Design and experimental evaluation of a local graph clustering algorithm

Christian Schulz

June 2, 2008

1232808

Student thesis

at

Institut for Theoretical Computer Science, Algorithmics I
Universitaet Karlsruhe (TH)

Supervisors:
Prof. Dr. Dorothea Wagner,
Robert Görke, Daniel Delling

Acknowledgment

I would like to express my gratitude to all those who gave me the possibility to complete this paper. I want to thank my advisors Daniel Delling and Robert Görke.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig angefertigt habe und nur die angegebenen Hilfsmittel und Quellen verwendet wurden.

Karlsruhe, den 2. Juni 2008

Abstract

Finding natural groups in large graphs is a field with many applications. Applications, such as analyzing social networks or analyzing the web, provide graphs with a node count of up to millions if not billions. The *best known algorithm* has a running time $O(n^2 \log n)$ for sparse graphs (see Related Work/Greedy). Roughly speaking, current algorithms are too slow to analyze such networks.
We present a new algorithm for clustering graphs, based on the *contraction* of dense regions with weight updates and inserting *shortcuts*. In contrast to most known algorithms, which work in a global way, we use *local operations* and we do not directly optimize an objective function. We detect dense regions with a local search approach. The contraction order is obtained by using a simple priority function measuring the expected weight per node. Our algorithm can be used as a clustering algorithm for itself or simply to *reduce the search space* for the greedy algorithm.
This thesis is a feasibility-study which points out that it is possible to implement a local clustering algorithm. We do not analyse the running time of our algorithm.

Contents

1 Introduction

Clustering is one of the most widely used methods for *investigative data analysis*. It has a very wide field of applications including data mining, gene analysis, protein domain decomposition and analyzing social networks. In round terms, clustering consists of *detecting natural groups* of elements in data sets which are similar. Graph clustering is an interesting variant of data clustering, in which the separation of sparsely connected dense subgraphs from each other is a main goal. In other words, a graph clustering algorithm discovers groups which are internally dense and only sparsely connected between each other.
A important application delivering very large graphs to be clustered, arises in the area of scientific computing ([14], [13]). Engineers make extensive use of Finite Element Methods (FEM) to analyze a variety of physical processes which can be described by partial differential equations (PDE). Therefore, the simulation area has to be dissected into simple geometric elements, e.g. triangles, to approximate the solution of the partial differential equation.
However, the approximation quality becomes better, the finer this discretization is performed. But dissecting the simulation area in more elements also results in growing computational complexity. State-of-the-art simulations make use of many millions of elements. To overcome the computational effort a parallel computer is used. The parallelization of numerical simulation algorithms usually follows the *Single-Program Multiple-Data* paradigm: Each of the P processors executes the same code on a different part of the data. Thus, the mesh has to be split into sub-domains, each being assigned to one processor.
For using a parallel system efficiently an even work load distribution onto the processors is required. All processors should roughly contain the same number of elements to minimize the overall computation time. Furthermore, since iterative algorithms perform mainly local operations, the parallel algorithm mostly requires communication at the partition boundaries. Communication between the processors should be kept minimal due to the relatively high latency caused by communications. Hence, the partition boundaries should be as small as possible due to the high communication costs involved.
The relationships between the elements can be *modeled as a graph*, where the computations are represented by nodes and the data dependencies by edges. A common algorithm to distribute the computational work onto the processors consists of dividing the nodes of this graph into equally sized sets (partitions) such that as few edges as possible connect vertices that are placed in different partitions. This matches the classical *graph partitioning problem*. Although the problem is a bit different from our clustering problem, it can be approximated using a clustering algorithm. First we generate cluster hierarchies. Then we take that step of the hierarchy with the minimal number of clusters higher than the count of processors k.

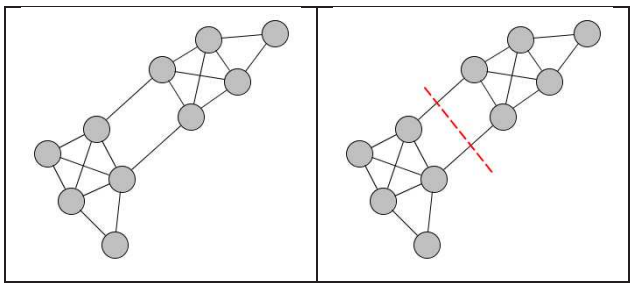


Figure 1.1: Example: Computations are represented by nodes and the data dependencies by edges. The cut minimizes the communication and has optimal work load.

Afterwards, it is possible to merge clusters greedily regarding size and quality until only k clusters in the clustering are left. The resulting clusters can be used for the work-load distribution.

2 Fundamentals

2.1 General Definitions

We are now going to repeat some general definitions from [11] and [12] which we need later. Throughout this paper we write $G = (V, E)$ for an undirected, unweighted, simple graph. The set of vertices is denoted by V and E is the set of edges. We denote $n = |V|$ and $m = |E|$. If the graph is weighted we write $G = (V, E, \omega)$, where the mapping $\omega : E \to [0, 1]$ is the *weight function*. The weight function represents the strength of the similarity relation between two nodes v_1, v_2 modeled by the edges. For the unweighted case, the weight function is assumed to be constantly one.
We say two nodes v, w are equal if and only if $\omega(\{v, w\}) = 1$ and for example two nodes are *50 percent similar* if and only if $\omega(\{v, w\}) = 0.5$.
In the following we use an abbreviation for summing up the weight of an edge subset $E' \subseteq E$:

$$\omega(E') = \sum_{e \in E'} \omega(e)$$

Definition 1 (Subgraph)**.** *A graph $C = (V_C, E_C)$ is called* subgraph *of a graph $G = (V, E)$ if $V_C \subseteq V$ and $E_C \subseteq E$. For $V' \subseteq V$ and $E(V') := \{\{v, w\} \in E \mid v, w \in V'\}$ we define*

$$G[V'] := (V', E(V'))$$

as the node-induced subgraph*.*

Definition 2 (Clustering)**.** *Given a graph $G = (V, E)$ a* clustering $\mathcal{C} = \{C_1, \cdots, C_k\}$ *is a* partition *of V i.e.*

1. $V = \bigcup_{i=1}^{k} C_i$
2. $i \neq j \Rightarrow C_i \cap C_j = \emptyset$
3. $C_i \neq \emptyset \quad \forall i \in \{1, \cdots, k\}$.

The node-induced subgraphs $G[C_i]$ are known as clusters*. In the following, we often identify a cluster C_i with its node-induced subgraph.*

The set $E(\mathcal{C}) := \bigcup_{i=1}^{k} E(C_i)$ is the set of *intra-cluster edges* and $E \backslash E(\mathcal{C})$ the *inter-cluster edges*. The number of intra-cluster edges is denoted by $m(\mathcal{C}) = |E(\mathcal{C})|$ and the number of inter-cluster edges by $\overline{m}(\mathcal{C}) = |E \backslash E(\mathcal{C})|$. Given a graph $G = (V, E, \omega)$ we define $\mathcal{A}(G)$ as the set of all possible clusterings of G. A clustering $\mathcal{C}_1 := \{C_1, \ldots, C_k\}$ is called a *refinement* of $\mathcal{C}_2 := \{C'_1, \ldots, C'_l\}$ iff

$$\mathcal{C}_1 \leq \mathcal{C}_2 :\Leftrightarrow \forall i \in \{1..k\}\ \exists j \in \{1, \ldots, l\} : C_i \subseteq C'_j$$

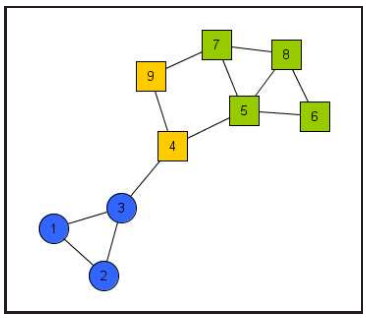


Figure 2.1: Two clusterings indicated by node colors and node shapes. The clustering indicated by node shapes is a coarsening of the clustering induced by node colors.

The clustering $\mathcal{C}_2$ is called a *coarsening* of $\mathcal{C}_1$. A subset ρ of $\mathcal{A}(G)$ such that every pair is comparable in ρ is a *hierarchy*.

Definition 3 (Neighborhood)**.** *Let $G = (V, E, \omega)$ be a weighted graph. For a node $v \in V$ the set*

$$N(v) := \{w \mid \{v, w\} \in E\}$$

is called the (standard)-neighborhood *and we denote the* Dijkstra neighborhood *with*

$$N_d(v) := \{w \mid dist(v, w) \leq d, v \neq w\}$$

where $dist(v, w)$ is the length of the shortest path *between v and w. d is the maximal distance to v in the set. Note that for $\omega(e) \equiv 1$, we get $N(v) = N_1(v)$.*

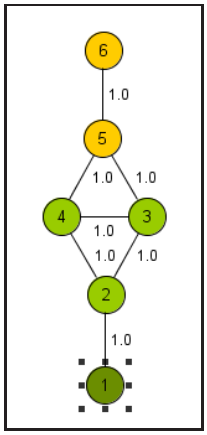


Figure 2.2: Dijkstra neighborhood $N_{2.0}(1)$

Definition 4 (Degree)**.** *Given a graph $G = (V, E, \omega)$, the* degree of a node $v \in V$ *is defined by*

$$deg(v) := |N(v)|$$

and regarding ω we denote the weighted degree of a node $v \in V$ *by*

$$deg_\omega(v) := \sum_{e=\{v,w\}, w \in N(v)} \omega(e)$$

2.2 Quality Indices

Clustering techniques are used to find groups of nodes that are internally dense and that are only sparsely connected with each other. The problem with this formulation is, that it is based on our intuition. For an algorithmic approach, we need a measure which tells us whether a clustering is *good or not*. Furthermore, such quality indices allow us to *compare* two different clusterings. We only give a short summary on quality indices. For more information we refer the reader to [11].

2.2.1 Coverage

The coverage($\mathcal{C}$) of a graph clustering $\mathcal{C}$ is the fraction of the *weight of all intra-cluster edges* within the complete weight of all edges, i.e.

$$\text{coverage}(\mathcal{C}) = \frac{w(E(\mathcal{C}))}{w(E)} \ .$$

We get the unweighted case by setting $\omega(e) \equiv 1$. Intuitively, the larger the value of coverage($\mathcal{C}$) the better the quality of a clustering $\mathcal{C}$. A disadvantage of coverage is that $\mathcal{C} = \{V\}$ and "mincuts" achieve the maximum value if the graph has more than one connected component. According to this quality index such trivial clusterings are optimal. However, these clusterings cannot be considered to be good clusterings for general graphs and therefore coverage is rarely used as the only quality measurement of a clustering.

2.2.2 Performance

For the unweighted case the performance($\mathcal{C}$) of a clustering $\mathcal{C}$ counts the number of "*correctly classified pairs of nodes*" in a graph. In this context a "correctly" classified pair of nodes means two nodes either belonging to the same cluster and connected by an edge, or belonging to different clusters and not connected by an edge. With $f(\mathcal{C}) = \sum_{i=1}^{k} |E(C_i)|$ which counts the edges inside clusters and $g(\mathcal{C}) = \sum_{u,v \in V} [\{u,v\} \notin E] \cdot [u \in C_i, v \in C_j, i \neq j]$ which counts the edges between clusters, we get

$$\text{performance}(\mathcal{C}) := \frac{f(\mathcal{C}) + g(\mathcal{C})}{\frac{1}{2}n \cdot (n-1)} \ .$$

The definition is given in *Iversion Notation*: the terms inside the parentheses can be any logical statement. True statements are evaluated to 1, false statements to 0. In [15] it is proved that it is NP-hard to calculate the maximum of $f + g$. Therefore $n \cdot (n-1)/2$ is used as upper bound for the maximum of $f + g$ (there are $n \cdot (n-1)/2$ different node pairs). Clusterings with high performance tend to have many clusters.
It is possible to define performance for the weighted case. On first sight, it is not clear how to assign a value to edges belonging to different clusters and not connected by an edge. Therefore, we need a meaningful maximum edgeweight M. We use the maximum weight of the graph $M := \max_{e \in E} \omega(e)$ and set

$$\text{performance}(\mathcal{C}) := \frac{\omega(\mathcal{C}) + M \cdot g(\mathcal{C})}{M\frac{1}{2}n \cdot (n-1)} \ .$$

2.2.3 Modularity

The quality index coverage has the disadvantage that $\mathcal{C} = \{V\}$ and mincuts achieve the maximum value 1 if there is more than one connected component. Thus it is not a good measure of community structure. To fix this problem a new quality index has been introduced by [7]. The main idea is to "subtract from [the measure] the expected value $[\cdots]$". We get

$$\begin{aligned} \text{modularity}(\mathcal{C}) &= cov(\mathcal{C}) - \mathbb{E}[cov(\mathcal{C})] \\ &= \frac{|E(\mathcal{C})|}{|E|} - \frac{1}{4|E|^2} \sum_{C \in \mathcal{C}} \left(\sum_{v \in C} deg(v) \right)^2 \end{aligned}$$

for the unweighted case and

$$\begin{aligned} \text{modularity}(\mathcal{C}) &= cov(\mathcal{C}) - \mathbb{E}[cov(\mathcal{C})] \\ &= \frac{\omega(E(\mathcal{C}))}{\omega(E)} - \frac{1}{4(\omega(E))^2} \sum_{C \in \mathcal{C}} \left(\sum_{v \in C} deg_\omega(v) \right)^2 \end{aligned}$$

for the weighted case.
It is shown in [3] that modularity maps into $(-\frac{1}{2}, 1]$ and may be negative. If the value of modularity for a given clustering $\mathcal{C}$ is close to zero, then $\mathcal{C}$ is not much better than a random clustering. A high value of modularity is a *good indicator for a significant clustering*. The quality index has a high acceptance in the community and is currently the standard measure for clusterings. For more information on modularity we refer the reader to [3].

2.2.4 Example

In Figure 2.3, we present an example to point out that the disadvantages of coverage and performance are not only theoretical results. We compare the clustering in which each node has its own cluster with the large clustering and the clustering induced by the colors of the nodes. Table 2.1 shows the numerical results achieved by the clusterings.

Quality Index	Singletons	Large cluster	Color-induced clustering
Coverage unweighted	0.0	**1.0**	0.833
Coverage weighted	0.0	**1.0**	0.826
Performance unweighted	0.769	0.231	**0.871**
Performance weighted	**0.831**	0.168	0.827
Modularity unweighted	-0.086	0.0	**0.498**
Modularity weighted	-0.088	0.0	**0.491**

Table 2.1: The results regarding the quality index and different clusterings for the sample graph in figure 2.3.

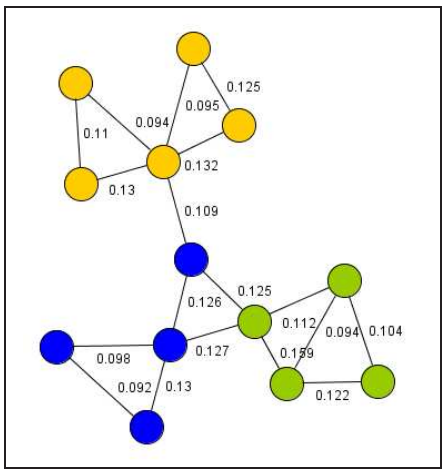


Figure 2.3: A sample graph with a clustering induced by node-colors. The weights of the edges are generated regarding the euclidean distances of the nodes. The evaluated quality indices are shown beneath.

The winners regarding the current quality index are emphasized. It is obvious that the large cluster and singletons are not "good" clusterings. Anyhow, the large cluster wins regarding coverage and the singletons clustering wins regarding performance. Furthermore, we can see that the difference for the unweighted performance value between the color-induced clustering and the singleton clustering is rather little. Looking at the modularity values, we can clearly see that the values for the singleton clustering and the large clustering are not better than a random clustering. The color-induced clustering achieves good modularity values.

3 Related work

This chapter presents four clustering algorithms related to this thesis. The first three algorithms make use of the *normalized adjacency matrix* of G. This matrix is defined by $M(G) := D(G)^{-1}A(G)$ where $A(G)$ is the adjacency matrix and $D(G) = diag(\deg_\omega(v_1), \cdots, \deg_\omega(v_n))$.

3.1 Iterative Conductance Cutting (ICC)

The *Iterative Conductance Cutting* clustering algorithm, proposed by [17], works in a hierarchical way. The algorithm is a top down approach. The idea is to cut the graph in two nearly equalized subgraphs with minimal edge count between the subgraphs and then proceed with the subgraphs until a threshold is reached. For the formulation of the algorithm we need the definition of *conductance cuts*. Conductance compares the weight of the cut with the edge weight in one of the two induced subgraphs. Thus, conductance can be seen as a measure for bottlenecks.

Definition 5 (Conductance)**.** *Let $G = (V, E, \omega)$ be a graph and $\mathcal{C}' = (C, V \backslash C)$ be a cut, then the* conductance of a cut *is defined by*

$$\phi(\mathcal{C}') = \begin{cases} 1 & , \text{ if } C \in \{\emptyset, V\} \\ 0 & , \text{ if } C \notin \{\emptyset, V\}, \omega(\overline{E(\mathcal{C}')}) = 0 \\ \frac{\omega(\overline{E(\mathcal{C}')})}{min(\sum_{v \in C} deg_\omega(v), \sum_{v \in V \backslash C} deg_\omega(v))} & , \text{ otherwise} \end{cases}$$

and the conductance of the graph G is denoted by

$$\phi(G) = \min_{C \subseteq V} \phi((C, V \backslash C))$$

Definition 6 (Intra-cluster Conductance)**.** *Let $G = (V, E, \omega)$ be a graph and $\mathcal{C} = \{C_1, \cdots, C_n\}$ a clustering. Then the* intra-cluster conductance α *is defined as the minimum conductance occurring in the cluster-induced subgraphs $G[C_i]$:*

$$\alpha(\mathcal{C}) = \min_{1 \leq i \leq k} \phi(G[C_i])$$

A cut has a *small conductance value* if its size is small relative to the density of either side of the cut. The algorithm starts by resetting the clustering to one large cluster and proceeds by iteratively cutting the cluster with a minimum conductance cut as long as the quality measure of the two resulting parts is below a threshold α^*. The iteration stops if it is not possible to cut a remaining cluster within the threshold. However, finding such cuts is NP-hard ([1]). Therefore, a heuristic has to be used. In [5] and [6] it is shown, that it is possible to *approximate*

Algorithm 1 Iterative Conductance Cutting (ICC)

Input: $G = (V, E, \omega)$, conductance threshold $0 < \alpha^* < 1$

1: $\mathcal{C} \leftarrow \{V\}$
2: **while** $\exists C \in \mathcal{C} : \phi(G[C]) < \alpha^*$ **do**
3: $\quad (C', C \backslash C') \leftarrow$ approximate minimum conductance cut in $G[C]$
4: $\quad \mathcal{C} \leftarrow (\mathcal{C} \backslash \{C\}) \cup \{C', C \backslash C'\}$
5: **end while**

the minimum conductance cut of a graph with a poly-logarithmic approximation guarantee in general. The approximation makes use of the eigenvector of $M(G[C_i]))$ associated with the second largest eigenvalue.
A disadvantage of the algorithm is that the estimation of a conductance cut is NP-hard and therefore it has to be approximated.

3.2 Geometric MST Clustering (GMC)

The *Geometric MST Clustering* algorithm was introduced by Gaertler in his master thesis ([10]). The algorithm first calculates a geometric embedding of the graph G and then uses an extension of the Minimum Spanning Tree clustering technique by [19]. The embedding of G is constructed from d distinct eigenvectors $x_1, \cdots, x_d$ of $M(G)$ associated with the largest eigenvalues less than 1. Note that eigenvectors are used due to their partitioning properties (Thm. 3.34 [10]). We now present the algorithm as it is proposed in [4].

Algorithm 2 Geometric MST Clustering (GMC)

Input: $G = (V, E, \omega)$, embedding dimension d, clustering valuation *quality*

1: $(1, \lambda_1, \cdots, \lambda_d) \leftarrow d + 1$ largest eigenvalues of $M(G)$
2: $d' \leftarrow \max\{i : 1 \leq i \leq d, \lambda_i > 0\}$
3: $x^{(1)}, \cdots, x^{(d')} \leftarrow$ eigenvectors of $M(G)$ associated with $\lambda_1, \cdots, \lambda_{d'}$
4: **for all** $e = (u, v) \in E$ **do**
5: $\quad \omega(e) \leftarrow \sum_{i=1}^{d'} |x_u^{(i)} - x_v^{(i)}|$
6: **end for**
7: $T \leftarrow$ MST of G with respect to ω
8: $\mathcal{C} \leftarrow \mathcal{C}(\tau)$ for which $quality(\mathcal{C}(\tau))$ is maximum over all $\tau \in \{\omega(e) : e \in T\}$

$C(\tau)$ is the clustering induced by the connected components of the forest induced by all edges of T with weight at most τ. It is remarkable that the algorithm has no parameters which fix any cluster property. A disadvantage is that it is time-consuming to calculate the embedding.

3.3 Markov Clustering (MCL)

MCL is short for *Markov Clustering Algorithm*. [16] proposed the algorithm in 2000. The main idea is that "a random walk in G that visits a dense cluster will likely not leave the cluster until many of its vertices have been visited". A *random walk* is a path starting at a random

vertex and then repeatedly moving to a neighbor in G with equal probabilities. The algorithm does not simulate random walks, but iteratively modifies a matrix of transition probabilities. We now present the algorithm as it is proposed in [4]. Note, the start matrix $M \leftarrow M(G)$ corresponds to random walks having a length not exceeding one.

Algorithm 3 Markov Clustering (MCL)

Input: $G = (V, E)$, expansion parameter e, inflation parameter r

1: $M \leftarrow M(G)$
2: **while** M is not a fixpoint **do**
3: *//simulate e steps of random walk*
4: $M \leftarrow M^e$
5: *//re-normalize the transition probabilities*
6: **for all** $u \in V$ **do**
7: **for all** $v \in V$ **do**
8: $M_{uv} \leftarrow M_{uv}^r$
9: **end for**
10: **for all** $v \in V$ **do**
11: $M_{uv} \leftarrow \frac{M_{uv}}{\sum_{w \in V} M_{uw}}$
12: **end for**
13: **end for**
14: **end while**
15: $H \leftarrow$ graph induced by non-zero entries of M
16: $\mathcal{C} \leftarrow$ clustering induced by connected components of H

It is argued in [16] that the algorithm in all likelihood ends up in a fixpoint or a recurrent state. A disadvantage of the algorithm is that it executes in every iteration a matrix multiplication which dominates the runningtime. This operation is very expensive for large n but can be accelerated using a parallel computer. The algorithm has no parameters which fix any cluster property which is remarkable.

3.4 Greedy (Newman)

The last algorithm we present, was introduced by Newman [7]. The algorithm starts with singletons and iteratively merges the clusters which lead to a maximal increasement of modularity. Therefore, this algorithm is a bottom up approach which is in contrast to the ICC algorithm working top down. Note, merging cluster pairs iteratively results in a hierarchy as well. This hierarchy can be represented as tree diagram also known as dendrogram. In [7] it is prooved that this algorithm can be implemented in $O(md \log n)$, where d is the depth of the resulting dendrogram. It is possible that $d \sim n$ if the dendrogram is a degenerated tree. This leads to a asymptotic running time $O(n^2 \log n)$ for sparse graphs $m \sim n$. For good-natured sparse graphs we get $d \sim \log n$ and thus a running time $O(n \log^2 n)$. An advantage of this algorithm is that it always achieves good results regarding modularity. Therefore, the calculated clustering mostly catches the intuition. On the other hand, the runningtime of the algorithm is too slow for really large graphs.

Algorithm 4 Greedy (Newman)

Input: $G = (V, E, \omega)$

1: *//start with singletons*
2: $\mathcal{C} \leftarrow \bigcup_{v \in V} \{\{v\}\}$
3: **while** $\mathcal{C}$ *is not the large cluster* $\{V\}$ **do**
4: merge the cluster pair increasing modularity the most
5: **end while**
6: **return** the occurred clustering with highest modularity value

4 Orca - Orca reduction clustering algorithm

We now formulate our algorithm. All currently known algorithms work in a global way. Our algorithm works in a local way by using *local operations*. This is promising since the distance of two nodes v_1, v_2 belonging to the same cluster is most likely small. Therefore it is intimating to use local operations to identify dense regions and iteratively contract them to reduce the size of the input. A global perspective seems to be overeager.
Mainly our algorithm consists of *three phases*. At the beginning we *remove* all *nodes* having degree one or less. These nodes are later assigned to the cluster of their neighbor or stay as singletons if they have degree 0. In the first phase of the algorithm we *find dense regions* which are contracted to a *super node* afterwards. We repeat this step until it is not possible to find anymore dense regions. To ensure that we can proceed with the first two steps, we *insert shortcuts*. The three phases are executed until we have only one node left. Note that this process creates hierarchies just as the greedy algorithm and the iterative conductance cutting algorithm.

4.1 Remove Nodes

As a first step of our algorithm, we remove nodes having degree 1 or less and iterate this procedure until all remaining nodes have a degree higher than 1. The removed nodes are later assigned to the cluster of their original neighbor. The idea behind this is that it seems obvious for nodes with degree 1 to belong to the cluster of their neighbor. In [3], it is shown that a clustering with maximum modularity has no cluster that consists of a single node with degree 1. Figure 4.1 gives an example. By not doing this we could have a singleton connected to a "dense" region. The pseudocode of this routine is given in Algorithm 5. Note that on the

Algorithm 5 DEGREE-ONE-REMOVAL

Input: $G = (V, E, \omega)$

1: **while** $\exists v \in V$ with $\deg(v) = 1$ **do**
2: remove v
3: **end while**

other hand, the approach is counterintuitive if the length of the removed path is too long. But long paths at dense regions do not occure often real world graphs. This part of our algorithm can be implemented in $O(m \max(\Delta, \log n))$ time (see [2]).

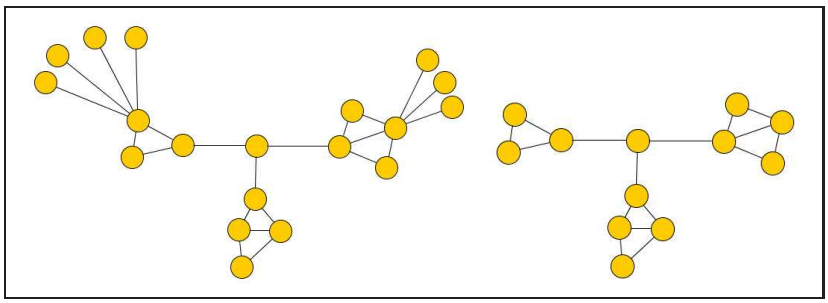


Figure 4.1: An execution of Degree One Removal. The nodes with degree 1 have been removed.

4.2 Fast Dense Region Detection

The next step of algorithm is the detection of dense regions. A dense region is a subset of V which is highly connected. The detected dense regions are later assigned to the same cluster and get contracted to a super node to reduce the search space and continue with the algorithm. To detect dense regions in the graph, we use a *local search approach*. Roughly speaking, we pick a node v of the graph and then compare its neighborhood with the neighborhood of its neighbors. If a neighbor has many neighbors of v, we add it to the current dense region. More precisely, we start at a node v and start then for every neighbor in the Dijkstra neighborhood a local search to determine their Dijkstra neighborhood. Since $\omega : E \to [0, 1]$ models similarities, we use $2-\omega$ as edge-weights for the Dijkstra-search. Consequently nodes which are less similar are more distant to each other in our search graph. What happens next is that every neighbor increments the "seen" attribute of all nodes in their Dijkstra neighborhood. A node with a high "seen" attribute can be accessed by many nodes in the Dijkstra neighborhood of the start node v. We take this as an indicator for a dense region and add this node to a potential dense region if it is higher than a number depending on a parameter γ and the size of the Dijkstra neighborhood. After we found a potential dense region starting from node v, we assign a

Algorithm 6 FAST-DENSE-REGION-DETECTION-LOCAL

Input: $G = (V, E, \omega)$, $\gamma \in \mathbb{R}^+$, search depth d, Start-node v
Output: Dense region

```
1:  Set denseregion ← {v}
2:  for all w ∈ N_d(v) do
3:     for all u ∈ N_d(w) do
4:        u.seen++
5:     end for
6:  end for
7:  for all w ∈ N_d(v) do
8:     if w.seen ≥ |N_d(v)| / γ then
9:        denseregion.add(w)
10:    end if
11: end for
12: return denseregion
```

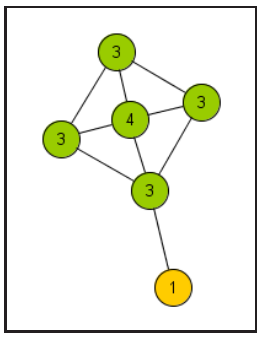


Figure 4.2: One iteration of our local search from node with number 4 with $\gamma = 2$ and search depth 1. The numbers indicate the "seen by neighbors" attribute. The green nodes have been recognized as dense region

priority to the region by using the function $\psi : \mathcal{P}(V) \to [0, 1]$.

$$\psi(\mathcal{D}) := \frac{\sum_{e \in E(\mathcal{D})} \omega(e)}{|\mathcal{D}|} \quad \text{for } \mathcal{D} \subseteq V$$

In other words, ψ measures the *expected weight per node*. Note, in the unweighted case $\psi(\mathcal{D}) = 1 \Leftrightarrow \mathcal{D}$ is a clique and furthermore $\psi(\mathcal{D}) = 0 \Leftrightarrow E(\mathcal{D}) = \emptyset$. The priorities deliver a contraction order and we contract regions with the highest priority first. Since $\psi(\mathcal{D}_1) \leq \psi(\mathcal{D}_2)$ implies that the weight per node of $\mathcal{D}_2$ is higher than the weight per node of $\mathcal{D}_1$, the contraction order trys to contract the densest regions first. By starting a local search for dense regions for every node $v \in V$, the contraction order is built up in Lines 2-5 of Algorithm 7. We need the procedure NOT-USED-FOR-LOCAL-SEARCH(Denseregion), so that the nodes which already belong to a dense region are not considered in further local-searches. This is necessary because we want to contract the regions and therefore a node belongs to one dense region only. In other words, the contraction can *change the priority* of other dense regions which means that the original priority can change. However, we do not update the priorities which have been calculated first since this would be too ineffizient. It is indeed possible to implement a priority lower bound for the contraction of regions.

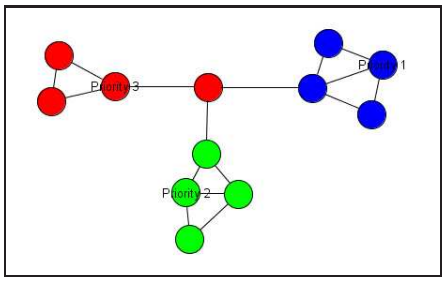


Figure 4.3: One iteration of the global dense region detection routine. The contraction order is shown through the priority labels and the corresponding dense regions are colored.

Algorithm 7 FAST-DENSE-REGION-DETECTION-GLOBAL

Input: $G = (V, E, \omega)$, $\gamma \in \mathbb{R}^+$, search depth d,

```
1: PriorityQueue pq ← ∅
2: for all v ∈ V do
3:    Denseregion ← FAST-DENSE-REGION-DETECTION-LOCAL(G, γ, d, v)
4:    pq.insert(v, ψ(Denseregion))
5: end for
6: List contractionlist
7: while !pq.isEmpty() do
8:    v ← pq.popMax()
9:    Denseregion ← FAST-DENSE-REGION-DETECTION-LOCAL(G, γ, d, v)
10:   NOT-USED-FOR-LOCAL-SEARCH(Denseregion)
11:   contractionlist.add(Denseregion)
12: end while
13: for all Denseregion ∈ contractionList do
14:   CONTRACTION(Denseregion)
15: end for
```

4.2.1 Contraction of Dense Regions

The next local operation is the contraction of dense regions. After we found dense regions, we contract them to a super node. This reduces the input and we can start the search for dense regions with the new smaller graph again. The process is repeated until it is not possible to find more dense regions in the current graph using the current γ. This can happen if the nodes are connected too weak.
After the contraction of a subset $\mathcal{D}$ to a super node s, the super node s gets all edges of their original nodes with new weights assigned. The weights are assigned regarding the size of $\mathcal{D}$ and the weight and multiplicity of the original edges. In the unweighted case the weight of a new edge is 1 if every node in the contracted region is connected to a node outside the region. That means if a region is highly connected to a node v, the created super node is highly connected to v too. The nodes in $\mathcal{D}$ are removed, after the creation of the super node. We give an example in Figure 4.4.

Algorithm 8 CONTRACTION

Input: $G = (V, E, \omega)$, Nodes to contract $\mathcal{D}$

```
1: create a super-node s in G
2: for all edges e = {v, w} with v ∈ D, w ∈ V\D do
3:    insert edge {s, w}
4:    ω({s, w}) ← (Σ_{ṽ∈D} ω({ṽ, w})) / |D|
5: end for
6: remove nodes D
```

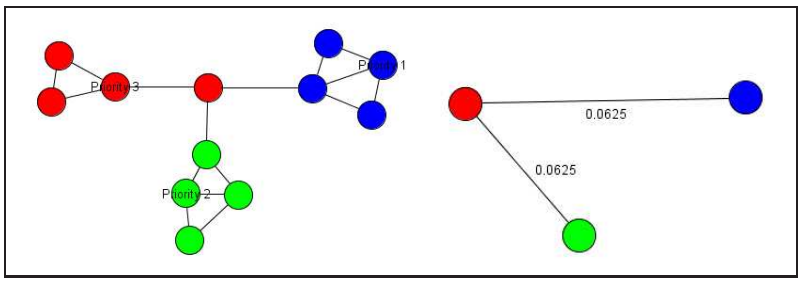


Figure 4.4: The found dense regions are contracted regarding the contraction order.

4.3 Densification via Shortcuts

The last local operation, we use in our algorithm, is the insertion of shortcuts. We mentioned in the section above that it is possible not to find a dense region if γ is too small or the graph is too weakly connected. In this case we insert shortcuts. Usually we call this part of our algorithm with $d = \delta := \min_{v \in V} \deg(v) > 1$. Using δ we ensure that the lost information is minimal. An example is given in Figure 4.5. The insertion of shortcuts with weight updates is done using the following Algorithm 9. The algorithm can be seen as a densifier since it removes a node and inserts edges. After applying this routine, we are able to continue with Algorithm 7. The new inserted edgeweight is the geometric mean of the weights of the incident edges.

Algorithm 9 SHORTCUTS

Input: $G = (V, E, \omega)$, degree $d > 1$

for all $v \in V$ **do**
  if $\deg(v) = d$ **then**
    for all pairs $p = \{v_1, v_2\}$ with $v_1, v_2 \in N(v)$ and $v_1 \neq v_2$ **do**
      if $\nexists$ edge between v_1 and v_2 **then**
        create edge between v_1 and v_2
      end if
      $\omega_1 \leftarrow \omega(v, v_1)$
      $\omega_2 \leftarrow \omega(v, v_2)$
      $\omega(v_1, v_2) \leftarrow \frac{1}{\frac{1}{\omega_1} + \frac{1}{\omega_2}}$
    end for
    remove v
  end if
end for

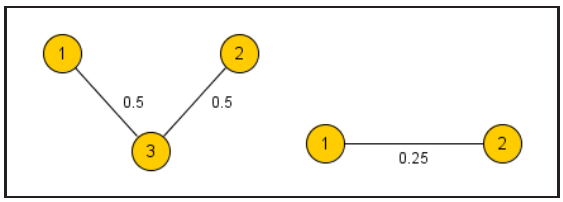


Figure 4.5: An execution with $d = 2$. A shortcut has been inserted from node 1 to 2.

4.4 Orca reduction clustering algorithm

With these *local operations* at hand, we are now able to assemble **O**ur **r**eduction **c**lustering **a**lgorithm. The global dense region contraction step is executed until it is not possible to find more dense regions. To proceed with the algorithm and ensure termination we insert shortcuts for all nodes having min-degree δ. The algorithm creates a hierarchy. The levels are set in line 5. Indeed, the hierarchies could be more granular if we set a new hierarchy level after the contraction of each dense region. To generate the final clustering on a hierarchy level, we

Algorithm 10 Our Algorithm

Input: $G = (V, E, \omega)$, $\gamma \in \mathbb{R}^+$, search depth d

```
1: DEGREE-ONE-REMOVAL(G)
2: while |V| > 2 do
3:    FAST-DENSE-REGION-DETECTION-GLOBAL(G, γ, d)
4:    while contracted nodes > 1 do
5:       //new hierarchy level
6:       FAST-DENSE-REGION-DETECTION-GLOBAL(G, γ, d)
7:    end while
8:    δ := max(min_{v∈V} deg(v), 2)
9:    SHORTCUTS(G,δ)
10: end while
```

assign nodes to the cluster of their original super node because they have been contracted to this node. Nodes which have been removed in the first step are assigned to the cluster of their neighbor. The resulting first hierarchy for the running example is shown in Figure 4.6.

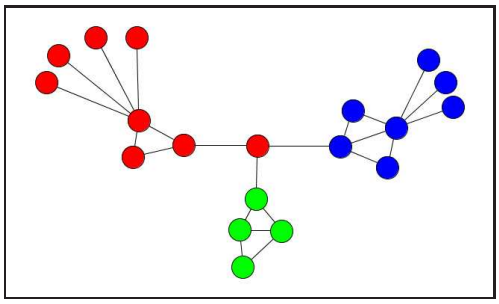


Figure 4.6: The resulting clustering on the first hierarchy level.

4.5 Post-Newman-Step

The greedy algorithm has a very slow start since it begins with singletons and its running time depends on the cluster count. Our algorithm can be used to overcome this slow start without a loss of quality, empirically.
On the first hierarchy level our algorithm produces a clustering which is *granular* if we choose a small γ. Choosing a small γ results in the fact, that nodes have to have nearly the same neighborhood to get into the same dense region. However, this *reduces the search space* for

clusterings and it is possible to apply Newman's algorithm on the given clustering. This algorithm merges iteratively the two clusters which increase modularity the most (see section 5). With this technique, we are able to increase the modularity value for our clustering in every hierarchy level. We present the pseudocode in Algorithm 11.

Algorithm 11 Post-Newman-Step

Input: $G = (V, E, \omega)$, $\gamma \in \mathbb{R}^+$, search depth d

1: apply our algorithm to obtain a clustering $\mathcal{C}$
2: apply Newman's greedy algorithm on the clustering $\mathcal{C}$

5 Experimental evaluation

To assure that our algorithm works, we started an experimental evaluation. Moreover, since our algorithm has two main parameters namely γ and search depth d, tests have been necessary to find the "best" parameters and to find parameters which make "no sense" at all. We started the systematic evaluation by using graph generators and then tested the algorithm with well known real world graphs. An advantage of a graph generator is that it creates a graph with a "hidden" clustering which serves for comparisons. The main goal is to find the "hidden" clustering or a clustering which has nearly the same quality. Through a generator it is possible to rate the capability of an algorithm to find the underlying structure and further to test it with a *large variety of graphs*. A disadvantage is that the generated graphs are perhaps a bit artificial. Therefore we used two different generators as well as two well known real world graphs for the evaluation.
We now present our test results. We first present the generators we used from [8] followed by the achieved results. At the end we present the results for two real world graphs.

5.1 Graph Generators

5.1.1 Attractor Generator

The attractor generator uses geometric properties to generate a significant clustering based on the following idea: k cluster centers, so called *attractor nodes*, are placed uniformly at random with a certain minimum distance t in the plane. Then, $n-k$ *satellite nodes* are assigned to the attractors and their corresponding clusters using the following policy. At a random position a satellite node u is inserted with probability $d(u,a)/t$, where $d(u,a)$ is the euclidean distance from u to the nearest attractor node a. If u is inserted, the edge $\{u,a\}$ is inserted.
The generator further takes a density parameter ρ controlling the *maximum distance* between two nodes that should be connected by an edge, $d_{max} = \frac{1}{2}t\rho$. After connecting all node pairs having a distance lower than the maximum distance, the generated cluster centers are deleted. We use an implementation which makes use of the unit square for modeling the plane and $t = \sqrt{2/(\pi k)}$ as a threshold minimum distance between attractor nodes. We give an example of the generation process in Figure 5.1.

5.1.2 Significant Gaussian Generators

First we describe the gaussian generator and after that the significant gaussian generator which we use for our tests. The gaussian generator requires three parameters: an approximate number of nodes n and two probabilities p_{in}, p_{out}. First an integer array P is generated. The array indicates the *partition of the nodes*, where $|P|$ is the number of clusters and each entry of the

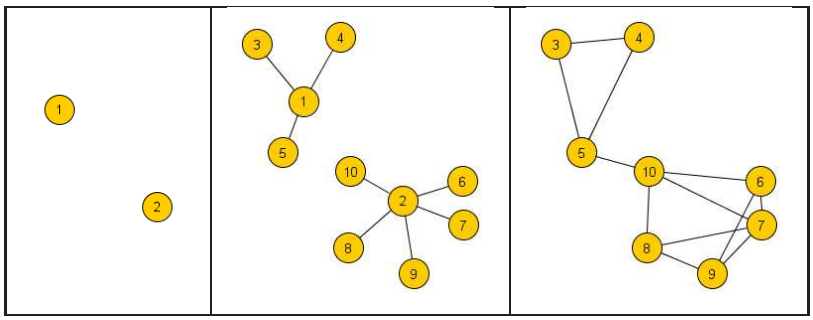


Figure 5.1: Left: *Two generated attractor nodes*, Middle: *After inserting satellite nodes*, Right: *After connecting nodes and deleting centers*

array indicates the size of the corresponding cluster. The number of clusters $|P|$ is chosen uniformly at random between $log_{10}(n)$ and $\sqrt{n}$. The mean of the entries of P is $a = \lfloor n/|P| \rfloor$ and the standard deviation d is $d = \lfloor a/4 \rfloor$. The sum of all entries of P approximately equals $|P| \cdot \lfloor n/|P| \rfloor$. The generator connects each node pair within the same cluster with probability p_{in} and a node pair in different clusters with probability p_{out}.
The significant gaussian generator substitutes the parameter p_{out} by an *edge-ratio* $\rho := \mathbb{E}(\overline{m}(\mathcal{C}))/\mathbb{E}(m(\mathcal{C}))$. This is needed because for increasing n and a fixed pair (p_{in}, p_{out}) the growth of inter-cluster edges exceeds the growth of intra-cluster edges. Note that ρ is highly dependent on the number of clusters. The generator initially creates a gaussian partition as described above, dynamically calculates p_{out} according to the following equation and calls the same procedure as the gaussian generator for building the edge set.
Since $\binom{n}{2}$ is the count of possibilities the grab two different nodes out of n and $|P|\binom{n/|P|}{2}$ corresponds to the count of possibilities the grab two nodes out of the same cluster, we get

$$\rho = \frac{\mathbb{E}(\overline{m}(\mathcal{C}))}{\mathbb{E}(m(\mathcal{C}))} = \frac{p_{out}(\binom{n}{2} - |P|\binom{n/|P|}{2})}{p_{in}|P|\binom{n/|P|}{2}} = \frac{p_{out}(n - n/|P|)}{p_{in}(n/|P| - 1)} \quad .$$

Algorithm 12 Significant Gaussian Generator

Input: approx. $\#n$, intra-cluster edge probability p_{in}, edge-ratio ρ

1: generateGaussianPartition P
2: compute p_{out}
3: createGaussianGraph(P, p_{in}, p_{out})

5.2 Attractor Generator Tests

5.2.1 Estimating Parameters

We started our tests with a set of parameters in which γ varied from 1 to 10, the search depth varied from 1 to 5 and the node count increased in steps of size 50 from 50 to 500. Every time we have been twice as bad as Newman regarding modularity we modified the parameter set by removing the current "bad" parameter. Table 5.1 gives an overview over the "best" parameters.

γ	Search Depth
2	1
3	1
4	1

Table 5.1: The "best" parameters.

A possible explanation for "bad" results with γ varying between 4 and 10 is that for increasing γ the algorithm converges very fast to a clustering with a low number of clusters. Thus, the resulting clustering has a coarse structure. As a consequence, the quality-index modularity has a very low score, but this does not indicate a mistake of our algorithm (see Appendix .1).

5.2.2 Setup

In the following we present the achieved results for the above parameters. In this test the density parameter ρ varied between 0.5 and 2.5 in steps of size 0.1. We then generated 30 graphs per density parameter using the attractor generator. For each graph our algorithm generated hierarchies. In the following Section 5.2.3, we present the qualities on the different hierarchy levels. In Section 5.2.4, we compared the best of the clusterings in the hierarchy regarding quality with the clustering of the greedy algorithm and clustering of the generator.
We mentioned earlier that our algorithm creates hierarchies. In the following hierarchy level i is the induced clustering after the i'th complete FAST-DENSE-REGION-DETECTION-GLOBAL step.

5.2.3 Hierarchies

In Figure 5.2, we present the results for Modularity, Coverage and Performance which we achieve on the first three hierarchy levels using the above described setup for the "best" parameters $\gamma = 2$ with search depth 1 and $\gamma = 4$ with search depth 1. The values for modularity and performance of our algorithm using $\gamma = 2$ are always better than the values we achieve by using $\gamma = 4$. The opposite is the case for coverage. We can further observe that both algorithms achieve nearly the same results on the first hierarchy level. Consequently the parameter $\gamma = 4$ seems to be to strong for continuing on the second hierarchy level. An advantage of a higher γ value is indeed the faster convergence, since more nodes are contracted on the first hierarchy level. Furthermore, with increasing hierarchy level modularity gets worse. Note that this does

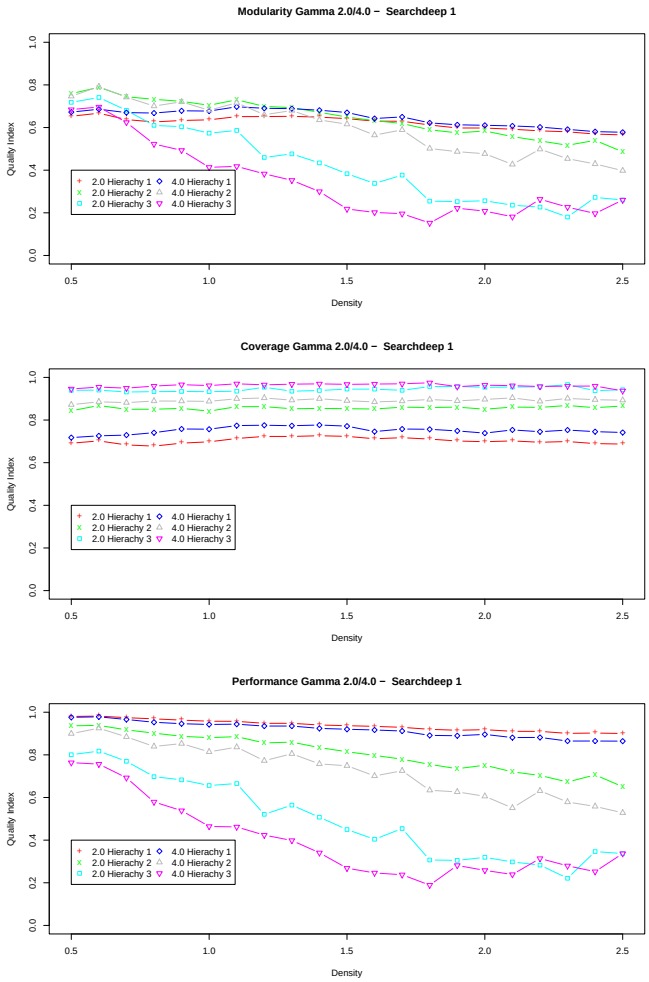


Figure 5.2: Hierarchies for Gamma 2 / 4 and Search depth 1

not indicate a mistake in our algorithm because we generate hierarchies (see appendix). For a fixed cluster count the clusterings calculated by our algorithm are still of good quality.

5.2.4 Comparison

In the Figure 5.3, we compare our achieved results for the given setup with the results obtained by the greedy approach and the results given by the generator. In this case, +N means that we have applied a Post-Newman-Step on our generated clustering and -N means the opposite. For every iteration and quality index, we take the highest value out of the first three hierarchy levels for our algorithm and the best value for the greedy approach. Since the number of clusters is very low in the third hierarchy level coverage tends to be very high. The high performance values arise from the fact, that there are many clusters in the first hierarchy level. Note, that we are on the same modularity level as the greedy approach if we apply the postnewman step. We can further see if we apply a postnewman step, the values for performance go down. This is mainly due to the fact that clusterings with good performance tend to have a high number of clusterings and by applying a postnewmanstep for increasing the modularity value, we reduce the number of clusters.

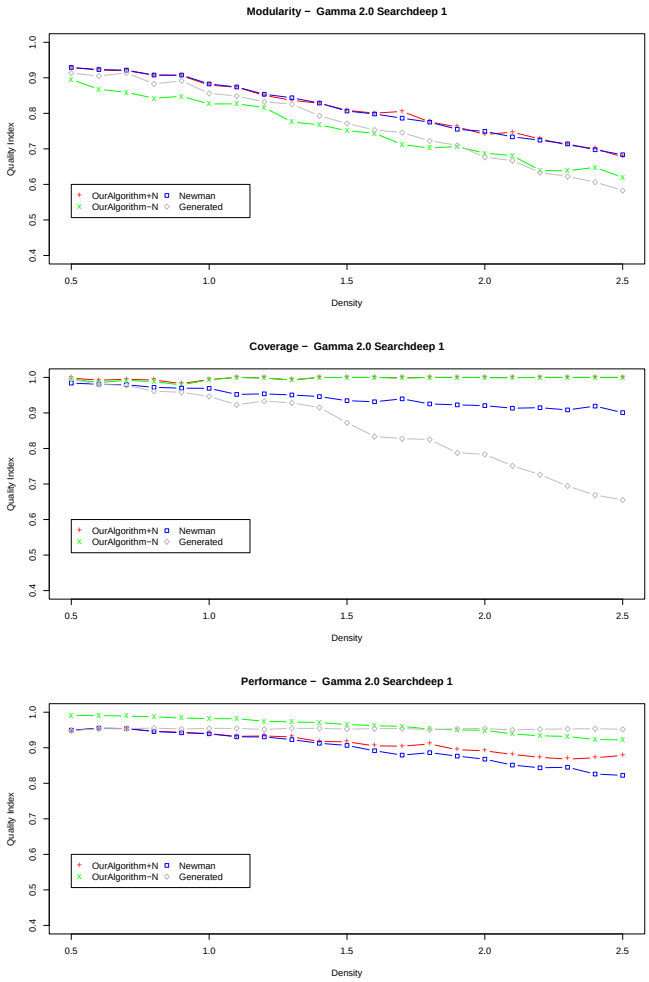


Figure 5.3: Comparison of Modularity, Coverage and Performance

5.3 Significant Gaussian Generator Tests

5.3.1 Estimating Parameters

We started our tests with a set of parameters in which γ varied from 2 to 10 in steps of size 2 and the search depth varied from 1 to 5. The generated graphs had 500 nodes, p_{in} varied between 0.1 and 0.7 in steps of size 0.1, ρ varied in steps of size 0.05 between 0.1 and 0.5. For each parameter set for the generator we generated 30 graphs. Note, $\rho = 0.5$ means that the are have as many inter-cluster edges as there are intra-cluster edges. After a few graphs we took the parameters with the *best average modularity* value to *continue our tests*. Again, increasing the search depth does not improve quality.

5.3.2 Results for Modularity

In Table 5.2, we present the "best" parameters considering the average value of modularity and in Figure 5.4 we present the corresponding contour plots for the in Section 5.3.1 given setup. The best parameters have a high value of γ. We conclude that we are able to get significant clusterings by using above parameters for our algorithm. The achieved quality equals the qual-

ity obtained by the greedy approach if our algorithm is used as a preprocessing step. Indeed, the quality gets worse for increasing ρ since the generated clustering is less significant.
If we use our algorithm without a Post-Newman-Step, we get an average quality of our clusterings which is 0.1 to 0.2 less than the average value generated by the greedy approach. But the values still indicate a significant clustering. Also it is not astonishing that we reach a quality which is less than the quality obtained by the greedy approach. The soul of the greedy approach is the optimization of modularity and our algorithm doesn't know modularity at all. We further conclude that we are able to compete with other algorithms.

γ	Average Modularity (-PNS)	Average Modularity (+PNS)	Level
4.0	0.4166	0.6385	2
6.0	0.4616	**0.6746**	1
8.0	0.5567	**0.6769**	1
10.0	0.5858	**0.6749**	1

Table 5.2: The "best" parameters. The average modularity value of the greedy approach has been 0.6680

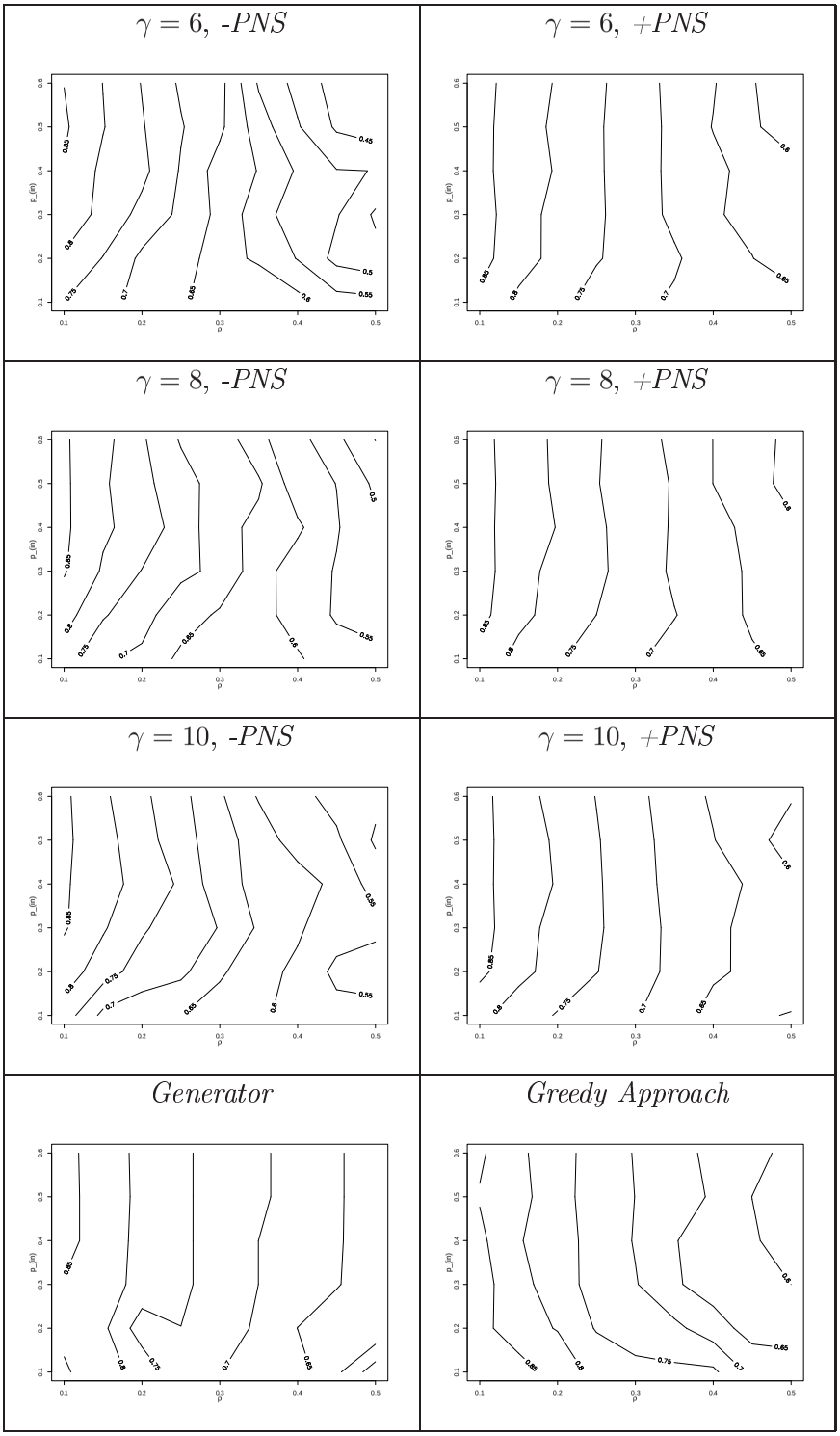


Figure 5.4: Gamma 6 to 10 (step size 2), Search depth 1

5.4 Hierarchies for Well Known Graphs

We now show some clustering results for graphs which are well known in the cluster community. First, we present a graph which is *clearly organized* in 16 small groups which themselves are organized in four groups. This graph was proposed by Santo Fortunato ([9]). The second example was compiled by Zachary while doing a Karate Club Study ([18]).

5.4.1 Hierarchy graph

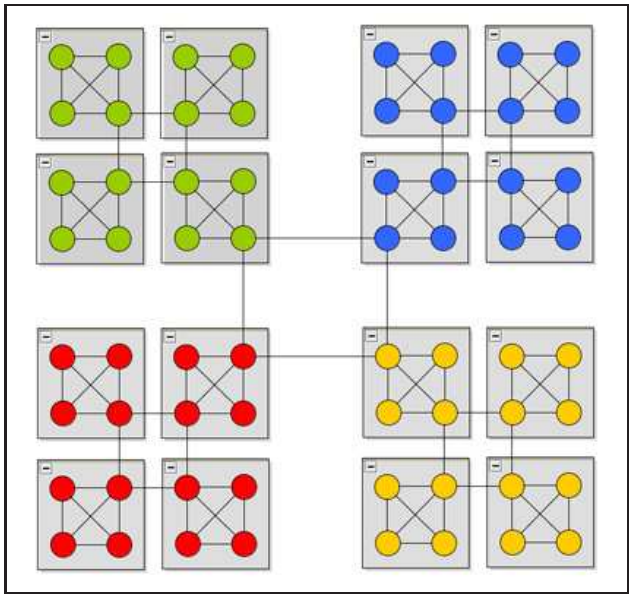


Figure 5.5: The hierarchy levels 1 and 3 produced by our algorithm with parameters $\gamma = 1.5, \mathbf{PAS} = \mathit{true}$, search depth 1. The first hierarchy level is indicated by the grouping, the third hierarchy level is indicated by the colors.

In the given graph the vertices have two levels of organization. The graph contains smaller groups, which are organized in bigger modules. An algorithm has to find all modules and their hierarchy to clearly characterize the roles of a vertice. When using the above parameters, we are able to find the 16 small groups on the first hierarchy level and then the 4 big groups on the third hierarchy level. That means we are able to identify the roles of each vertice in the graph.

5.4.2 Zachary's Karate Club

The Zachary Karate Club graph: the graph consists of 34 nodes representing people from a karate club. The friendship between the people is modeled by the edges. This graph was compiled by Zachary. During the course of Zachary's study of the karate club, a dispute came up between the members of the club which then split in two. The resulting fractions are indicated by the clustering in the following graph.

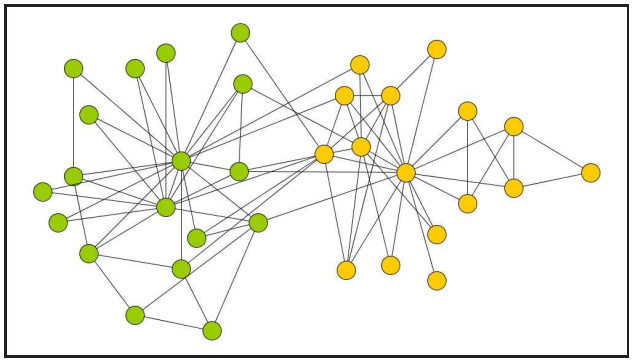


Figure 5.6: The resulting fractions of the Zachary Karate Club after a dispute.

We used our algorithm with parameters $\gamma = 2$ and search depth $= 1$ to cluster the karate club graph. The produced hierarchy levels 1 to 3 are shown beneath. The first clustering is

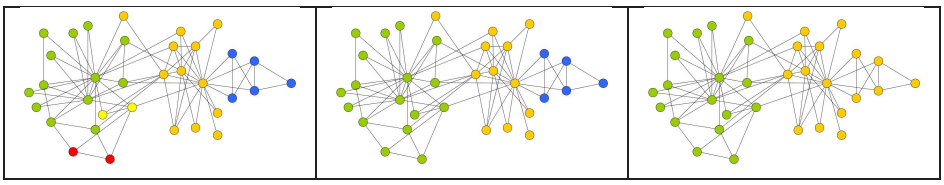


Figure 5.7: From left to right hierarchy levels 1 to 3 using $\gamma = 2$ and search depth $= 1$

granular and the clustering on the third hierarchy level is very coarse since it has only two clusters. Note that the clustering on the third hierarchy level is nearly the same as the real world clustering. Only one node differs from the original splitting and this node which has degree 2 is connected equipollent to both clusters. Roughly speaking, we would have been able to predict the resulting fractions of the Karate Club. Furthermore, we have to add that the modularity value for the original splitting is 0.3715 and 0.3718 for our clustering on the third hierarchy level and we achieve 0.399 on the second hierarchy level.

6 Final remarks

6.1 Conclusion

The main contribution of this thesis is a new algorithm for clustering graphs based on *local operations* namly local detection of dense regions and inserting shortcuts. We do not directly optimize a quality index. Most currently known algorithms work in a global way. The proposed algorithm can be used as alonestanding algorithm as well as a preprocessing step for the greedy approach to *reduce the searchspace*.
We tested our algorithm with real world graphs and two graph generators namly Significant Gaussian Generator and Attractor Generator. Through the experimental evalution we found out that increasing the search deep, which results in increasing runningtime, has not a positive effekt on the quality of the calculated clustering. We reach the best quality by using search deep 1. Roughly speaking, we have seen that the results regarding quality, using our algorithm as preprocessing step, do not differ. When using the algorithm without Newman's algorithm as a postprocessing step, we still find significant clusterings but with a lower modularity value. This however is not astonishing, since newmans algorithm optimizes modularity. We conclude that our algorithm can compete with other algorithms.

6.2 Future work

There are fields in which further investigation is possible. For example it could be possible to gain a speed up by applying another priority calculation to obtain a contraction order and to let γ vary dependent on the density of the graph G. However, it is not clear on the first sight how the new obtained contraction order affects the quality of the resulting clustering.
In this thesis, our main concern was to find out if it is possible to create a local algorithm for clustering graphs. For the future we plan to do an efficient implemenatation and test the algorithm for really large real world graphs as well as analyse the running time of our algorithm.

6.2.1 Calculating Priorities

It is possible to obtain a *contraction order* by another faster algorithm for calculating priorities. Instead of calculating the priority for every node through local searches, the following locally propagative approach could be feasible. We calculate the product of all degrees in the neighborhood of a node v and use this as an indicator for a dense region. The idea behind this approach is that if a region is dense, it is most likely that many degrees are high in this region.

$$\text{priority}(v) := \text{degree}(v) \cdot \prod_{w \in N(v)} \text{degree}(w)$$

Note that this can be implemented in $O(m+n)$ by traversing edges.

Algorithm 13 Priorities

Input: $G = (V, E, \omega)$
Output: mapping priority: $V \to \mathbb{N}$

1: **for all** $v \in V$ **do**
2: priority$(v) \leftarrow$ degree(v)
3: **end for**
4: **for all** $e = \{v, w\} \in E$ **do**
5: priority$(v) \leftarrow$ priority$(v) \cdot$ degree(w)
6: priority$(w) \leftarrow$ priority$(w) \cdot$ degree(v)
7: **end for**

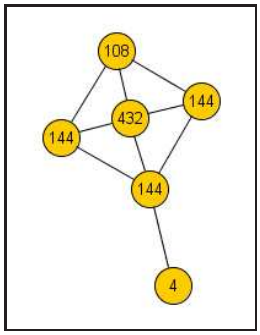


Figure 6.1: The calculated priorities using the alternative approach.

6.2.2 Density Parameter

We currently use a fixed density parameter γ. Roughly speaking, the parameter controls for a neighbor w of a starting node v, how many neighbors have to be equal to get into the dense region of v. It seems possible to improve the quality for a clustering if we do not use a fixed value for γ. Instead, we should use a mapping γ : Set of all graphs $\to \mathbb{R}_{>1}$ determining a good value for γ. This could be done *regarding the density* of a given graph.
A further possibility to deal with the density parameter is the *combination of different values for* γ in the priority search. That means, for a node $v \in V$ several local searches with a varying γ are started and only the region with the highest priority ψ is pushed on the priority queue.

Algorithm 14 FAST-DENSE-REGION-DETECTION-GLOBAL

Input: $G = (V, E, \omega)$, Set $\Gamma = \{\gamma_1, \cdots, \gamma_k \mid \gamma_i \in \mathbb{R}_{>1}\}$, search deep d

1: PriorityQueue pq $\leftarrow \emptyset$
2: **for all** $v \in V$ **do**
3: Denseregion $\leftarrow$ argmin$_{\gamma \in \Gamma} \psi$ (F-D-R-D-LOCAL(G, γ, d, v))
4: pq.insert(v, ψ(Denseregion))
5: **end for**
6: //the rest of the algorithm

Appendix

.1 A Note on Quality Indices for Hierarchies

In this paper we compared the quality of different hierarchy levels with the generated quality and the quality generated by the greedy approach. It seems difficult to compare clusterings which have a fixed number of clusters with all clusterings we get from the hierarchy, because the number of clusters decreases with increasing hierarchy level. As we can see for example in the hierarchy plots for γ 10, the quality gets worse as we increase the hierarchy level and it seems that the clustering is not good at all. But the comparison is perhaps too strong.

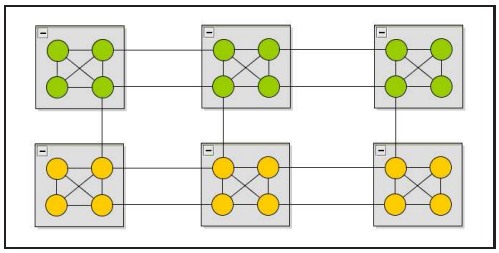


Figure .2: A graph with two clusterings. The clustering induced by node-colors is a min-cut (k = 2), has optimal modularity value under all clusterings with cluster count 2, but the quality is worse the clustering induced by the grouping.

As we see in Figure .2, it is possible that the clusterings found by our algorithm are very good when we hold the number k of cluster constant. Meanwhile we want to optimize modularity with the cluster count as an additional constraint. To get a "fair" comparison we should instead of simply comparing the value for modularity, compare the following value regarding the number of clusters

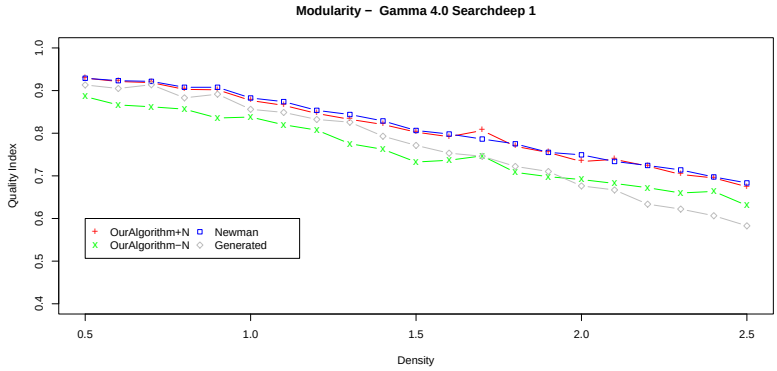
$$\text{sizemod}(k) := \max_{\mathcal{C} \in \mathcal{A}(G), |\mathcal{C}|=k} \text{mod}(\mathcal{C}) \ .$$

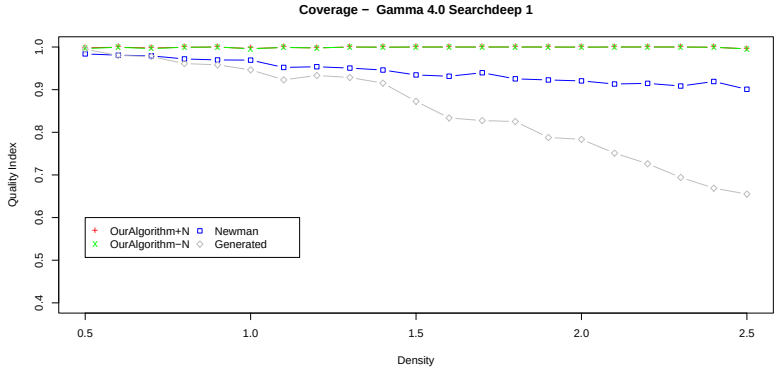
We get furthermore

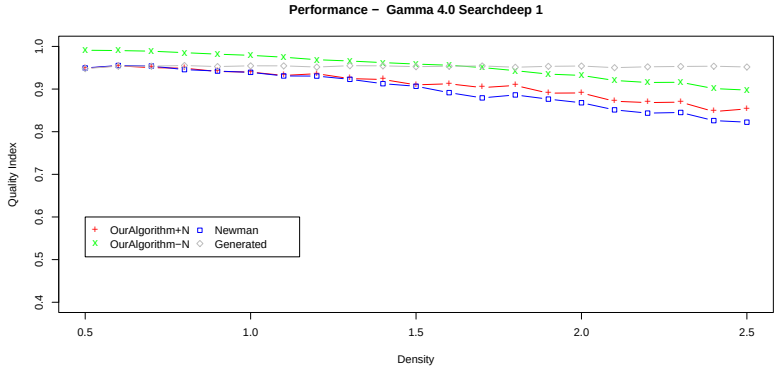
$$\max_{\mathcal{C} \in \mathcal{A}(G)} \text{mod}(\mathcal{C}) = \max_{k \in \{1,\cdots,|V|\}} \text{sizemod}(k) \ .$$

.2 More Experimental Results of Attractor Tests

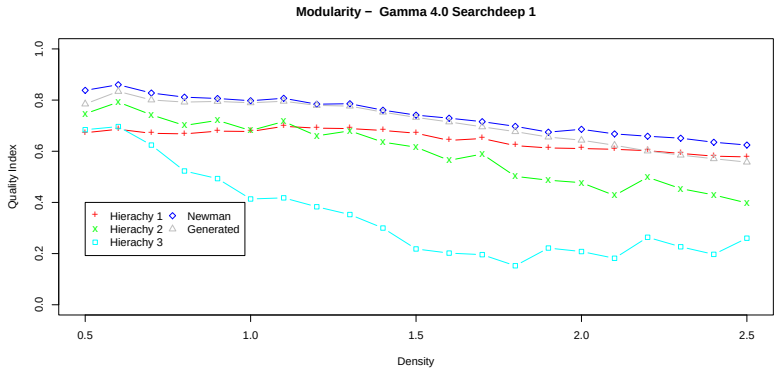
.2.1 Results for Gamma 4 / Search Depth 1

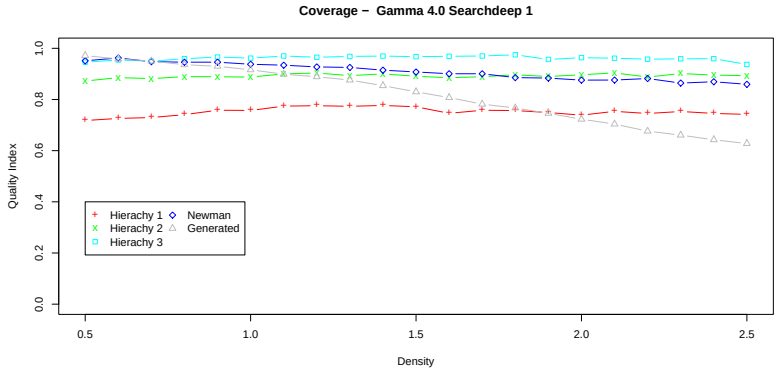


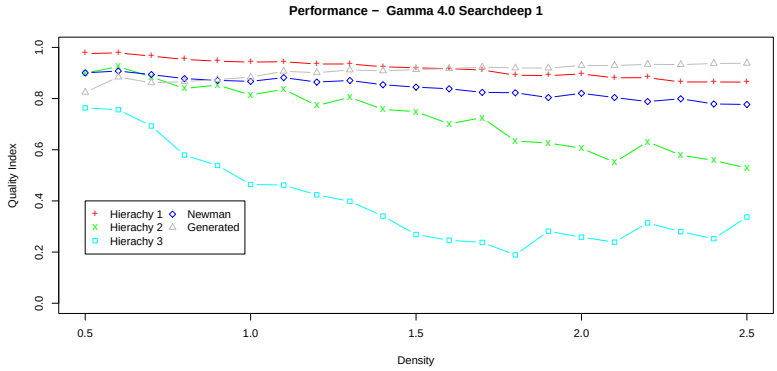




.2.2 Hierarchies for Gamma 4 / Search Depth 1







.3 More Experimental Results of Significant Gaussian Tests

.3.1 Hierarchies for Gamma 6 / Search Depth 1

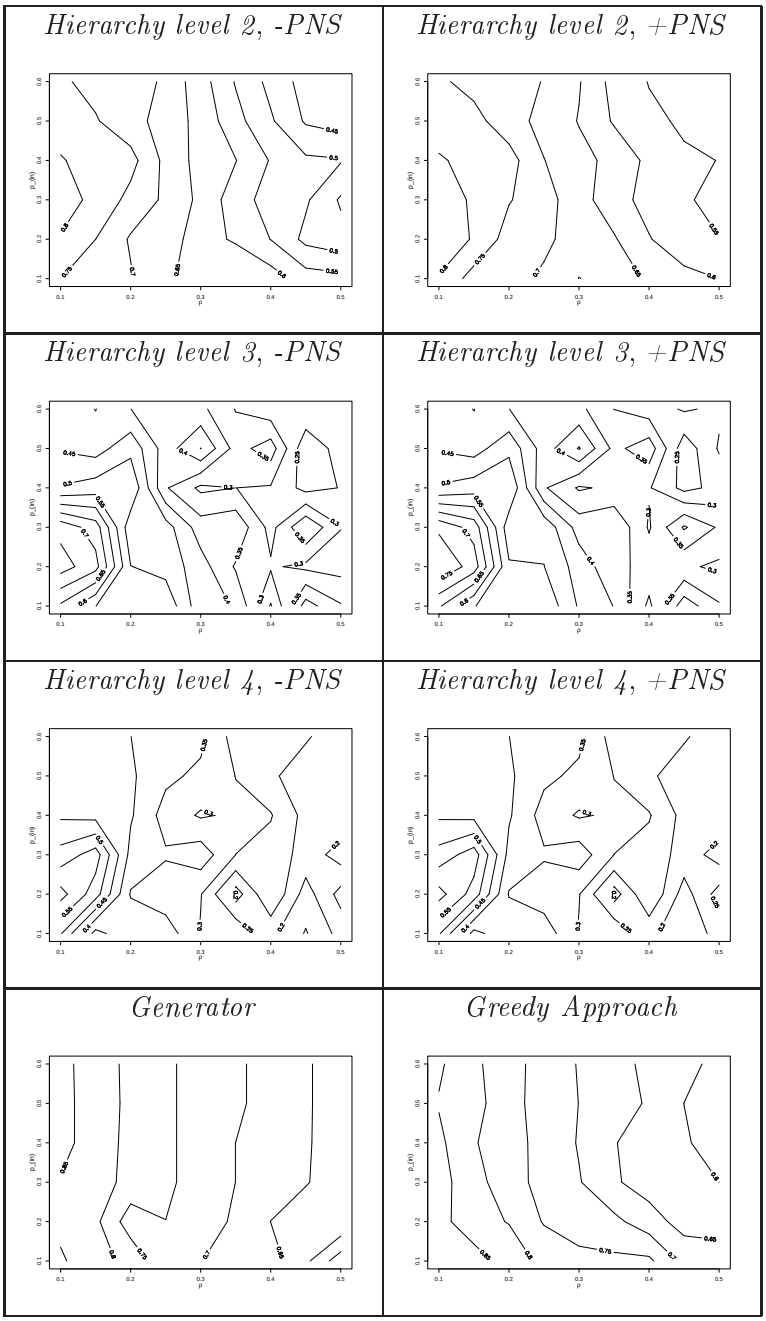


Figure .3: Hierarchies for $\gamma = 6$, Search Depth 1, Modularity

.3.2 Hierarchies for Gamma 8 / Search Depth 1

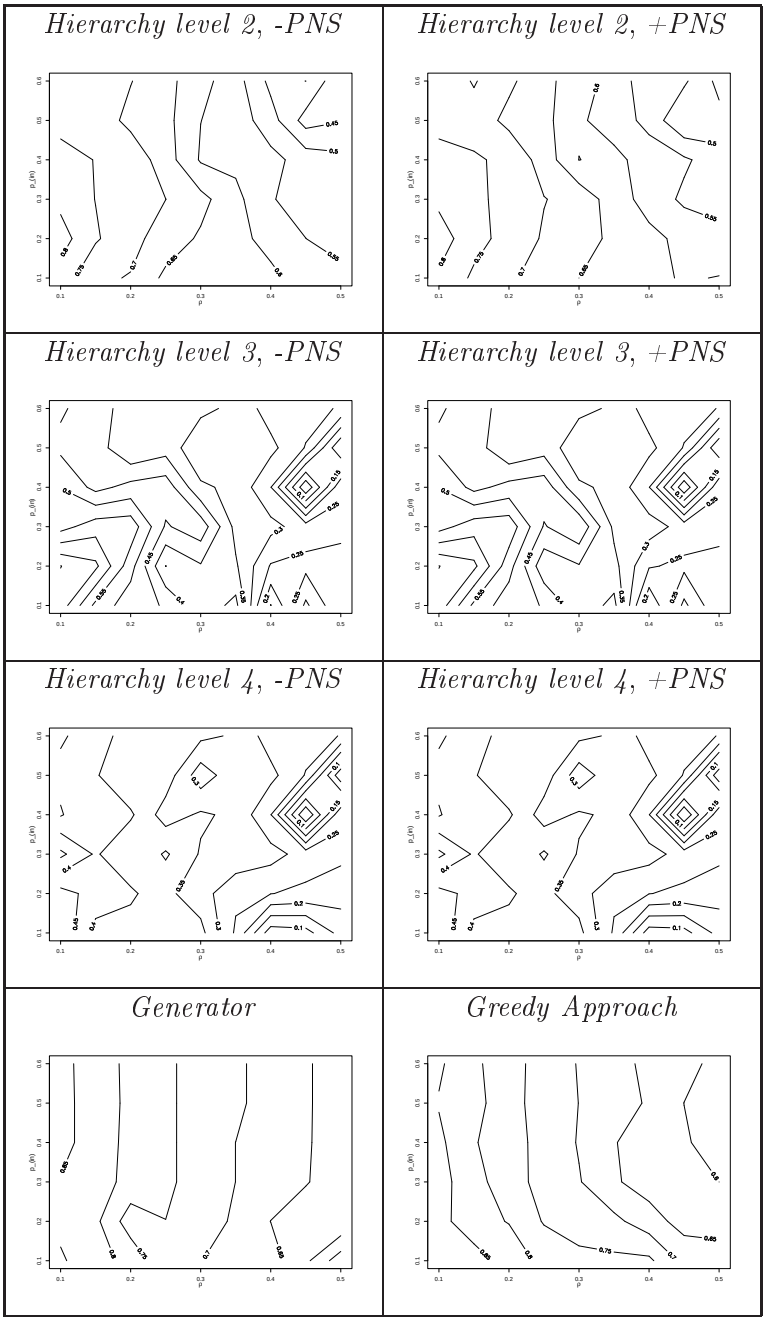


Figure .4: Hierarchies for $\gamma = 8$, Search Depth 1, Modularity

.3.3 Hierarchies for Gamma 10 / Search Depth 1

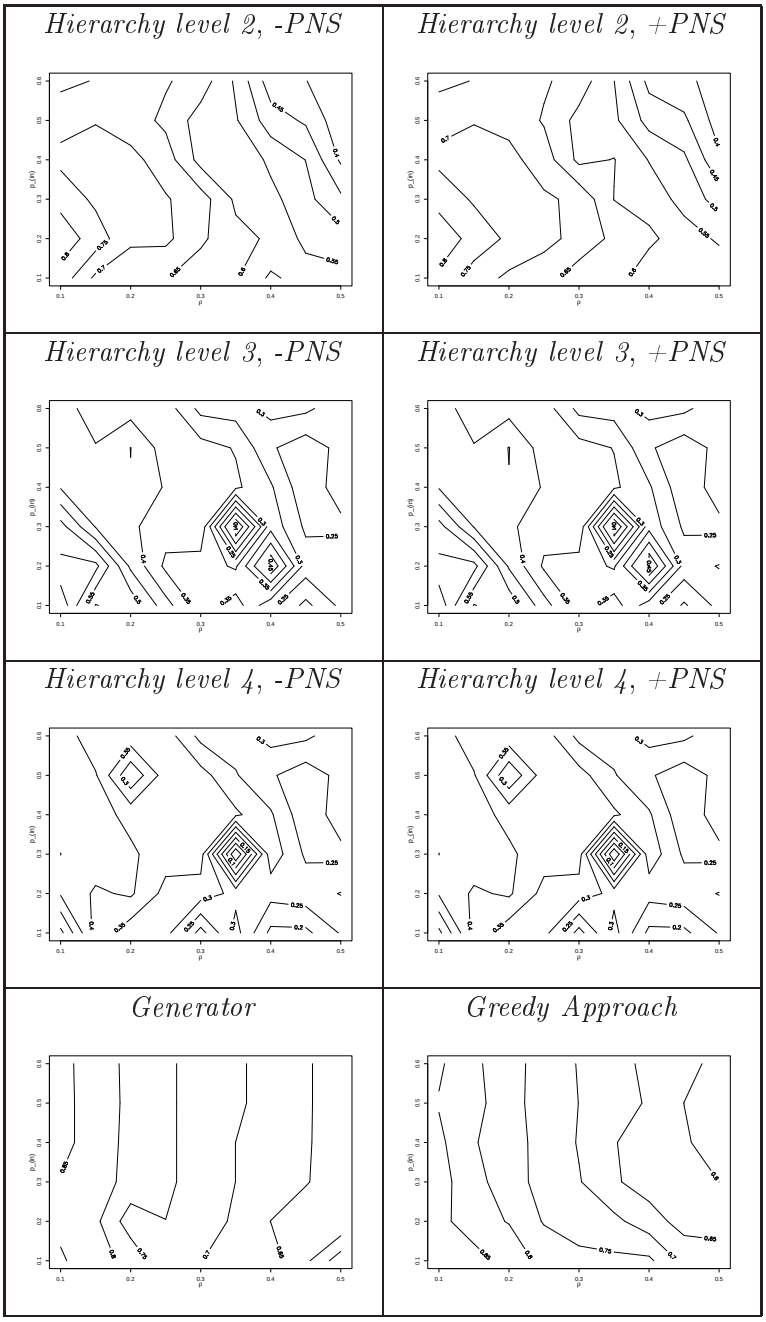


Figure .5: Hierarchies for $\gamma = 10$, Search Depth 1, Modularity

Bibliography

[1] Giorgio Ausiello, Pierluigi Crescenzi, Giorgio Gambosi, Viggo Kann, and Alberto Marchetti-Spaccamela. *Complexity and Approximation - Combinatorial Optimization Problems and Their Approximability Properties*. Springer, 2nd edition, 2002.

[2] Vladimir Batagelj and Matjaž Zaveršnik. Generalized Cores. Preprint 799, IMFM Ljublana, Ljubljana, 2002.

[3] Ulrik Brandes, Daniel Delling, Martin Höfer, Marco Gaertler, Robert Görke, Zoran Nikoloski, and Dorothea Wagner. On Finding Graph Clusterings with Maximum Modularity. In Andreas Brandstädt, Dieter Kratsch, and Haiko Müller, editors, *Proceedings of the 33rd International Workshop on Graph-Theoretic Concepts in Computer Science (WG'07)*, volume 4769 of *Lecture Notes in Computer Science*, pages 121–132. Springer, October 2007.

[4] Ulrik Brandes, Marco Gaertler, and Dorothea Wagner. Experiments on Graph Clustering Algorithms. In *Proceedings of the 11th Annual European Symposium on Algorithms (ESA'03)*, volume 2832 of *Lecture Notes in Computer Science*, pages 568–579. Springer, 2003.

[5] Fan R. K. Chung and S.-T. Yau. A Near Optimal Algorithm for Edge Separators. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing*, pages 1–8. ACM Press, 1994.

[6] Fan R. K. Chung and S.-T. Yau. Eigenvalues, Flows and Separators of Graphs. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pages 1–8. ACM Press, 1997.

[7] Aaron Clauset, Mark E. J. Newman, and Cristopher Moore. Finding community structure in very large networks. *Physical Review E*, 70(066111), 2004.

[8] Daniel Delling, Marco Gaertler, and Dorothea Wagner. Generating Significant Graph Clusterings. In *Proceedings of the European Conference of Complex Systems (ECCS'06)*, September 2006.

[9] Santo Fortunato and Claudio Castellano. Community Structure in Graphs. 2007.

[10] Marco Gaertler. Clustering with Spectral Methods. Diplomarbeit, Fachbereich Informatik und Informationswissenschaft, Universität Konstanz, March 2002.

[11] Marco Gaertler. Clustering. In Ulrik Brandes and Thomas Erlebach, editors, *Network Analysis: Methodological Foundations*, volume 3418 of *Lecture Notes in Computer Science*, pages 178–215. Springer, February 2005.

[12] Dieter Jungnickel. *Graphs, Networks and Algorithms*, volume 5 of *Algorithms and Computation in Mathmatics*. Springer, 1999.

[13] Henning Meyerhenke, Burkhard Monien, and Stefan Schamberger. Accelerating Shape Optimizing Load Balancing for Parallel FEM Simulations by Algebraic Multigrid. 2006.

[14] Henning Meyerhenke and Stefan Schamberger. A Parallel Shape Optimizing Load Balancer. 2006.

[15] Ron Shamir, Roded Sharan, and Dekel Tsur. Cluster Graph Modification Problems. In *Proceedings of the 28th International Workshop on Graph-Theoretic Concepts in Computer Science (WG'02)*, volume 2573 of *Lecture Notes in Computer Science*, pages 379–390. Springer, 2002.

[16] Stijn M. van Dongen. *Graph Clustering by Flow Simulation*. PhD thesis, University of Utrecht, 2000.

[17] Santosh Vempala, Ravi Kannan, and Adrian Vetta. On Clusterings - Good, Bad and Spectral. In *Proceedings of the 41st Annual IEEE Symposium on Foundations of Computer Science (FOCS'00)*, pages 367–378, 2000.

[18] Wayne W. Zachary. An Information Flow Model for Conflict and Fission in Small Groups. *Journal of Anthropological Research*, 33:452–473, 1977.

[19] C.T̃. Zahn. Graph-Theoretical Methods for Detecting and Describing Gestalt Clusters. *IEEE Transactions on Computers*, C-20:68–86, 1971.