Experimental Evaluation of Dynamic Graph Clustering Algorithms

Christian Staudt

Supervisors:
Dipl. Math. tech. Robert Görke and Prof. Dr. Dorothea Wagner

February 7, 2010

Abstract

Graph clustering is concerned with identifying the group structure of networks. Existing static methods, however, are not the best choice for evolving networks with evolving group structures. We discuss dynamic versions of existing clustering algorithms which maintain and modify a clustering over time rather than recompute it from scratch. We developed an extensible software framework for the evaluation of these algorithms, and present experimental results on real-world and synthetic graph instances. Our focus on clustering quality, clustering smoothness, and runtime. We conclude that dynamically maintaining a clustering on an evolving graph is superior in terms of all criteria. We demonstrate that dynamic algorithms are able to react quickly and appropriately to changes in the cluster structure. Our results allow us to give sound recommendations for the choice of an algorithm.

Contents

Chapter 1

Introduction

Graph clustering, the partition of complex networks into natural groups, is an active area of research. A variety of static clustering algorithms allows us to efficiently identify group structures. It is a current task to extend this knowledge in order to deal with networks that change and evolve over time. We can base this undertaking on the assumption that the effects of a single change of the graph on the overall group structure are necessarily local. Large, global changes may result as the sum of smaller changes, but they do not manifest suddenly. Ideally, dynamic clustering algorithms will react to small, local changes with small, local modifications of their previous work. Compared to static algorithms, there is obviously the opportunity for significant runtime reduction, but also the opportunity to cluster smoothly, i.e. to avoid abrupt changes to the clustering which might be undesirable. The question arises whether this approach entails a significant tradeoff between the runtime and the quality of the clustering. The aim of this work is to introduce several dynamic algorithms as well as a toolkit for their evaluation, and then present an assessment of the benefits and tradeoffs of the dynamic approach.

In the course of this work, we implemented a software framework which allows us to apply clustering algorithms to dynamic graph instances, measuring and comparing their performance in many respects. These instances include both real world graphs and synthetic graphs generated according to a probabilistic model.

Before we discuss the clustering algorithms, we will review some preliminary definitions. It is then necessary to specify what constitutes a 'natural' group, and how well a given clustering represents this natural structure of the graph, which leads to a number of *quality indices*. We also need to state formally what a 'small' change to a clustering is, leading to several *distance measures*. These are the main theoretical tools for clustering evaluation. We will then discuss our data sources, introduce our candidate algorithms in turn, and finally present and assess our results.

Insights gained from this work have been integrated into the paper *Modularity-Driven Clustering of Dynamic Graphs* [GMSW], to which we refer the reader for both a summary and further information on the topic.

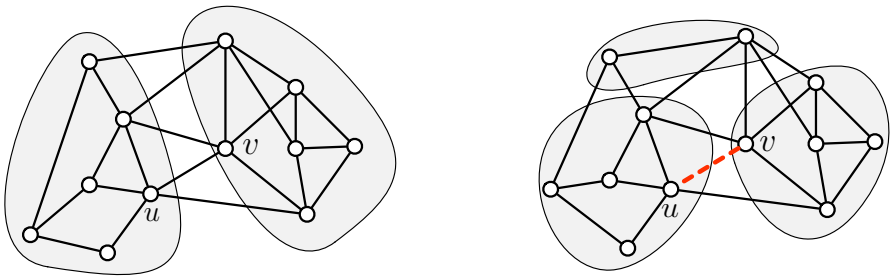


Figure 1.1: Clustering with optimal *modularity* (Paragraph 1.2) before and after the removal of the edge $\{u, v\}$

1.1 Preliminaries

Notation.

In this work we use IVERSON notation where it is convenient. This means that Formulas may contain bracketed logical statements which evaluate to one if the statement evaluates to true. Note that multiplication of brackets results in the conjunction of the logical statements.

Definition 1. *Let P be a logical statement. Then*

IVERSON NOTATION

$$[P] := \begin{cases} 1 & P = \textit{true} \\ 0 & P = \textit{false} \end{cases} \tag{1.1.1}$$

For convenience we use the a short notation for extending and reducing sets as well as replacing elements of the set.

Definition 2. *Given a set A and elements e and e', the following definitions hold:*

$$A + e := A \cup \{e\} \tag{1.1.2}$$
$$A - e := A \setminus \{e\} \tag{1.1.3}$$
$$\tag{1.1.4}$$

Dynamic Graphs

A graph represents a network of entities (as *nodes*) and connections (as *edges*). Throughout this work, the conventional notation for graphs is used.

Definition 3. *A graph is a tuple $G = (V, E)$ where V is the set of vertices. An edge in E connects two vertices. The graph is a directed graph if the edges are ordered pairs $E \subseteq V \times V$ or an undirected graph if the edges are unordered pairs $E \subseteq \binom{V}{2}$.* GRAPH

We use $n = |V|$ and $m = |E|$ as shorthand for the number of nodes and edges in a graph. All graphs considered in this work are undirected. All graphs used for the evaluation are simple graphs, excluding self-loops and multiple edges between a pair of nodes. If there is an edge $\{u, v\}$, it is called *incident* to u and v, and the nodes u and v are called *adjacent*. We will also occasionally refer to the set $\bar{E}$ of non-adjacent nodes which is induced by E.

$$\bar{E} := \{\{u, v\} \in \tbinom{V}{2} : \{u, v\} \notin E\} \tag{1.1.5}$$

Definition 4. *The degree of a node is the number of its incident edges.* DEGREE

$$deg(u) := |\{\{u, v\}, v \in V\}| \tag{1.1.6}$$

Definition 5. *Let ω be a weight function* WEIGHTED GRAPH

$$\omega : \begin{cases} E \to \mathbb{R} \\ \{u, v\} \mapsto x \end{cases} \tag{1.1.7}$$

Then $G = (V, E, \omega)$ is called a weighted graph.

Edges can carry a weight, which expresses the strength of the tie they represent; for readability, the weight of an edge $\{u, v\}$ as is written as $\omega(u, v)$ instead of $\omega(\{u, v\})$. We consider both weighted and unweighted graphs. Although we make the effort to distinguish weighted and unweighted case in following definitions, note that the unweighted case can also be seen as a special weighted case where $\forall\{u, v\} \in E : \omega(u, v) = 1$. If nothing else is said, the weight function is non-negative by definition.

By extension, we also define a weight function for a set of edges (1.1.8), as well as a single node, giving the total weight of its incident edges (1.1.9). This is a generalization of a node's degree. We use $\omega_{\max}(E)$ to denote the maximum weight of a set of edges.

$$\omega(E) := \sum_{\{u,v\} \in E} \omega(u, v) \tag{1.1.8}$$

$$\omega(u) := \sum_{\{u,v\} \in E} \omega(u,v) \tag{1.1.9}$$

In order to model the development of a network over time, a sequence of graphs can be regarded as a dynamic graph.

Definition 6. *A dynamic graph $\mathcal{G} = (G_0, \ldots, G_{t_{max}})$ is a sequence of graphs, with $G_t = (V_t, E_t)$ being the state of the dynamic graph at time step t.* DYNAMIC GRAPH

The difference between two states of the graph can be represented as a sequence of changes, which we call *graph events*. They are atomic in the sense that they cannot be subdivided into smaller changes. One could think of events on a larger scale, for instance because the removal of a node implies the removal of its incident edges. Nevertheless, our framework is agnostic in this respect. Such a larger-scale event would be presented to the algorithms as a sequence of unrelated atomic events in which all incident edges are removed in turn and finally the node itself.

Definition 7. *A graph event is one of the following atomic changes to the graph:* GRAPH EVENT

- *creation of a node:* $V_t \leftarrow V_{t-1} + u$
- *removal of an isolated node:* $V_t \leftarrow V_{t-1} - u$
- *creation of an edge:* $E_t \leftarrow E_{t-1} + \{u, v\}$
- *removal of an edge:* $E_t \leftarrow E_{t-1} - \{u, v\}$
- *weight increase:* $\omega_t(u,v) \leftarrow \omega_{t-1}(u,v) + x \quad x > 0$
- *weight decrease:* $\omega_t(u,v) \leftarrow \omega_{t-1}(u,v) - x \quad x > 0$

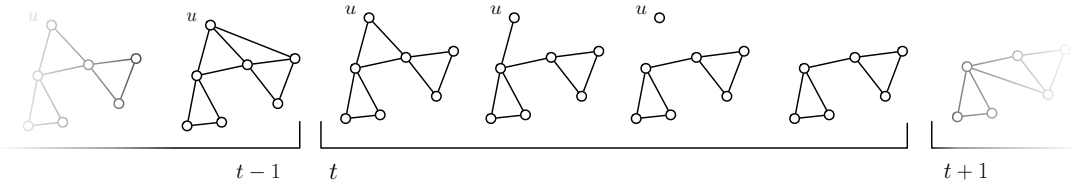


Figure 1.2: One time step — one node deletion — a sequence of graph events

Definition 8. *A path P from u to v is a sequence of edges* PATH

$$P(u,w) := (\{u, v_1\}, \{v_1, v_2\}, \ldots, \{v_{k-1}, w\})$$

Clustering

Clustering is the subdivision of a graph's node set into groups, which we formalize as follows:

Definition 9. *A clustering $\zeta(G)$ of a graph $G = (V, E)$ is a partition of V into disjoint, non-empty subsets $\{C_1, \ldots, C_k\}$. Each subset is a cluster $C_i \in \zeta$.* CLUSTERING

ζ is written instead of $\zeta(G)$ when there is no danger of ambiguity. We abbreviate the number of clusters in a clustering with $k = |\zeta|$. If a cluster contains only one node, it is called a *singleton*; accordingly, a *singleton clustering* consisting only of singletons. The other trivial clustering is the *1-clustering*, a cluster which contains all nodes. An alternative view is that of a clustering as a function which assigns nodes to clusters. It is convenient to denote the cluster to which u currently belongs by $\zeta(u)$.

$$\zeta : \begin{cases} V \to \zeta \\ v \mapsto C \end{cases} \tag{1.1.10}$$

With a given clustering, we can also distinguish two categories of edges, namely *intra-cluster edges* and *inter-cluster edges*. We will refer to the set of intra-cluster edges of a cluster C as defined in Equation 1.1.11 and the set of inter-cluster edges between clusters C_i and C_j as defined in Equation 1.1.12.

$$E(C) := \{\{u, v\} \in E : u \in C \wedge v \in C\} \tag{1.1.11}$$

$$E(C_i, C_j) := \{\{u, v\} \in E : u \in C_i \wedge v \in C_j\} \quad i \neq j \tag{1.1.12}$$

If we consider the clustering as a whole, the set of all intra-cluster edges (Equation 1.1.13) and the set of all inter-cluster edges (Equation 1.1.14) are also relevant.

INTER/INTRA-CLUSTER EDGES

$$E(\zeta) := \bigcup_{C \in \zeta} E(C) \tag{1.1.13}$$

$$E(\zeta)^c = \bigcup_{C_i \neq C_j} E(C_i, C_j) = E \setminus E(\zeta) \tag{1.1.14}$$

Additionally, we can divide the non-adjacent node pairs into *intra-cluster pairs* and *inter-cluster pairs*.

$$\bar{E}(\zeta) := \{\{u, v\} \in \bar{E} : \zeta(u) = \zeta(v)\} \tag{1.1.15}$$

$$\bar{E}(\zeta)^c = \{\{u, v\} \in \bar{E} : \zeta(u) \neq \zeta(v)\} = \bar{E} \setminus \bar{E}(\zeta) \tag{1.1.16}$$

Contracted Graphs

Some algorithms operate by contracting the original graph, combining several nodes and edges into one. This produces a weighted graph, possibly with self-loops.

Definition 10. *A graph $\hat{G} = (\hat{V}, \hat{E}, \hat{\omega})$ is a contraction of a graph $G = (V, E)$ if the nodes in $\hat{V}$ are pairwise disjoint subsets of V.* GRAPH CONTRACTION

$$\hat{V} = \{U_i : \bigcup U_i = V \wedge U_i \cap U_j = \emptyset, i \neq j\} \tag{1.1.17}$$

$$\hat{E} = \{\{U_i, U_j\} \in \binom{\hat{V}}{2} : \exists \{u \in U_i, v \in U_j\} \in E\} \tag{1.1.18}$$

$$\hat{\omega}(U_i, U_j) = \sum_{\{u,v\} \in E} [u \in U_i \wedge v \in U_j] \cdot \omega(u, v) \tag{1.1.19}$$

1.2 Quality Indices

Quality indices can be used to measure how well a given clustering fits the 'natural' group structure of the underlying graph. All indices discussed in this work are based on the paradigm of *intra-cluster density versus inter-cluster sparsity*. This means that a high quality clustering should identify densely connected groups of nodes which are only sparsely connected with other groups. Several indices following this paradigm will be reviewed in short. Note that the unweighted formulations are equal to the weighted ones if all edge weights are 1.

Coverage *Coverage* is a simple quality index which divides the number (or weight) of edges contained within clusters to the total number (or weight). Maximizing *coverage* means minimizing the number of inter-cluster edges. *Coverage* maps onto $[0, 1]$, with the *singleton clustering* and the *1-clustering* occupying the two extremes.

Definition 11. *For a graph $G = (V, E)$, an optional weight function ω and a clustering ζ of G, coverage is defined as* *coverage*

$$\mathcal{C}(G, \zeta) := \sum_{C \in \zeta} \frac{|E(C)|}{|E|} \tag{1.2.1}$$

$$\mathcal{C}_\omega(G, \zeta) := \sum_{C \in \zeta} \frac{\omega(E(C))}{\omega(E)} \tag{1.2.2}$$

Modularity. The fact that the *1-clustering* achieves optimal *coverage* but rarely constitutes a meaningful result is an obvious shortcoming of the *coverage* index. *Modularity* remedies this by looking at the statistical significance of the clustering. We obtain *modularity* by subtracting from *coverage* its expected value. This is, roughly speaking, the expected *coverage* the clustering would achieve if the graph had the same degree distribution but was randomly connected. Note that now the *1-clustering* is bound to have an index value of 0, because it achieves the same *coverage* for the actual edge structure of the graph as can be expected by chance. *Modularity* maps onto $[-1, 1]$.

Despite some drawbacks such as nonlocal effects possibly leading to counter-intuitive results, *modularity* agrees with human intuition in many instances. *Modularity* optimization has become one of the primary methods for graph clustering. Therefore, we focus on *modularity* in this work. The index can be computed in linear time, but the problem of finding a clustering with maximum *modularity*, ModOpt, is $\mathcal{NP}$-hard. The problem of optimally updating a given clustering following graph events, DynModOpt, is also $\mathcal{NP}$-hard: Since any graph can be constructed from a sequence of graph events, solving a linear number of DynModOpt instances would yield a solution for ModOpt.

Definition 12. *For a graph $G = (V, E)$, an optional weight function ω and a clustering ζ of G, modularity is defined as* MODULARITY

$$\begin{aligned}\mathcal{M}(G, \zeta) &:= \mathcal{C}(G, \zeta(G)) - \mathbb{E}[\mathcal{C}(G, \zeta(G)] \\ &= \sum_{C \in \zeta} \frac{|E(C)|}{|E|} - \sum_{C \in \zeta} \frac{\left(\sum_{v \in C} deg(v)\right)^2}{(2 \cdot |E|)^2}\end{aligned} \tag{1.2.3}$$

$$\begin{aligned}\mathcal{M}_\omega(G, \zeta) &:= \mathcal{C}(G, \zeta, \omega) - \mathbb{E}[\mathcal{C}(G, \zeta, \omega)] \\ &= \sum_{C \in \zeta} \frac{\omega(E(C))}{\omega(E)} - \sum_{C \in \zeta} \frac{\left(\sum_{v \in C} \omega(v)\right)^2}{(2 \cdot \omega(E))^2}\end{aligned} \tag{1.2.4}$$

Performance. According to this index, clustering quality means that two connected nodes should belong to the same cluster while two unconnected nodes should reside in different clusters. Each pair which is in this sense correctly classified contributes positively to *performance*. In the weighted version, the factor $\omega_{\max}(E)$ stems from the need to assign weights to non-existent edges, although other normalizations are possible (compare [Gae05]).

Definition 13. *For a graph $G = (V, E)$ and a clustering $\zeta(G)$, performance is defined as* PERFORMANCE

$$\mathcal{P}(G,\zeta) := \frac{|E(\zeta)| + |\bar{E}(\zeta)^c|}{\frac{1}{2}\cdot n\cdot(n-1)} \tag{1.2.5}$$

$$\mathcal{P}_\omega(G,\zeta) := \frac{\omega(E(\zeta) + \omega_{\max}(E)\cdot|\bar{E}(\zeta)^c| - \big(\omega_{\max}(E)\cdot|\bar{E}(\zeta)| - \omega(\bar{E}(\zeta))\big)}{\frac{1}{2}\cdot n\cdot(n-1)\cdot\omega_{\max}(E)} \tag{1.2.6}$$

Significance-Performance. This index relates to *performance* in the same way *modularity* relates to *coverage*. *Significance-performance* and *modularity* are in fact instances of a general class of *significance* indices. We refer to [GHWG09] for details on the significance paradigm. It has been shown that *modularity* and *significance-performance* are equivalent in the sense that

$$\mathcal{SP}(G,\zeta_1) > \mathcal{SP}(G,\zeta_2) \iff \mathcal{M}(G,\zeta_1) > \mathcal{M}(G,\zeta_2) \tag{1.2.7}$$

Definition 14. *For a graph $G = (V,E)$ and a clustering $\zeta(G)$, significance-performance is defined as*

SIGNIFICANCE-PERFORMANCE

$$\mathcal{SP}(G,\zeta) := \mathcal{P}(G,\zeta) - \mathbb{E}[\mathcal{P}(G,\zeta)] \tag{1.2.8}$$

$$= \frac{|E(\zeta)| + |\bar{E}(\zeta)^c|}{\frac{1}{2}\cdot n\cdot(n-1)} - \frac{\sum_{C\in\zeta}\left((\sum_{v\in C} deg(v))^2\frac{1}{m} - |C|^2\right) + n^2 - 2m}{n(n-1)}$$

$$\mathcal{SP}_\omega(G,\zeta) := \mathcal{P}_\omega(G,\zeta) - \mathbb{E}[\mathcal{P}_\omega(G,\zeta)] \tag{1.2.9}$$

$$= \frac{|E(\zeta)| + |\bar{E}(\zeta)^c|}{\frac{1}{2}\cdot n\cdot(n-1)} - \frac{\sum_{C\in\zeta}\left((\sum_{v\in C} deg(v))^2\frac{1}{m} - |C|^2\right) + n^2 - 2m}{n(n-1)}$$

Inter-Cluster-Conductance. The *inter-cluster-conductance* index is based on the relationship between clusterings and cuts. Technically, a cut corresponds to a clustering with two clusters. The *conductance* of a cut is low if its weight is small compared to the density of the induced subgraphs. Clearly, a high-quality clustering implies low-weight cuts between clusters and high-weight cuts within clusters. *Inter-cluster conductance* measures clustering quality this way and maps it onto the interval $[0,1]$.

Definition 15. *For a graph $G = (V, E)$, a cut $\theta = (U, V \setminus U)$ is a partition of the node set into two subsets. For a weighted graph $G = (V, E, \omega)$, the weight $\omega(\theta)$ of the cut is defined as* CUT WEIGHT

$$\omega(\theta) := \sum_{\{u,v\} \in E(U, V \setminus U)} \omega(\{u, v\}) \tag{1.2.10}$$

The conductance-weight of one side of the the cut is

$$a(U) := \sum_{\{u,v\} \in E(U,V)} \omega(\{u, v\}) \tag{1.2.11}$$

Definition 16. *For a graph $G = (V, E, \omega)$ and a clustering ζ, the conductance for the cut $\theta = (C, V \setminus C)$ is defined as* CONDUCTANCE

$$\phi(C) := \begin{cases} 1 & if\ C \in \{\emptyset, V\} \\ 0 & if\ C \notin \{\emptyset, V\} \wedge \omega(\overline{E(\zeta)}) = 0 \\ \frac{\omega(\overline{E(\zeta)})}{\min(a(C), a(V \setminus C))} & else \end{cases} \tag{1.2.12}$$

Definition 17. *For a graph $G = (V, E)$ and a clustering $\zeta(G)$, inter-cluster-conductance is defined as* ICC

$$ICC(\zeta) := 1 - \sum_{C_i \in \zeta} \frac{\phi(C_i)}{|\zeta|} \tag{1.2.13}$$

1.3 Distance Measures

Several measures have been introduced to formalize the notion of (dis)similarity between two clusterings of a graph. They can be broadly categorized into node-structural measures, which depend only on the partition of the node set, and graph-structural measures, which take into account the edge structure of the graph. These categories can be further divided into measures relying on pair-counting, cluster overlap or entropy. Each of the measures considered here maps two clusterings into the interval $[0, 1]$ with 0 indicating equality and 1 indicating maximum dissimilarity, which is of course a matter of definition. The following and additional distance measures are discussed in depth in [Del06] and [DGGW07].

Node-Structural Measures

Basics on Pair Counting. Several node-structural measures count pairs of nodes noting whether they are clustered together or not. The sets defined in

Equation (1.3.1) are frequently used. Definitions of some node-structural pair counting measures follow.

$$
\begin{aligned}
S_{11} &:= \{\{u,v\} \in \tbinom{V}{2} : \zeta(u) = \zeta(v) \wedge \zeta'(u) = \zeta'(v)\} \\
S_{00} &:= \{\{u,v\} \in \tbinom{V}{2} : \zeta(u) \neq \zeta(v) \wedge \zeta'(u) \neq \zeta'(v)\} \\
S_{10} &:= \{\{u,v\} \in \tbinom{V}{2} : \zeta(u) = \zeta(v) \wedge \zeta'(u) \neq \zeta'(v)\} \\
S_{01} &:= \{\{u,v\} \in \tbinom{V}{2} : \zeta(u) \neq \zeta(v) \wedge \zeta'(u) = \zeta'(v)\}
\end{aligned}
\tag{1.3.1}
$$

Rand. Introduced by Rand (1971), this measure counts S_{11} and S_{00} and normalizes their sum by the total number of node pairs. It is obvious that $\mathcal{R}$ has the minimum value of 0 if ζ and ζ' are identical.

Definition 18. *For a pair of clusterings $\{\zeta, \zeta'\}$ of a graph G, the Rand measure is defined as* rand

$$
\mathcal{R}(\zeta, \zeta') := 1 - \frac{2 \cdot (|S_{11}| + |S_{00}|)}{n \cdot (n-1)} \tag{1.3.2}
$$

Jaccard. The *Jaccard* measure $\mathcal{J}$ relies on the same pair counting sets as $\mathcal{R}$. The singleton clustering and any other clustering compare with a distance of 1.

Definition 19. *For a pair of clusterings $\{\zeta, \zeta'\}$ of a graph G, the Jaccard measure is defined as* jaccard

$$
\mathcal{J}(\zeta, \zeta') := \begin{cases} 1 - \frac{2 \cdot |S_{11}|}{n \cdot (n-1) - 2 \cdot |S_{00}|} & \text{if } \; n \cdot (n-1) - 2 \cdot |S_{00}| > 0 \\ 0 & \text{else} \end{cases} \tag{1.3.3}
$$

Fowlkes-Mallows. Likewise, the *Fowlkes-Mallows* measure $\mathcal{FM}$ has the property that the singleton clustering achieves the maximum distance of 1 to any other clustering.

Definition 20. *For a pair of $\{\zeta, \zeta'\}$ of a graph G, the Fowlkes-Mallows measure is defined as* Fowlkes-Mallows

$$
\mathcal{FM}(\zeta, \zeta') := \begin{cases} 1 - \frac{|S_{11}|}{\sqrt{(|S_{11}|+|S_{10}|) \cdot (|S_{11}|+|S_{01}|)}} & \text{if } \; |S_{01}|, |S_{10}| > 0 \vee |S_{11}| > 0 \\ 1 & \text{if } |S_{11}|, |S_{10}| = 0 \vee |S_{11}|, |S_{01}| = 0 \\ 0 & \text{else.} \end{cases} \tag{1.3.4}
$$

Basics on Entropy. Some measures are based on *node entropy*, which we explain here briefly: The probability that a cluster C contains a node v chosen at random is

$$P(C) := P[v \in C] = \frac{|C|}{n} \tag{1.3.5}$$

Using this probability, we can assign an entropy value to a clustering. The entropy can be interpreted as the uncertainty measured in bits for the result of $\zeta(v)$ if we select v at random:

Definition 21. *The node entropy of a clustering ζ is defined as*

NODE ENTROPY

$$\begin{aligned}\mathcal{H}(\zeta) &:= -\sum_{C\in\zeta} P(C)\cdot\log_2(P(C)) \\ &= -\sum_{C\in\zeta} \frac{|C|}{n}\cdot\log_2\left(\frac{|C|}{n}\right)\end{aligned} \tag{1.3.6}$$

The value of $\mathcal{H}(\zeta)$ ranges from 0 for the 1-clustering to $\log_2(n)$ for the singleton clustering. Closely related is *mutual node information* which measures the change in uncertainty for one clustering if the other clustering is known. *Mutual node information* is bounded by $0 \leq \mathcal{I}(\zeta,\zeta') \leq \min\{\mathcal{H}(\zeta),\mathcal{H}(\zeta')\}$.

Definition 22. *Let $\{\zeta,\zeta'\}$ be a pair of clusterings defined on the same node set V. Then their mutual node information is*

NODE CORRELATION INFORMATION

$$\mathcal{I}(\zeta,\zeta') := \sum_{C\in\zeta}\sum_{C'\in\zeta'} \frac{|C\cap C'|}{n}\cdot\log_2\left(\frac{|C\cap C'|\cdot n}{|C|\cdot|C'|}\right) \tag{1.3.7}$$

Fred-Jain. This entropy based measure incorporates both *node entropy* and *node correlation information*.

Definition 23.

FRED-JAIN

$$\mathcal{FJ}(\zeta,\zeta') := \begin{cases} 1 - \frac{2\cdot\mathcal{I}(\zeta,\zeta')}{\mathcal{H}(\zeta)+\mathcal{H}(\zeta')} & \mathcal{H}(\zeta)+\mathcal{H}(\zeta') \neq 0 \\ 0 \end{cases} \tag{1.3.8}$$

Maximum-Match. The basis of this measure is a *confusion matrix* $M(\zeta,\zeta') \in \mathbb{N}^{|\zeta|\times|\zeta'|}$ where each entry holds the number of nodes in the intersection of two clusters from the different clusterings.

Definition 24. *For a pair of clusterings $\{\zeta,\zeta'\}$ of a graph G, the maximum match measure is defined as*

MAXIMUM MATCH

$$m_{ij} := |C_i \in \zeta \cap C_j \in \zeta'| \tag{1.3.9}$$

$$\mathcal{MM}(\zeta, \zeta') := 1 - \frac{1}{n} \cdot \mathit{MaxMatch}(\zeta, \zeta') \tag{1.3.10}$$

Algorithm 1: MaxMatch

Input: $M(\zeta, \zeta') \in \mathbb{N}^{|\zeta| \times |\zeta'|}$: confusion matrix

$\Sigma \leftarrow 0$
while $|M| > 0$ **do**
  $m_{ab} \leftarrow \max m_{ij} \in M$
  $\Sigma \leftarrow \Sigma + m_{ab}$
  $M \leftarrow [[m_{ij}]]_{i \neq a, j \neq b}$
return Σ

Graph-Structural Measures

All of the measures described so far ignore the edges of the underlying graph. However, they can be easily modified to take the edges into account. We can define pair counting sets which are analogous to (1.3.1) with the difference that only pairs connected with an edge are considered. For each of the measures based on pair counting ($\mathcal{R}$, $\mathcal{J}$ and $\mathcal{FM}$) , substituting these sets for their counterparts yields their graph-structural variants ($\mathcal{R}_g$, $\mathcal{J}_g$ and $\mathcal{FM}_g$)

$$\begin{aligned} E_{11} &:= \{\{u, v\} \in E : \zeta(u) = \zeta(v) \wedge \zeta'(u) = \zeta'(v)\} \\ E_{00} &:= \{\{u, v\} \in E : \zeta(u) \neq \zeta(v) \wedge \zeta'(u) \neq \zeta'(v)\} \\ E_{10} &:= \{\{u, v\} \in E : \zeta(u) = \zeta(v) \wedge \zeta'(u) \neq \zeta'(v)\} \\ E_{01} &:= \{\{u, v\} \in E : \zeta(u) \neq \zeta(v) \wedge \zeta'(u) = \zeta'(v)\} \end{aligned} \tag{1.3.11}$$

Likewise, there are graph-structural versions of entropy-based and overlap-based measures.

Distance Measures on Dynamic Graphs

The introduced measures generally assume that the clusterings to compare are defined on the same graph. It is not completely obvious how the dissimilarity of two consecutive clusterings on a dynamic graph should be defined. We chose to consider the intersection of the two graph states - a node or edge which is not present in both of them is ignored.

Chapter 2

Data Sources

2.1 Generator for Dynamic Clustered Random Graphs

In the run-up to this work we implemented a versatile generator for dynamic clustered graphs. We keep the following description short and refer the reader to the technical report [GS09] for a thorough documentation. The aim of our implementation was to include features not present in existing generators. It allows us to generate graphs which are

- *dynamic*, i.e. representing the change of a network in the course of discrete time
- *clustered*, i.e. exhibiting a clustered structure based on intra-cluster densitiy versus inter-cluster sparsity of edges
- *random*, i.e. generated according to a probabilistic model

The generator creates edges with certain probabilities - called p_{in} and p_{out} - depending on whether they reside in the same cluster or not. The resulting graph has a group structure with a significance that can be controlled by parameters. We use the term *ground-truth clustering* for the clustering that is used for the assignment of edge probabilities. In order to simulate clustering evolution, this *ground-truth clustering* can change over time. Its clusters can merge or split, controlled by probabilistic parameters. Edge probabilities are updated accordingly. Since the actual edge distribution is gradually adjusted, the de-facto clustering (the clustering observable by clustering algorithms) lags behind the *ground-truth*. However, a good clustering algorithm should quickly recognize the new *ground-truth* as soon as the merger or split becomes apparent.

The actions of the generator during one iteration through the main loop can involve several atomic changes (as described in Definition 7). More precisely, the generator can produce one of the following events affecting elements of the graph per iteration:

- an edge $\{u, v\}$ is created with probability $p(u, v)$
- an edge $\{u, v\}$ is removed with probability $1 - p(u, v)$
- a node u is created, then edges $\forall v \in V - u : \{u, v\}$ are created with probability $p(u, v)$
- a node u is removed following the removal of all incident edges $\{u, v\}$

When a threshold number η of edge events is exceeded, a time step in the dynamic graph is triggered (recall Definition 6). This parameter controls the amount of change from one time step to the next and can be used to balance the impact of node events.

In addition to the clustering $\zeta_{\text{gen}}(G_t)$ determining the edge probabilities, the generator maintains a separate clustering $\zeta_{\text{ref}}(G_t)$ which serves as a reference to which we can compare the results of our candidate algorithms. $\zeta_{\text{gen}}(G_t)$ is not suitable for this purpose, because the edge density in the early stages of an ongoing cluster event does not yet reflect the aspired clustering. The generator therefore inspects the ongoing cluster events and judges whether they are complete. If an event is considered complete, it is incorporated into $\zeta_{\text{ref}}(G_t)$.

2.2 E-Mail Graph

The *e-mail graph* ($\mathcal{G}_e$) is the real world data set used in this evaluation. It was obtained by logging the e-mail communication between members of the faculty of computer science at Universität Karlsruhe (KIT). A node represents an anonymized e-mail address, generally corresponding to a person. An edge represents a communication link with its weight being the number of e-mails sent. The graph has already served as real world data for clustering experiments in [DGGW08]. Each e-mail contributes to the weight only for a limited amount of time so that edge weights can also decrease again. Similar to the generated graph, the development of the graph is subdivided into time steps. However, a single time step involves the creation of at most two nodes (sender and recipient, if not already there) or the creation or weight update of exactly one edge. Edge weights can be incremented by one due to an email sent or decremented by one due to a timeout. A node is immediately excluded from the graph if it becomes isolated in the process.

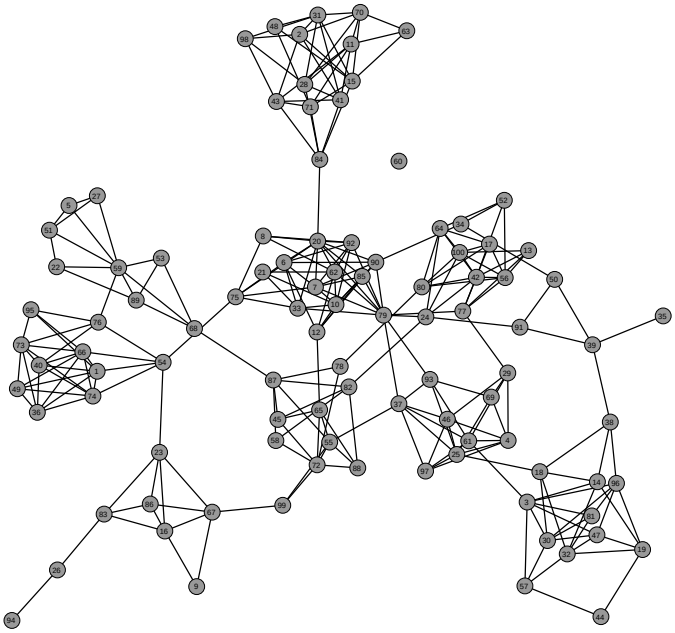


Figure 2.1: Rendering of a generated graph instance with $n = 100$, $k = 10$, $p_{in} = 0.5$ and $p_{out} = 0.005$

Chapter 3

Evaluation Framework

3.1 Framework Components

Evaluation framework and algorithms are written in the `Java` programming language. Some pre-existing and newly implemented software components are described in this section.

Static Graphs A basic component of our framework is the `yFiles` framework, which provides a versatile graph data structure. The documentation can be obtained at http://www.yworks.com/products/yfiles/doc/.

Dynamic Graphs We extended the `yFiles` graph with a class which incrementally reads from a data source, and modifies the graph. The data source is a file which stores either a generated graph or a real world data set such as the email graph. To the outside world this class presents itself as a dynamic graph over whose states can be iterated in ascending order, yielding a static graph at each timestep.

Clustering Framework Thanks to the work of our colleagues Bastian Katz and Daniel Delling we could build on an existing graph clustering framework. At the core of the this framework is a data structure `Clustering` which maintains a mapping between the nodes of a graph and the clusters of possibly multiple clusterings. It is implemented as a *listener* (or *observer* as described in [GHJ+]) to a `yFiles` graph, which is automatically notified of graph events and will react to them. By default, newly added nodes are not yet assigned to a cluster (e.g. by creating their own singleton cluster) but kept in a "pool" of unclustered nodes. The framework provides a `ClusteringListener`. Instances of classes inheriting from `ClusteringListener` will be notified of cluster events as well as forwarded graph

events. We based our dynamic clustering algorithms on this class as described in the following subsection. The framework also includes implementations for all of the quality indices and distance measures described.

3.2 Dynamic Clustering Algorithms

The abstract base class `DynamicClusteringAlgorithm` extends the class `ClusteringListener`. All algorithm implementations can therefore register as a listener to their respective clustering, and indirectly receive graph events and clustering events. Additionally, they receive a *time step event* after each time step. Remeber that a time step may comprise more than one graph event. It is up to the concrete algorithm implementation whether to perform the main work directly after each graph event or delay it until the time step event.

Figure 3.1 is a class diagram in UML syntax detailing the type hierarchy of `DynamicClusteringAlgorithm` as well as its relations to other classes of the framework. Arrows represent inheritance, dashed arrows represent interface extension, lines represent associations with certain cardinalities.

3.3 Experimental Setup and Visualization

A graphical frontend and plotting environment written for `Mathematica` complements our suite. It allows us to compose an experimental setup quickly and conveniently and plot its results. After algorithms, measures and a graph instance have been selected, an evaluation run can be started. After each run, result data is available for plotting. The plots show both attributes of the graph and its clustering evolution or data concerning the performance of the algorithms. The following measures are plotted by default. Time is charted on the x-axis. Examples of these plots follow in Chapter 5 .

- number of nodes and edges in G_t
- number of ongoing cluster split events and cluster merge events (only for generated graphs)
- runtime in milliseconds per time step for all selected algorithms
- number of clusters for all selected algorithms
- quality values for all selected quality indices and algorithms
- distance values for all selected distance measures and algorithms

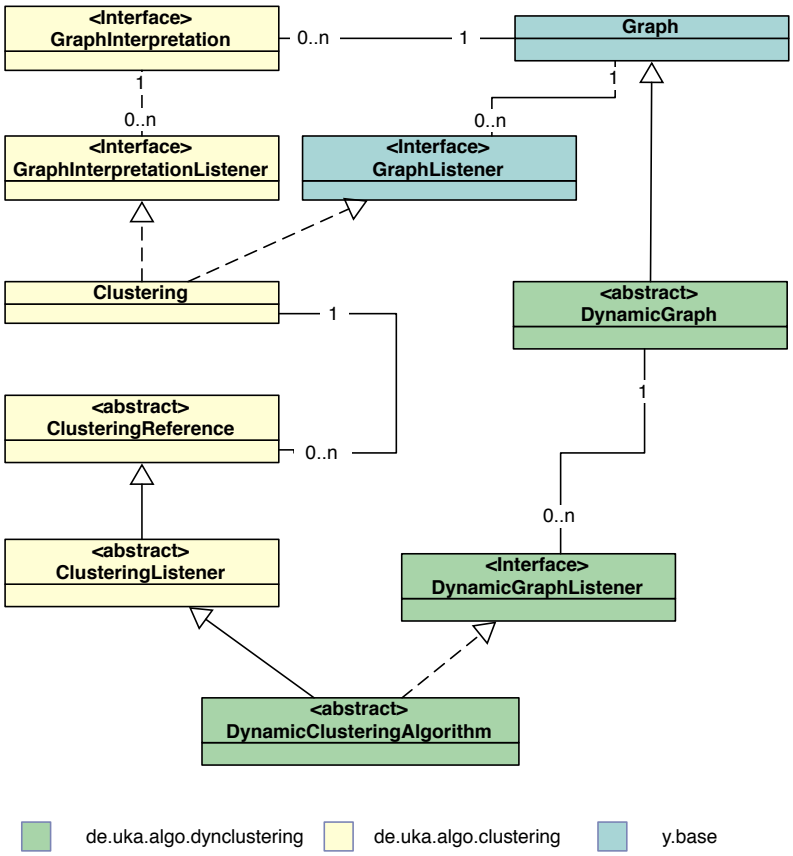


Figure 3.1: Class diagramm of framework components

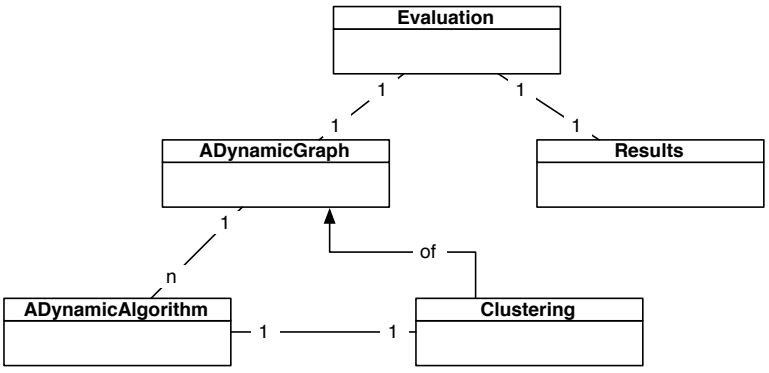


Figure 3.2: Object diagram of an evaluation setup

Chapter 4

Strategies and Algorithms

Formally, a dynamic clustering algorithm is a procedure which, given the previous state of a dynamic graph G_{t-1}, a sequence of graph events $\Delta(G_{t-1}, G_t)$ and a clustering $\zeta(G_{t-1})$ of the previous state, returns a clustering $\zeta(G_t)$ of the current state. While the algorithm may discard $\zeta(G_{t-1})$ and simply start from scratch, a good dynamic algorithm will harness the results of its previous work and revise them according to the changes.

Deciding which part of the clustering is up for discussion and what remains fixed is the task of a *prep strategy*. The strategy prepares a partially finished clustering in response to changes, and passes it on to the algorithm for completion. We consider several strategies which can be combined with different algorithms as described later.

This chapter also contains descriptions of our algorithm candidates. Among them are fully dynamic algorithms, but it is also possible to incorporate static clustering algorithms into the framework by letting them pose as dynamic ones. The candidates can also be categorized into heuristics (dGlobal and dLocal including their static counterparts) and exact optimizers (pILP and EOO).

4.1 Prep Strategies

As mentioned before, a dynamic clustering algorithm will draw upon a previous clustering and react to changes with local modifications. An *prep strategy* determines the problem that the algorithm has to solve after changes have occured, which is generally a part of the static problem. Our dynamic algorithms can employ one of several strategies: Both the *breakup strategy* (BU) and the two *traversal strategies* designate a subset $\tilde{V}$ of V needing reassessment. How to deal with the nodes marked by the strategy is up to the algorithm. We will refer to this class of *prep strategies* as *subset strategies*. In contrast, the *backtrack strategy*

operates on the clustering history of agglomerative algorithms.

Breakup Strategy (BU). A simple reaction to an edge event $\{u, v\}$ is to mark the clusters $\zeta(u)$ and $\zeta(v)$ entirely. These nodes can then be handled by the algorithm, for example by breaking up the clusters into singletons. The *breakup strategy* is applicable to all of the algorithms described. The strategy itself requires very little computation. Figure 4.1 illustrates an example.

Name	Event	Reaction
edge creation	$E + \{u, v\}$	$\tilde{V} + (\zeta(u) \cup \zeta(v))$
edge removal	$E - \{u, v\}$	$\tilde{V} + (\zeta(u) \cup \zeta(v))$
weight increase	$\omega(u, v) + x$	$\tilde{V} + (\zeta(u) \cup \zeta(v))$
weight decrease	$\omega(u, v) - x$	$\tilde{V} + (\zeta(u) \cup \zeta(v))$

Table 4.1: BU Strategy

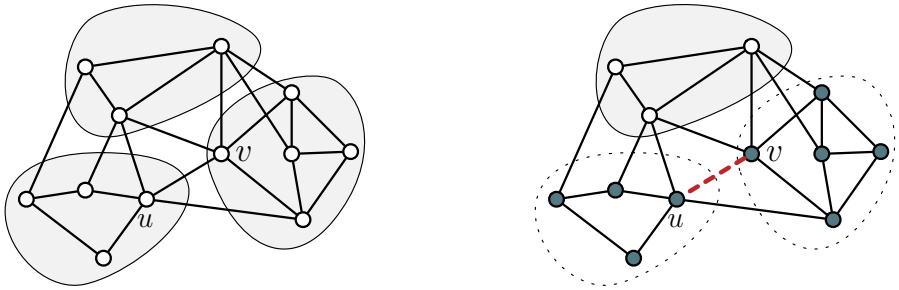


Figure 4.1: The clusters affected by the edge event $\{u, v\}$ freed by the BU strategy

Traversal Strategies. All of the dynamic algorithms we discuss here are built on the assumption that the effect graph events have regarding an optimal clustering are local, i.e., that nodes in the vicinity of the event are more likely to need reassessment than remote nodes. However, the nodes in the affected clusters are not necessarily the nodes closest to the event, and too many nodes might be freed by the *breakup strategy*. Instead, graph traversal (e.g. breadth first search) can be used to quickly obtain a suitable neighborhood of the event. For definitions of neighborhoods and descriptions of possible traversal algorithms, we refer the reader to Appendix A. Suffice it to say that: a) these strategies free nodes up to a certain distance d from the event (*neighborhood strategy* N_d) or a certain

number k of nodes near the event (*bounded neighborhood strategy* BN_k); b) they can easily be implemented as BFS-like algorithms. Figure 4.1 shows an example neighborhood.

Name	Event	Reaction
edge creation	$E + \{u, v\}$	$\tilde{V} + N(\{u, v\})$
edge removal	$E - \{u, v\}$	$\tilde{V} + N(\{u, v\})$
weight increase	$\omega(u, v) + x$	$\tilde{V} + N(\{u, v\})$
weight decrease	$\omega(u, v) - x$	$\tilde{V} + N(\{u, v\})$

Table 4.2: Traversal strategy (N or BN)

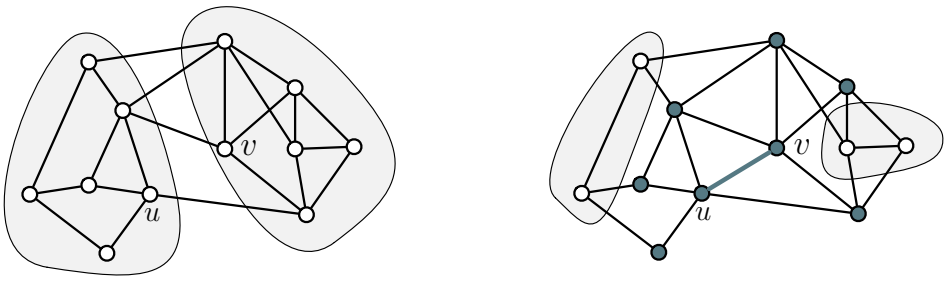


Figure 4.2: The 1-neighborhood of the edge event $\{u, v\}$ freed by the N^1 strategy

Backtrack Strategy (BT). This strategy is applicable to greedy agglomerative algorithms, such as dGlobal. It maintains a history of the merge operations that led to the current clustering. Triggered by a graph event, it can backtrack up to a certain point, giving the algorithm a chance to incorporate changes. The strategy is based on a data structure U which stores the agglomeration history, comparable to a traditional *union-find-backtrack* data structure. Such a data structure allows for several different reactions to a graph event. It supports the operation backtrack(v), which steps back in the agglomeration history and splits the cluster containing v into the two subsets it resulted from (see Algorithm 2). This operation can be used to formulate the operations isolate(v) and separate(u, v) as specified in Algorithms 3 and 4. The given pseudocode is a high level description of the operations. See Appendix B for the specifics of a possible implementation. The strategy reacts to a graph according to Table 4.3. After each time step, the algorithm is applied to the resulting clustering.

Name	Event	Reaction	
node creation	$V + u$	$U.\mathsf{add}(v)$	
node removal	$V - u$	$U.\mathsf{remove}(v)$	
edge creation	$E + \{u, v\}$	$U.\mathsf{separate}(u, v)$	$\zeta(u) = \zeta(v)$
		$U.\mathsf{isolate}(u), U.\mathsf{isolate}(v)$	$\zeta(u) \neq \zeta(v)$
edge removal	$E - \{u, v\}$	$U.\mathsf{isolate}(u), U.\mathsf{isolate}(v)$	$\zeta(u) = \zeta(v)$
		$-$	$\zeta(u) \neq \zeta(v)$
weight increase	$\omega(u, v) + x$	$U.\mathsf{separate}(u, v)$	$\zeta(u) = \zeta(v)$
		$U.\mathsf{isolate}(u), U.\mathsf{isolate}(v)$	$\zeta(u) \neq \zeta(v)$
weight decrease	$\omega(u, v) - x$	$U.\mathsf{isolate}(u), U.\mathsf{isolate}(v)$	$\zeta(u) = \zeta(v)$
		$-$	$\zeta(u) \neq \zeta(v)$
cluster merge	$C_i \cup C_j \to C_k$	$U.\mathsf{union}(u \in C_i, v \in C_j)$	

Table 4.3: BT strategy

Algorithm 2: backtrack splits a cluster containing v into the two parts it resulted from

Input: v: a node

1 $\zeta(v) \to \zeta'(v) \cup \zeta(v) \setminus \zeta'(v)$

Algorithm 3: isolate backtracks until v is contained in a singleton

Input: $v \in V$

1 **while** $|\zeta(v)| \neq 1$ **do**

2 $\quad$ backtrack(v)

Algorithm 4: separate backtracks until u and v are in different clusters

Input: $u, v \in V$

1 **while** $\zeta(u) = \zeta(v)$ **do**

2 $\quad$ backtrack(u)

4.2 Algorithms

In the following we describe our algorithm candidates: Two heuristic approaches, dGlobal and dLocal, including their static precursors sGlobal and sLocal; one approach performing local exact optimization via integer linear programming, pILP; and one approach aiming at optimization through a number of elemental operations, EOO.

4.2.1 sGlobal

sGlobal implements the widely known clustering method introduced by Newman et al. in [CNM04], a globally greedy agglomerative algorithm. For the static version, the clustering is reset to singletons after each time step. The clustering is then passed to the procedure run_sGlobal (Algorithm 5). For each pair of clusters, it determines the increase $\Delta_{\mathcal{M}}(C_i, C_j)$ in modularity that can be achieved by merging the pair, and merges the pair for which the increase is maximal. This is repeated until no more improvement is possible.

Name	Event	Reaction
node creation	$V + u$	$\zeta + \{u\}$
node removal	$V - u$	$\zeta(u) - u$
edge creation	$E + \{u, v\}$	$-$
edge removal	$E - \{u, v\}$	$-$
weight increase	$\omega(u, v) + x$	$-$
weight decrease	$\omega(u, v) - x$	$-$
time step	$t + 1$	run_sGlobal$(G, \text{singletons}(V))$

Table 4.4: sGlobal behavior

4.2.2 dGlobal

sGlobal can be made dynamic by not resetting the clustering completely, but revising it according to a *prep strategy*. Starting from the revised clustering $\tilde{\zeta}$, the dynamic dGlobal performs greedy agglomerations just like the static version. The main factor is the choice of the *prep strategy* (see Section 4.1). If a *subset strategy* is employed, all marked nodes are turned into singletons. When using the *backtrack strategy*, the clustering is set to the clustering induced by the backtrack data structure U.

$$\mathsf{revise}(S,\zeta) := \begin{cases} \tilde{\zeta} \leftarrow \zeta.\mathsf{singletons}(\tilde{V}) & S \text{ is a subset strategy} \\ \tilde{\zeta} \leftarrow U.\mathsf{getClustering}() & S \text{ is the backtrack strategy} \end{cases} \tag{4.2.1}$$

parameter	domain
prep strategy	BT, N, BU, BN

Table 4.5: dGlobal parameters

Name	Event	Reaction
node creation	$V + u$	$\zeta + \{u\}$
node removal	$V - u$	$\zeta(u) - u$
edge creation	$E + \{u, v\}$	$\rightarrow S$
edge removal	$E - \{u, v\}$	$\rightarrow S$
weight increase	$\omega(u, v) + x$	$\rightarrow S$
weight decrease	$\omega(u, v) - x$	$\rightarrow S$
time step	$t + 1$	$\mathsf{revise}(S, \zeta)$, $\mathsf{run\_sGlobal}(G, \tilde{\zeta})$

Table 4.6: dGlobal behavior

Algorithm 5: run_sGlobal

Input: graph G, clustering $\zeta(G)$
Output: clustering $\zeta(G)$

```
while ∃(C_i, C_j) ∈ (ζ 2) : Δ_M(C_i, C_j) > 0 do
    (C_a, C_b) ← arg max_{(C_i,C_j)∈ (ζ 2)} Δ_M(C_i, C_j)
    ζ.merge(C_a, C_b)
return ζ(G)
```

4.2.3 sLocal

Blondel et al. propose a locally greedy algorithm based on *modularity* in [BGLL08]. Rather than performing the globally best merger, sLocal considers the nodes in turn, moves them to the best neighboring cluster and contracts the graph for the next iteration. In the following description, $\Delta_{\mathcal{M}}(u, v)$ denotes the improvement in *modularity* which can be achieved by move(u, v), i.e. moving u to the cluster of v. The operation contract(G, ζ) returns a *contracted graph* where $\hat{V} = \zeta$. $N(u)$ is the direct neighborhood of a node.

Algorithm 6 is a description of the core procedure. The inner loop contains an iteration over all $u \in V$, where $\Delta_{\mathcal{M}}(u, v)$ is evaluated for all v in $N(u)$. If a positive improvement can be achieved, u is moved to the cluster of the neighbor with the maximum improvement. The inner loop breaks if the last pass over V yields no changes to the clustering. Then, the outer loop continues by contracting the graph according to ζ, and starts again with a singleton clustering on the contracted graph. This is repeated until a pass through the outer loop body yields no more changes. At this point the hierarchy of contracted graphs induces a clustering of the original graph, which is obtained via the unfurl operation.

The exact outcome of the algorithm depends on the order in which nodes are visited (hence the parameter *node order strategy*) but Blondel et al. claim in [BGLL08] that this does not have a significant effect on the resulting clustering quality. A *stopping criterion* can be specified, although the only option so far is to stop when the first peak in *modularity* is reached. This is a reasonable choice, since no agglomeration will improve the result any further at this point.

parameter	domain
node order strategy	Array, InverseArray,Random
stopping criterion	FirstPeak
update policy	Affected, Neighbors

Table 4.7: sLocal parameters

Name	Event	Reaction
node creation	$V + u$	$\zeta + \{u\}$
node removal	$V - u$	$\zeta(u) - u$
edge creation	$E + \{u, v\}$	$-$
edge removal	$E - \{u, v\}$	$-$
weight increase	$\omega(u, v) + x$	$-$
weight decrease	$\omega(u, v) - x$	$-$
time step	$t + 1$	$\mathsf{run\_sLocal}(G)$

Table 4.8: sLocal behavior

Algorithm 6: run_sLocal

Input: graph G
Output: clustering $\zeta(G)$

$\hat{G}^0 \leftarrow G$
repeat
  $\zeta \leftarrow \{\{u\} : u \in V\}$
  repeat
    for u **in** V **do**
      if $\max_{v \in N(u)} \Delta_{\mathcal{M}}(u, v) \geq 0$ **then**
        $w \leftarrow \arg\max_{v \in N(u)} \Delta_{\mathcal{M}}(u, v)$
        $\mathsf{move}(u, \zeta(w))$
  until *no more changes*
  $\hat{G}^{h+1} \leftarrow \mathsf{contract}(\hat{G}^h, \zeta)$
until *no more changes*
$\zeta(G) \leftarrow \mathsf{unfurl}(\hat{G}^{h_{max}})$
return $\zeta(G)$

4.2.4 dLocal

We present a dynamic version of the sLocal algorithm. Like its static counterpart, dLocal creates a hierarchy of contracted graphs from $\hat{G}^0 = G$ to $\hat{G}^h$. Like other dynamic algorithms, it first forwards a graph event to the *prep strategy*, which yields a set $\tilde{V}$ of nodes needing reassessment. In the first iteration, these nodes are considered in a certain order. If a quality improvement is possible, nodes are moved, changing the clustering on the lowest level. This in turn induces graph events in the contracted graph of the next higher level. We do not apply the same *prep strategy* at the higher levels of the hierarchy, since the number of affected elementary nodes could heavily vary and easily become to large. Instead, the parameter *update policy* determines which nodes are revisited in this level - either only the nodes directly affected (policy Affected), or also their direct neighbors (policy Neighbors). From there the procedure continues - creating new hierarchy levels if necessary - until a stable singleton cluster arises at the highest level.

parameter	domain
node order strategy	Array, InverseArray,Random
stopping criterion	FirstPeak
update policy	Affected,Neighbors

Table 4.9: dLocal parameters

Name	Event	Reaction
node creation	$V + u$	$\zeta + \{u\}$
node removal	$V - u$	$\zeta(u) - u$
edge creation	$E + \{u, v\}$	$\rightarrow S$
edge removal	$E - \{u, v\}$	$\rightarrow S$
weight increase	$\omega(u, v) + x$	$\rightarrow S$
weight decrease	$\omega(u, v) - x$	$\rightarrow S$
time step	$t + 1$	run_dLocal($\tilde{V}$)

Table 4.10: dLocal behavior

4.2.5 EOO

The EOO (Elemental Operations Optimizer) performs a limited number of elemental operations at each time step, trying to optimize the clustering quality according to a given measure. The elemental operations available are listed in Table 4.2.5. Although rather limited in its options, the EOO is often used as a post-processing tool ([NR09]).

parameter	domain
optimization strategy	Optimizer, SimulatedAnnealing, Minimizer
quality index	$\mathcal{M}$, $\mathcal{C}$, $\mathcal{P}$, ...
max. number of operations	$\mathbb{N}$
allow merge	Boolean
allow shift	Boolean
allow split	Boolean

Table 4.11: EOO parameters

Name	Event	Reaction
node creation	$V + u$	$\zeta + \{u\}$
node removal	$V - u$	$\zeta(u) - u$
edge creation	$E + \{u, v\}$	$-$
edge removal	$E - \{u, v\}$	$-$
weight increase	$\omega(u, v) + x$	$-$
weight decrease	$\omega(u, v) - x$	$-$
time step	$t + 1$	run_EOO(G_{t+1}, ζ)

Table 4.12: EOO behavior

Operation	Effect
merge(u,v)	$\zeta(u) \cup \zeta(v)$
shift(u,v)	$\zeta(u) - u, \zeta(v) + u$
split(u)	$(\{u\}, \zeta(u) \setminus u) \leftarrow \zeta(u)$

Table 4.13: EOO operations

Algorithm 7: runEOO

Input: graph G, clustering ζ, maximum number of operations max, quality index Q

Output: clustering ζ

```
for u ∈ randomOrder(V) do
    if i < max then
        if merge allowed then
            v ∈ N(u)
            if Q(ζ.merge(u, v)) > Q(ζ) then
                ζ.merge(u, v)
                i ← i + 1
        if shift allowed then
            v ∈ N(u)
            if Q(ζ.shift(u, v)) > Q(ζ) then
                ζ.shift(u, v)
                i ← i + 1
        if split allowed then
            if Q(ζ.split(u)) > Q(ζ) then
                ζ.split(u)
                i ← i + 1
```

4.2.6 pILP

When looking for an optimal clustering, one approach is to formulate the clustering problem as an *Integer Linear Program (ILP)* in order to give an instance of the problem to a solver (such as the free `lp_solve` or the commercial `CPLEX`). The solver performs an optimization of an objective function subject to a set of constraints, yielding an optimal solution. In this case the objective function expresses a quality index while the constraint set ensures that the result is a valid clustering. In practive, this is very costly in terms of computing time, and becomes infeasible for more than a few hundred nodes.

However, ILP optimization could still be a feasible approach in a dynamic setting. Using a *prep strategy*, one could let the solver work out a part of the problem small enough to be completed in time. Within our framework, we implemented the partial ILP clustering algorithm as presented by HÜBNER in [Hüb08]. A description of the ILP follows.

parameter	domain
subset strategy	BU, N, BN
objective function	$\mathcal{P}, \mathcal{P}_\omega, \mathcal{M}, \mathcal{M}_\omega$
solver	`CPLEX`, `lp_solve`
variant	NM, M

Table 4.14: pILP parameters

Name	Event	Reaction
node creation	$V + u$	$\zeta + \{u\}$
node removal	$V - u$	$\zeta(u) - u$
edge creation	$E + \{u, v\}$	$\rightarrow S$, run_pILP$(\tilde{V})$
edge removal	$E - \{u, v\}$	$\rightarrow S$, run_pILP$(\tilde{V})$
weight increase	$\omega(u, v) + x$	$\rightarrow S$, run_pILP$(\tilde{V})$
weight decrease	$\omega(u, v) - x$	$\rightarrow S$, run_pILP$(\tilde{V})$
time step	$t + 1$	–

Table 4.15: pILP behavior

ILP Formulation of the Clustering Problem. A clustering can be understood as an equivalence relation on V:

$$u \sim v \Leftrightarrow \zeta(u) = \zeta(v) \tag{4.2.2}$$

Algorithm 8: run_pILP

Input: $\tilde{V}$: subset of V
Output: ζ: a clustering of G

```
freeNodes(Ṽ)
ilp ← createILP(Ṽ, ζ̃)
solution ← solve(ilp, obj)
ζ ← readClustering(solution)
```

This view is suited for an ILP formulation of the clustering problem. Let V be a set of nodes. Then we can define a decision variable X_{uv} for each pair of nodes, which yields the set

$$\mathcal{X}(V) := \{X_{uv} : \{u, v\} \in \binom{V}{2}\} \tag{4.2.3}$$

The value of this variable will be 0 if both nodes belong to the same cluster, and 1 otherwise.

$$X_{uv} = \begin{cases} 0 & \zeta(u) = \zeta(v) \\ 1 & \zeta(u) \neq \zeta(v) \end{cases} \tag{4.2.4}$$

Since this corresponds to a notion of distance, we call it the *metric model* (as opposed to the *equivalence model*, which is the inverse case). Both models are equivalent, but the metric model is preferred for reasons of much better performance with `lp_solve`.

Reflexivity (4.2.5), symmetry (4.2.6) and transitivity (4.2.7) of the equivalence relation can be expressed as ILP constraints. The constraints for reflexivity and symmetry can be omitted for the implementation because they are not part of the objective function.

$$\forall u \in V : X_{uu} \leq 0 \tag{4.2.5}$$

$$\forall \{u, v\} \in \binom{V}{2} : X_{uv} - X_{vu} \leq 0 \tag{4.2.6}$$

$$\forall \{u, v, w\} \in \binom{V}{3} : \begin{cases} X_{uv} + X_{vw} - X_{uw} \geq 0 \\ X_{uv} + X_{uw} - X_{vw} \geq 0 \\ X_{uw} + X_{vw} - X_{uv} \geq 0 \end{cases} \tag{4.2.7}$$

Partial Re-Clustering. After the *prep strategy* has reacted to an edge event by marking some nodes, the algorithm turns them into a subset $\tilde{V} \subseteq V$ of *free nodes* not belonging to any cluster. This leaves a clustering $\tilde{\zeta}$ restricted to the remaining nodes. The free nodes correspond to the decision variables $\mathcal{X}(\tilde{V})$. Additonally, we need a set of decision variables for pairs of free nodes and remaining clusters.

$$\mathcal{Y}(\tilde{V}, \tilde{\zeta}) := \{Y_{uC} : \{u, C\} \in \tilde{V} \times \tilde{\zeta}\} \tag{4.2.8}$$

The value of these variables will determine the cluster to which the free node is reassigned in the following way:

$$Y_{uC} = \begin{cases} 0 & u \in C \\ 1 & u \notin C \end{cases} \tag{4.2.9}$$

Using these decision variables, we can formulate weighted *modularity* as an objective function as follows. Because the decision variables represent a distance, the objective function has to be minimized. It is easy to see that this is equivalent to a maximization when using $(1 - X_{uv})$ etc.

$$\begin{aligned} \mathcal{M}_{\mathsf{pILP(NM)}}(G, \zeta) = &\sum_{X_{uv} \in \mathcal{X}(\tilde{V})} \left(\omega(u, v) - \frac{\omega(u) \cdot \omega(v)}{2\omega(E)} \right) X_{uv} \\ &+ \sum_{Y_{uC} \in \mathcal{Y}(\tilde{V}, \tilde{\zeta})} \left(\sum_{w \in C} \left(\omega(u, w) - \frac{\omega(u) \cdot \omega(w)}{2\omega(E)} \right) \right) Y_{uC} \end{aligned} \tag{4.2.10}$$

In order to construct an instance of the ILP, a number of constraints are added. Equation (4.2.11) expresses transitivity. The relation between a cluster and a free node is subject to similar transitivity constraints, as expressed in Equation (4.2.12). Additionally, the constraints in Equation (4.2.13) guarantee that a node is not assigned to more than one cluster.

$$\forall \{u, v, w\} \in \binom{\tilde{V}}{3} : \begin{cases} X_{uv} + X_{vw} - X_{uw} \geq 0 \\ X_{uv} + X_{uw} - X_{vw} \geq 0 \\ X_{uw} + X_{vw} - X_{uv} \geq 0 \end{cases} \tag{4.2.11}$$

$$\forall \{u, v, C\} \in \binom{\tilde{V}}{2} \times \tilde{\zeta} : \begin{cases} X_{uv} + Y_{vC} - Y_{uC} \geq 0 \\ X_{uv} + Y_{uC} - Y_{vC} \geq 0 \\ Y_{uC} + Y_{vC} - X_{uv} \geq 0 \end{cases} \tag{4.2.12}$$

$$\forall u \in \tilde{V} : \sum_{C \in \tilde{\zeta}} Y_{uC} \geq k - 1 \tag{4.2.13}$$

So far we have not allowed two clusters in $\tilde{\zeta}$ to merge in order to improve the clustering (the NM variant of pILP). But this can be achieved easily by modifying the set of constraints, yielding the M variant of pILP. We need to introduce another set of decision variables which determine whether two clusters should be merged.

$$\mathcal{Z}(\zeta) := \{Z_{CD} : \{C, D\} \in \binom{\zeta}{2}\} \tag{4.2.14}$$

$$Z_{CD} = \begin{cases} 0 & \mathsf{merge}(C, D) \\ 1 & - \end{cases} \tag{4.2.15}$$

We apply two additional constraint sets (4.2.16) and (4.2.17) which ensure the validity of the merge operations. Equation (4.2.18) shows the extended objective function which takes the new variables into account.

$$\forall \{C, D, E\} \in \binom{\tilde{\zeta}}{3} : \begin{cases} Z_{CD} + Z_{DE} - Z_{CE} \geq 0 \\ Z_{CD} + Z_{CE} - Z_{DE} \geq 0 \\ Z_{CE} + Z_{DE} - Z_{CD} \geq 0 \end{cases} \tag{4.2.16}$$

$$\forall \{u, C, D\} \in \tilde{V} \times \binom{\tilde{\zeta}}{2} : \begin{cases} Z_{CD} + Y_{uD} - Y_{uC} \geq 0 \\ Z_{CD} + Y_{uC} - Y_{uD} \geq 0 \\ Y_{uC} + Y_{uD} - Z_{CD} \geq 0 \end{cases} \tag{4.2.17}$$

$$\begin{aligned} \mathcal{M}_{\mathsf{pILP(M)}}(G, \zeta) = & \sum_{X_{uv} \in \mathcal{X}(\tilde{V})} \left(\omega(u, v) - \frac{\omega(u) \cdot \omega(v)}{2\omega(E)} \right) X_{uv} \\ & + \sum_{Y_{uC} \in \mathcal{Y}(\tilde{V}, \tilde{\zeta})} \left(\sum_{w \in C} \left(\omega(u, w) - \frac{\omega(u) \cdot \omega(w)}{2\omega(E)} \right) \right) Y_{uC} \\ & + \sum_{Z_{CD} \in \mathcal{Z}(\zeta)} \left(\sum_{x \in C} \sum_{y \in D} \left(\omega(x, y) - \frac{\omega(x) \cdot \omega(y)}{2\omega(E)} \right) \right) Z_{CD} \end{aligned} \tag{4.2.18}$$

Name	Constraint Set
NM	(4.2.11), (4.2.12), (4.2.13)
M	(4.2.11), (4.2.12), (4.2.16), (4.2.17)

Table 4.16: pILP variants

4.3 Reference

Since it is possible to emulate a clustering algorithm by using information provided by the dynamic graph generator (see Section 2.1), we implemented it as the Reference algorithm. At each time step, Reference reads clustering data from the file containing the generated graph and assigns nodes according to $\zeta_{\text{ref}}(G_t)$. For significant clusterings, the clustering delivered by the generator should be considered a good reference and present an upper bound on quality. For unclear clusterings however, clustering algorithms should be able to find a better, hidden cluster structure. Algorithms should also surpass Reference while cluster events are in progress, which will be falling behind in terms of quality until the event is considered complete by the generator. In the result plots, sudden drops in reference quality indicate changes in the ground-truth clustering. The quality catches up again when the merge or split event is considered complete. This is a useful feature of Reference which helped to assess the ability of the actual algorithms to react to clustering changes.

Chapter 5

Evaluation Results

5.1 Evaluation Criteria

Our data strongly suggests that all of the distance measures are qualitatively equivalent for our purposes, and differ only in scale (Plot C.5 illustrates this). Therefore we chose the graph-structural *Rand* measure $\mathcal{R}_g$ as a good representative in the following experiments. In the following, *smoothness* generally refers to low values of $\mathcal{R}_g$. Since maximizing *modularity* has become the primary method for finding clusterings, we chose it as the main quality index. In the following, *quality* generally refers to high values of $\mathcal{M}$.

What follows is a summary of the relevant evaluation criteria. All of them are functions of t.

Runtime The computation time required by prep strategy and clustering algorithm combined, measured in *ms*, is plotted.

Cluster Count The number of clusters in a clustering produced by an algorithm is plotted.

Quality The quality of a clustering produced by an algorithm, mainly measured in $\mathcal{M}$, is plotted.

Smoothness The distance between the current and the previous clustering produced by an algorithm, mainly measured in $\mathcal{R}_g$, is plotted. Lower distance values indicate a smoother dynamic clustering.

5.2 Plots

Plotting the raw data for many measures may produce cluttered graphics. In most plots, we apply a central moving average function (Equation (5.2.1)) to smoothen

the data and make our plots more readable. The function maps a raw data point to a smoothed data point. The width w of the sliding window is an estimate depending on the total number of time steps, namely $\frac{1}{50}t_{max}$. As a caveat, note that a nonzero curve for the smoothed distance values should not be interpreted as the algorithm changing its clustering in every single time step.

$$s[i] := \frac{\sum_i^{i+w} d[i]}{w} \quad (i + w \leq |d|) \tag{5.2.1}$$

For historical reasons, the plot legends use different names for algorithms and strategies than this paper. Table 5.2 shows the corresponding names.

Text name	Plot name
sGlobal	StaticNewman2
dGlobal	Newman2
sLocal	StaticBlondel
dLocal	Blondel
pILP	PartialILP
BU	Breakup
BT	Backtrack
N	Neighborhood
BN	BoundendNeighborhood

Table 5.1: Legacy names in plot legends

5.3 Results

5.3.1 Performance of Static Heuristics

To set the stage, we first give an overview of the performance of the static heuristics, sGlobal and sLocal. We believe that sLocal is slower due to implementation overhead in spite of its lower theoretical runtime. It requires a more complex implementation, giving sGlobal a speed advantage for smaller graphs. It is evident from Plots C.15 and C.16 that sLocal is superior to sGlobal both in terms of quality and smoothness. As shown in Plot C.14, sGlobal fails to recognize a number of clusters as high as Reference.

5.3.2 Discarding EOO

Plots C.6, C.7 and C.8 show a run of all the candidates on a small generated instance. We can see that the quality produced by EOO remains below those of competitors and deteriorates. Increasing the number of elemental operations does not seem to have a significant influence on either the quality or the runtime. Since its runtime increases strongly with instance size, relying on EOO is infeasible for large networks. This leads us to the conclusion that we can discard EOO as a viable candidate for now. We pass on a thorough examination of its parameters in favor of the other candidates.

5.3.3 Algorithm Parameters

sLocal and dLocal Blondel et. al. state that the order in which their algorithm considers the nodes does not affect the overall quality of its result. In order to check this assumption, we compared three different node order strategies in the static case: Array visits the nodes in the order they are stored in the data structure, which remains more or less constant, InvArray is the inverse order, Random visits the nodes in a random order. Our results indicate that the assumption is generally true. Plot C.9 shows some variation, but no overall advantage for a single strategy. However, it becomes clear that the Random node order clusters less smoothly, as illustrated by Plot C.10.

The question remains whether this is also true for the dynamized version of the algorithm. Additionally, a decision has to be made whether to consider only the Affected nodes in the contraction hierarchy or also their Neighbors. We applied all combinations of these parameters to $\mathcal{G}_e$. Although the search space is larger with Neighbors, there seems to be no effect on the runtime. As shown by Plots **??** and **??**, there are minor variations in quality and distance, but no choice of parameters stands out.

Variants of pILP Two different constraint sets lead to two variants of pILP - pILP(M) allows for the merger of existing clusters, pILP(NM) does not. The two variants differ most clearly in the number of clusters they recognize, which is generally lower for M. As illustrated by Plot C.17, they roughly bound the cluster count of the dynamic heuristics from above and below. Generally, M takes longer and leads to a coarser clustering with slightly lower *modularity*. We therefore reject it, and limit the discussion to the NM variant of pILP in the following.

5.3.4 Adapting to Cluster Events

Our results show that the dynamic approaches have the ability to adapt quickly and appropriately to cluster events. After a small decline in quality at the beginning of the event, they usually catch up quickly (as illustrated by Plot C.18).

5.3.5 Heuristics versus Optimization

As a striking result, heuristics (dGlobal and dLocal) are consistently better than the exact optimizer (pILP) in terms of quality (see Plot C.19). This is true regardless of whether pILP allows the merging of clusters or not. The deficit does not improve over time. We conjecture that this phenomenon can be explained in the following way: pILP gets caught in local optima which it cannot easily escape, a behavior that can be likened to *overfitting*. Since pILP is also the most processor-intensive algorithm, it seems ill-suited for updates on large graphs. This leaves the heuristic algorithms as the best candidates for dynamic scenarios.

5.3.6 Behavior of the Dynamic Heuristics

For dLocal, a gradual improvement of quality and smoothness over time could be observed. Apparently the overall clustering benefits from repeated local disturbances. This effect is reminiscent of *simulated annealing*.

We cannot make a general statement on whether dLocal is superior to dGlobal or vice-versa. The best choice in terms of quality may depend on the nature of the target graph: dLocal surpasses dGlobal on almost all generated graphs, dGlobal is superior on our real-world instance $\mathcal{G}_e$. We speculate that this is due to $\mathcal{G}_e$ featuring a power law degree distribution in contrast to the Erdös-Renyi-type generated instances.

5.3.7 Prep Strategies

Many combinations of algorithms with *prep strategies* and strategy parameters are possible. We evaluated as many as practical in order to determine the strategy best suited for an algorithm. The strategies BU, N_d and BN_s are applicable to both dGlobal and dLocal. By nature, BT is only suitable for dGlobal. For the parametrized strategies, we tried N_d for $d \in \{0, 1, 2, 3\}$ and BN_s for $s \in \{2, 4, 8, 16, 32\}$. Our aim was to give a sound recommendations for the choice of a strategy. The focus is on the trade-off between runtime on the one hand and quality and smoothness on the other.

Considering only the nodes incident to a changing edge (i.e. N_0 and BN_2) proved insufficient, and is therefore ignored in the following. For dLocal with N_d,

increasing d has only marginal effect on quality and smoothness, while runtime grows sublinearly (see Plot C.37). This suggests N_1 as an appropriate strategy. For dGlobal, N_d risks high runtimes for depths $d > 1$, depending on the density of the graph. Depths $d > 1$ also seem to deteriorate quality, a strong indication that large search spaces contain local optima. Smoothness approaches the unwanted values of sGlobal for $d > 2$. Again, $d = 1$ is the depth of choice. For BN_s, increasing s is essentially equivalent to increasing d, only on a finer scale. Consequently, we can report similar observations. For dLocal, BN_4 turned out slightly superior. The quality produced by dLocal benefits from increasing s in the range $[2, 32]$ (see Plot C.23), but at the cost of speed and smoothness. We conclude that BN_{16} is a reasonable choice. The BU strategy generally falls behind the other strategies in terms of all criteria (see Plots C.43 and C.45). BU tends to produce overly large search spaces, which also depend on the size of the affected clusters. Algorithms equipped with the BU strategy often mimic the behavior of their static counterparts. We conclude that breaking up entire clusters is not an advisable strategy. A major point for the BT strategy is its speed. dGlobal combined with BT is by far the fastest algorithm (see Plot C.41). It also yields competitive quality, but at the expense of smoothness, which is in the range of the static algorithms. A speedup of up to a factor of 1k was observed at 1k nodes. For scenarios where speed is decisive, dGlobal with BT is the best candidate.

As a general result, expanding the search space beyond a small neighborhood of the event is not justified when trading off runtime against quality and smoothness. In accord with the locality assumption, very small search spaces yield good results.

Chapter 6

Conclusion

6.1 Summary of Insights

Overall, the outcomes of our evaluation are very favorable for the dynamic approach. The dynamic algorithms prove their ability to react quickly and appropriately to changes in the ground-truth clustering. We show that dynamically maintaining a clustering does not only save time, but also yields higher clustering quality and guarantees much smoother clustering dynamics than static recomputation (see Plot C.8).

Interestingly, the heuristic algorithms turned out to be superior to locally exact optimization. We assume that this is due to an effect akin to overfitting. Together with its high runtimes, pILP is not suited for updates on large graphs. dLocal and dGlobal turned out to be the most promising algorithms, performing strongly for all criteria. The choice between the two may depend on the characteristics of the target graph.

We observed that dLocal is less succeptible to an increase of the search space than dGlobal. However, our results argue strongly for the locality assumption in both cases - expanding the search space beyond a very limited range is not justified when trading off runtime against quality. On the contrary, quality and smoothness may even suffer. Surprisingly small search spaces work best, avoid trapping local optima well and adapt quickly and aptly to changes in the ground-truth clustering. This observation strongly argues for the assumption that changes in the graph ask for local updates on the clustering.

Breaking up entire clusters (BU) tends to free too many nodes. Consequently, *prep strategies* N and BN with a limited range are capable of producing high-quality clusterings while excelling at smoothness. The BT strategy yields competitive quality at unrivaled speed, but at the expense of smoothness; a constraint is that it is only applicable to dGlobal.

6.2 Starting Points for Follow-up Work

While designing *prep strategies*, the idea came up that freeing only the 'right' nodes in the neighborhood of an event might result in speedup. One could explore the potential benefits of prioritizing certain nodes when using a *traversal strategy*, e.g. according to the degree of the node. The framework provides provides starting points for this approach (for details see App. A), but it was not pursued further. Since small neighborhoods yielded the best results, reducing the number of nodes this way might not be necessary.

Since the implementation of the evaluation framework is a test-bed for ideas rather than code fine-tuned for efficiency, even higher speedup factors might be achieved with optimized implementations.

Trying to recognize the clustering structure of synthetic clustered graphs was a suitable method for exploring and evaluating the dynamic approach. Applying the insights gained in a compelling real world scenario, where useful information can be extracted from a dynamic clustering, is something that remains to be done. The dynamic algorithms could be suited as online algorithms, processing incoming changes to a graph-like set data.

Appendix A

Mini-Framework for Graph Traversal

With the locality assumption, the question arises how much of the previous clustering should be revised, i.e. which and how many nodes should be reassesed. It is straightforward that the nodes in the "neighborhood" of a graph event are the most promising candidates. How we choose this neighborhood is relevant for dynamic clustering algorithms, since its size and composition should reflect a good tradeoff between runtime and clustering quality. In this appendix we are going to formalize the notion of *neighborhood of a graph event*, and describe algorithms for obtaining it.

Obviously the neighboring nodes of an event should be near the nodes affected by the event, which calls for a definition of distance:

Definition 25. *The distance $d(u, v)$ of a pair of nodes in a graph is the length of the shortest path connecting the nodes.* NODE DISTANCE

$$\forall \{u, v\} \in V^2 : d(u, v) := \begin{cases} 0 & u = v \\ \infty & \nexists P(u, v) \\ \min\{|P(u, v)|\} \end{cases}$$

We can therefore define a neighborhood of a node as all nodes with at most a certain distance to it:

Definition 26. *The d-neighborhood N^d of a node s is the set of nodes connected to s by a path no longer than d.* d-NEIGHBORHOOD

$$N^d(s) := \{v \in V : d(s, v) \leq d\}$$

Breadth-first search (BFS) will visit and return nodes from the neighborhood of a source node s in the order of their distance $d(s, v)$. Because a sequence of

edge events affect at least two nodes, it is useful to generalize the algorithm from the neighborhood of a single node to the joint neighborhood of multiple nodes.

Definition 27. *The distance $d(S, v)$ of a node $v \in V$ from set $S \subseteq V$ of nodes is defined as*

NODE DISTANCE

$$\forall \{S, v\} \in \mathcal{P}(V) \times V : d(S, v) := \begin{cases} 0 & v \in S \\ \infty & \forall s \in S : \nexists P(s, v) \\ \min\{d(s, v) : s \in S\} \end{cases}$$

Definition 28. *The joint d-neighborhood of a set of nodes S is the set of nodes $v \in V$ with a distance no greater than d from S.*

JOINT d-NEIGHBORHOOD

$$N^d(S) = \{v \in V : d(S, v) \leq d\}$$

Proposition 1. *The joint d-neighborhood is equal to the union of all d-neighborhoods of the members of S.*

$$N^d(S) = \bigcup_{s \in S} N^d(s)$$

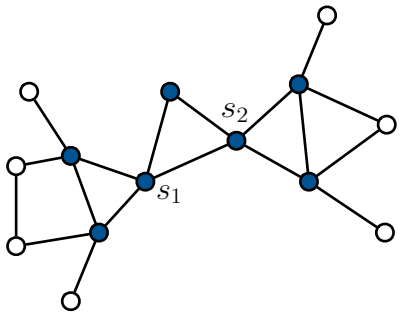


Figure A.1: The joint 1-neighborhood of $S = \{s_1, s_2\}$

$N^d(S)$ can be obtained via MultiSourceBFS (Algorithm 9). The algorithm is initialized with a set S of source nodes and assembles an ordering O of $V' \subseteq V$ according to $d(S, v)$. Throughout the search the following invariant for $O = (u_1, \ldots, u_l)$ holds:

$$\forall v \in V : v \notin O \implies (d(S, v) \geq d(s, u_l)) \tag{A.0.1}$$

Proof. Since MultiSourceBFS assembles O by performing O.append(Q.dequeue()), the order O follows from the order in which nodes are enqueued. Let u_l be

the last node appended to O. All unvisited neighbors of u_l are enqueued, having the distance $d(S, u_l) + 1$. It follows that $d(S, q_i)$ for $Q = (q_1, \ldots, q_l)$ is nondecreasing. It also follows that $d(S, u_l) \leq d(S, q_i) \leq d(S, u_l) + 1$ for $Q = (q_1, \ldots, q_l)$. Consequently, if a node v is not in O, it is either in Q, therefore $d(S, u_l) + 1 \geq d(S, v) \geq d(S, u_l)$, or in $V \setminus (Q \cup O)$, therefore $d(S, v) \geq d(S, u_l)$. □

It follows for the ordering returned that

$$\forall u_i, u_j \in O : i < j \implies (d(S, u_i) \leq d(S, u_j)) \tag{A.0.2}$$

The size of the neighborhood is exponential in d with the exact size depending strongly on the structure of the graph. For some algorithms, a d-neighborhood might be too small while a $d + 1$-neighborhood would be intractably large. It is therefore desirable to parametrize the BFS with a maximum number of nodes that will be visited.

Definition 29. *The k-bounded neighborhood of a set of source nodes S is defined as* K-BOUNDED NEIGHBORHOOD

$$N_k(S) := \{v_1, \ldots, v_k\} \subseteq O \tag{A.0.3}$$

Algorithm 9: MultiSourceBFS

Input: $S \subset V$: a set of source nodes, condition: exit condition
Output: O: an ordering of $V' \subseteq V$

```
1  A ← {}, Q ← (), O ← ()
2  for s in S do
3      Q.enqueue(s)
4      A ← A + s
5  while Q ≠ () do
6      u ← Q.dequeue()
7      if condition then
8          break
9      else
10         O.append(u)
11         for v in {v ∈ V : ∃{u, v} ∈ E} do
12             if v ∉ A then
13                 Q.enqueue(v)
14                 A ← A + v
```

MultiSourceBFS can be parametrized with an exit condition. Depending on the condition, it yields either $N^d(S)$ or $N_k(S)$.

$$\textsf{condition} := \begin{cases} d(S,u) > d & N^d(S) \\ |O| + 1 > k & N_k(S) \end{cases} \tag{A.0.4}$$

Using pure BFS as a traversal strategy leads to high-degree nodes having a higher probability of being freed than low-degree nodes. At the same time though, high-degree nodes are probably densely connected within their community - and less likely to change their cluster as a result of an edge event in their vicinity. Prioritizing low-degree nodes has therefore been suggested in [Hüb08].

Generally, nodes could be prioritized not only according to distance from the source (which BFS effectively does), but according to an arbitrary priority function $\rho : V \to \mathbb{R}$. This can be achieved by replacing the simple queue with a priority queue. Since this is a min-queue, a dequeue operation will return the element with the lowest priority. We will refer to a member of the resulting family of traversal algorithms as *priority search* or *priority traversal*.

PRIORITY TRAVERSAL

A possible priority function considering both distance and degree could be

$$\rho_{d,deg}(S,u) = w_1 \cdot d(S,u) + w_2 \cdot deg(u) \tag{A.0.5}$$

where w_1 and w_2 are weighting factors to be determined.

Algorithm 10: MultiSourcePrioritySearch search with distance-degree priority

Input: $S \subset V$: a set of source nodes, $\rho(d(S,u), deg(u))$: priority function
Output: O: an ordering of V

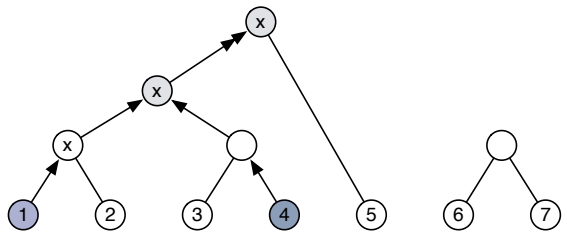
```
1  A ← {}, O ← (), Q ← ()
2  for s in S do
3      Q.enqueue(s, ρ(0, deg(s)))
4      A ← A + s
5  while Q = (u_1, ..., u_l) ≠ () do
6      u ← Q.dequeue()
7      if condition then
8          break
9      else
10         O.append(u)
11         for v in {v ∈ V : ∃{u_1, v} ∈ E} do
12             if v ∉ A then
13                 Q.enqueue(v, ρ(d(S, v), deg(v)))
14                 A ← A + v
```

Appendix B

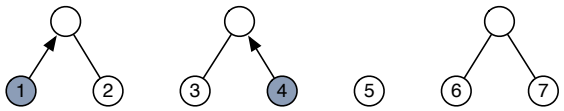
Data Structure for Unions and Backtracking

The *backtrack strategy* for greedy agglomerative algorithms requires a data structure which keeps a history of the mergers performed and allows the algorithm to reverse selected mergers. Short of implementing a dynamic *union-find-backtrack* data structure, we chose a simplified version. In a *union-find-backtrack* data structure, the actual nodes of the graph would comprise all the nodes of the union forest. Making such a structure fully dynamic, i.e. allowing for node deletions and node insertions into specific clusters turned out to be quite complex. Our simplified data structure maintains a binary forest instead where only the leaf nodes L contain the actual elements while all other tree nodes U are internal *union nodes*. In contrast to a proper *union-find* data structure, the elements themselves are not used as representatives of their set. This simplicity is bought by needing twice as many nodes in the union forest as a *union-find* structure (although a fully dynamic one would require a memory overhead, too). The public operations union, backtrack (Algorithm 12), separate (Algorithm 11) and isolate (Algorithm 15) are provided, while find (Algorithm 14) and split (Algorithm 13) are private operations. Finally, getPartition returns the partition of the element set induced by the forest. Since the structure is a binary forest, find, backtrack and separate operation run in $\Theta(\log n)$.

Figure B.1 illustrates the state of an example instance before and after the operation separate(1,4). The operation starts from 1 and follows the parent pointers until it reaches a root, marking each union node on the path. Then the path from 4 to the root is traversed and any union node that is marked is deleted.



(a) before



(b) after

Figure B.1: separate$(1, 4)$

Algorithm 11: separate

Input: $l, m \in L$

```
1  u ← parent(l)
2  repeat
3  |  mark u
4  |  u ← parent(u)
5  until u is a root
6  u ← parent(m)
7  repeat
8  |  if u is marked then
9  |  |  split(u)
10 |  u ← parent(u)
11 until u is a root
```

Algorithm 12: backtrack

Input: $l \in L$

```
1  r ← find(l)
2  if r ≠ l then
3  |  split(r)
```

Algorithm 13: split

Input: $u \in U$

```
1 makeRoot(leftChild(u))
2 makeRoot(rightChild(u))
```

Algorithm 14: find

Input: $s \in T$

Output: $t \in T$, root of t or t if t is a singleton

```
1 t ← s  while t is not a root do
2 |  t ← parent(t)
3 return t
```

Algorithm 15: isolate

Input: $l \in L$

```
1 while l is not a root do
2 |  split(find(l))
```

Appendix C

Result Plots and Tables

C.1 Graph Properties

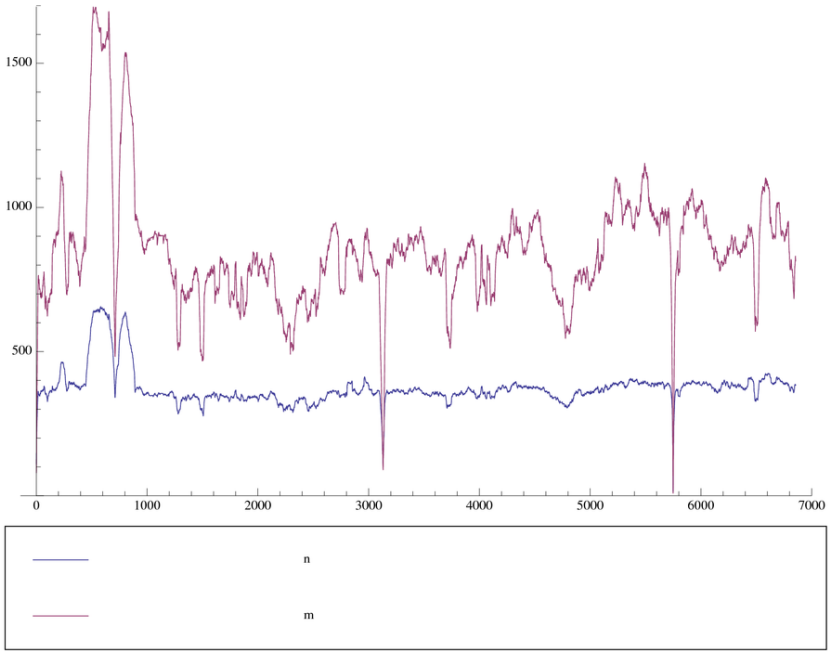


Figure C.1: Node and edge count of the e-mail graph $\mathcal{G}_e$, sampled at 100 time steps

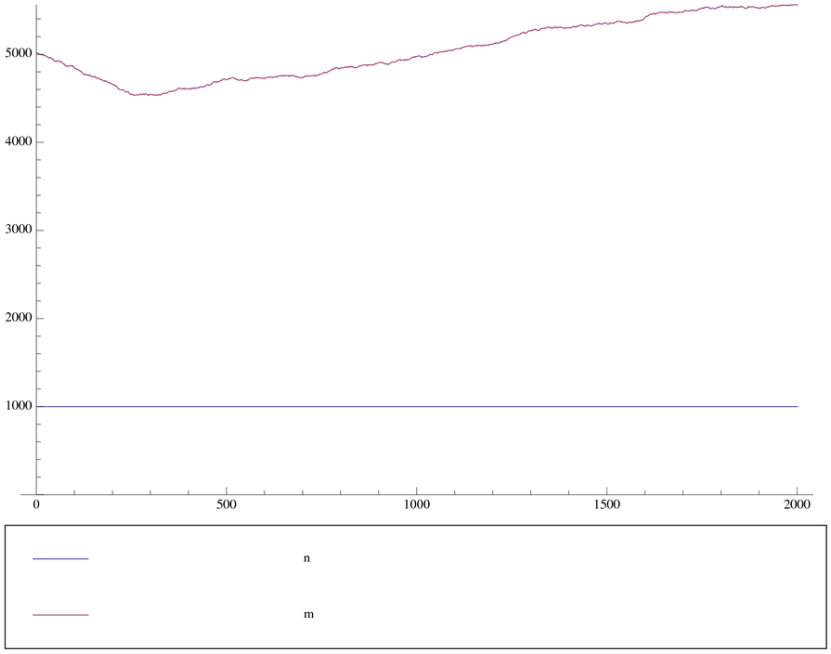


Figure C.2: Node and edge count of a generated dynamic graph

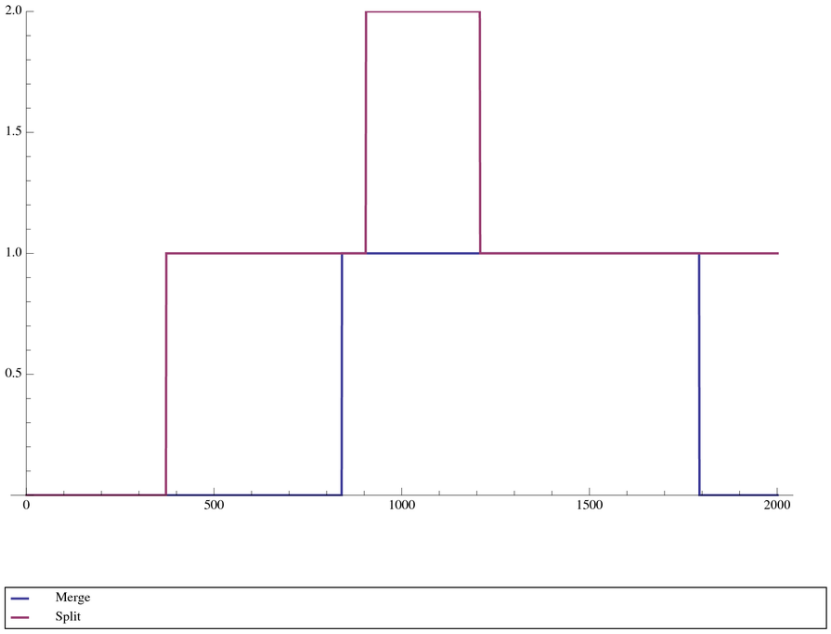


Figure C.3: Cluster events of the graph above

C.2 Measures

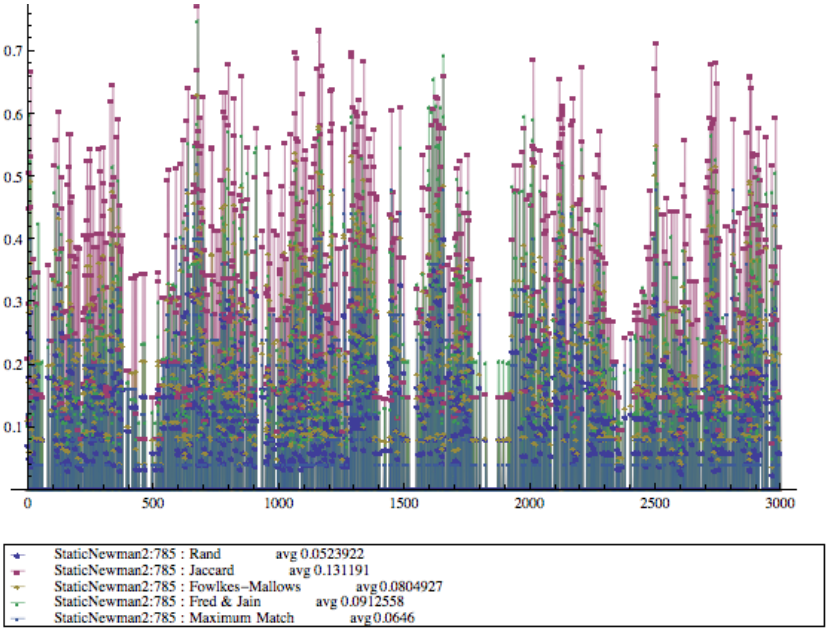


Figure C.4: Raw data for several distance measures

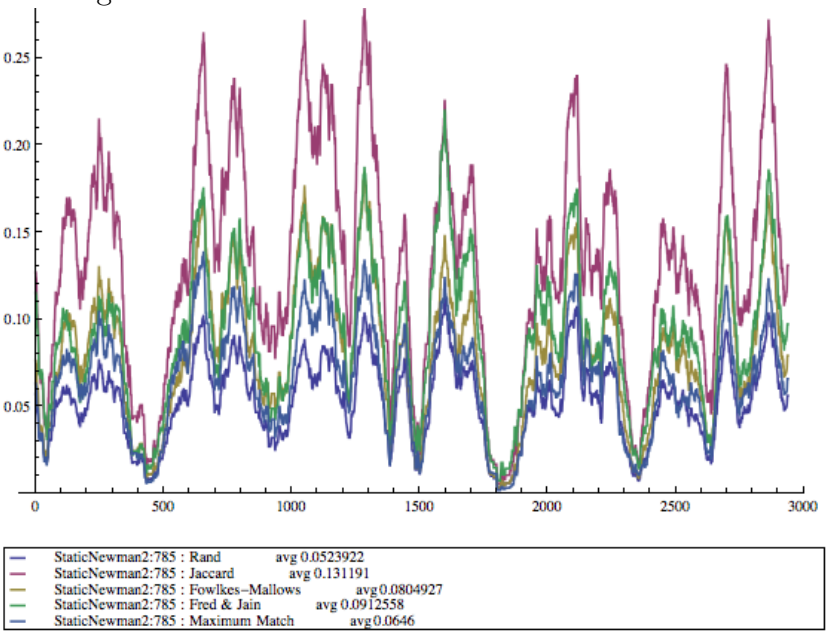


Figure C.5: Smoothed data corresponding to C.4

C.3 Evaluation Results

C.3.1 General Results

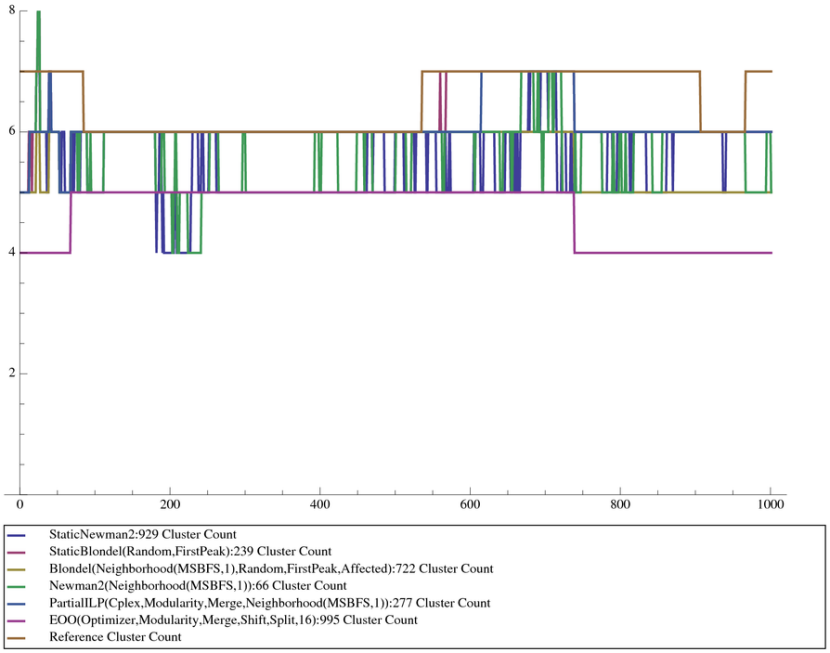


Figure C.6: Cluster counts for different algorithms

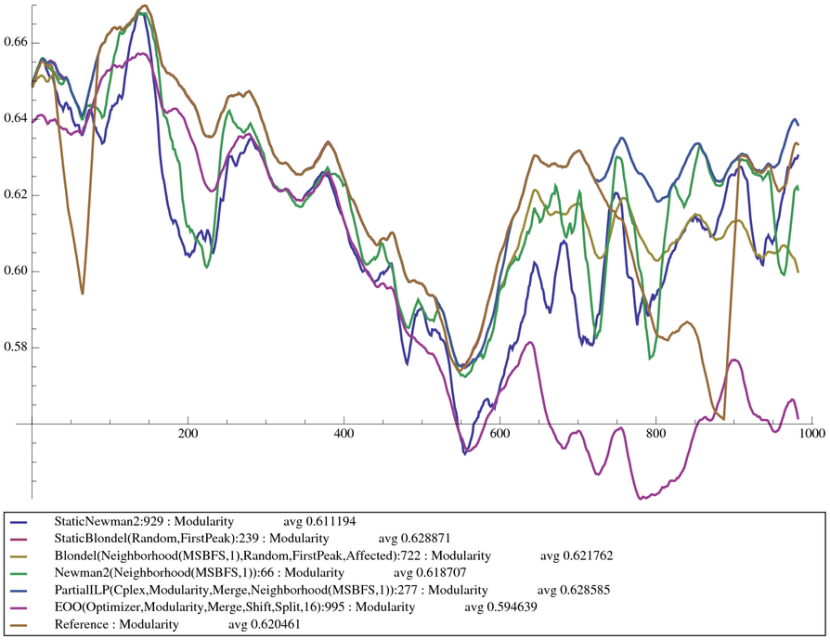


Figure C.7: Quality for all candidates in comparison

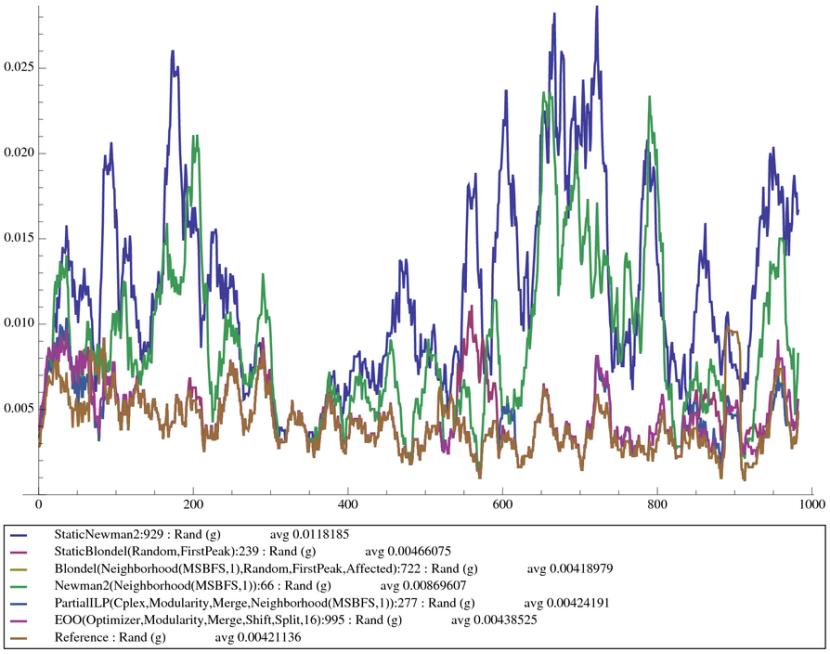


Figure C.8: Statics cluster less smoothly than dynamics.

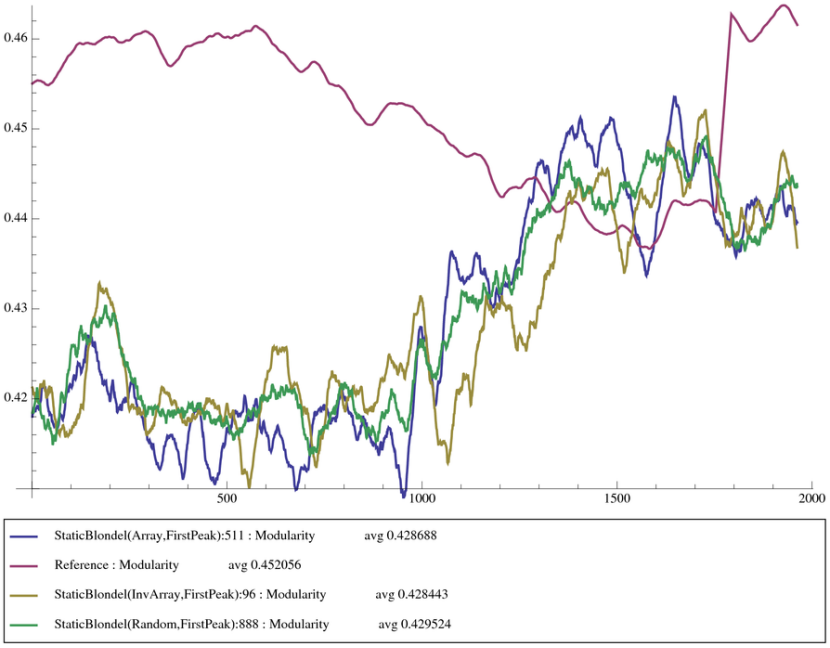


Figure C.9: Node order does not influence the overall quality of sLocal

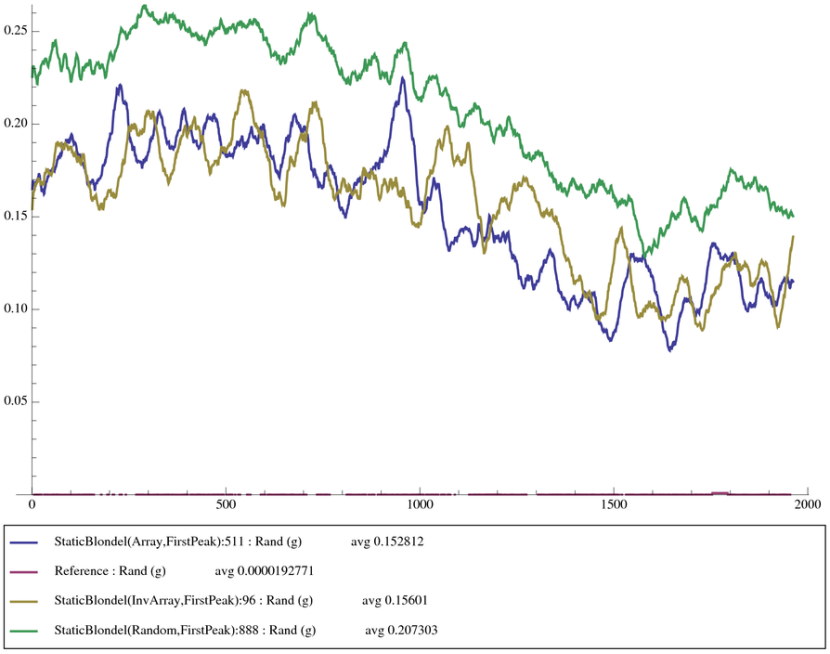


Figure C.10: sLocal clusters less smoothly with Random node order

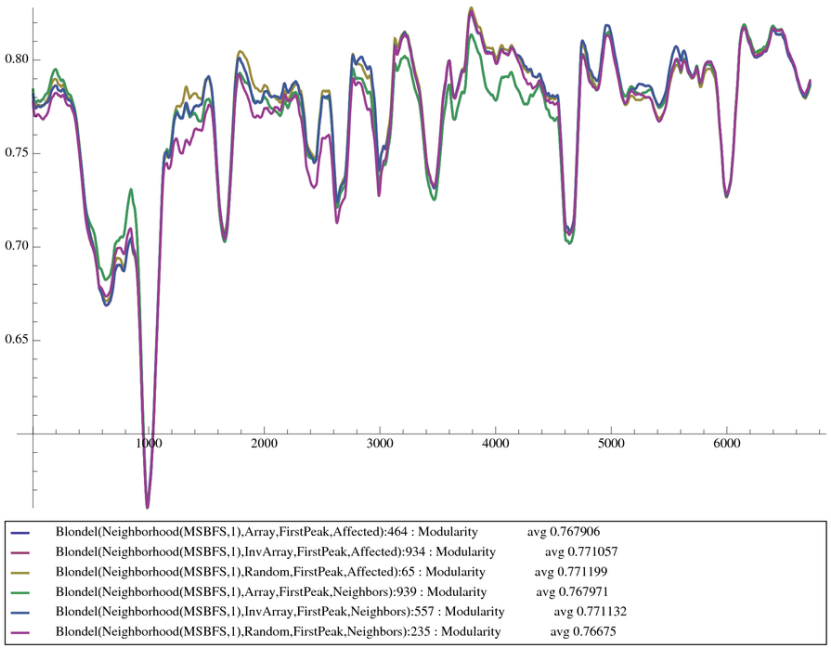


Figure C.11: Quality for dLocal with different node orders and update policies

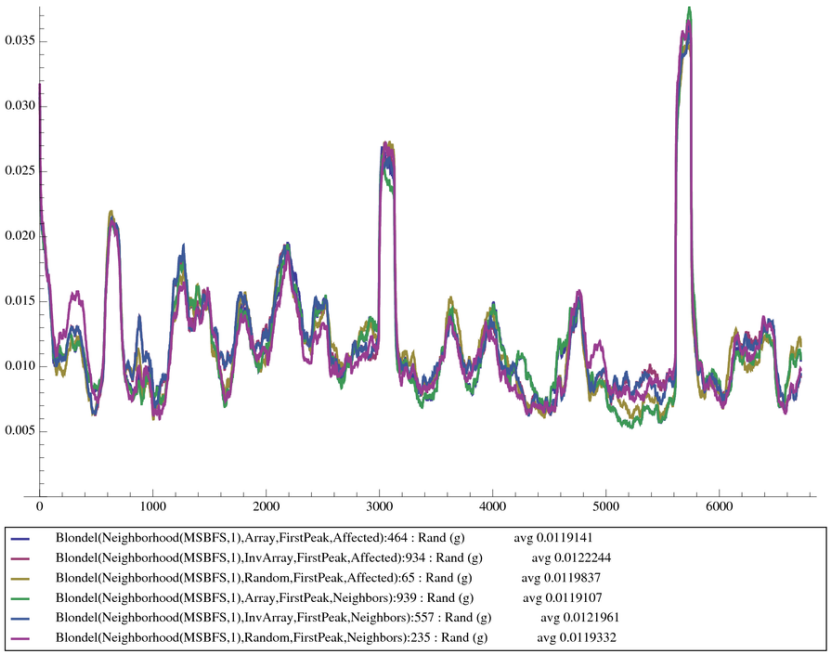


Figure C.12: Distance for dLocal with different node orders and update policies

C.3.2 Static Heuristics

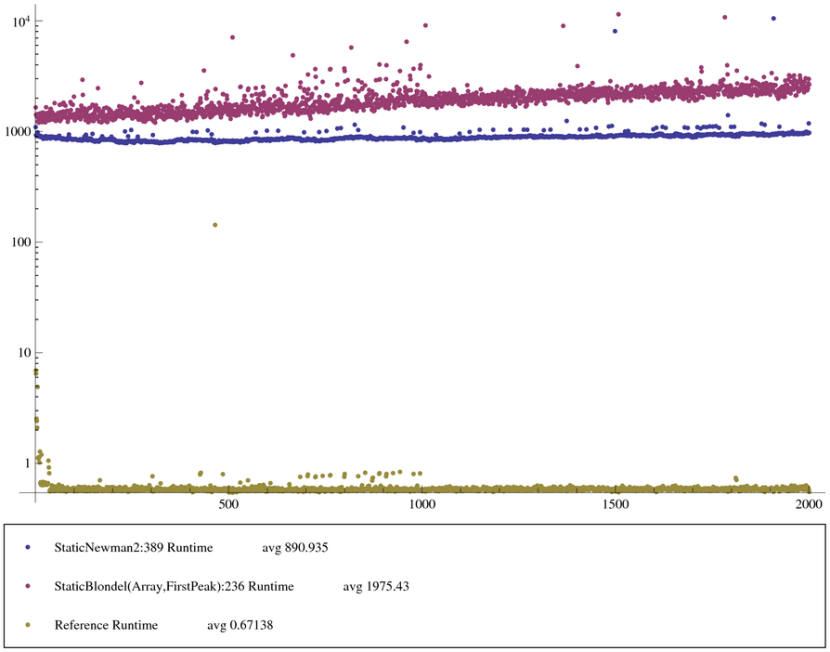


Figure C.13: Runtime for sGlobal and sLocal

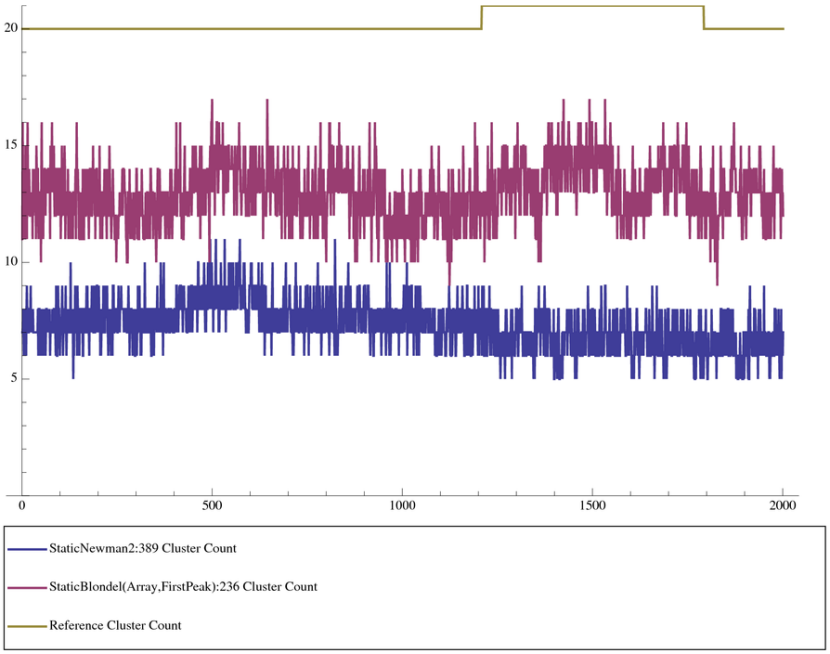


Figure C.14: sGlobal produces less clusters than sLocal

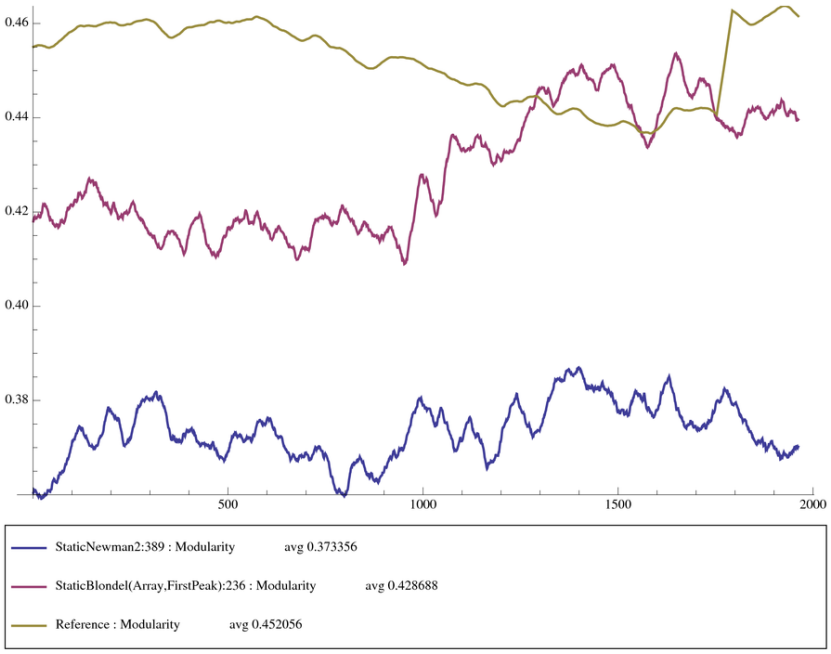


Figure C.15: sLocal is superior to sGlobal in terms of quality

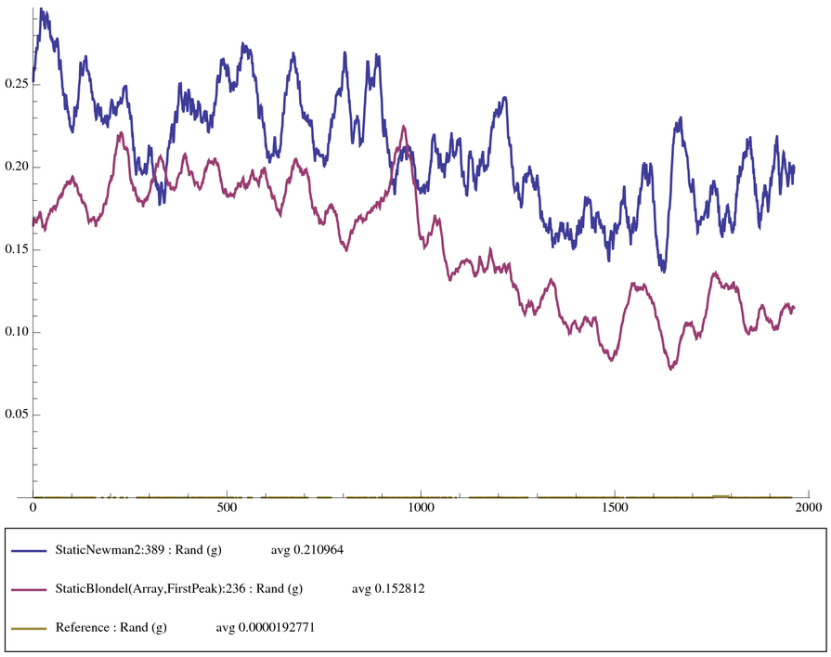


Figure C.16: sLocal is superior to sGlobal in terms of smoothness

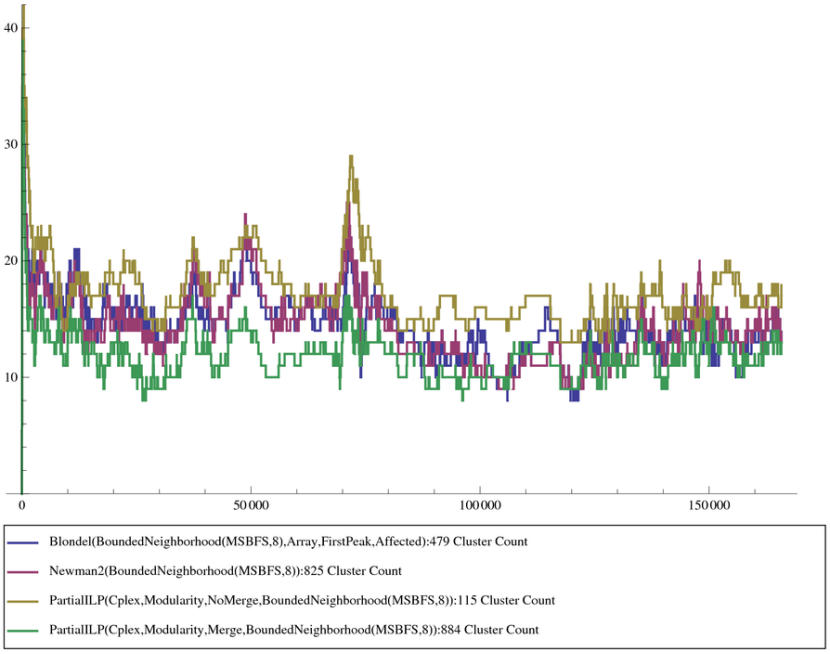


Figure C.17: Cluster count for dLocal, dGlobal, pILP(M) and pILP(NM)

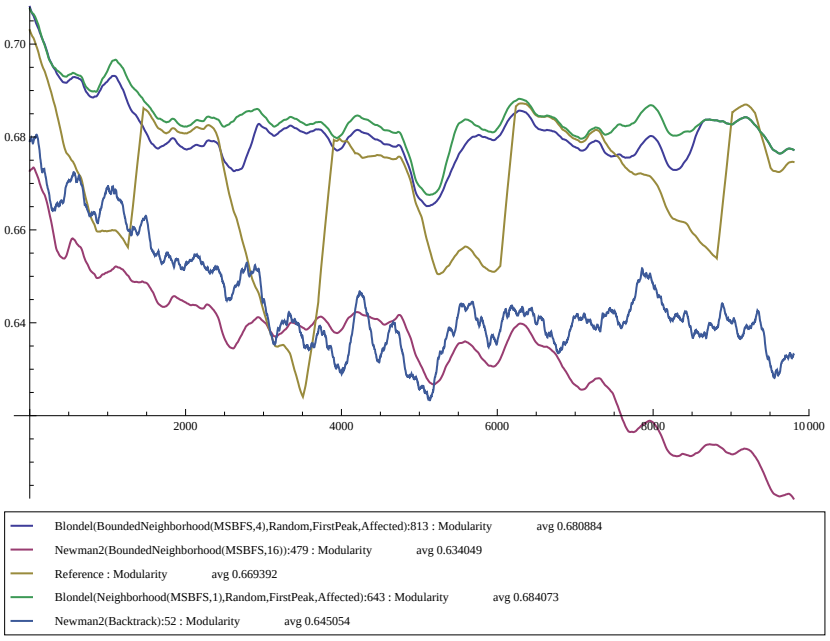


Figure C.18: Quality: Dynamics adapt quickly to cluster events

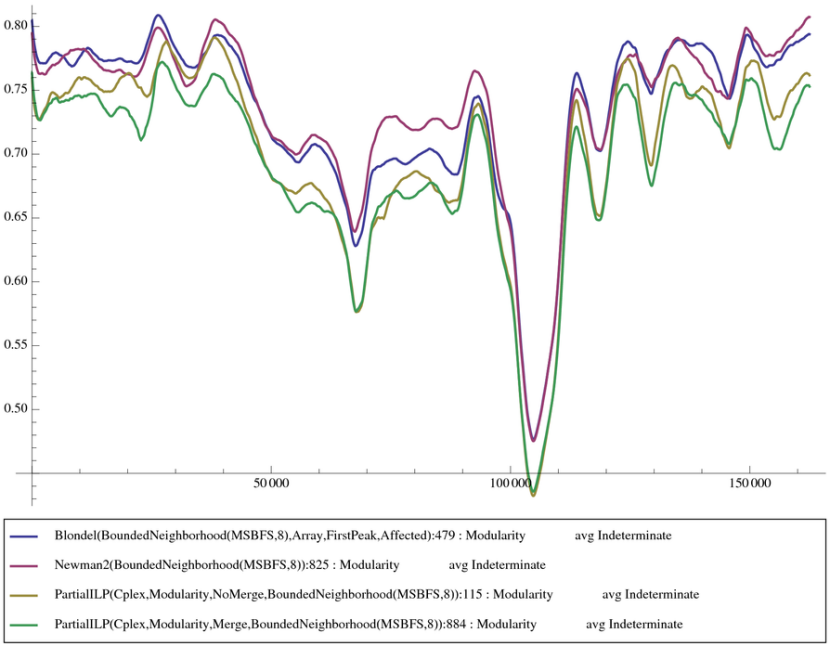


Figure C.19: Quality: Heuristics are superior to pILP

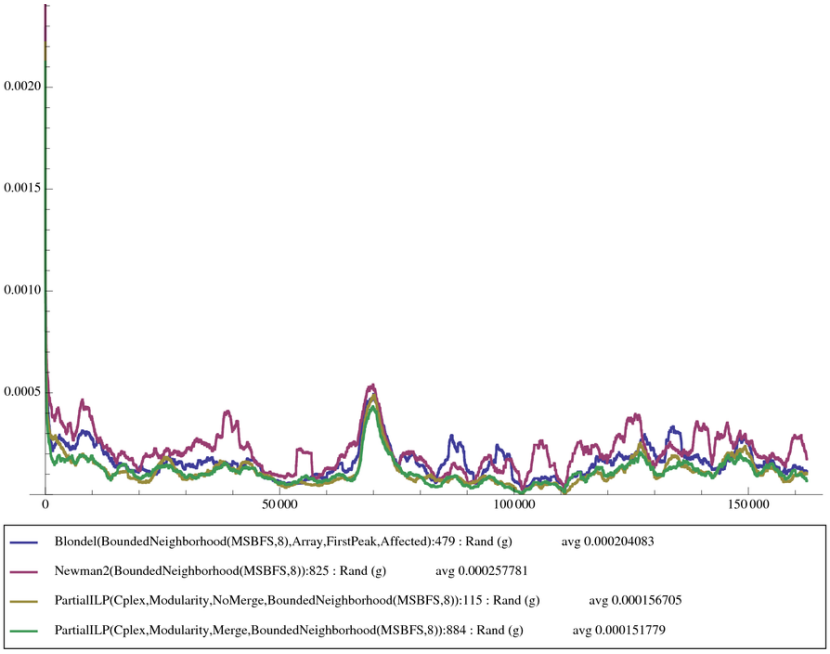


Figure C.20: Distance: Differences between heuristics and pILP are neglibible

C.3.3 Prep Strategies

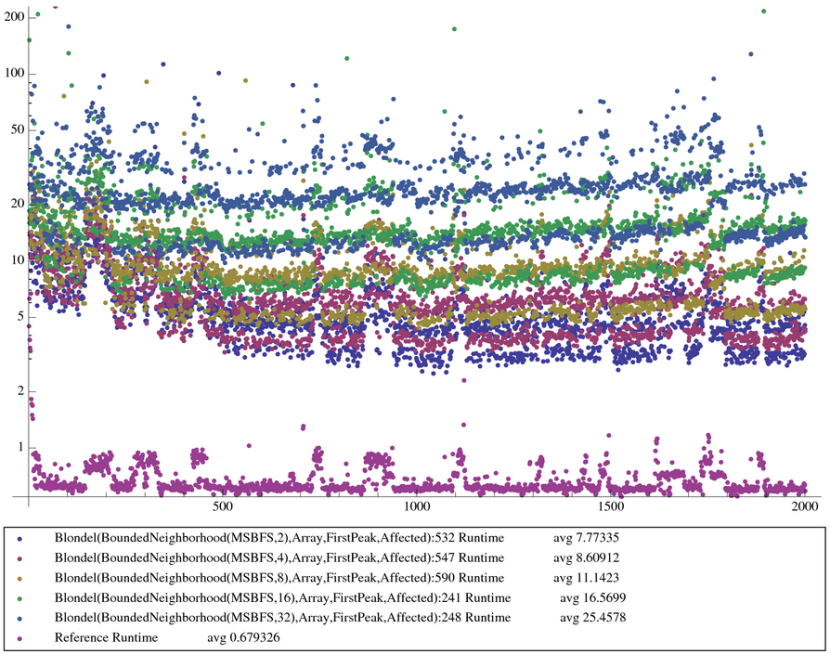


Figure C.21: Runtime for dLocal with BN strategy

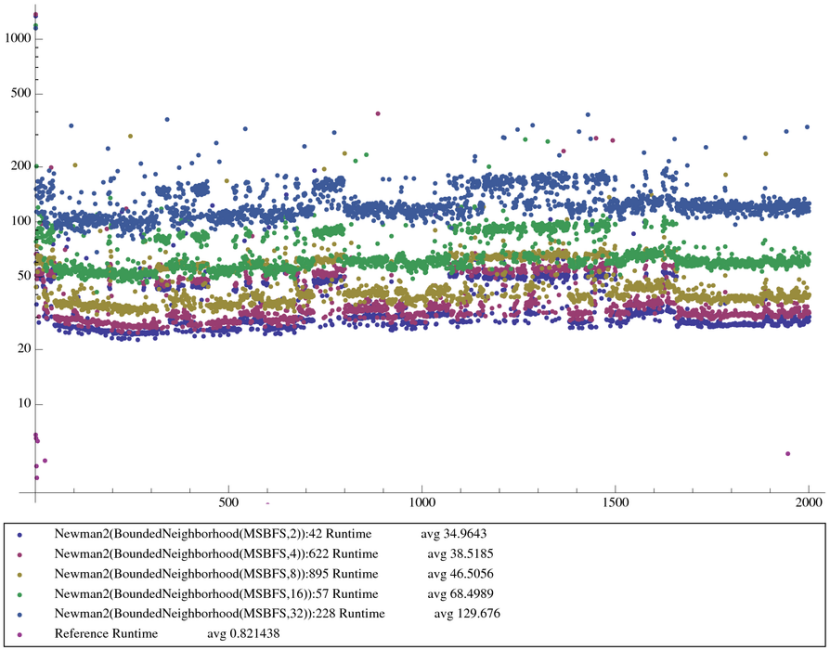


Figure C.22: Runtime for dGlobal with BN strategy

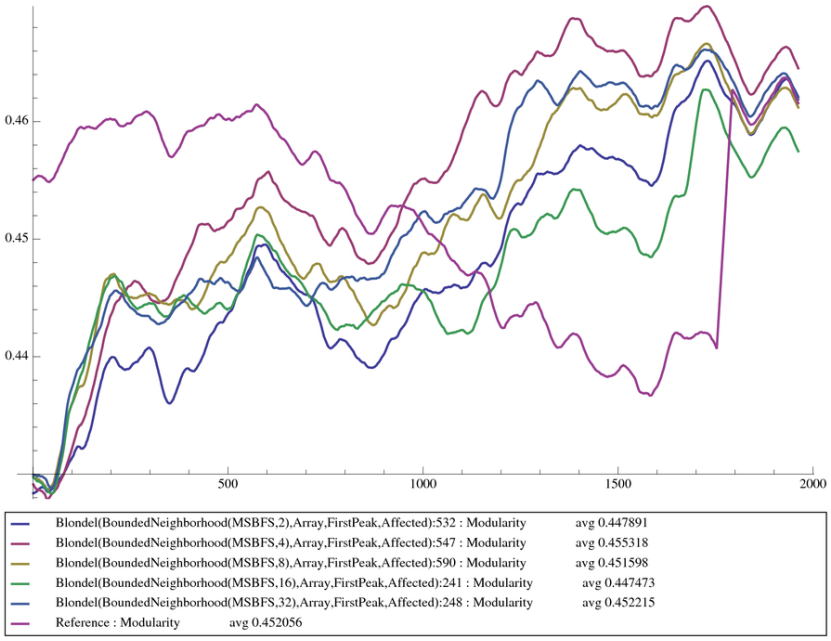


Figure C.23: Quality for dLocal with BN strategy

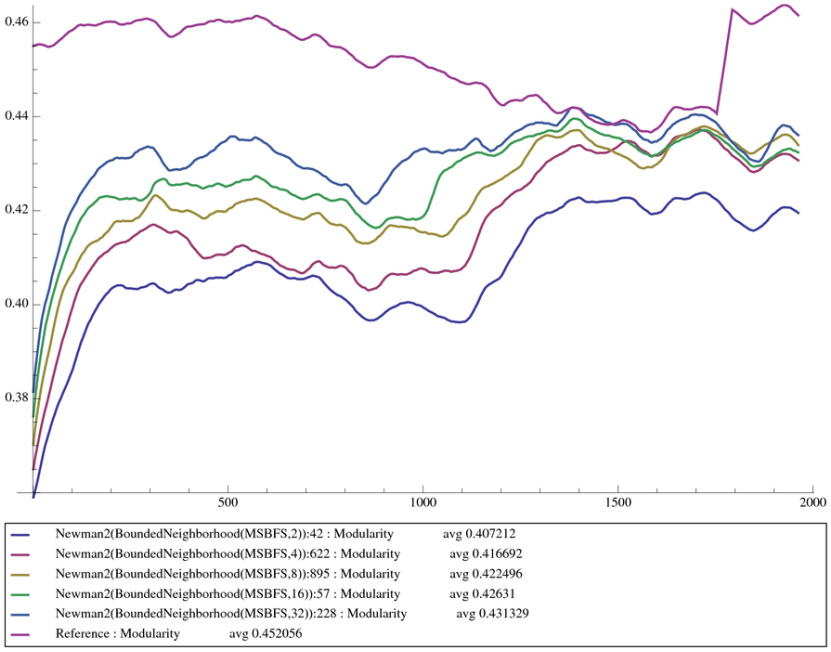


Figure C.24: Quality for dGlobal with BN strategy

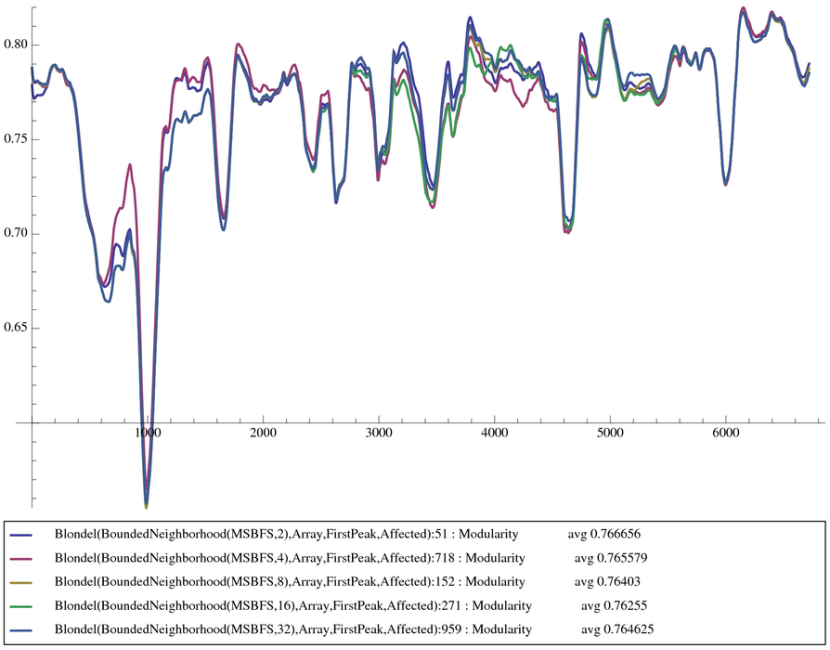


Figure C.25: Quality for dLocal with BN strategy

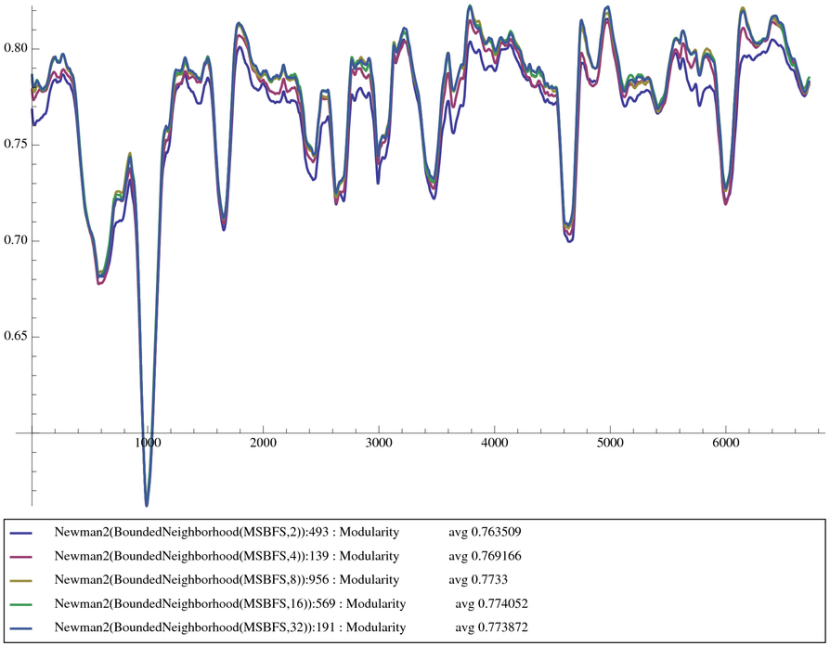


Figure C.26: Quality for dGlobal with BN strategy

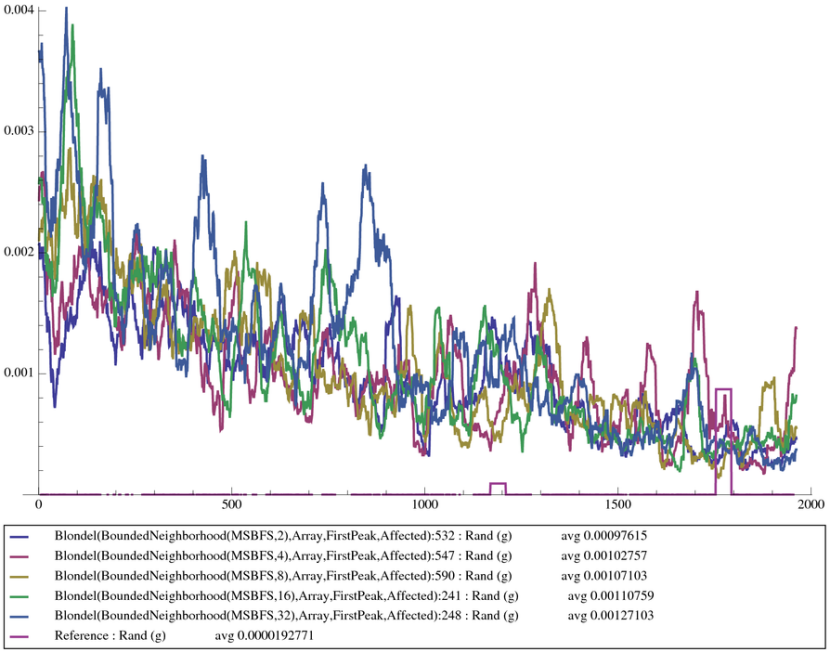


Figure C.27: Distance for dLocal with BN strategy

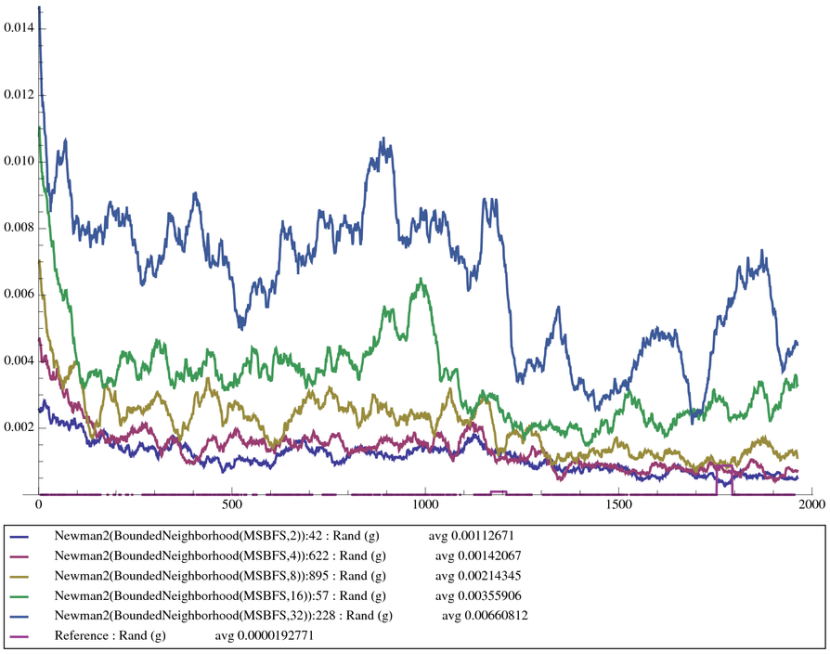


Figure C.28: Distance for dGlobal with BN strategy

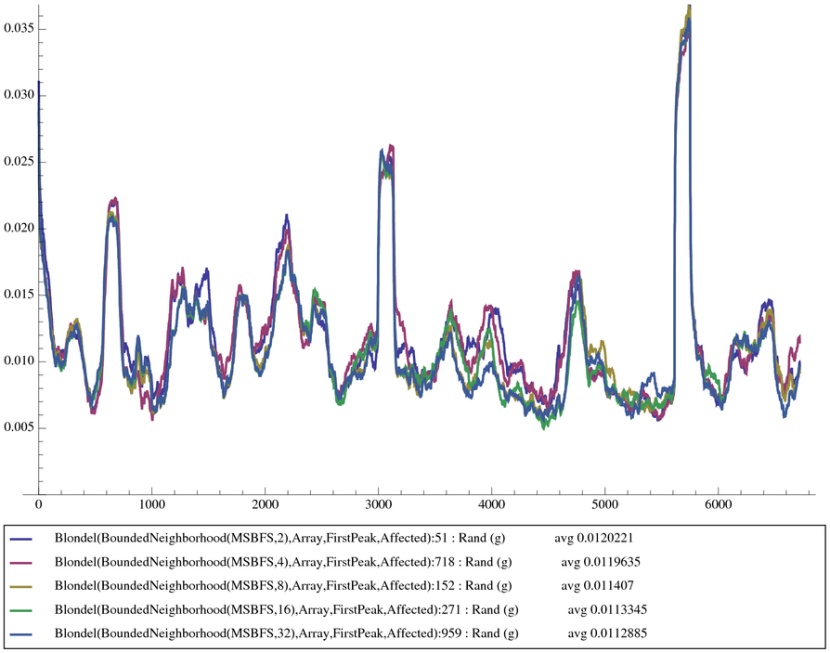


Figure C.29: Distance for dLocal with BN strategy

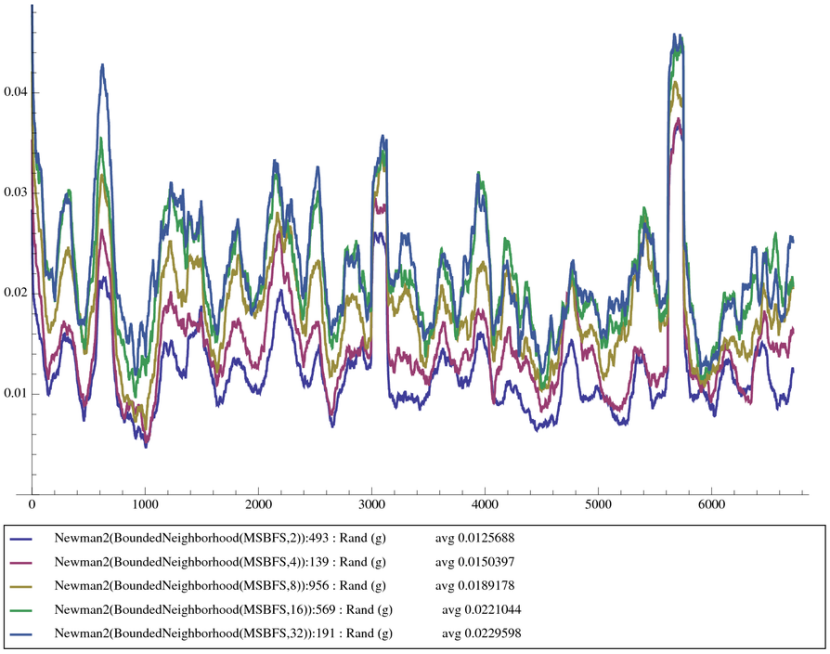


Figure C.30: Distance for dGlobal with BN strategy

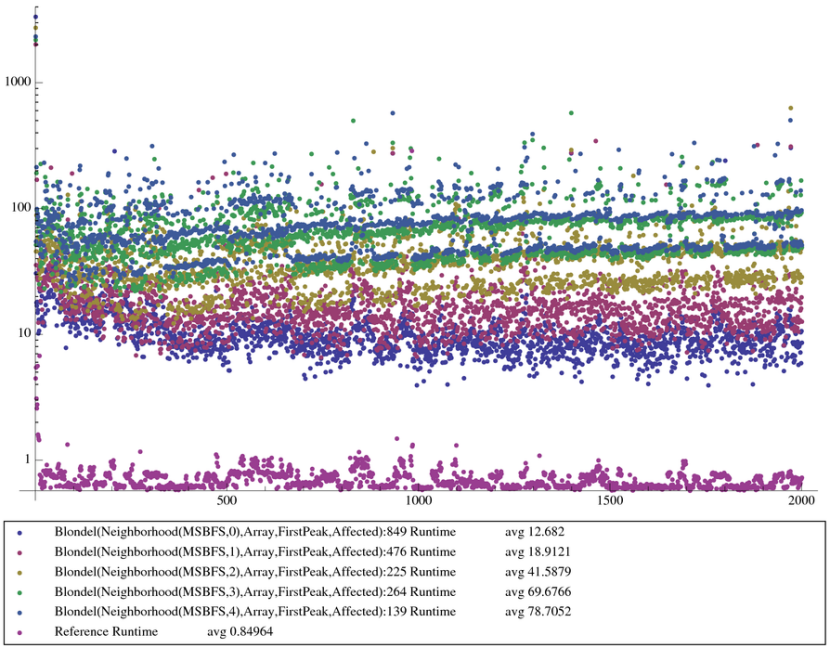


Figure C.31: Runtime for dLocal with N strategy

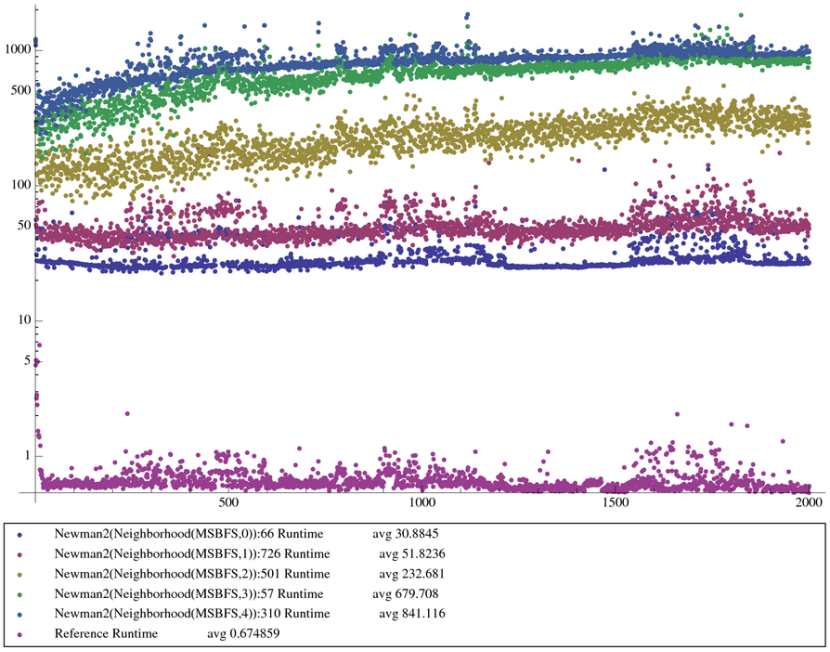


Figure C.32: Runtime for dGlobal with N strategy

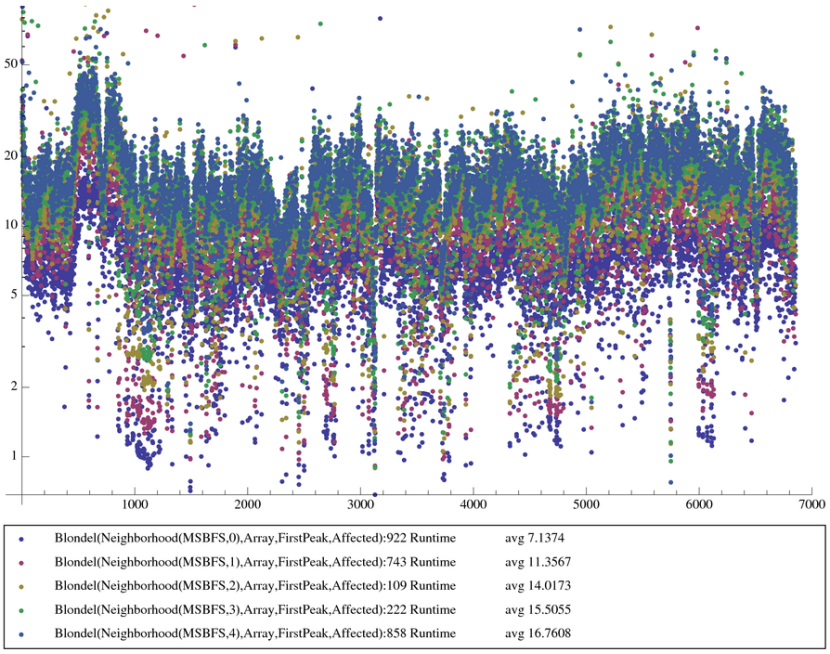


Figure C.33: Runtime for dLocal with N strategy

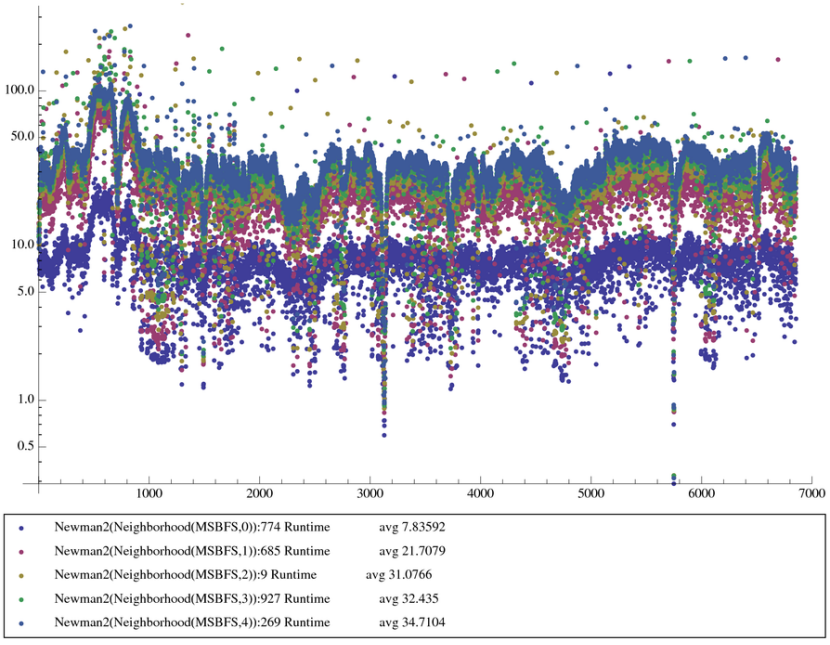


Figure C.34: Runtime for dGlobal with N strategy

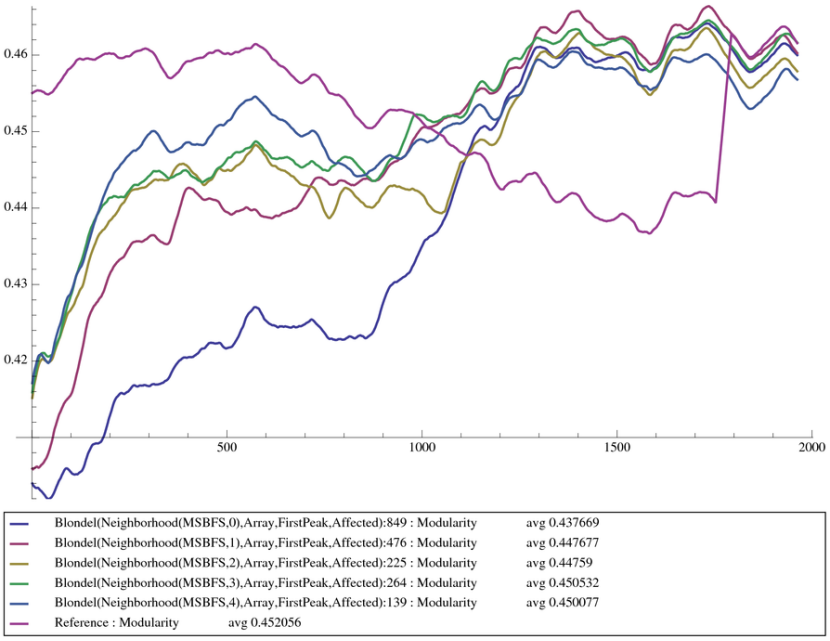


Figure C.35: Quality for dLocal with N strategy

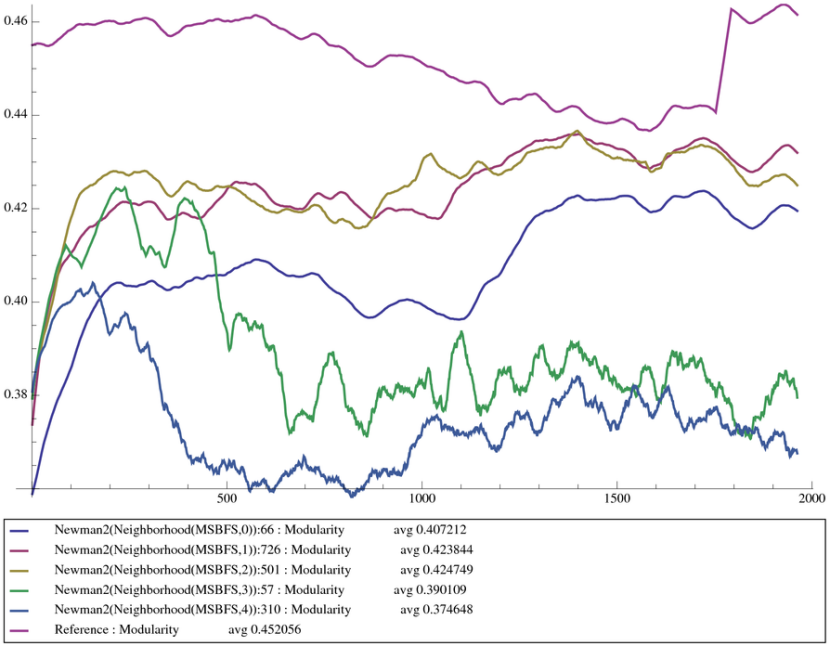


Figure C.36: Quality for dGlobal with N strategy

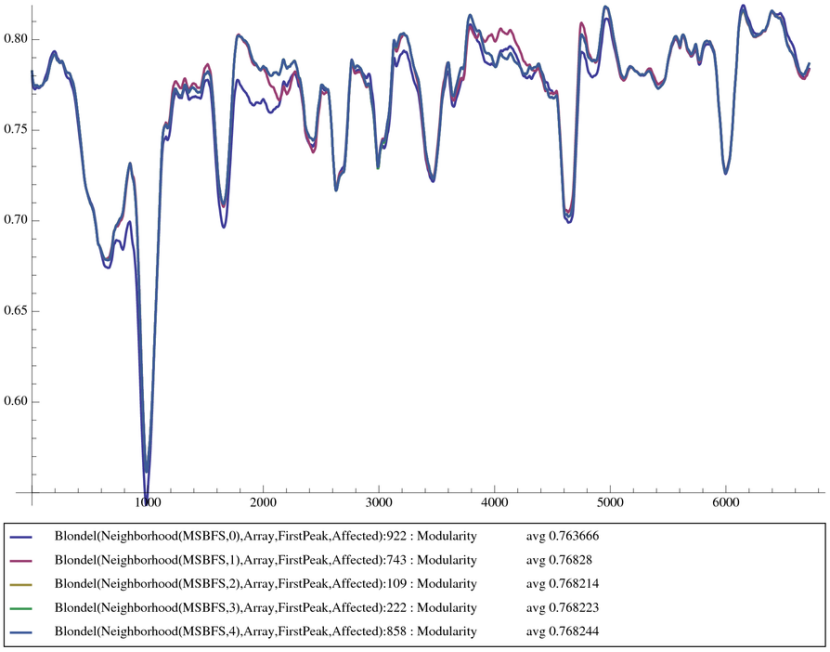


Figure C.37: Quality for dLocal with N strategy

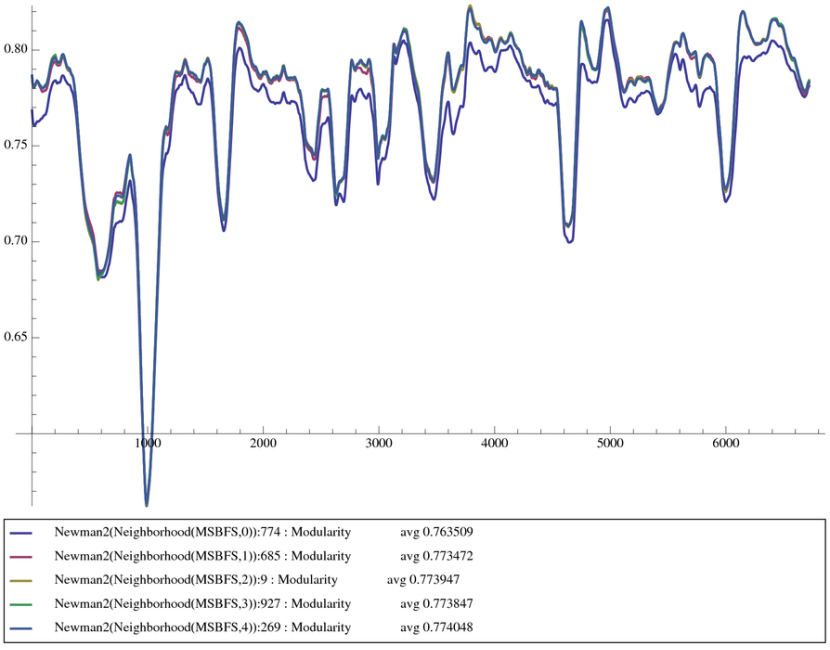


Figure C.38: Quality for dGlobal with N strategy

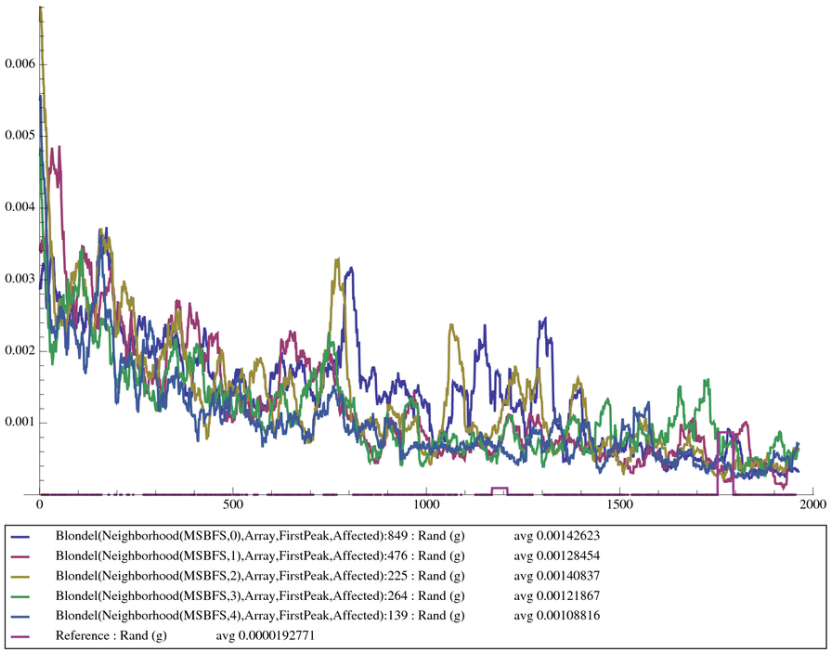


Figure C.39: Distance for dLocal with N strategy

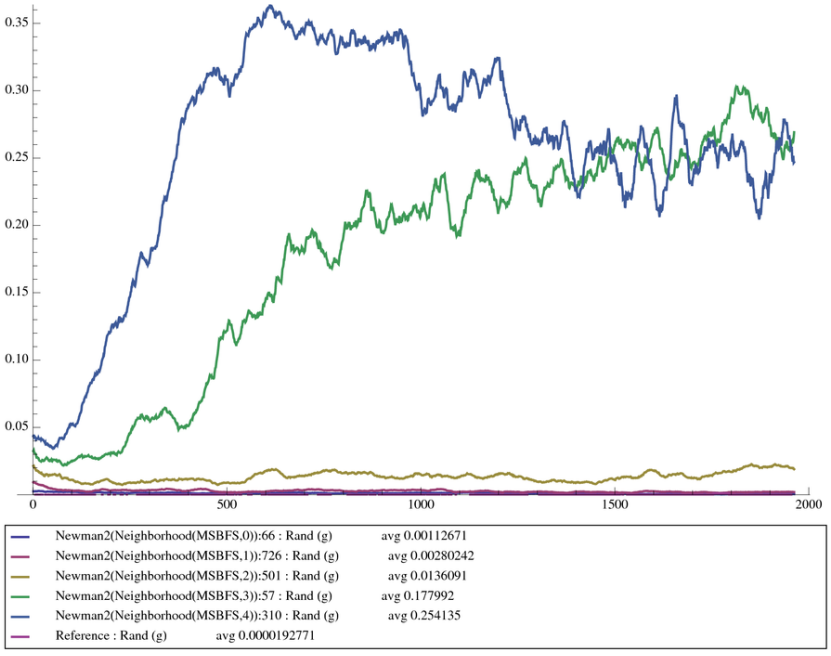


Figure C.40: Distance for dGlobal with N strategy

Prep strategies for dGlobal

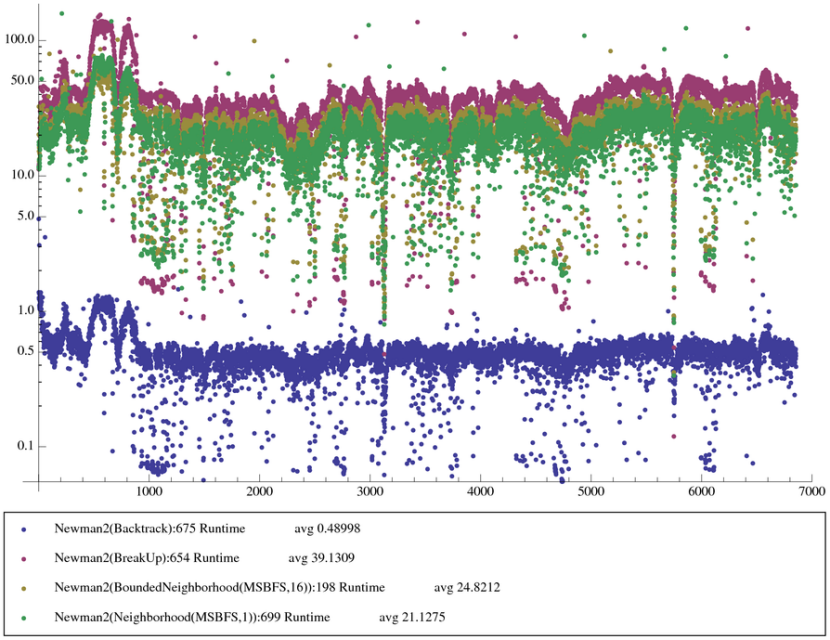


Figure C.41: Runtime for dGlobal with different *prep strategies*

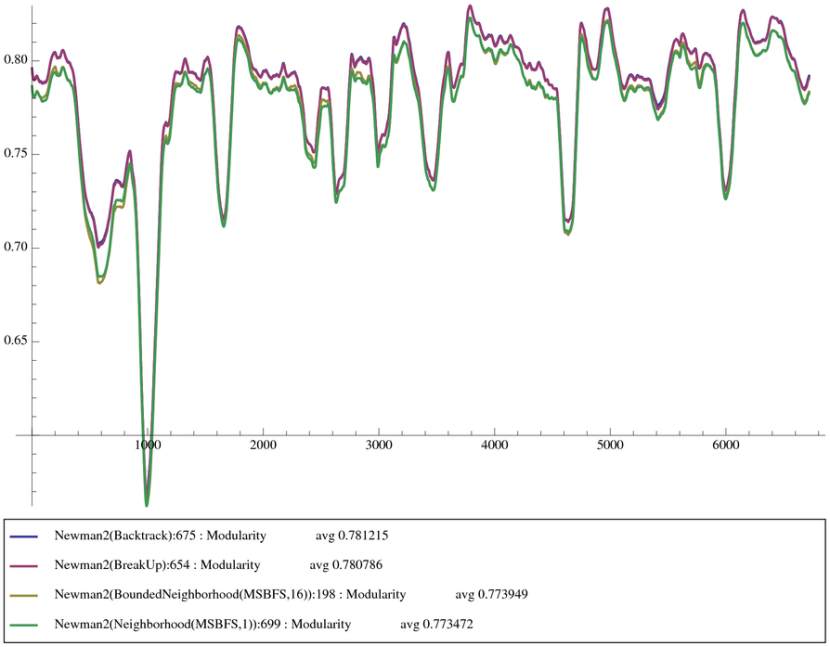


Figure C.42: Quality for dGlobal with different *prep strategies*

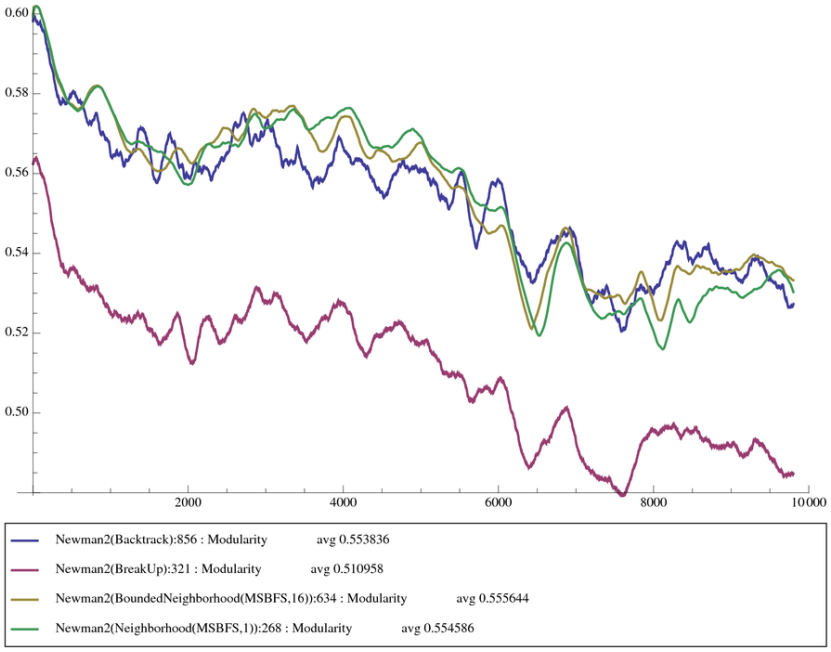


Figure C.43: Quality for dGlobal with different *prep strategies*

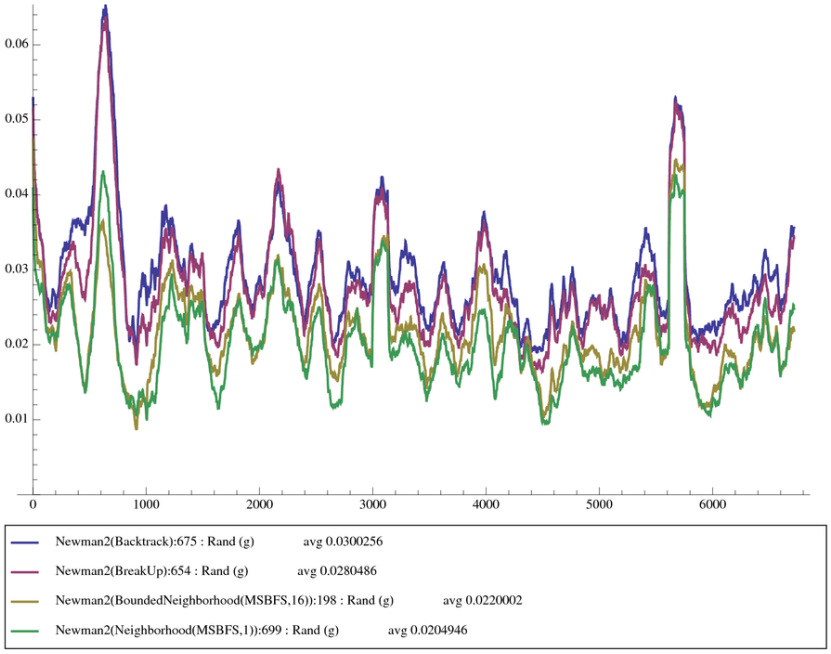


Figure C.44: Distance for dGlobal with different *prep strategies*

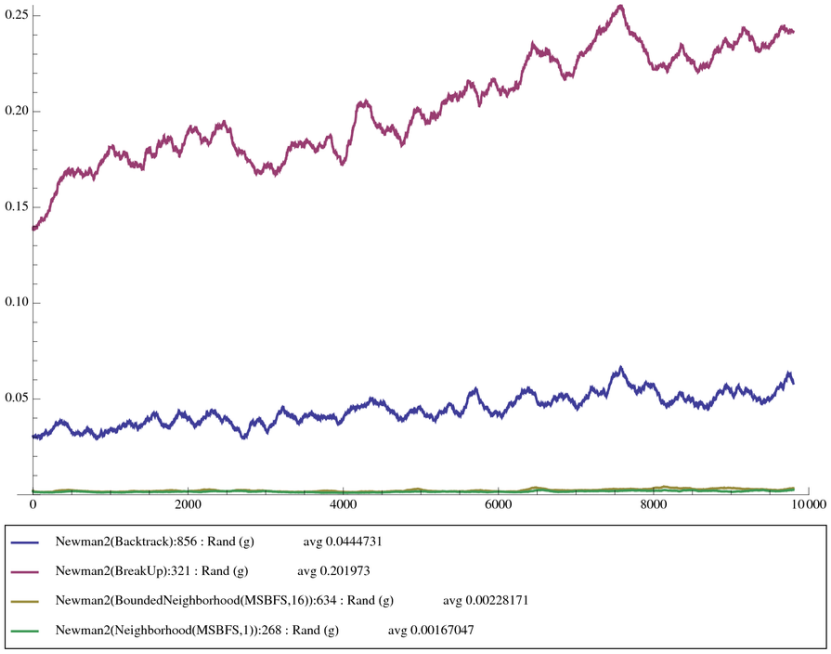


Figure C.45: Distance for dGlobal with different *prep strategies*

List of Tables

List of Figures

List of Algorithms

Bibliography

[BGLL08] Vincent Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of community hierarchies in large networks. ARXIV: 2008arXiv0803.0476B, March 2008.

[CNM04] Aaron Clauset, Mark E. J. Newman, and Cristopher Moore. Finding community structure in very large networks. *Physical Review E*, 70(066111), 2004.

[Del06] Daniel Delling. Analyse und Evaluierung von Vergleichsmaßen für Graphclusterungen. Diplomarbeit, Universität Karlsruhe (TH), Fakultät für Informatik, February 2006.

[DGGW07] Daniel Delling, Marco Gaertler, Robert Görke, and Dorothea Wagner. Engineering Comparators for Graph Clusterings. In *Proceedings of the European Conference of Complex Systems (ECCS'07)*, October 2007. As poster.

[DGGW08] Daniel Delling, Marco Gaertler, Robert Görke, and Dorothea Wagner. Engineering Comparators for Graph Clusterings. In *Proceedings of the 4th International Conference on Algorithmic Aspects in Information and Management (AAIM'08)*, volume 5034 of *Lecture Notes in Computer Science*, pages 131–142. Springer, June 2008.

[Gae05] 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.

[GHJ+] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Q.R.H. Montreal, and I.R.J. Urbana. Back DESIGN PATTERNS-Elements of Reusable Object-Oriented Software.

[GHWG09] Robert Görke, Florian Hübner, Dorothea Wagner, and Marco Gaertler. Computational Aspects of Significance-Driven Graph Clustering . Submitted to JGAA, 2009.

[GMSW] Robert Görke, Pasca Maillard, Christian Staudt, and Dorothea Wagner. Modularity-driven clustering of dynamic graphs.

[GS09] Robert Görke and Christian Staudt. A Generator of Dynamic Clustered Random Graphs. Development notes, to be submitted to ESA09, 2009.

[Hüb08] Florian Hübner. The Dynamic Graph Clustering Problem - ILP-Based Approaches Balancing Optimality and the Mental Map. Master's thesis, Universität Karlsruhe (TH), May 2008. Informatik Diplomarbeit.

[NR09] Andreas Noack and Randolf Rotta. Multi-level Algorithms for Modularity Clustering. In Jan Vahrenhold, editor, *Proceedings of the 8th International Symposium on Experimental Algorithms (SEA'09)*, volume 5526 of *Lecture Notes in Computer Science*, pages 257–268. Springer, June 2009.