusters by pulling in cut edges (i.e. stealing a direct neighbor vertex from an adjacent cluster), or splitting low-density clusters, or joining two clusters such that the sum of edge weights of the merged cluster is maximized.

An agglomerative hierarchical genetic algorithm is presented by Lipczak and Milios in [23]. The authors represent each cluster as one individual to avoid perceived problems with integer encoding when k and n are large. This localized representation format requires quite a few changes to make all genetic operators work as when using a more standard representation.

For starters, selection involves picking a random node and choosing the associated cluster as parent. Next, all neighbors of this node that belong to another cluster are checked, and the most similar one is picked. The cluster to which this node belongs is the second parent. Standard

crossover is chosen as a recombination operator while compensating for possibly differing parent sizes. No mutation operator is defined.

4 Evolutionary Graph Clustering

In this chapter, we discuss the core principles employed by our proposed algorithm, including initialization, selection and replacement strategies, as well as recombination and mutation operators.

Our algorithm is an evolutionary algorithm utilizing a few methods previously described in Chapter 3. Most notably, we make use the maximum overlap procedure introduced by Ovelgönne and Geyer-Schulz [32] in CGGC, as well as using a variant of Label Propagation [25, 35] and high quality graph partitioning provided by KaHIP [36, 38], and the Louvain method [8].

4.1 Outline

The basic outline of the here presented algorithm is as shown in Algorithm 4.1.1. The governing operations are the four recombinators, which drive the evolution. Their main goals are to either combine two clusterings and then focus on their differing parts, or to introduce a new clustering to try and move the evolution in a new direction. As such, they are not pure recombination operators, but have a bit of a mutation flavor to them.

We also define a pure mutation operator which focuses on splitting clusters using a balanced cut, which aims to help with the known resolution limit of modularity as well as increasing the diversity of the population.

Additionally, newly recombined individuals may be input to a local search routine for refinement, with the goal of shuffling some fringe vertices to adjacent clusters if it is locally favorable.

4.2 Initialization

The population is initialized by generating p clusterings using the Louvain method. Due to the non-deterministic nature of local search—the order of nodes visited is randomized—we can expect the initial population to be somewhat diverse.

Algorithm 4.1.1 Outline of evolutionary graph clustering

```
procedure EGC(G, time limit)
    pop ← generate population with randomized base-algorithm
    repeat
        if dice roll: mutation then
            m ← MUTATE(pop)
            REFINE(m)
            evict some individual for m
        else if dice roll: recombination then
            select a recombination operation by dice roll
            c ← RECOMBINE(pop)
            optional: REFINE(c)                    ▷ refinement is done by local search
            evict some individual for c
        end if
    until time runs out
    return best specimen in pop
end procedure
```

The size of the population is fixed to be $p \leq 100$. As generating that many individuals may take considerable time with large graphs, we have specified that the maximum time to spend on population building is 10% of the time limit.

A large population is harmful to the evolution process as it severely limits the competition between specimens. Darwin referred to competition as a 'struggle for existence', usually brought on by the scarcity of some vital resource. In the context of an evolutionary algorithm, competition is hardly given when populations are so large that full replacement (assuming one replacement per iteration) takes many iterations, i.e. even unfit individuals live long and may be selected for reproduction often by mere chance—not much struggling involved. Ideally, large parts of the population are quick to be replaced, and only the best individuals survive for longer periods.

Another relevant concern are actions that might consider the whole population before coming to a conclusion, such as when choosing an individual for replacement. Evaluating a large population will delay these decisions and slow down the algorithm.

4.3 Selection

How individuals are selected for reproduction is, of course, a deciding factor for the quality of the offspring. A naïve selection mechanism might be to always select the two best individuals in the population. This would result in a very high *selection pressure* [26]—which is the degree by

which the best individuals are preferred in the selection process. Such would certainly lead any evolutionary algorithm to quick convergence, but with a very high probability that the found solution would be sub-optimal, as only relatively few genes were involved—indeed, heavily elitist reproduction would be positively incestuous as long as the offspring was of comparable quality as their parents, as the new offspring would then be selected to co-parent with one of their parents when generating the next offspring.

With a low selection pressure, reaching convergence takes longer, but the process is also more likely to reach the real optimum. As so often, a trade-off must be arranged between convergence speed and solution quality.

One approach to selection, and the one that we chose, is tournament selection. Tournament selection works by selecting s individuals of the population, the partakers of the tournament, and then selecting the best one of the lot. This method makes it easy to adjust the selection pressure [26]; a small tournament size (e.g., $s = 2$) equals low selection pressure, while a tournament size closer to p would be the selection of only the very best individuals, i.e. very high selection pressure.

We see no need for a particularly high selection pressure with our approach, as we can expect most of the population to consist of reasonably good clusterings. Furthermore, even if a bad clustering would be selected, its genes could still have a beneficial influence when combined with a good clustering—great good can, theoretically, come from quite unexpected places, and it can be argued that worse clusterings may provide a valuable perspective that can nudge the optimization process into a completely new, but ultimately good direction. Thus, we chose a tournament size of $s = 2$.

4.4 Recombination Operators

A recombination operator combines two (or more) specimen, the parents, into a new individual with traits from all parents. Recombination is the driving force behind evolution—a good recombination operation produces offspring that is on average significantly different from its parents and peers, and ideally often improves on the parents.

We define four recombination actions, described in the following subsections. As the algorithm runs, one of them is randomly selected and applied to two parent clusterings selected by tournament selection or obtained using an alternative algorithm (see Section 4.4.3 and Section 4.4.4). In general, the two clusterings are combined using the maximum overlap routine introduced in CGGC [32] (see Algorithm 4.4.1), and the resulting overlap is used to contract the input

Algorithm 4.4.1 Maximum overlap

```
procedure MaximumOverlap(C1, C2)
    M ← empty mapping
    C3 ← empty clustering
    c ← 0
    for all a, b in Zip(C1, C2) do
        if M(a, b) is undefined then
            M(a, b) ← c
            c ← c + 1
        end if
        C3(v) ← M(a, b)                          ▷ v is the current vertex
    end for
    return C3
end procedure
```

graph. This contracted graph is then input to the Louvain method, and the resulting clustering constitutes the offspring.

On a higher level, this approach isolates core clusters, which consist of all nodes that are in the same cluster in both parents, from vertex groups that have conflicting cluster assignments. The contraction of the graph then ensures that those core clusters are effectively left alone, as they are likely already classified correctly, and only predominantly conflicting vertices are moved by the Louvain method.

The majority of our recombination operators produce at worst no increase in offspring fitness compared to the best parent, because the better parent clustering is applied to the offspring clustering and then further refined by applying the Louvain method, during which vertices are only moved to another cluster iff it results in a gain in modularity. The only operator where a worse clustering could be achieved is the plain version (as discussed shortly), as the Louvain method will start from scratch when clustering the contracted graph. However, we observed that the Louvain method is usually able to cluster the contracted graph such that it is at least as fit as its fitter parent.

As for notation, we will be using $\mathcal{C}_1$ and $\mathcal{C}_2$ to refer to the two parent clusterings, and $\mathcal{C}_3$ to refer to the resulting overlap.

4.4.1 Plain Recombination

This recombination operator works as generally described in the previous section, and is the most basic variant. Two clusters are picked from the population per tournament selection and

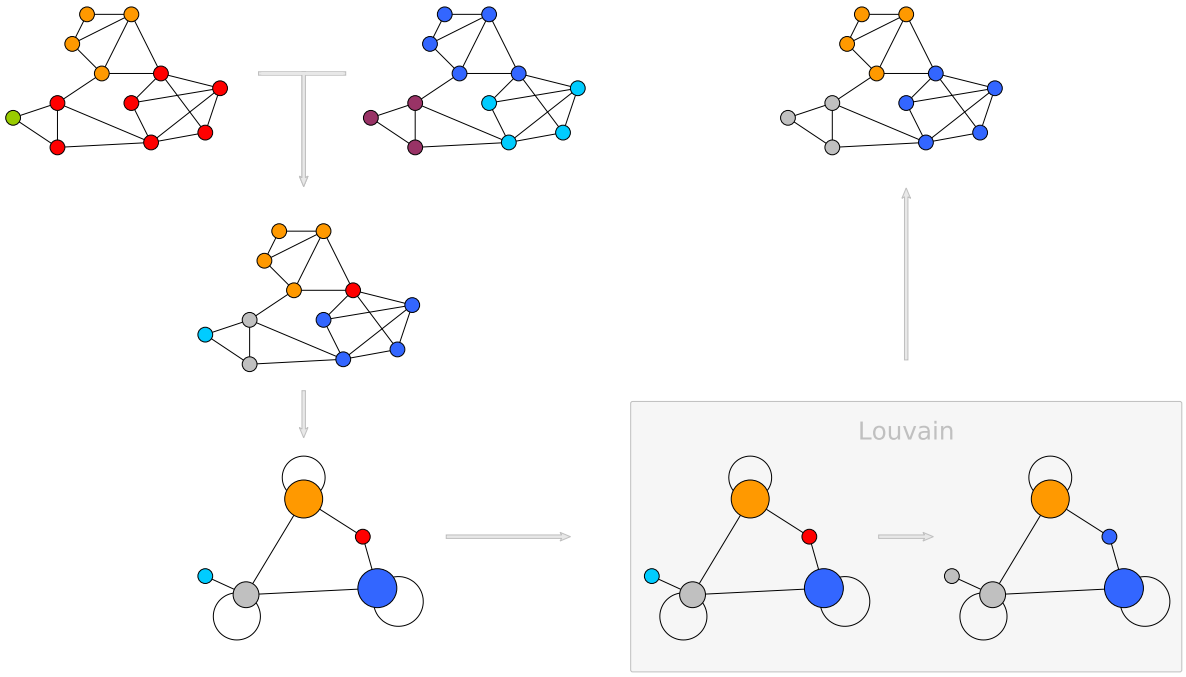


Figure 4.1: Overview of a general recombination operator

overlapped. This overlap is then used to contract the input graph, and the Louvain method is run, starting from singleton clusters. This aims to focus on certain cluster regions of the graph upon which the two parent clusterings disagree and tries to find a fitter partition that still contains the core clusters as defined by $\mathcal{C}_1$ and $\mathcal{C}_2$. Algorithm 4.4.2 shows this routine, as well as Figure 4.1.

4.4.2 Recombination with Applied Clustering

This operator is very similar to the previous, but starts off the Louvain method by using the better of the two parent clusterings as a starting point instead of singleton clusters. The

Algorithm 4.4.2 Plain recombination

procedure RecombinePlain(G, *pop*)
  $\mathcal{C}_1, \mathcal{C}_2 \leftarrow$ select two parents by tournament selection
  $\mathcal{C}_3 \leftarrow$ MaximumOverlap($\mathcal{C}_1$, $\mathcal{C}_2$)
  $G' \leftarrow$ contract graph G by clustering $\mathcal{C}_3$
  $\mathcal{C}'_3 \leftarrow$ run Louvain on G' starting from singleton clusters
  return $\mathcal{C}'_3$
end procedure

Algorithm 4.4.3 Recombination with Applied Clustering

procedure RecombineApplied(G, *pop*)
  $\mathcal{C}_1, \mathcal{C}_2 \leftarrow$ select two parents by tournament selection
  $\mathcal{C}_3 \leftarrow$ MaximumOverlap($\mathcal{C}_1$, $\mathcal{C}_2$)
  $G' \leftarrow$ contract graph G by clustering $\mathcal{C}_3$
  $\mathcal{C}'_3 \leftarrow$ run Louvain on G' starting from the better of $\mathcal{C}_1$ and $\mathcal{C}_2$
  return $\mathcal{C}'_3$
end procedure

Algorithm 4.4.4 Recombine by SCLP

procedure RecombineSCLP(G, *pop*)
  $\mathcal{C}_1 \leftarrow$ select first parent by tournament selection
  $\mathcal{C}_2 \leftarrow$ compute second parent by using SCLP
  $\mathcal{C}_3 \leftarrow$ MaximumOverlap($\mathcal{C}_1$, $\mathcal{C}_2$)
  $G' \leftarrow$ contract graph G by clustering $\mathcal{C}_3$
  $\mathcal{C}'_3 \leftarrow$ run Louvain on G' starting from $\mathcal{C}_1$
  return $\mathcal{C}'_3$
end procedure

resulting offspring can be expected to be less diverse as when using the plain recombination operator, but might more significantly improve upon the parent. Algorithm 4.4.3 shows this operator.

4.4.3 Recombination by SCLP

This operator differs from the previous two operators insofar that only one of the parents is obtained from the population. The other parent is manufactured on the spot by running size constrained label propagation (SCLP) [25] on the input graph, which yields a potentially very different clustering compared to the other parent, which aims to help with discovering new and better partitions. We picked the size constraint to be $v \sim U([10, |V|])$. Algorithm 4.4.4 outlines this operator.

SCLP extends upon LPA by disallowing that any block exceeds the given size constraint. This is done with regard to graph partitioning, as it is difficult to redistribute partitions that have violated the balance constraint. As such, SCLP finds a crude partitioning of a graph in linear time.

Algorithm 4.4.5 Recombination by partitioning

procedure RecombinePartitioning(G, *pop*)
  $\mathcal{C}_1 \leftarrow$ select first parent by tournament selection
  $\mathcal{C}_2 \leftarrow$ compute second parent by partitioning the graph
  $\mathcal{C}_3 \leftarrow$ MaximumOverlap($\mathcal{C}_1$, $\mathcal{C}_2$)
  $G' \leftarrow$ contract graph G by clustering $\mathcal{C}_3$
  $\mathcal{C}'_3 \leftarrow$ run Louvain on G' starting from $\mathcal{C}_1$
  return $\mathcal{C}'_3$
end procedure

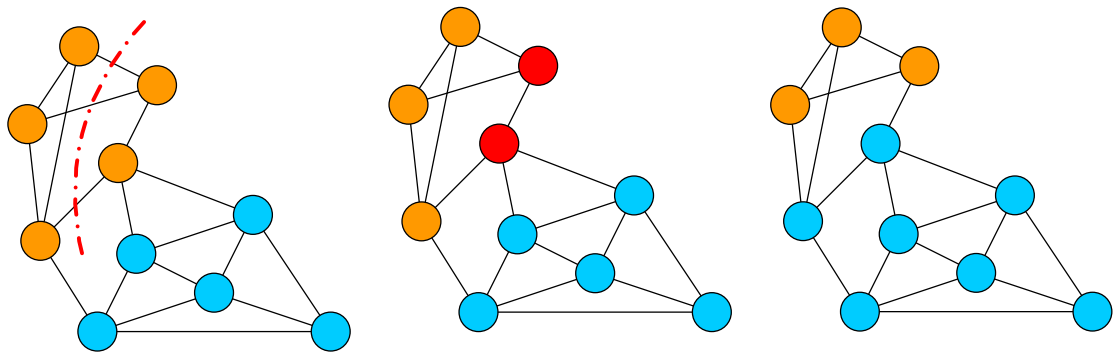
4.4.4 Recombination by Partitioning

With this operator, the same approach as the previous one is taken, but the secondary parent clustering $\mathcal{C}_2$ is generated using a partitioning routine taken from KaHIP by Schulz [38]. The partitioning algorithm is parameterized using $k \sim U([2, 64])$ and $\varepsilon \sim U([0.03, 0.5])$ with a preconfiguration of `fastsocial`. As no good clustering can be expected from a partitioner, we use $\mathcal{C}_1$ as a starting clustering for Louvain and hope that the partitioning uncovers interesting regions of the graph that can be refined into a better clustering. Algorithm 4.4.5 summarizes this operator.

4.5 Mutation Operator

Usually [3], the mutation operator is applied to the offspring with some probability p_m immediately after recombination. We, instead, define p_m to be the probability that mutation is done on some individual in the population instead of performing recombination, i.e. p_m is the fraction of time spent doing mutation. If $p_m = 1$, no recombination steps are done.

Our approach to mutation is to select a random percentage ℓ of clusters, with $\ell \sim U([0.1, 0.7])$, and split them in half. This reduces fitness, but combined with a local search it may increase the likelihood of a better partition down the road as vertices are more prone to be moved out of smaller clusters by local search, thereby increasing the diversity of the population. The splitting is done using a min cut with $\varepsilon \sim U([0, 0.5])$. Replacement for the new individual is done by tournament selection with inverse fitness, i.e. the worst clustering of the tournament is chosen to be replaced. Figure 4.2 shows an example clustering that is mutated by a cut along the red dashed line.



(a) (b) (c)

Figure 4.2: Mutation example, (a) shows the cut, (b) the new cluster after the split in red, and (c) the state after refinement

4.6 Eviction Strategies

The eviction strategy, sometimes also called survivor or replacement selection[1], pertains to how newly generated individuals are merged into the size-constrained population.

The two canonical categories for eviction are *age-based* and *fitness-based* replacement. The first removes individuals after a given number of iterations, paralleling death of old age in nature, while the other picks particular candidates that are deemed unfit, simulating death due to bad adaptation to the environment, i.e. disadvantageous phenotype.

As the first approach does not consider fitness, it is not guaranteed that the mean fitness of the population does not decrease as highly fit individuals may be replaced before a similarly fit or fitter individual is generated. Due to this, age-based replacement is usually not implemented as-is, but paired with elitist replacement selection [3], i.e. the fittest member(s) of the population are never to be replaced.

We chose to disregard age and only act on the fitness of individuals. We developed two strategies for replacement, a simple replace-worst strategy, and a strategy that considers diversity compared to the new individual and will only replace specimen that are similar to, but less fit than the new individual.

[1] In general, if the algorithm produces $n > p$ offspring, the term *survivor selection* is more commonly used. If only few of the old specimen must yield, the term is *replacement selection*.

Algorithm 4.6.1 Diversity-sustaining eviction

```
procedure Replace(G, P, x)                          ▷ x is the new individual
    C_x ← CollectCutedges(G, x)
    W ← ∅
    for all i ∈ P do
        if i is less fit than x then
            W ← W ∪ i
        end if
    end for
    replace w ∈ W where CollectCutedges(G, w) is most similar to C_x
end procedure

procedure CollectCutedges(G, i)
    C_i ← ∅
    for all (v, w) ∈ E(G) do
        if C_i(v) ≠ C_i(w) then
            C_i ← C_i ∪ (v, w)
        end if
    end for
    return C_i
end procedure
```

4.6.1 Replace Worst

One approach to replacement is to evict the worst individual of the population. This approach is likely to quickly converge, but is also likely to prematurely converge in a suboptimal solution [3] as more diverse, but less fit individuals are quickly shed, especially if the population is small.

4.6.2 Diversity-Sustaining Eviction

A more sensible approach to eviction would be to consider only the specimen that are worse than the newly generated specimen, and then replace the most similar one of them. This maintains the current level of diversity in the population, ensuring that few genes are lost.

How similar two clusterings are can be determined in $O(m)$ time by inspecting the cut edges, i.e. all edges whose sink and source lie in different clusters. A clustering with close to the same cut edges can be regarded as similar. Algorithm 4.6.1 shows how this is done.

5 Experimental Evaluation

In this chapter we evaluate our previously proposed and outlined algorithm. We tune crucial parameters such as mutation probability and eviction strategy, present the problem instances used, and finally benchmark our algorithm against a set of graphs with known best values for modularity.

5.1 Instances

We obtained the graph instances used for evaluation from the 10th DIMACS implementation challenge [1]. These graphs have been used for benchmarking various top-of-the-field clustering algorithms [7], which provides us with a point of reference regarding modularity.

We evaluated 24 of the 30 graphs included in the DIMACS data set; the other 6 graphs were skipped as they were particularly large in either n or m.

The majority of the used graphs are some kind of social network, either modeling interactions between scientists or showing a subset of the internet. We have also used 2 street networks (belgium.osm and luxembourg.osm) as well as mesh graphs (e.g. ldoor and 333SP), as well as some synthesized networks. Almost all of the graphs exhibit a strong community structure according to modularity.

The instances used for tuning are listed in Table 5.1, and all instances used for benchmarking are listed in Table 5.2.

graph	n	m
power	4941	6594
wordassociation-2011	10,617	63,788
email-EuAll	16,805	60,260
cond-mat-2005	40,421	175,691
preferentialAttachment	100,000	499,985
caidaRouterLevel	192,244	609,066

Table 5.1: Instances used for tuning sorted by n

graph	n	m
celegans_metabolic	453	2025
email	1133	5451
polblogs	1490	16,715
power	4941	6594
PGPgiantcompo	10,680	24,316
astro-ph	16,706	121,251
memplus	17,758	54,196
as-22july06	22,963	48,436
cond-mat-2005	40,421	175,691
kron_g500-simple-logn16	65,536	6,289,992
G_pin_pout	100,000	501,198
preferentialAttachment	100,000	499,985
smallworld	100,000	499,998
luxembourg.osm	114,599	119,666
rgg_n_2_17_s0	131,072	728,753
caidaRouterLevel	192,244	609,066
coAuthorsCiteseer	227,320	814,134
citationCiteseer	268,495	1,156,647
coPapersDBLP	540,486	15,245,729
eu-2005	862,664	16,138,468
ldoor	952,203	22,785,136
in-2004	1,382,908	13,591,473
belgium.osm	1,441,295	1,549,970
333SP	3,712,815	11,108,633

Table 5.2: Instances used for benchmarking sorted by n

5.2 Setup and Methodology

All experiments were run on a four Quad-Core AMD Opteron 8356 machine, with each core running one experiment in parallel, equipped with 64 GB memory. Every experiment was repeated five times and then averaged over time. The averaged runs were then further condensed by calculating the geometric mean over instances in the case of the tuning data sets.

We average n data sets by collating all timestamps, sorting them and then, for every timestamp t', collecting the n time-modularity pairs (t, m) for which $t \leq t'$ holds and t is maximal, i.e. the data points that are most recent to t'. All timestamps for which less than n values are available are skipped. The n values are then averaged by calculating the geometric mean, i.e. $\overline{m} = (\prod_i m_i)^{\frac{1}{n}}$.

5.3 Parameter Tuning

Our algorithm has many parameters that could be considered for tuning. These include:

- Population size
- Likelihoods of different recombination operators
- Mutation probability
- Selection strategy
- Tournament size (selection pressure)
- Replacement strategy
- Any implementation-specific parameters

We shall focus on the mutation probability, compare the two proposed eviction strategies with each other, as well as evaluate the impact of the additional refinement operation after combination. This is only a selection of parameters, as tuning every single parameter would prove tedious, and to make matters worse, would in all likelihood provide only marginal quality increases. Configurations for subalgorithms (such as partitioning routines, or the Louvain method) were chosen based on preliminary experiments.

5.3.1 Mutation Probability

Results are shown in Figure 5.1. Our algorithm seems rather robust, with only drastically large values of p_m leading to a significant decrease in solution quality. When $p_m = 1$, no recombination actions are done, which obviously leads to hardly any improvement during the course of the algorithm.

As most resulting plots were pretty similar, they were omitted in the figure. It turns out that $p_m = 0.55$ yielded the best results, but this is somewhat skewed by small graphs. As the algorithm does many more iterations on a small graph than on a larger graph, and most of them yield no improvement anyway, a large mutation ratio does less harm than on larger graphs. Thus, we suggest that $p_m = 0.3$ is actually a better value for general use. Further of note is that the differences between different runs are so small that they could very well still be a product of chance, as it is not impossible that five abnormally good or bad runs occur in a row for one parameter configuration (although unlikely).

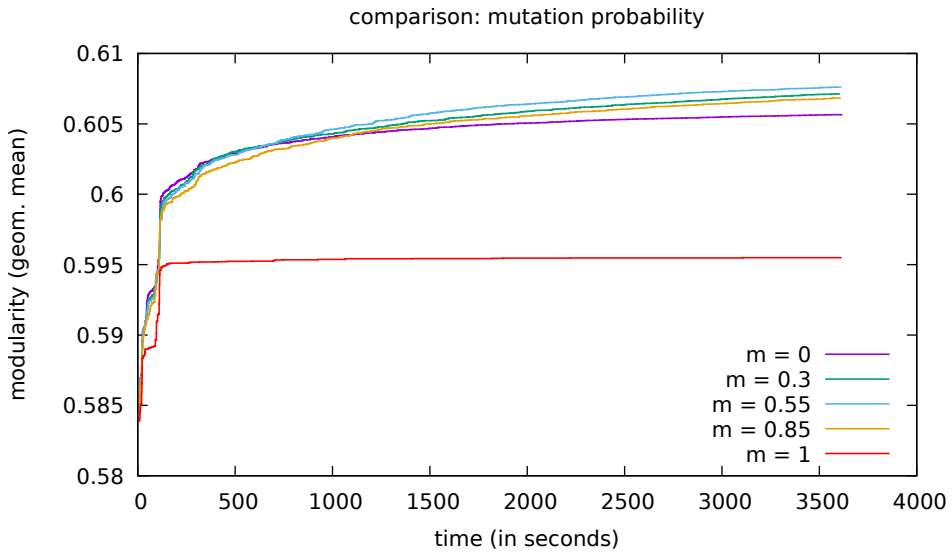


Figure 5.1: Effect of mutation probabilities

The impact of the mutation varies quite strongly between graphs. On some graphs, it is indeed (slightly) beneficial, while on some graphs the mutation almost seems like wasted cycles, as the result is comparable even with mutation disabled—or even worse with some graphs. It is difficult to propose a generalized optimum value for the mutation probability, thus we will pick two values for further evaluation: $p_m = 0$ and $p_m = 0.3$.

5.3.2 Eviction Strategies

As is discussed in Section 4.6, we expect that the plain replacement, where the worst specimen is evicted, will lead to a quicker convergence, but converge in a suboptimal solution, while the diversity-sustaining eviction will be perhaps a little slower to converge, but will ultimately do so in a better solution.

This has proven to be true on the subset of graphs that were used for tuning. Without fail, the more advanced eviction strategy yielded better end results, but moved more slowly. Figure 5.2 shows a comparison.

5.3.3 Additional Refinement

Figure 5.3a shows the impact of the additional refinement (LS) done in every iteration. This extra step provides a much steeper increase in quality early on. The differences even out a bit,

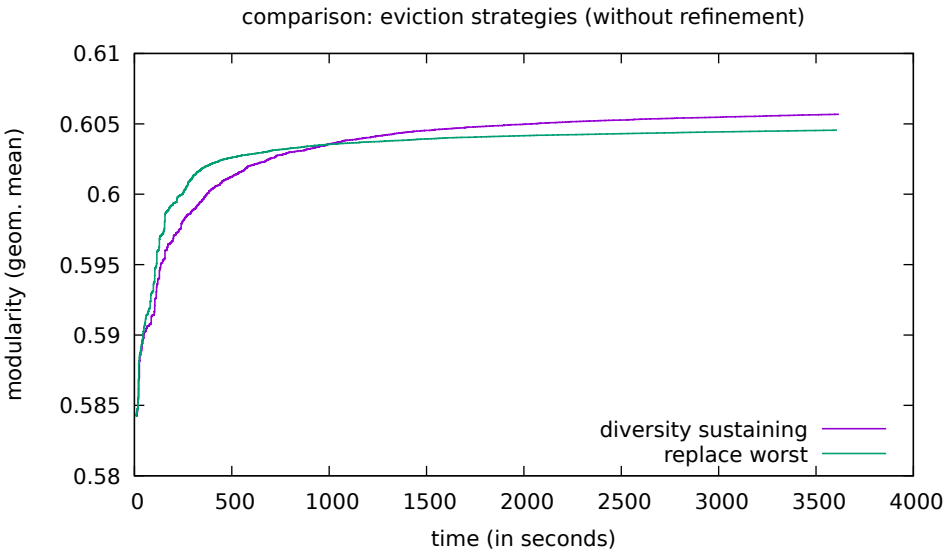
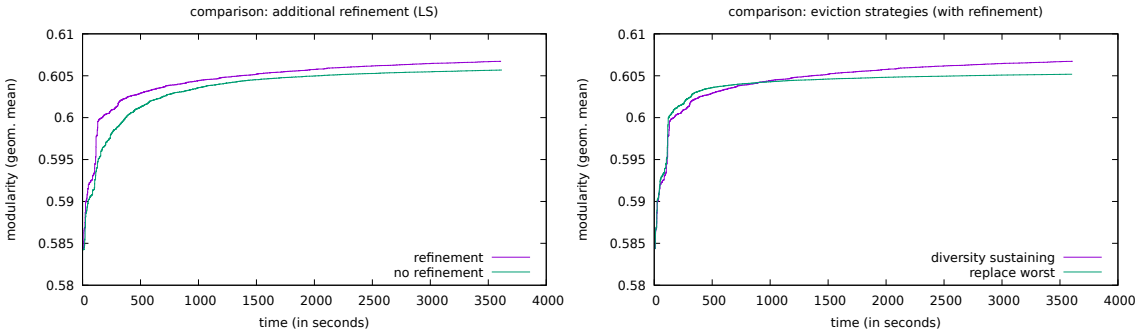


Figure 5.2: Comparison of eviction strategies

but the end result is still better with additional refinement. It is notable, however, that on the smallest graph, the added local search seems to do harm, with the end result being a smidgen better when run without LS. This is possibly due to LS harming the ability of the algorithm to leave local optima. This was not observed on the larger graphs, thus we consider the additional LS to be a net gain with some caveats pertaining to very small graphs.

The steeper quality increase is able to catch up with the faster convergence of the replace-worst eviction strategy, as visible in Figure 5.3b when compared to Figure 5.2. LS provides a boost to quality in early iterations that is comparable to the boost apparent with the replace-worst strategy, closing the gap in early iterations while still having a better end result.



(a) Comparison of refinement

(b) Refinement with the two eviction strategies

Figure 5.3: (a) Refinement, (b) refinement with different eviction strategies

config A	
plain	on
applied	off
SCLP	off
partitioning	off
p_m	0.0

config AM	
plain	on
applied	off
SCLP	off
partitioning	off
p_m	0.3

config B	
plain	on
applied	on
SCLP	off
partitioning	off
p_m	0.0

config BM	
plain	on
applied	on
SCLP	off
partitioning	off
p_m	0.3

config C	
plain	on
applied	on
SCLP	on
partitioning	off
p_m	0.0

config CM	
plain	on
applied	on
SCLP	on
partitioning	off
p_m	0.3

config D	
plain	on
applied	on
SCLP	on
partitioning	on
p_m	0.0

config DM	
plain	on
applied	on
SCLP	on
partitioning	on
p_m	0.3

Table 5.3: Overview over benchmarking configurations

5.4 Benchmark Results

We benchmarked our algorithm on the problem instances listed in Table 5.2. To investigate the impact of the different recombination operators in combination with mutation, we will test eight configurations. Table 5.3 lists them. Recombinators are enabled cumulatively, hence we expect that results should on average improve between configurations if the recombinator is beneficial, i.e. the impact is positive. Of course, the impact of recombinators may vary graph to graph, and randomness is at play; differences—especially if very small—should be taken with a grain of salt.

We grouped the graphs into 3 classes based on size and average iteration length which depends not only on graph size, but also on the general graph structure. One class, consisting of 15 smaller graphs, ran for 2 hours each, while the other class consisting of four medium-sized graphs ran for 8 hours. Finally, the five largest graphs that we benchmarked ran for 16 hours each.

For the large graphs, adjustments regarding the mutation operation had to be made as the operator was found to scale badly with the chosen distribution for ℓ. We reduced ℓ to be $\ell \sim U([0.1, 0.2])$.

The results are presented in Table 5.4. We list the minimum achieved modularity value as well as the maximum and average among all configurations, specify how long the algorithm was run, and the reference value taken from the benchmark with the name of the algorithm that achieved

graph	time	config	reference	*min*	*max*	*avg*
as-22july06	2h	CM	0.678267[1]	0.678449	**0.679308**	0.679073
astro-ph	2h	D	0.744621[2]	0.745462	**0.746194**	0.746066
belgium.osm	8h	C	0.994940[1]	0.995038	**0.995059**	0.995052
caidaRouterLevel	2h	D	0.872042[1]	0.872417	**0.872700**	0.872596
celegans_metabolic	2h	BM,CM	0.453248[2]	0.452447	*0.453248*	0.453186
citationCiteseer	8h	D	0.823930[1]	0.824422	**0.825269**	0.825026
coAuthorsCiteseer	8h	D	0.905297[1]	0.906133	**0.906612**	0.906398
cond-mat-2005	2h	C	0.746254[1]	0.748689	**0.749637**	0.749264
coPapersDBLP	16h	DM	0.866794[1]	0.867563	**0.867880**	0.867767
email	2h	all	0.582829[2]	0.582387	*0.582829*	0.582824
eu-2005	16h	D	0.941554[1]	0.941516	**0.941574**	0.941558
G_n_pin_pout	2h	A	0.500098[1]	0.50035	**0.500473**	0.50045
in-2004	16h	D	0.980622[1]	0.980574	**0.980685**	0.980617
kron_g500-simple-logn16	8h	C	0.065056[2]	0.0621052	0.0638011	0.063306
ldoor	16h	D	0.969370[3]	0.970038	**0.970415**	0.970298
luxembourg.osm	2h	BM	0.989621[2]	0.989618	**0.989647**	0.989639
memplus	2h	DM	0.700473[1]	0.646283	**0.701275**	0.695058
PGPgiantcompo	2h	AM	0.886564[1]	0.886625	**0.886826**	0.886781
polblogs	2h	all	0.427105[2]	0.427105	*0.427105*	0.427105
power	2h	CM,DM	0.940851[2]	0.940716	**0.940925**	0.940893
preferentialAttachment	2h	CM	0.315994[2]	0.296291	0.305846	0.302909
rgg_n_2_17_s0	2h	C	0.978324[2]	0.978331	**0.978413**	0.978387
smallworld	2h	AM	0.793042[2]	0.793153	**0.793169**	0.793166
333SP	16h	C	0.989096[3]	0.989025	**0.989155**	0.989121

[1] CGGCi_RG [32], [2] VNS [5], [3] ParMod [12]

Table 5.4: Benchmarking results

this value indicated by the superscript. The configuration(s) as given by Table 5.3 with which the best value was achieved is noted as well. Maximum values that surpass the benchmark value are printed in bold, and values that reached but did not surpass the benchmark value are written in italic. More detailed results split up by configurations are available in Appendix B.

5.4.1 Observations

Our algorithm was able to achieve comparable or better results on all but two of the graphs. On 14 of these graphs, all eight configurations were able to reach or surpass the benchmark value even in the worst run. One notable graph is the memplus instance, on which we observed significant variance between runs—the worst run managed to achieve a modularity of 0.646283, while the best run achieved 0.701275, which is a bit better than the benchmark value. A similar situation—where the minimum run is worse than the benchmark value, but the maximum is better—can be found on the 333SP instance, although the range of the results on this graph is

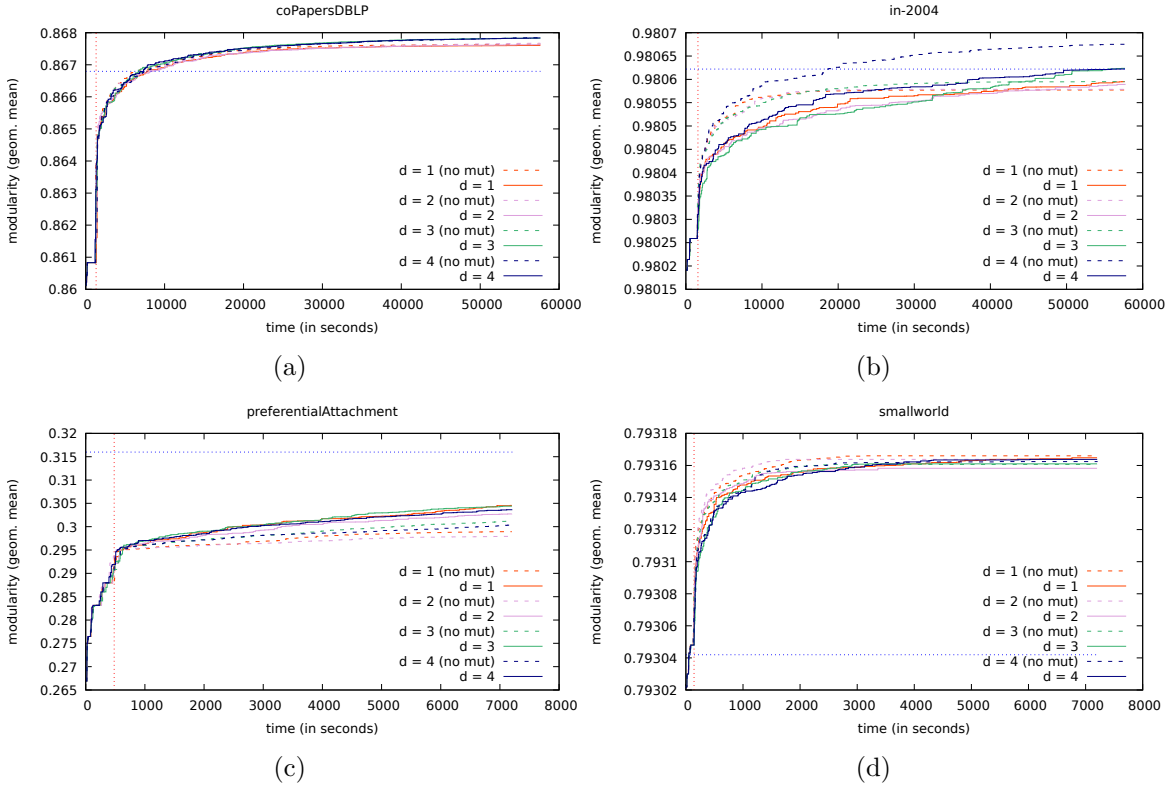


Figure 5.4: A selection of result plots, more in Appendix B

much smaller. Both of these graphs are instances of sparse matrices, and in the case of 333SP a bit low on iterations as it is rather large.

It is notable that the graphs on which no competitive result could be achieved have little natural cluster structure which is evident by the low modularity, which is $mod(\mathcal{C}) \lessapprox 0.3$. This may indicate that our scheme has difficulties with graphs that have no pronounced clustering structure.

Figure 5.4 shows a representative selection of 4 result plots. Plots for all instances are available in Appendix B. These figures, especially 5.4a and 5.4d, show the typical behavior of our algorithm. Early iterations are marked by an steep increase in quality, followed by a gradual tapering off during later iterations, where only marginal refinements are achieved. The vertical red dashed line marks the end of the population building phase, i.e. after this point, any improvements made are the result of our work. The legend on these figures specifies the degree of the algorithm by d. This number specifies the recombinators that were active and correspond to the configurations given in Table 5.3. On most graphs, the benchmark value indicated by the horizontal blue dashed line is reached within the first third of the allocated time frame.

The coPapersDBLP instance is shown in Figure 5.4a, on which all configurations achieved very similar results, but the D and DM configurations got the best results by a small margin, with very little difference between the two.

Figure 5.4b depicts the plot resulting from the in-2004 instance. On this graph it is apparent that the D configuration without mutation has a small advantage and surpasses the benchmark value, while the same configuration with mutation is the next best curve, but barely reaches the benchmark value.

Figure 5.4c shows results from the preferentialAttachment graph, which is one of the two on which no competitive result could be achieved by our algorithm. The plot shows that the starting point generated by the Louvain method is rather far off from the benchmark value, and the curve is quite flat, although this is exaggerated by the scale of the plot. The configurations with mutation enabled fared better on this graph instance.

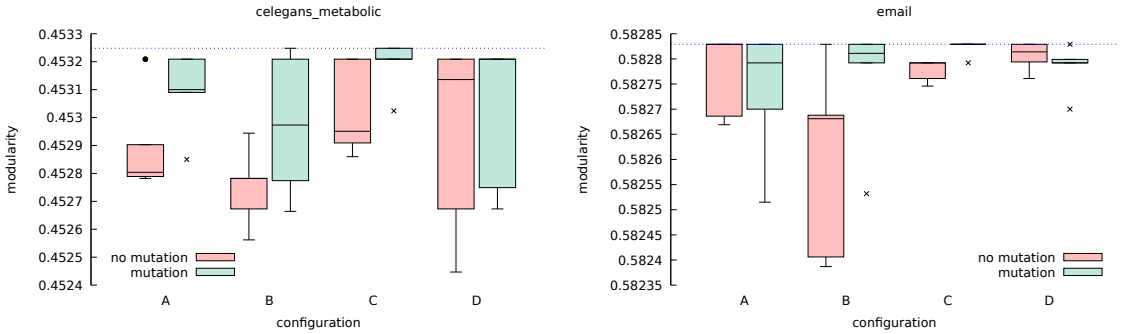
Figure 5.4d pictures the results arising from the smallworld graph. On this graph, the Louvain method alone was able to achieve a result that surpasses the benchmark value. Another interesting point is that on this graph, the plain configuration A and AM were able to achieve results that were on average slightly better than the full-featured configurations D and DM.

Impact of Recombination Operators

There is hardly any clear separation between the curves portraying different recombination setups on most of the smaller graphs. On some graphs, it becomes apparent that the configuration utilizing the partitioning recombination operator suffers slightly or does not significantly improve upon configuration C.

Nevertheless, the configurations with the partitioning recombinator enabled were able to achieve the best result on 10 graphs, and of these, 3 configurations also had the mutation operator enabled. These configurations were able to achieve the best result on 4 of the 5 large graphs, although we consider the differences to be rather marginal. In general, the results pertaining to the best configuration are rather mixed, with smaller graphs achieving better results with a mix of the configurations C and B and in some cases even just A.

The slight inferiority of the partitioning recombinator is likely because of the larger cost of the partitioning, as well as a lack of value brought to the table by the partition. Although we did adjust the algorithm to account for the larger cost of the partitioning, making all other recombination operators about four times as likely to be selected over the partitioning recombinator, this effect is still apparent with large graphs. Further punishing the partitioning



(a) (b)

Figure 5.5: Effects of recombinators on range of end results

recombination operator would result in it being called so rarely in relation to other recombinators that it would likely cease to have any significant effect—for better or worse.

On some graphs, configurations C and D had an interesting effect on the range of end results. Figure 5.5 shows the two graphs that exhibited this effect most strongly, but in opposite directions. While on the email instance, aforesaid configurations had a much smaller range (although not always capable of reaching the benchmark), the range actually increased on the celegans_metabolic instance. Interestingly, these graphs are also among the smallest in the data set.

Impact of the Mutation Operator

A little surprising was the scaling behavior of the mutation operator. While the mutation step on smaller graphs usually ran faster than a recombination, and thus more iterations were performed when the mutation probability was set to $p_m = 0.3$, the behavior on larger graphs was very much the opposite. In some cases, about 2 000 iterations were performed without mutation, while only about 300 were made with mutation. This is a very significant discrepancy and severely handicaps the algorithm when run with mutation enabled.

After we tweaked the value range for ℓ, this improved to about 1 700 iterations in the same time span. Although this is an improvement, it is still suboptimal. We suggest that the mutation probability should be lowered to $p_m = 0.1$ when run on very large graphs for best results.

The runs with mutation enabled show a flatter curve (see 5.4b and to lesser extent 5.4d), which is explained by the fewer recombination operations done in early iterations. However, in most cases, the curves catch up and the actual difference in end results is rather low.

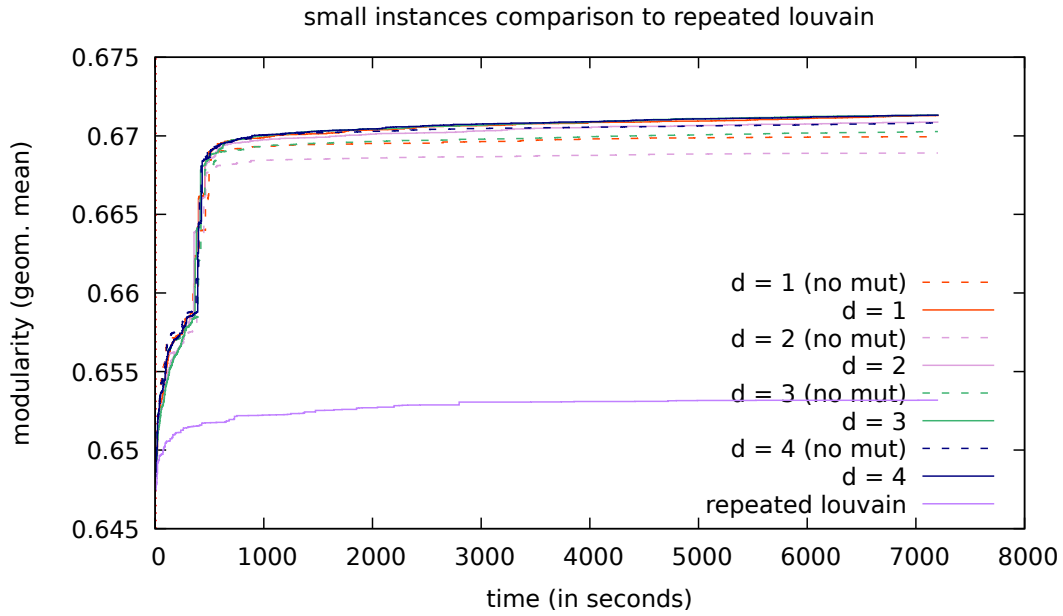


Figure 5.6: Comparison to random repeats of the Louvain method

All in all, the end results fail to unambiguously show the significance of the mutation operator, and in the aforementioned cases of large graphs deteriorate the quality of the final clustering. However, we want to point out that the goal of the mutation operator is to increase the diversity in the population which is constantly decreasing as the algorithm runs. The diversity of the population must remain high, or the algorithm will completely stagnate. As such, a lack of diversity in the population is only really apparent when it is negatively impacting the algorithm, and it may not be readily apparent that something is amiss. Thus, we argue that the best course of action would not be to completely ditch the mutation operator, but perhaps lower the mutation probability and extend the run time to ensure that the same amount of recombination operations are done as in a run without mutation.

5.4.2 Comparison to Random Repeats of Louvain

The results show that our evolutionary algorithm is able to improve upon the population generated by at most $p = 100$ repeats of the Louvain method. We now want to show that our algorithm produces better results even compared to random repeats of the Louvain algorithm over the same time frame, i.e. more than 100 repeats.

For this, we selected all small graphs that were run for 2 hours for the benchmark, and repeatedly ran the Louvain method on them, also for 2 hours, noting the maximum modularity score

thus far after every iteration. Every run was repeated 5 times. Figure 5.6 shows a comparison between the averages for all 8 configurations and the average of all runs of random repeats of the Louvain method, both averaged over all small instances. The gap between the two is quite pronounced, which means that our algorithm attains better results by a noticeable margin. This is true for all instances where any improvement is possible, i.e. all instances where the optimum is not already found by the Louvain method.

6 Conclusion

This thesis introduced the graph clustering problem, with a focus on modularity maximization. We surveyed notable work in this field as well as evolutionary algorithms and a few examples of applying evolutionary algorithms to solve the graph clustering problem.

We further presented our evolutionary graph clustering algorithm. We proposed four recombination operators, as well as a mutation operator, an eviction strategy that aims to uphold the current level of diversity in the population and an additional refinement operation to be applied to newly recombined offspring.

We then evaluated the behavior of our algorithm when run with different configurations, particularly when run with some recombination operators dis- or enabled, as well as with the mutation operator turned off or on. We have also tried to find parameters that are close to optimal for general use of our algorithm, which proved to be a challenge.

Of further importance is the comparison of the results achieved by our algorithm to the results achieved by various top-of-the-field algorithms. We were able to compete with those algorithms on most graphs of the DIMACS benchmarking testbed that we evaluated.

6.1 Future Work

Evolutionary algorithms lend themselves very well to parallelization, which would very likely speed up our algorithm significantly. Major components of the algorithm like the building of the initial population and recombination and mutation can be done in parallel, resulting in more iterations in the same time frame and thus potentially better results.

The algorithm itself could further be tuned, although this would likely need to be done on a case-by-case basis. The parameter configurations chosen in the evaluation were sufficient, but we suspect that better results could still be achieved on some of the graphs with different parameters. In particular, we believe that on the two problematic graphs a smaller population size would be beneficial, as well as an more aggressive mutation probability, as those graphs seem to need many refinement passes on the same clustering, which will not happen if the

population pool is large. This may be due to the weak community structure inherent to these graphs.

Furthermore, it may be worthwhile to investigate more or different recombination operators, as we have noted that there is not much of a strong case for the different recombination operators that we have proposed. It is simple to extend the algorithm by further operations, although the recombinators need to be balanced to achieve the best results, which is again a tuning effort.

An especially interesting approach to extending the recombinators would be to introduce multilevel recombinators, which go through several levels of graph contraction instead of contracting only once. This technique is commonly used in graph partitioning.

Bibliography

[1] 10th DIMACS Implementation Challenge. `https://www.cc.gatech.edu/dimacs10/`. Accessed on August 21, 2017.

[2] MCL - a cluster algorithm for graphs. `https://micans.org/mcl/index.html`. Accessed on July 18, 2017.

[3] J.E. Smith (auth.) A.E. Eiben. *Introduction to Evolutionary Computing*. Natural Computing Series. Springer-Verlag Berlin Heidelberg, 2 edition, 2015.

[4] L. Akoglu, H. Tong, and D. Koutra. Graph based anomaly detection and description: A survey. *Data Min. Knowl. Discov.*, 29(3):626–688, May 2015.

[5] D. Aloise, G. Caporossi, P. Hansen, L. Liberti, S. Perron, and M. Ruiz. Modularity maximization in networks by variable neighborhood search. In *Graph Partitioning and Graph Clustering*, volume 588 of *Contemporary Mathematics*, pages 113–128. American Mathematical Society, 2012.

[6] T. Back, U. Hammel, and H. P. Schwefel. Evolutionary computation: comments on the history and current state. *IEEE Transactions on Evolutionary Computation*, 1(1):3–17, Apr 1997.

[7] D. A. Bader, H. Meyerhenke, P. Sanders, C. Schulz, A. Kappes, and D. Wagner. Benchmarking for graph clustering and partitioning. In *Encyclopedia of Social Network Analysis and Mining*, pages 73–82. 2014.

[8] 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(10):P10008, 2008.

[9] U. Brandes, D. Delling, M. Gaertler, R. Görke, M. Hoefer, Z. Nikoloski, and D. Wagner. *On Finding Graph Clusterings with Maximum Modularity*, pages 121–132. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[10] U. Brandes, M. Gaertler, and D. Wagner. Experiments on graph clustering algorithms. 2003.

[11] T. N. Bui and C. Jones. Finding Good Approximate Vertex and Edge Partitions is NP-hard. *Inf. Process. Lett.*, 42(3):153–159, May 1992.

[12] Ü. V. Çatalyürek, K. Kaya, J. Langguth, and B. Uçar. A partitioning-based divisive clustering technique for maximizing the modularity. In *Graph Partitioning and Graph Clustering*, volume 588 of *Contemporary Mathematics*, pages 171–186. American Mathematical Society, 2012.

[13] A. Clauset, M. E. J. Newman, and C. Moore. Finding community structure in very large networks. *Phys. Rev. E*, 70:066111, Dec 2004.

[14] J. Demme and S. Sethumadhavan. Approximate graph clustering for program characterization. *ACM Trans. Archit. Code Optim.*, 8(4):21:1–21:21, January 2012.

[15] A. A. Diwan, S. Rane, S. Seshadri, and S. Sudarshan. Clustering techniques for minimizing external path length. In *Proceedings of the 22th International Conference on Very Large Data Bases*, VLDB '96, pages 342–353, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[16] S. Fortunato and M. Barthélemy. Resolution limit in community detection. *Proceedings of the National Academy of Sciences*, 104(1):36–41, 2007.

[17] L. C. Freeman. A Set of Measures of Centrality Based on Betweenness. *Sociometry*, 40(1):35–41, March 1977.

[18] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some Simplified NP-complete Problems. In *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, STOC '74, pages 47–63, New York, NY, USA, 1974. ACM.

[19] G. Karypis and V. Kumar. A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. *University of Minnesota, Department of Computer Science and Engineering, Army HPC Research Center, Minneapolis, MN*, 1998.

[20] J. Kohout and R. Neruda. Two-phase genetic algorithm for social network graphs clustering. In *2013 27th International Conference on Advanced Information Networking and Applications Workshops*, pages 197–202, March 2013.

[21] A. Lancichinetti and S. Fortunato. Community detection algorithms: A comparative analysis. *Phys. Rev. E*, 80:056117, Nov 2009.

[22] S. Lancichinetti, A. Fortunato. Limits of modularity maximization in community detection. *Phys. Rev. E*, 84:066122, Dec 2011.

[23] M. Lipczak and E. E. Milios. Agglomerative genetic algorithm for clustering in social networks. In Franz Rothlauf, editor, *GECCO*, pages 1243–1250. ACM, 2009.

[24] S. McFarling. Program optimization for instruction caches. *SIGARCH Comput. Archit. News*, 17(2):183–191, April 1989.

[25] H. Meyerhenke, P. Sanders, and C. Schulz. Partitioning complex networks via size-constrained clustering. In *SEA*, volume 8504 of *Lecture Notes in Computer Science*, pages 351–363. Springer, 2014.

[26] B. L. Miller and D. E. Goldberg. Genetic algorithms, selection schemes, and the varying effects of noise. *Evol. Comput.*, 4(2):113–131, June 1996.

[27] M. E. J. Newman. Properties of highly clustered networks. *Physical Review E*, 68(2):026121, 2003.

[28] M. E. J. Newman. Fast algorithm for detecting community structure in networks. *Phys. Rev. E*, 69:066133, Jun 2004.

[29] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Phys. Rev. E*, 69:026113, Feb 2004.

[30] M. Ovelgönne and A. Geyer-Schulz. Cluster cores and modularity maximization. In *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*, ICDMW '10, pages 1204–1213, Washington, DC, USA, 2010. IEEE Computer Society.

[31] M. Ovelgönne and A. Geyer-Schulz. A comparison of agglomerative hierarchical algorithms for modularity clustering. In Wolfgang Gaul, Andreas Geyer-Schulz, Lars Schmidt-Thieme, and Jonas Kunze, editors, *GfKl*, Studies in Classification, Data Analysis, and Knowledge Organization, pages 225–232. Springer, 2010.

[32] M. Ovelgönne and A. Geyer-Schulz. An ensemble learning strategy for graph clustering. In *10th DIMACS Implementation Challenge Graph Partitioning and Graph Clustering*, 2012.

[33] J. B. Pereira-Leal, A. J. Enright, and C. A. Ouzounis. Detection of functional modules from protein interaction networks. *Proteins: Structure, Function, and Bioinformatics*, 54(1):49–57, 2004.

[34] I. Pitas. *Graph-Based Social Media Analysis*. Chapman and Hall and CRC Press, Dec 2015.

[35] U. N. Raghavan, R. Albert, and S. Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Phys. Rev. E*, 76:036106, Sep 2007.

[36] P. Sanders and C. Schulz. Think Locally, Act Globally: Highly Balanced Graph Partitioning. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *LNCS*, pages 164–175. Springer, 2013.

[37] S. E. Schaeffer. Survey: Graph clustering. *Comput. Sci. Rev.*, 1(1):27–64, August 2007.

[38] C. Schulz. *High Quality Graph Partitioning.* PhD thesis, Karlsruhe Institute of Technology, 2013.

[39] M. Tasgin and H. Bingol. Community detection in complex networks using genetic algorithm. In *ECCS '06: Proc. of the European Conference on Complex Systems*, April 2006.

[40] J. Ugander and L. Backstrom. Balanced label propagation for partitioning massive graphs. In *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*, WSDM '13, pages 507–516, New York, NY, USA, 2013. ACM.

[41] S. van Dongen. A cluster algorithm for graphs. Technical report, Amsterdam, The Netherlands, 2000.

[42] S. van Dongen. *Graph Clustering by Flow Simulation.* PhD thesis, University of Utrecht, Utrecht, May 2000. http://www.library.uu.nl/digiarchief/dip/diss/1895620/inhoud.htm.

[43] K. Wakita and T. Tsurumi. Finding community structure in mega-scale social networks. *CoRR*, abs/cs/0702048, 2007.

[44] Y. Xu, V. Olman, and D. Xu. Clustering gene expression data using a graph-theoretic approach: an application of minimum spanning trees. *Bioinformatics*, 18(4):536–545, 2002.

Appendix A

Implementation

The implementation is written in C++14. We used the most recent implementation of the Louvain method provided by the authors[2], and modified it somewhat to fit our requirements such as using unsigned integers as cluster IDs and removing the need for the binary file format, as well as allowing the algorithm to start from a precomputed clustering. The program integrates into KaHIP[3].

Input files must be of the format used in the METIS graph partitioning software package [19]. Our implementation only supports unweighted graphs with no self loops, no node weights and no parallel edges. The main routine accepts the graph in a compressed sparse row (CSR) format, similar to the routines included in METIS and KaHIP.

The following listing shows the helptext of the program which summarizes all possible command line arguments and their default values. The degree parameter (`-d`) controls which recombinators are available to choose from in each iteration. A value of 1 is equivalent to plain recombination only, while a value of 4 is the full-fledged version with all recombination operators enabled.

Listing A.1: Helptext of our implementation

```
$ ./bin/cluster -h
usage: ./bin/cluster opts FILE
where opt in opts is one of:

    -p $value            ... minimum population size, default: 10
    -P $value            ... maximum population size, default: 100
    -t $value            ... maximum time to spend in seconds, default: 60
    -r $value            ... fraction of maximum time to spend building initial
        ↪ population, default: 0.1
```

[2] https://sourceforge.net/projects/louvain/
[3] https://github.com/schulzchristian/KaHIP

```
    -i $value            ... number of iterations to be done, turns off maximum time
        ↪ for main loop. (population building is still affected)
    -m $value            ... mutation probability, default: 0.1
    -d $value            ... degree of recombinators, default: 4
    -h                   ... this message

    --[no-]ls            ... toggle additional local search, default: on
    --[no-]interim-print ... toggle interim mod printing, default: on
    --[no-]print-time    ... toggle timestamp when interim printing, default: on
    --[no-]better-evict  ... toggle "better" evict, default: on

and FILE is a metis-style graph file.
```

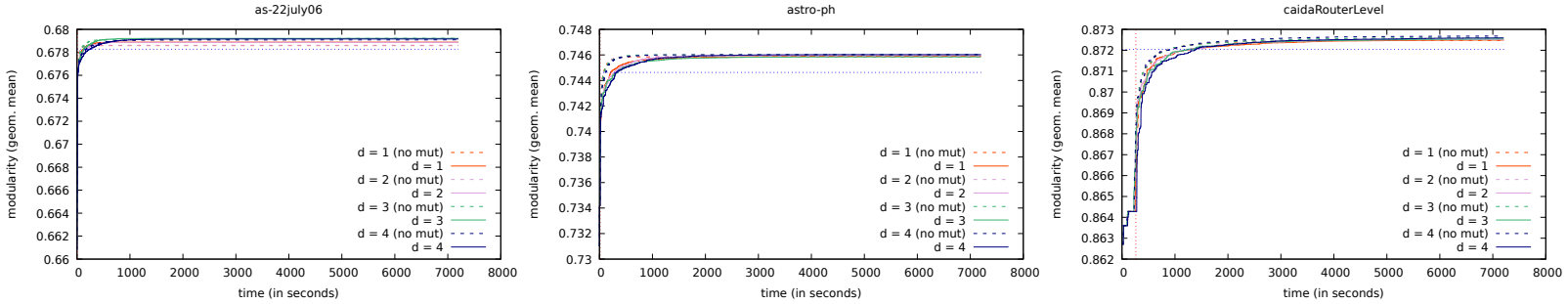
Appendix B

Detailed Results

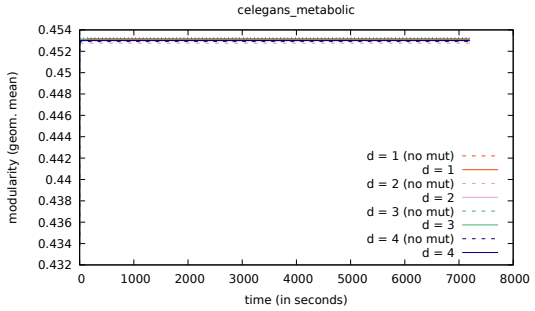
On the next few pages, we list and summarize all data that was raised during evaluation. Values are marked according to whether they reach the benchmark value or not. Tabulated data is not averaged and is result of one of five runs. The data represented in the figures is averaged by geometric mean over five runs.

B.1 Small Graphs

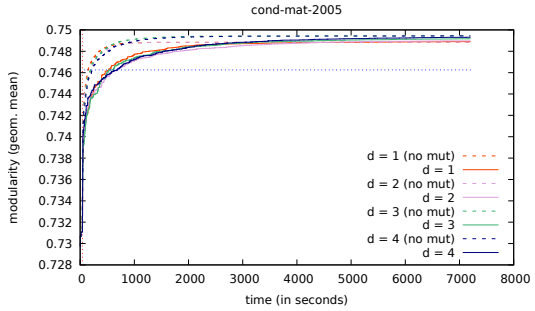
B.1.1 Plots



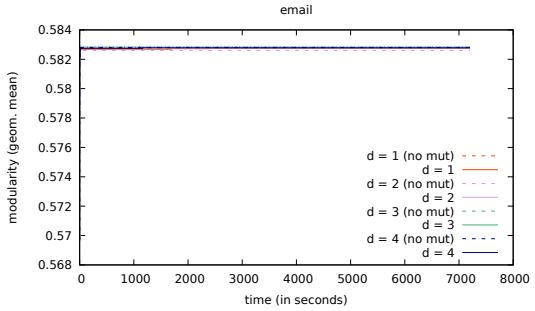
(a) (b) (c)


celegans_metabolic
modularity (geom. mean)
time (in seconds)
d = 1 (no mut)
d = 1
d = 2 (no mut)
d = 2
d = 3 (no mut)
d = 3
d = 4 (no mut)
d = 4

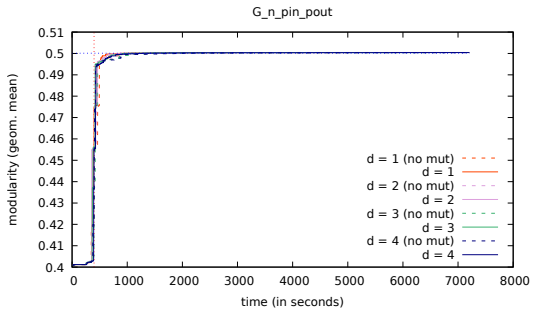
(d)


cond-mat-2005
modularity (geom. mean)
time (in seconds)
d = 1 (no mut)
d = 1
d = 2 (no mut)
d = 2
d = 3 (no mut)
d = 3
d = 4 (no mut)
d = 4

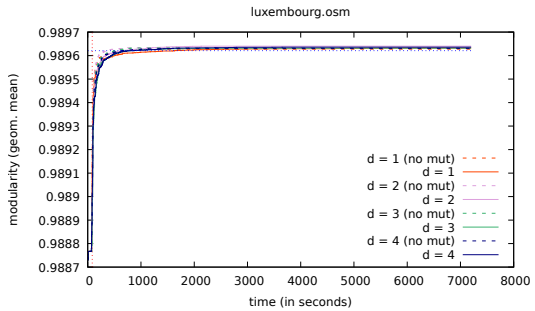
(e)


email
modularity (geom. mean)
time (in seconds)
d = 1 (no mut)
d = 1
d = 2 (no mut)
d = 2
d = 3 (no mut)
d = 3
d = 4 (no mut)
d = 4

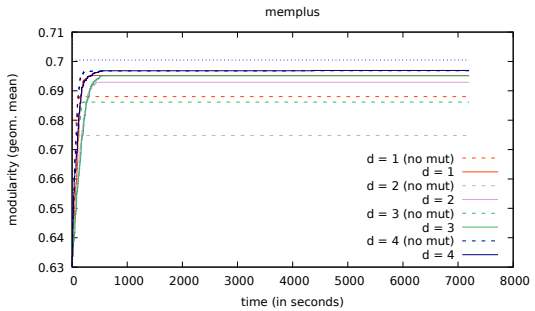
(f)


G_n_pin_pout
modularity (geom. mean)
time (in seconds)
d = 1 (no mut)
d = 1
d = 2 (no mut)
d = 2
d = 3 (no mut)
d = 3
d = 4 (no mut)
d = 4

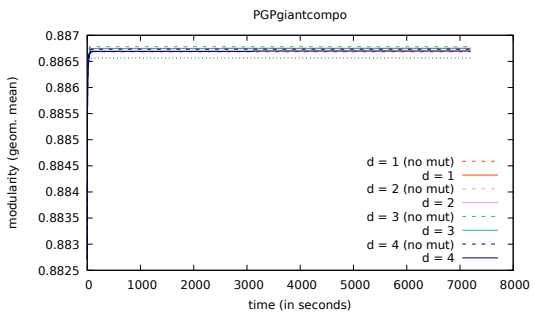
(g)


luxembourg.osm
modularity (geom. mean)
time (in seconds)
d = 1 (no mut)
d = 1
d = 2 (no mut)
d = 2
d = 3 (no mut)
d = 3
d = 4 (no mut)
d = 4

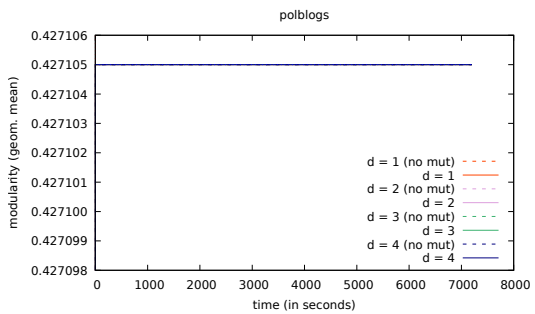
(h)


memplus
modularity (geom. mean)
time (in seconds)
d = 1 (no mut)
d = 1
d = 2 (no mut)
d = 2
d = 3 (no mut)
d = 3
d = 4 (no mut)
d = 4

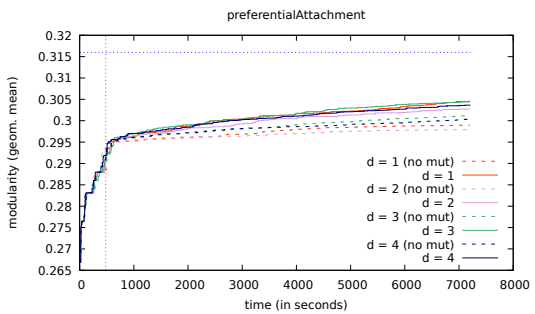
(i)


PGPgiantcompo
modularity (geom. mean)
time (in seconds)
d = 1 (no mut)
d = 1
d = 2 (no mut)
d = 2
d = 3 (no mut)
d = 3
d = 4 (no mut)
d = 4

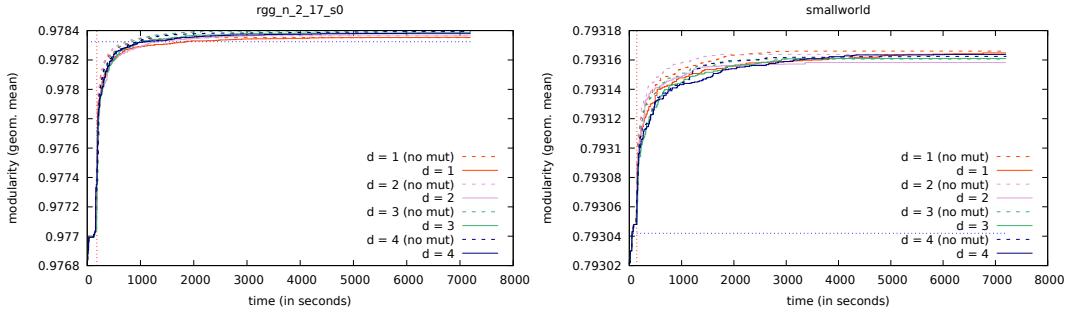
(j)


polblogs
modularity (geom. mean)
time (in seconds)
d = 1 (no mut)
d = 1
d = 2 (no mut)
d = 2
d = 3 (no mut)
d = 3
d = 4 (no mut)
d = 4

(k)


preferentialAttachment
modularity (geom. mean)
time (in seconds)
d = 1 (no mut)
d = 1
d = 2 (no mut)
d = 2
d = 3 (no mut)
d = 3
d = 4 (no mut)
d = 4

(l)



(m) (n)

Figure B.1: Result plots of small graphs data set

B.1.2 Tabulated Data

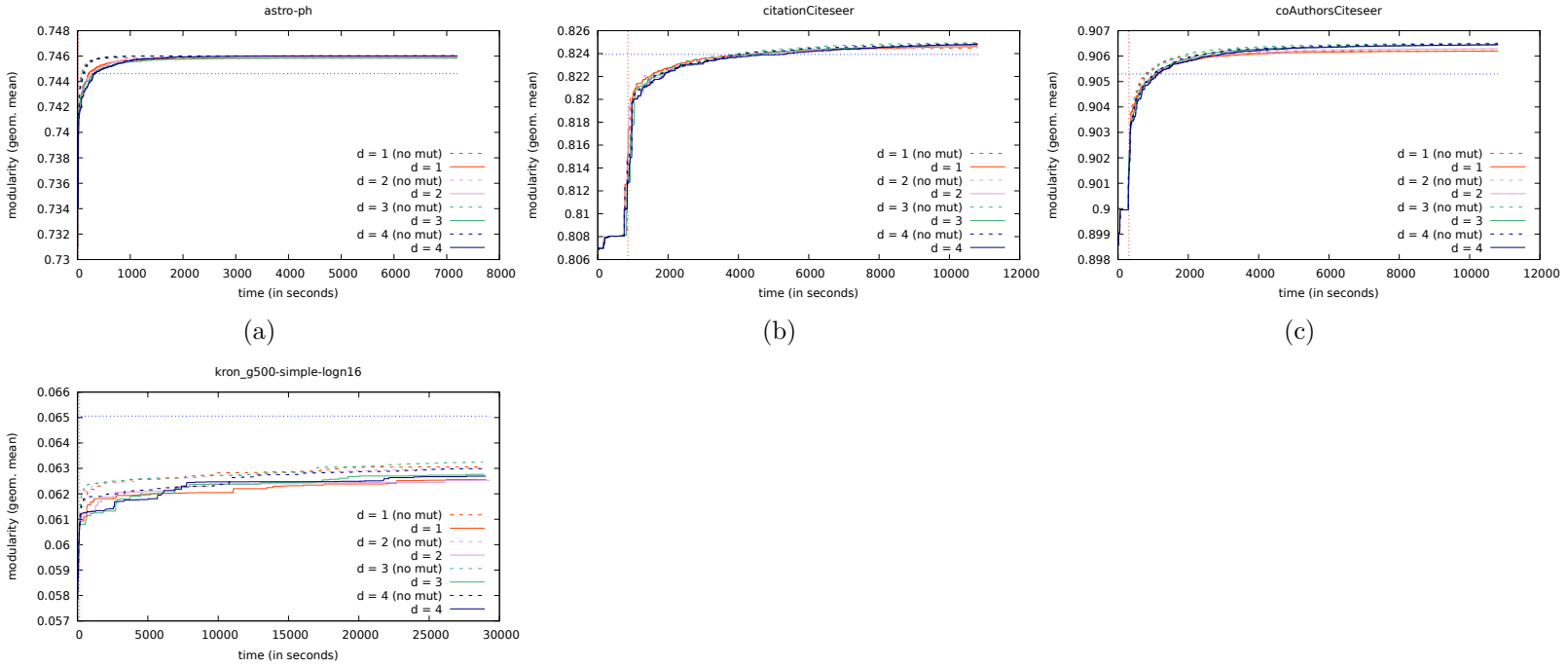
graph (configuration)	reference	mutation disabled *min*	mutation disabled *max*	mutation disabled *avg*	mutation enabled *min*	mutation enabled *max*	mutation enabled *avg*
as-22july06 (A)	0.678267	**0.678449**	**0.678792**	**0.678594**	**0.678715**	**0.679014**	**0.678891**
as-22july06 (B)	— " —	**0.678564**	**0.678660**	**0.678615**	**0.678594**	**0.679067**	**0.678867**
as-22july06 (C)	— " —	**0.679212**	**0.679232**	**0.679227**	**0.679101**	**0.679308**	**0.679216**
as-22july06 (D)	— " —	**0.678918**	**0.679238**	**0.679087**	**0.678877**	**0.679273**	**0.679165**
astro-ph (A)	0.744621	**0.745658**	**0.745998**	**0.745848**	**0.745846**	**0.746013**	**0.745909**
astro-ph (B)	— " —	**0.745730**	**0.746010**	**0.745889**	**0.745879**	**0.746029**	**0.745967**
astro-ph (C)	— " —	**0.745929**	**0.746084**	**0.746012**	**0.745462**	**0.746074**	**0.745842**
astro-ph (D)	— " —	**0.745883**	**0.746194**	**0.746029**	**0.745905**	**0.746127**	**0.746023**
caidaRouterLevel (A)	0.872042	**0.872438**	**0.872509**	**0.872472**	**0.872417**	**0.872551**	**0.872489**
caidaRouterLevel (B)	— " —	**0.872476**	**0.872577**	**0.872533**	**0.872490**	**0.872531**	**0.872515**
caidaRouterLevel (C)	— " —	**0.872567**	**0.872658**	**0.872606**	**0.872481**	**0.872594**	**0.872546**

caidaRouterLevel (D)	— " —	**0.872660**	**0.872700**	**0.872681**	**0.872519**	**0.872645**	**0.872594**
celegans_metabolic (A)	0.453248	0.452782	0.453209	0.452897	0.45285	0.453209	0.453092
celegans_metabolic (B)	— " —	0.452562	0.452944	0.452749	0.452664	*0.453248*	0.452974
celegans_metabolic (C)	— " —	0.45286	0.453209	0.453028	0.453024	*0.453248*	0.453188
celegans_metabolic (D)	— " —	0.452447	0.453209	0.452935	0.452673	0.453209	0.45301
cond-mat-2005 (A)	0.746254	**0.748689**	**0.748941**	**0.748837**	**0.748772**	**0.749030**	**0.748906**
cond-mat-2005 (B)	— " —	**0.748749**	**0.748968**	**0.748841**	**0.748891**	**0.749165**	**0.748974**
cond-mat-2005 (C)	— " —	**0.749207**	**0.749637**	**0.749433**	**0.749089**	**0.749320**	**0.749174**
cond-mat-2005 (D)	— " —	**0.749239**	**0.749528**	**0.749441**	**0.749131**	**0.749522**	**0.749315**
email (A)	0.582829	0.582669	*0.582829*	0.582768	0.582515	*0.582829*	0.582733
email (B)	— " —	0.582387	*0.582829*	0.582598	0.582532	*0.582829*	0.582759
email (C)	— " —	0.582746	0.582792	0.582777	0.582792	*0.582829*	0.582822
email (D)	— " —	0.582761	*0.582829*	0.582805	0.5827	*0.582829*	0.582782
G_n_pin_pout (A)	0.500098	**0.500443**	**0.500473**	**0.500461**	**0.500442**	**0.500466**	**0.500454**
G_n_pin_pout (B)	— " —	**0.500418**	**0.500445**	**0.500432**	**0.500437**	**0.500465**	**0.500453**
G_n_pin_pout (C)	— " —	**0.500413**	**0.500450**	**0.500433**	**0.500405**	**0.500449**	**0.500434**
G_n_pin_pout (D)	— " —	**0.500350**	**0.500414**	**0.500379**	**0.500408**	**0.500436**	**0.500422**
luxembourg.osm (A)	0.989621	0.989618	**0.989642**	**0.989627**	**0.989626**	**0.989639**	**0.989632**
luxembourg.osm (B)	— " —	**0.989627**	**0.989636**	**0.989634**	**0.989635**	**0.989647**	**0.989640**
luxembourg.osm (C)	— " —	**0.989628**	**0.989639**	**0.989633**	**0.989623**	**0.989636**	**0.989632**
luxembourg.osm (D)	— " —	**0.989627**	**0.989634**	**0.989630**	**0.989632**	**0.989639**	**0.989636**
memplus (A)	0.700473	0.68175	0.692157	0.688052	0.69484	0.695898	0.695223
memplus (B)	— " —	0.646283	0.685575	0.674964	0.688382	0.695057	0.692933
memplus (C)	— " —	0.676225	0.692612	0.686188	0.69184	0.698423	0.695112
memplus (D)	— " —	0.695134	0.699466	0.696837	0.694206	**0.701275**	0.696945

PGPgiantcompo (A)	0.886564	**0.886639**	**0.886745**	**0.886684**	**0.886672**	**0.886826**	**0.886749**
PGPgiantcompo (B)	— " —	**0.886625**	**0.886745**	**0.886684**	**0.886663**	**0.886742**	**0.886714**
PGPgiantcompo (C)	— " —	**0.886732**	**0.886825**	**0.886781**	**0.886713**	**0.886790**	**0.886749**
PGPgiantcompo (D)	— " —	**0.886693**	**0.886806**	**0.886733**	**0.886647**	**0.886767**	**0.886697**
polblogs (A)	0.427105	*0.427105*	*0.427105*	*0.427105*	*0.427105*	*0.427105*	*0.427105*
polblogs (B)	— " —	*0.427105*	*0.427105*	*0.427105*	*0.427105*	*0.427105*	*0.427105*
polblogs (C)	— " —	*0.427105*	*0.427105*	*0.427105*	*0.427105*	*0.427105*	*0.427105*
polblogs (D)	— " —	*0.427105*	*0.427105*	*0.427105*	*0.427105*	*0.427105*	*0.427105*
power (A)	0.940851	0.940774	**0.940864**	0.940815	0.940798	**0.940903**	**0.940854**
power (B)	— " —	0.940716	**0.940865**	0.940797	0.940788	0.940844	0.940819
power (C)	— " —	0.940792	**0.940918**	**0.940869**	**0.940890**	**0.940925**	**0.940912**
power (D)	— " —	0.940841	**0.940900**	**0.940875**	**0.940867**	**0.940925**	**0.940895**
preferentialAttachment (A)	0.315994	0.29793	0.299944	0.298928	0.303177	0.305404	0.304564
preferentialAttachment (B)	— " —	0.296291	0.300171	0.297922	0.301903	0.304154	0.302757
preferentialAttachment (C)	— " —	0.30045	0.302587	0.301199	0.30311	0.305846	0.304406
preferentialAttachment (D)	— " —	0.299596	0.300946	0.300336	0.302672	0.304219	0.303634
rgg_n_2_17_s0 (A)	0.978324	**0.978332**	**0.978363**	**0.978352**	**0.978331**	**0.978380**	**0.978354**
rgg_n_2_17_s0 (B)	— " —	**0.978356**	**0.978369**	**0.978364**	**0.978360**	**0.978381**	**0.978370**
rgg_n_2_17_s0 (C)	— " —	**0.978376**	**0.978413**	**0.978396**	**0.978365**	**0.978399**	**0.978384**
rgg_n_2_17_s0 (D)	— " —	**0.978368**	**0.978402**	**0.978393**	**0.978358**	**0.978392**	**0.978382**
smallworld (A)	0.793042	**0.793162**	**0.793168**	**0.793166**	**0.793160**	**0.793169**	**0.793165**
smallworld (B)	— " —	**0.793162**	**0.793167**	**0.793164**	**0.793153**	**0.793164**	**0.793158**
smallworld (C)	— " —	**0.793154**	**0.793167**	**0.793161**	**0.793154**	**0.793166**	**0.793161**
smallworld (D)	— " —	**0.793161**	**0.793165**	**0.793162**	**0.793162**	**0.793165**	**0.793164**

B.2 Medium-Sized Graphs

B.2.1 Plots



(a) (b) (c)

(d)

Figure B.2: Result plots of medium graphs data set

B.2.2 Tabulated Data

		mutation disabled			mutation enabled		
graph (configuration)	reference	*min*	*max*	*avg*	*min*	*max*	*avg*
belgium.osm (A)	0.99494	**0.995043**	**0.995045**	**0.995044**	**0.995038**	**0.995044**	**0.995041**
belgium.osm (B)	— " —	**0.995051**	**0.995058**	**0.995055**	**0.995051**	**0.995055**	**0.995054**
belgium.osm (C)	— " —	**0.995053**	**0.995059**	**0.995056**	**0.995046**	**0.995048**	**0.995047**
belgium.osm (D)	— " —	**0.995053**	**0.995057**	**0.995055**	**0.995045**	**0.995050**	**0.995047**
citationCiteseer (A)	0.82393	**0.824422**	**0.824554**	**0.824485**	**0.824783**	**0.824952**	**0.824850**
citationCiteseer (B)	— " —	**0.824640**	**0.824766**	**0.824683**	**0.824649**	**0.824950**	**0.824849**
citationCiteseer (C)	— " —	**0.825192**	**0.825263**	**0.825230**	**0.825124**	**0.825216**	**0.825177**
citationCiteseer (D)	— " —	**0.825210**	**0.825269**	**0.825244**	**0.825150**	**0.825239**	**0.825201**
coAuthorsCiteseer (A)	0.905297	**0.906133**	**0.906219**	**0.906177**	**0.906155**	**0.906205**	**0.906185**
coAuthorsCiteseer (B)	— " —	**0.906219**	**0.906262**	**0.906248**	**0.906234**	**0.906288**	**0.906254**
coAuthorsCiteseer (C)	— " —	**0.906499**	**0.906527**	**0.906512**	**0.906459**	**0.906509**	**0.906485**
coAuthorsCiteseer (D)	— " —	**0.906557**	**0.906612**	**0.906576**	**0.906513**	**0.906565**	**0.906539**
kron_g500-simple-logn16 (A)	0.065056	0.0627865	0.063472	0.063059	0.0621052	0.0632589	0.062566
kron_g500-simple-logn16 (B)	— " —	0.062691	0.0632864	0.063003	0.0623929	0.0627201	0.062531
kron_g500-simple-logn16 (C)	— " —	0.0630288	0.0638011	0.063247	0.0625878	0.0631697	0.062763
kron_g500-simple-logn16 (D)	— " —	0.0625694	0.0636884	0.062991	0.0624493	0.0630553	0.062693

B.3 Large Graphs

B.3.1 Plots

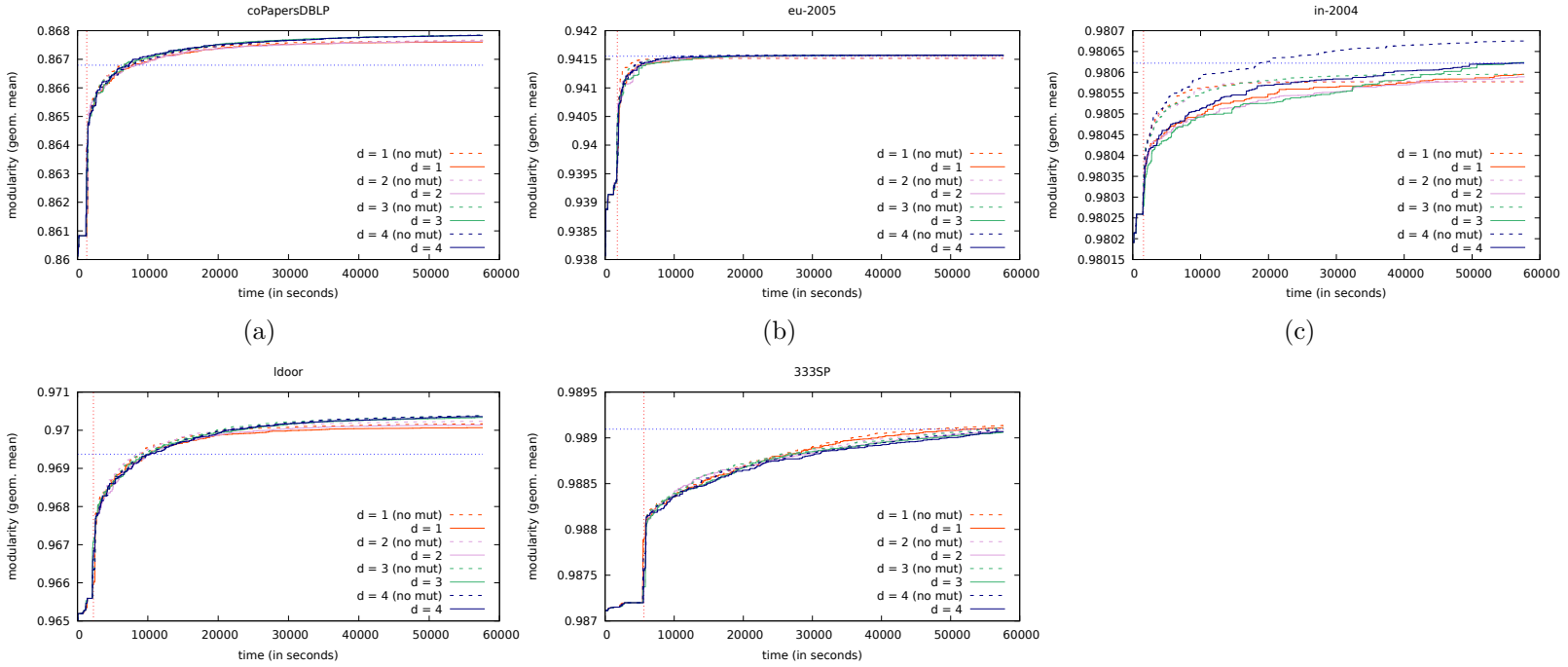


Figure B.3: Result plots of large graphs data set

B.3.2 Tabulated Data

graph (configuration)	reference	mutation disabled			mutation enabled		
		min	*max*	*avg*	*min*	*max*	*avg*
coPapersDBLP (A)	0.866794	**0.867563**	**0.867683**	**0.867641**	**0.867581**	**0.867643**	**0.867602**
coPapersDBLP (B)	— " —	**0.867653**	**0.867694**	**0.867674**	**0.867609**	**0.867657**	**0.867628**
coPapersDBLP (C)	— " —	**0.867784**	**0.867864**	**0.867841**	**0.867781**	**0.867851**	**0.867832**
coPapersDBLP (D)	— " —	**0.867823**	**0.867867**	**0.867844**	**0.867806**	**0.867880**	**0.867835**
eu-2005 (A)	0.941554	0.941516	0.941518	0.941517	**0.941555**	**0.941569**	**0.941564**
eu-2005 (B)	— " —	0.941516	0.941518	0.941517	**0.941561**	**0.941569**	**0.941566**
eu-2005 (C)	— " —	**0.941570**	**0.941572**	**0.941571**	**0.941570**	**0.941572**	**0.941571**
eu-2005 (D)	— " —	**0.941572**	**0.941574**	**0.941573**	**0.941571**	**0.941572**	**0.941572**
in-2004 (A)	0.980622	0.980574	0.98058	0.980577	0.980577	0.980603	0.980595
in-2004 (B)	— " —	0.980574	0.980583	0.980579	0.980577	0.980603	0.980589
in-2004 (C)	— " —	0.980579	0.980608	0.980595	0.980597	**0.980639**	**0.980624**
in-2004 (D)	— " —	**0.980672**	**0.980685**	**0.980675**	0.980595	**0.980634**	*0.980622*
ldoor (A)	0.96937	**0.970136**	**0.970207**	**0.970167**	**0.970038**	**0.970116**	**0.970068**
ldoor (B)	— " —	**0.970198**	**0.970287**	**0.970234**	**0.970109**	**0.970173**	**0.970137**
ldoor (C)	— " —	**0.970350**	**0.970399**	**0.970382**	**0.970270**	**0.970405**	**0.970334**
ldoor (D)	— " —	**0.970351**	**0.970415**	**0.970381**	**0.970334**	**0.970380**	**0.970358**
333SP (A)	0.989096	**0.989119**	**0.989148**	**0.989135**	0.989091	**0.989130**	**0.989109**
333SP (B)	— " —	0.989089	**0.989136**	**0.989107**	0.989042	0.989092	0.98907
333SP (C)	— " —	0.989091	**0.989155**	**0.989110**	0.98903	0.989084	0.98906
333SP (D)	— " —	0.989037	**0.989132**	0.989089	0.989025	0.989091	0.989068

Affidavit

Ich versichere, dass die Arbeit von mir selbständig verfasst wurde und dass ich keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Weiterhin wurde diese Arbeit keiner anderen Prüfungsbehörde übergeben.

Sonja Biedermann
Wien, den 14. September 2017