Insert here your thesis’ task.
Bachelor’s thesis

Implementation of the Chameleon Clustering Algorithm

Tomáš Brůna

Supervisor: Ing. Tomáš Bartoň

10th May 2015
I would like to thank my supervisor, Ing. Tomáš Bartoň, for his patience, support and advice throughout this project.
Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the “Work”), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on 10th May 2015
Czech Technical University in Prague
Faculty of Information Technology
© 2015 Tomáš Brůna. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic.
It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author’s permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Cílem této práce je analýza a implementace shlukovacího algoritmu Chameleon.

Chameleon je hierarchický shlukovací algoritmus navržený Vipinem Kumesem, Georgem Karypisem a Eui-Hong (Sam) Hanem. Algoritmus řeší problémy a nedostatky stávajících shlukovacích metod použitím dynamického modelování pro určování podobnosti mezi shluky.

Celý algoritmus se nám podařilo úspěšně implementovat a integrovat jej do programu Clueminer – interaktivní platformy pro shlukování. Analýza a výsledky dosažené naší implementaci prokázaly, že Chameleon skutečně dosahuje lepších výsledků než většina dostupných shlukovacích algoritmů a je schopen identifikovat shluky různých velikostí, tvarů a hustot.

**Klíčová slova** Chameleon, shlukování, hierarchické shlukování, Clueminer, strojové učení, rozdělování grafů
Abstract

The goal of this thesis is to analyze and implement the Chameleon algorithm.

Chameleon is a hierarchical clustering algorithm proposed by Vipin Kumar, George Karypis and Eui-Hong (Sam) Han. It overcomes limitations of existing clustering methods by using dynamic modeling to determine the similarity between pairs of clusters.

We successfully implemented the algorithm and integrated it into the Clueminer clustering platform. The analysis and results achieved by our implementation proved that the algorithm is in fact superior to most clustering algorithms and is able to identify clusters of different sizes, shapes and densities.

Keywords  Chameleon, clustering, hierarchical clustering, Clueminer, machine learning, graph partitioning
Introduction

1 Theoretical background
  1.1 Data mining ............................................. 3
  1.2 Machine learning ..................................... 3
  1.3 Clustering .............................................. 4
  1.4 Examples of graph and hierarchical clustering algorithms ..... 6
  1.5 Clueminer .............................................. 10

2 Chameleon analysis
  2.1 Description ............................................ 11
  2.2 Graph construction ................................... 12
  2.3 Graph partitioning .................................. 12
  2.4 Merging ................................................. 19

3 Chameleon implementation
  3.1 Chameleon module ................................... 25
  3.2 Graph interface ....................................... 25
  3.3 k-NN .................................................... 27
  3.4 Partitioning ........................................... 27
  3.5 Merging ................................................ 28
  3.6 Chameleon class ...................................... 29
  3.7 Algorithm parameters ................................ 30
  3.8 Clueminer integration ................................ 31
  3.9 Time complexity ..................................... 32

4 Results and comparisons
  4.1 Comparison of bisection algorithms ..................... 35
  4.2 Comparison of cluster similarity measures ............... 37
  4.3 Results ................................................. 37
List of Figures

1.1 Diamonds dataset ........................................... 7
1.2 The limitation of interconnectivity based algorithms .......... 9

2.1 The overview of the Chameleon approach ....................... 12
2.2 k-NN algorithm ............................................... 13
2.3 The limitation of equal-sized bisection ......................... 15
2.4 The problem with weighted edges in K-L bisection .......... 16
2.5 Iterative Fiduccia-Mattheyses ............................... 18
2.6 The importance of relative cluster properties ............... 21
2.7 User-specified threshold merging strategy .................... 22

3.1 Module diagram ............................................... 26
3.2 Class diagram ................................................ 26
3.3 Parts of Chameleon graphical user interface .................. 33

4.1 Comparison of bisection algorithms ......................... 35
4.2 Bisection speed comparison ................................ 36
4.3 Result improvement during eleven F-M iterations ............ 36
4.4 Comparison of standard and improved similarity measures .. 38
4.5 Overall results .............................................. 40
4.6 Comparison with the reference ................................ 41

B.1 Iris scatterplot matrix ....................................... 53
B.2 Ecoli scatterplot matrix ...................................... 54
B.3 Thyroid disease scatterplot matrix ............................ 54
B.4 Dendrogram of a clustering on dataset with 32 dimensions .... 55
Introduction

Clustering is an important machine learning problem with a variety of uses ranging from medicine and biology to marketing and document categorisation. Most of the current clustering algorithms are designed for specific datasets and fail when clusters do not follow the expected model.

Chameleon, hierarchical clustering algorithm proposed by V. Kumar, G. Karypis and E.-H. Han [1] aims to overcome this limitation by combining proximity and interconnectivity of items and also by taking into account internal properties of clusters during hierarchical merging.

The goal of this thesis is to analyze Chameleon and identify where its key features lie. Next, we will implement the algorithm and integrate it into the Clueminr clustering platform.

In [1], Chameleon is described only theoretically and the existence of a partial implementation in the CLUTO software package is mentioned. Apart from being closed-source, CLUTOS’s implementation is in fact incomplete and contains only half of the process which makes up the Chameleon algorithm. That is why, in order to use the algorithm and analyze its result, our own implementation is necessary.

Structure of this thesis

In chapter 1, we introduce the reader to clustering, clustering algorithms and the limitations of commonly used clustering methods.

In chapter 2, we introduce the Chameleon algorithm and analyze its key parts.

In chapter 3, we describe how we implemented the algorithm and how it is integrated into the Clueminr clustering platform.

In chapter 4, we present the result achieved by our implementation and compare the performance of different methods used throughout the algorithm.

In the last chapter, we consider which parts of the process could be sped up by parallelization.
Chapter 1

Theoretical background

1.1 Data mining

Data mining, sometimes called knowledge discovery, is the analysis of large datasets in order to extract a non-trivial and useful information [2]. In [3], Bissantz and Hagedorn provide general characteristics of various data-mining methods which are:

- Users will receive the required information in an easy-to-understand form (text or a chart).
- Measures are given to evaluate the reliability and certainty of the extracted knowledge.
- The generated information is not trivial compared to the existing knowledge.
- The run time of the underlying algorithm is acceptable and should not exceed that of a low-level polynomial.

1.2 Machine learning

Machine learning is very similar to data mining. Both look for patterns in the data, however, instead of extracting data for human comprehension, machine learning uses the data to improve the program’s own understanding. Program’s actions are adjusted according to patterns found in the data [4].

Machine learning can be divided into three significant groups: supervised, reinforcement and unsupervised learning [5].

In supervised machine learning the input data are associated with their desired outputs – labels. The task is to find a deterministic function based on this data which can predict the label of any future input. If the output
1. Theoretical background

is continuous, a regression function is learnt. Otherwise, in case of discrete values, we call the process classification [6].

Reinforcement learning is used when no optimal input-output mapping is available for a given state and the algorithm has to find an action which maximizes the expected reward over time.

In unsupervised learning we are given a set of data without labels and the goal is to find various patterns and connections among the data.

1.3 Clustering

Clustering is one of the most important unsupervised learning problems. The goal is to group similar objects into same clusters and minimise the similarity of objects in different clusters [7]. Clustering is used in many fields, for instance in marketing to find customers with similar buying behaviour; medicine, where clustering can be used to group patients with similar symptoms or in a document categorisation to find documents with related content [8].

Clustering algorithms can be divided into the following groups based on their cluster model:

- hierarchical clustering
- centroid clustering
- density based clustering
- distribution based clustering
- graph clustering

Another way to look at clustering is to distinguish exclusive and fuzzy clustering. In an exclusive clustering, every item belongs to exactly one cluster. In a fuzzy clustering, every object belongs to few clusters with certain degree.

Categories of clustering algorithms are not strictly defined, some algorithms (like Chameleon) use multiple approaches. Since Chameleon is a hierarchical clustering algorithm using a graph representation of the data, we will further describe hierarchical and graph clustering.

1.3.1 Hierarchical clustering

Kumar [9] defines hierarchical clustering as a set of nested clusters organized as a tree. The tree is usually visualized in the form of a dendrogram. Dendrogram shows which clusters were merged at which level. Also, by cutting the dendrogram at a certain level, we obtain a clustering result which assigns items to clusters according to the given dendrogram level.

There are two types of hierarchical clustering: agglomerative and divisive.
Agglomerative clustering algorithms successively merge similar pairs of clusters until all items have been assigned into one group. Usually, initial clusters are formed by individual input items. However, clusters at the beginning can be created by other methods, such as graph partitioning.

Divisive approach starts with one cluster which contains all items. In each step, the cluster is hierarchically divided into smaller groups.

To obtain one single result which divides the data into exclusive clusters, we need to cut the dendrogram at a certain level. It is often hard to determine where to make the best cut as many optimal cuts at different levels can exist for hierarchically structured data.

### 1.3.2 Graph theory

A graph is defined as a pair \( G = (N, E) \), where \( N \) is a set of nodes or vertices and \( E \) is a set of edges [10]. The pair \((u, v)\) denotes an edge between nodes \( u \) and \( v \). If the pair is ordered, the edge is directed, otherwise it is undirected. Graph is called weighted if we assign weight to each edge.

Graphs with its nodes and edges have many properties, here we will provide only a few definitions which are important for graph clustering.

If an edge exists between nodes \( u \) and \( v \), we say that \( v \) is a neighbor of \( u \). The degree of a node corresponds to the number of its neighbors.

A subgraph \( H \) of a graph \( G \) is a graph whose nodes and edges are subsets of nodes and edges from graph \( G \).

A path is a sequence of edges that connects two nodes. Nodes \( u \) and \( v \) are connected if a path exists between them. If a path exists between all nodes in a graph, the graph is connected, otherwise disconnected.

### 1.3.3 Graph clustering

The goal of graph clustering is to find similar groups of vertices in the graph structure. The data are usually represented by unordered weighted graphs. According to Schaeffer [10], graph’s clusters should meet the following criteria:

- Each cluster should be connected – there should be at least one, preferably more paths connecting each pair of nodes inside a cluster.

- If a vertex \( u \) cannot be reached from vertex \( v \), they should not be a part of the same cluster.

- Vertices should have more connections to other vertices inside a cluster compared to connections to vertices outside the cluster.
1. Theoretical background

1.4 Examples of graph and hierarchical clustering algorithms

The purpose of this section is to describe typical graph clustering algorithms and discuss their strengths and limitations. We will also look at hierarchical clustering to introduce commonly used approaches.

1.4.1 Graph clustering algorithms

1.4.1.1 Jarvis-Patrick

Jarvis-Patrick clustering [11] is a very simple method based on the examination of nodes’ neighbors. Two nodes are placed in the same cluster if they follow these conditions:

- They are connected with an edge.
- They share at least \( l \) neighbors.

If the input of the algorithm is not a graph but individual items, another parameter \( k \) is given and the graph is built by using mutual \( k \)-nearest neighbor approach. In this approach, an edge is created between two items \( i \) and \( j \) if \( i \) is among the \( k \) most similar items of \( j \) and \( j \) among the \( k \) most similar items of \( i \).

Main advantages of this approach are its speed and simplicity. Due to its graph nature, it can handle convex clusters of various sizes.

Perhaps the most serious disadvantage of Jarvis-Patrick algorithm is that the parameters \( k \) and \( l \) have to be determined by user and even small changes significantly change the output. The algorithm is also unable to separate clusters of different densities because the parameters cannot change during the clustering process.

1.4.1.2 Minimum spanning tree

Spanning tree of a connected, undirected graph is a subgraph which is a tree and connects all the nodes in the graph. Minimum spanning tree (MST) is a spanning tree with the minimal possible sum of edge weights.

MST clustering is a hierarchical divisive technique with the following steps:

1. Determine a minimum spanning tree of the graph;
2. while No more edges remain in the spanning tree do
3. Split the graph into clusters by removing the edge with the lowest weight;
4. end
5. return hierarchical structure;

To find a minimum spanning tree of a graph, well known Kruskal’s or Prim’s algorithms can be used.
1.4. Examples of graph and hierarchical clustering algorithms

Figure 1.1: Diamonds dataset. Many algorithms based on closeness or edge weights will not correctly separate these clusters because they are too close to each other and connected with short edges.

The algorithm is not limited by clusters’ shapes or sizes. However, when two different clusters are close to each other or when a bridge exists between them, MST algorithm will fail. Such dataset is shown in figure 1.1. Even though there is a small number of edges connecting the clusters in the picture, the edges are short (with high weights) and thus will not be removed during the first steps of the process.

1.4.1.3 Complex graph clustering approaches

More sophisticated graph clustering algorithms are based on the analysis of random walks on a graph [12]. If we start at any graph node and randomly travel to another connected node, we will likely stay in the same cluster because there are more edges inside clusters than between them. Therefore, clusters can be identified by focusing on the regions where the flow tends to gather.

Another widely used clustering method is based on using eigenvectors of graph’s Laplacian matrix. This is called spectral clustering and the method is further described in section 2.3.1.3 where it is used for graph bisection.

1.4.2 Hierarchical agglomerative clustering methods

In hierarchical agglomerative clustering, each item forms an individual cluster; afterwards, clusters are successively merged until they form one single cluster. Each step merges two clusters which are the most similar. Various methods to determine similarity between two clusters are shown below.
1. Theoretical background

1.4.2.1 Basic similarity measures

The basic methods to determine cluster similarity are:

**Single linkage**

The similarity between two clusters is determined by the closest pair of items from each cluster. Main limitations of this approach are noise and outliers because the similarity is determined by only one point from each cluster.

**Complete linkage**

Opposite to the single linkage. The similarity between two clusters is determined by the furthest pair of items from each cluster. This method is also affected by outliers and it cannot handle clusters of convex shapes.

**Centroid or medoid linkage**

In this method, similarity between clusters is determined by the distance between clusters’ centroids or medoids. Centroid is the geometrical center of the cluster and medoid is an item closest to the center. The method works well on equal-sized ellipsoidal clusters but fails otherwise.

**Average linkage**

The similarity is defined by the average distance between all pairs of the two clusters’ items. The limitations of this method are similar to that of centroid linkage.

As we can see, none of these simple methods is optimal. Therefore, more complex similarity measures exist, the CURE algorithm uses one of them.

1.4.2.2 CURE

Similarity function in CURE [13] is based on the combination of single and centroid linkage. In CURE, each cluster is represented by selecting a certain number of well scattered points from the cluster. Then, the points are shrunk towards the cluster’s centroid according to the shrinking factor. After that, the similarity is determined by the closest pair of representative points from each cluster.

More than one representative allows CURE to recognize clusters of convex shapes and the shrinking helps to overcome limitations of single linkage – the effects of noise and outliers.

1.4.2.3 Interconnectivity based algorithms

All of the previously discussed methods were based only on distance computed by various methods. With graph representation of the data, it is advantageous to use interconnectivity determined by edges crossing between clusters. Basically, the more edges between clusters exist, the higher their similarity is.
1.4. Examples of graph and hierarchical clustering algorithms

Figure 1.2: The limitation of interconnectivity based algorithms. Similarity functions based only on interconnectivity would consider blue and green clusters to be the most similar pair, even though red and blue clusters are very close to each other and obviously more similar.

The problem with using only distance between clusters is demonstrated in figure 1.1. Although the “diamonds” in the figure are close to each other, they are not very connected and should not be merged.

The most obvious way to determine interconnectivity is to make a sum of all edges connecting the two clusters and select cluster pair with the maximum sum. One major drawback of this approach is that big clusters tend to have higher interconnectivities simply because they contain more nodes. That is why, in more advanced algorithms, like ROCK [14], the aggregate interconnectivity is normalized by the size of clusters.

Using only interconnectivity, however, is also not optimal. Sometimes the distance is more important than connectivity, as shown in figure 1.2. Therefore, an optimal hierarchical clustering algorithm should use a combination of closeness and interconnectivity.

1.4.2.4 Common limitations

All the methods described so far suffer from the fact that they use only external cluster properties – how close or connected they are. Information about the clusters itself, for example what is the average distance between points inside the cluster, is not used at all. An explanation why the internal cluster properties are important and how they are incorporated in the Chameleon merging scheme is provided in the next chapter in section 2.4.
1. **Theoretical background**

1.5 **Clueminер**

Clueminер is an interactive data-mining platform with a special focus on clustering. The program is written in Java, built on top of NetBeans Platform and consists of many separate modules which are connected via Maven.

For end users, Clueminер offers graphical interface which allows comparing of clustering algorithms on various dataset while the modular architecture makes it easy for developers to extend. Source code and more information about Clueminер can be found at https://github.com/deric/clueminer.
Chameleon analysis

In this chapter, we analyze Chameleon as a whole, describe the algorithm’s most important aspects and propose methods which can be used to implement its key parts.

2.1 Description

Chameleon [1] is a hierarchical agglomerative clustering algorithm that uses dynamic modeling to determine the similarity between pairs of clusters. Chameleon’s key feature is that it accounts for both interconnectivity and closeness in identifying the most similar cluster pair. Furthermore, it considers clusters’ internal properties, therefore, the algorithm does not depend on user-supplied information describing the expected cluster structure but automatically adapts to the internal characteristics of the merged clusters [1].

2.1.1 Three-step approach

To find clusters, Chameleon uses a three-step approach. First two steps aim to partition the data into many small subclusters and the last repeatedly merges these subclusters into final result. The process is illustrated in figure 2.1.

The whole algorithm works with a graph representation of the data. For this reason we must first build a graph from the input. This is the first step of the algorithm. If the graph is given, we can skip the first step and start at the second.

Second step partitions the created graph. Goal of partitioning is to produce equal-sized partitions and minimize the number of cut edges [15]. This creates a lot of small clusters with few highly connected nodes in each.

Final step merges these clusters together according to various metrics. The general goal is to merge clusters which are connected, close to each other and have similar densities.
2. **Chameleon analysis**

![Diagram of Chameleon approach]

Figure 2.1: The overview of the Chameleon approach. The image was taken from [1].

2.2 **Graph construction**

Input of the algorithm is usually created by individual items with many attributes. We use these attributes to compute distances between items according to various measures. In order to create graph representation of the data, we use k-Nearest Neighbour algorithm (k-NN). K-NN creates \( k \) edges for each node which connect most similar (nearest) nodes. We also assign weights to edges, the shorter the edge the bigger weight it is assigned. The edge weight between two nodes is thus computed accordingly:\(^1\)

\[
\frac{1}{D(N_1, N_2)}
\]

where \( D \) is a distance function. The default distance function is the Euclidean distance but any other similarity measure can be used.

The graph building process with k-NN neighbor algorithm is demonstrated in figure 2.2.

Graph created this way is optimal for clustering because distant nodes are not connected at all and many edges are created among nodes close together – potential clusters.

2.3 **Graph partitioning**

Partitioning tries to split the graph into many similar-sized subclusters in a way that the edge cut is minimized. The edge cut is the total number of edges (or their weights) connecting nodes in different subclusters.

The difference to graph clustering is that in clustering, the desired number of clusters is usually not given and no constraints on cluster sizes exist.

Graph partitioning falls under the category of NP-hard problems, that is why we have to use approximation algorithms to find a solution.

\(^1\)It is important to bear in mind that two different items can have the same attributes and so the distance between them is 0. If we tried to compute the weight of edge created between those items, we would divide by 0. That is why we set very small \( \epsilon \) distance even between items which are at the same position.
2.3. Graph partitioning

(a) Input data  
(b) Graph constructed by k-NN

Figure 2.2: The process of graph creation with k-NN algorithm where $k = 4$. Edges with bigger weights are thicker.

Some algorithms try to partition the graph all at once but most use the process of recursive bisection. During recursive bisection, we recursively split the graph into two halves until we obtain the desired number of partitions or until the partitions are smaller than a specified threshold.

2.3.1 Bisection

Bisection tries to split the graph into two similar-sized parts and minimize the edge cut. It is basically a partitioning into two subsets.

In Chameleon, bisection is useful both in the partitioning process for recursive bisection and in merging for determining internal cluster properties. Bisection algorithms used in this work are Kernighan-Lin, Fiduccia-Mattheyses and spectral bisection.

2.3.1.1 Kernighan-Lin

Kernighan-Lin (K-L) [16] is a bisection algorithm based on iterative improvement of random initial partition. The algorithm comprises of the following steps:

1. Randomly assign nodes into two groups.

2. Find the difference between external and internal edge cost for each node. External edge cost $EC$ of a node is the sum of weights of edges which connect the node with nodes in other group. Internal cost $IC$ is the sum of weights of edges connecting it with the nodes in the same group. The difference is computed in this way:

$$D = EC - IC$$

The higher the difference the more the node does not belong to its current group.
3. Compute the gains of all pairs of nodes from two different groups. The gain tells us how much the bisection is improved by swapping the two nodes. It is computed accordingly:

\[ G_{1,2} = D_1 + D_2 - 2 \times w_{1,2} \]

\(D_1\) and \(D_2\) are the differences of the two nodes and \(w_{1,2}\) is the weight of the edge connecting the nodes.

4. Find the pair which maximizes the gain and swap these nodes.

5. Update the differences of all nodes neighboring the swapped pair, this is done in the following way: Find all neighbors of each node in the pair. If the neighbor was in the same group as the node in the pair, increase its difference by \(2 \times w_{n,p}\) – weight of the edge between the node from the pair and its neighbor. If the neighbor belonged to different group, decrease its difference by \(2 \times w_{n,p}\).

6. Mark the swapped nodes as used – they cannot be switched again.

7. Repeat steps 3-6 until all nodes were marked and keep track of the swapped pairs and its gains.

8. Find the sequence of swaps which maximizes the overall swap gain and use the result at the end of this sequence. This is the result of the bisection.

During the swapping, certain swaps can reduce the quality of the overall partition. The algorithm does not stop at this point, this local maximum, because sometimes a series of worsening swaps is needed to create a better result in the end [17]. That is why we have step number 8 in the algorithm. It inspects all swaps done, finds the global maximum of the gains and returns bisection which occurred at this point.

**Computational time**  
Computational time of K-L bisection is \(O(n^3)\). Finding the best pair takes \(O(n^2)\) steps because we have to find gains for all possible pairs of nodes. This is done \(O(n)\) times for we need to find the best pair before each swap.

**Equal-sized partitions**  
Bisection into two same-sized partitions is not always optimal. When the graph does not have a structure comprising of two clusters of the same size (and most graphs do not), the result is flawed by the attempt to create two equal-sized groups. The problem is illustrated in 2.3. Kernighan-Lin can be easily modified to create unequal-sized result [16] but we have to specify how many elements will be in each subset. Since we do not have any information about the graph which would tell us sizes of the
2.3. Graph partitioning

Figure 2.3: The limitation of equal-sized bisection. The constraint to create equal-sized partitions can make it impossible to achieve optimal bisection.

subsets, setting the parameters to create equal-sized partitions is the most reasonable choice.

**Weighted edges in K-L** Using edge weights during K-L bisection in combination with equal-sized partitions can lead to some unexpected results. Example of such bisection is shown in 2.4. The algorithm tries to minimize the cost of cut edges. Because the edges connecting the leftmost node with other nodes are long, they have low weights. K-L correctly does not want to cut short edges inside the cluster and so it decides to assign the leftmost node to the right cluster. This way only long edges are cut and the algorithm sees this result as optimal. But as we can see, even though the cut itself is good, the result is far from optimal and it would ruin the merging process. This is the reason why the edge weights are ignored by default in the K-L bisection.

We can afford to ignore the edge weights due to the k-NN algorithm used for graph construction. K-NN naturally creates low number of edges between distant nodes which makes them likely to be cut during the bisection.

**Disconnected clusters** Unfortunately, the previously described disconnected clusters can occur even when the edge weights are ignored. To fix this situation, we use a simple method called flood fill which finds connected components in the graph. The process of partitioning with the flood fill algorithm is shown in algorithm 1. Result of the described algorithm is a list of connected components where each component is represented by a list of nodes.

2.3.1.2 Fiduccia-Mattheyses

Fiduccia-Mattheyses [18] is a variant of K-L with computational speed of $O(E)$ where $E$ is the number of edges in the graph. Its structure is very very
2. **Chameleon analysis**

1. discover partitions of the graph using a partitioning algorithm;
2. remove the edges crossing between different partitions;
3. \( partitions = \text{new empty list}; \)
4. repeat
5. \( \text{node} = \text{find the first unmarked node in the graph}; \)
6. \( \text{partition} = \text{new empty list}; \)
7. \( \text{floodFill}(\text{node}, \text{partition}); \)
8. \( \text{partitions}.\text{add}(\text{partition}); \)
9. until All nodes are marked;
10. return \( \text{partitions}; \)
11. function \( \text{floodFill}(\text{node, partition}) \)
12. if \( \text{node}.\text{marked} \) then return;
13. \( \text{node}.\text{marked} = \text{true}; \)
14. \( \text{partition}.\text{add}(\text{node}); \)
15. foreach neighbor of the node do \( \text{floodFill}(\text{neighbor, partition}); \)
16. return;

**Algorithm 1:** Flood fill algorithm

![Graph](image1)

(a) Graph to bisect  
(b) Bisected graph  
(c) Bisected graph

**Figure 2.4:** The problem with weighted edges in K-L bisection. By K-L measures, this is a very good cut – only long edges were cut and important edges inside clusters were not. However, the bisection produced disconnected cluster which is very bad from the clustering perspective.
2.3. Graph partitioning

similar to the basic K-L. It also starts with a random bisection and iteratively improves it by swapping the nodes from one partition to the other. In contrast to K-L, Fiduccia-Mattheyses swaps only one node at a time, the one with the highest difference.

The nodes are selected from both partitions in turns to keep the result balanced. After the move, the node is locked and all its neighbors are updated. The process continues until all nodes in the graph are used. The best solution found during the execution is then used as the final bipartition.

The algorithm in its basic form works only on graphs without weighted edges. As we showed in the previous section, considering edge weights in K-L does not lead to better results and that is why we can afford to ignore the weights.

**Bucket data structure** The most important feature of the F-M algorithm is that it uses a bucket data structure to store the nodes. The data structure has an array of lists, where each list contains nodes in the same partition which cause the same change to the cutset when moved [19]. In other words, each bucket contains nodes with the same difference. The highest possible difference of a node is the maximum degree $\Delta(G)$ of any node in the graph. The lowest possible difference is the negative maximum degree. Thus, the number of buckets we need to create is:

$$2 \times \Delta(G) + 1$$

**Computational time** Finding the node with the highest difference is simple – it is any node in the bucket with the highest difference which is not empty. The operation takes $O(1)$ steps. This is significantly better than $O(n^2)$ in the case of Kernighan-Lin. After the best node is found, it is removed from the bucket and never used again.

Update of one neighboring node is more interesting. We have to store the nodes not only in the buckets but also in the array to achieve fast execution. Having the nodes in the array, we can access any neighbor instantly. It can then be removed from its current bucket in $O(1)$ time because the buckets are nothing more than double-linked lists. Adding it to the beginning of a new list is instant too. Thus, update of a neighbor is also a constant operation.

The overall time complexity is $O(E)$ because for each node being swapped we have to visit all its neighbors and update them.

**Iterative execution** The significant speed improvement of Fiduccia-Mattheyses allows us to execute the algorithm repeatedly. In repeated F-M, output of one bisection is used as the initial partition of the next run. Only the first starting partition is random. This way, we can iteratively improve the result until no improving step can be done. Result of the iterative application is shown in figure 2.5 and also in the chapter with results in 4.3.
2. Chameleon analysis

(a) One iteration
(b) Six iterations

Figure 2.5: The Improvement achieved by repeated application of Fiduccia-Mattheyses algorithm.

2.3.1.3 Spectral bisection

Spectral bisection is based on the eigenvector belonging to the second smallest eigenvalue of the Laplacian matrix.

Laplacian matrix $L$ is an $n \times n$ matrix computed in the following way [20]:

$$L = \Delta - A$$

where $A$ is the adjacency matrix of the graph and $\Delta$ is a diagonal matrix with node degrees on the diagonal. In other words the Laplacian matrix has degrees of nodes on the diagonal, $-1$ on position $(L)_{ij}$ if the nodes $i$ and $j$ are connected and 0 otherwise.

Miroslav Fiedler proved several properties of the second smallest eigenvalue and its corresponding eigenvector of the Laplacian matrix [21]. These second smallest eigenvalue and eigenvector have become known as Fiedler value and Fiedler vector.

Work in [22] shows how the Fiedler vector can be used to partition the graph. Graph vertices are mapped on their corresponding Fiedler vector components and sorted according to the components’ values. Specifically, median of the values is found and all nodes associated with values lower than median are put into one group. The rest of the nodes goes into the second part of the bisection. This way, the bisection is quite balanced, roughly half of the nodes goes into each group. However, nodes with the same corresponding Fiedler vector component value always remain in the same partition which prevents the algorithm from separating the most similar nodes.

**Computational time** The spectral algorithm provides high quality bisection and thus, when recursively applied, high quality partitioning. Unfortu-
nately, the algorithm, especially the eigenvector computation, takes a long time. The time complexity of eigenvalue decomposition used in this paper is $O(n^3)$ with large multiplicative constants which makes it too slow for the datasets with thousands of items.

2.4 Merging

After the partitioning, we have many small clusters which must be further merged to create final result. To determine which clusters are similar, we compute the following properties: internal and external interconnectivity and closeness. The importance of computing both interconnectivity and closeness was discussed in 1.4.2.3.

It is important to remember that every small cluster is a subgraph of the original k-NN graph – all these properties are computed from this graph and its subgraphs.

2.4.1 Internal cluster properties

Internal interconnectivity $IIC$ tells us how much the items in cluster are linked together. Internal closeness $ICL$ gives us the information about distances among items in the cluster.

We use bisection to compute these values. The sum of weights of the edges which were cut to split the graph gives us the internal interconnectivity value. The average of the weights is the internal closeness.

2.4.2 External cluster properties

External interconnectivity $EIC$ determines how much the clusters are connected to each other. External closeness $ECL$ represents the distance between clusters.

To gain external characteristics between two clusters we need to look at the edges connecting them. The sum of weights of the edges between the clusters is their external interconnectivity and the average is the closeness.

2.4.3 Relative cluster properties

The merging itself is based on the relative properties. They are computed from internal and external values to take into account both the distance between clusters and the similarity in their structures.

Relative interconnectivity of clusters $i$ and $j$ is computed from the internal and external in this way [1]:

$$ RIC(C_i, C_j) = \frac{EIC(C_i, C_j)}{IIC(C_i) + IIC(C_j)} $$

2.4. Merging
As we can see, external interconnectivity is normalized by the average of internal interconnectivities.

External closeness is also normalized by the internal, only this time their average is weighted by the number of items inside a cluster [1]:

$$RCL(C_i, C_j) = \frac{ECL(C_i, C_j)}{|C_i| + |C_j| ICL(C_i) + \frac{|C_i|}{|C_i| + |C_j|} ICL(C_j)}$$

This is done to increase importance of the bigger clusters because the bigger the cluster is the more information it contains.

The average is not weighted in the case of internal interconnectivity for one simple reason. It is computed from the sum of weights of the edges crossing different partitions after the bisection. The more items cluster has the more edges are cut and thus added to the sum. This way, internal interconnectivities of bigger clusters are naturally higher.

**Importance of internal properties** Combining external and internal properties is one of the key points of this clustering method. Most existing algorithms use only external properties of clusters. This way they do not correctly identify clusters of different densities. It can be best seen in figure 2.6. It is obvious that green and orange clusters are the best candidates for merging. They have similar structure and the distance between them is the same as the distance among items inside the clusters. However, algorithm based solely on external properties would merge red and blue clusters. They are closer together and are connected with more edges. Chameleon would not make that mistake because it would notice that even though they are quite connected, the items inside the clusters are even closer and more connected together. Relative to their structure, orange and green clusters are connected more and they would be correctly chosen for the merge.

### 2.4.4 Merging strategies

Once we have relative interconnectivity and closeness, we can merge the clusters. There are several strategies for deciding which clusters should be merged.

#### 2.4.4.1 User-specified threshold

The first strategy proposed in [1] is to merge clusters which exceed user-specified thresholds for relative interconnectivity and closeness. The method goes through all pairs of clusters and looks for pairs which exceed the given thresholds. If a cluster has more neighbors which satisfy this condition, the most interconnected neighbor is selected for merging. This means that during one merging step, every cluster is merged either with its nearest neighbor or not at all.
2.4. Merging

Figure 2.6: The importance of relative cluster properties. Algorithm based only on external properties would incorrectly choose to merge red and blue clusters because they are closer and more connected to each other. Chameleon, considering internal cluster structures, would not make this mistake and would merge orange cluster with the green instead.

The merging process is repeated until there is no pair of clusters which exceeds the thresholds.

The main problem with this approach is that the thresholds have to be specified by user and vary for different datasets. Even a small change in the parameters can bring huge changes to the result. Example of such change is shown in figure 2.7. Interconnectivity and closeness thresholds in the given picture were changed by only 0.1 and the change caused completely different results. Furthermore, none of the results is correct – clusters in the first result need to be further merged and in the second, separate clusters were merged together. That is why, function-defined optimization is a better merging scheme.

2.4.4.2 Function-defined optimization

The second recommended strategy is to define a function which combines relative interconnectivity and closeness. In each steps, cluster pair which maximizes this function is merged.

Original similarity function  Original similarity function from [1] combines RCL and RIC in the following way:
2. Chameleon analysis

(a) Dataset to cluster
(b) IC = 0.8, CL = 0.6
(c) IC = 0.7, CL = 0.5

Figure 2.7: The Problem with user-specified threshold merging. IC and CL are relative interconnectivity and closeness thresholds. As we can see, even a small change in the thresholds brings big changes to the final result.

\[
\text{Sim}(C_i, C_j) = RIC(C_i, C_j) \times RCL(C_i, C_j)^\alpha
\]

where \(\alpha\) is a user specified parameter. If \(\alpha < 1\), the algorithm gives a higher importance to the relative interconnectivity, when \(\alpha > 1\), higher importance is given to the relative closeness. Because we merge two clusters at each step, a standard dendrogram can be built from the sequence of results.

**Improved similarity function** Shatovska and colleagues [23] propose a modified similarity function which should outperform the previously mentioned original similarity measure.

The improved measure also incorporates internal and external cluster properties to determine the relative similarity but in a slightly different way. The internal cluster properties are computed from all edges in the graph. This way, the measure does not have to rely on the quality of the bisection algorithm.

Additionally, the result is multiplied by the quotient of clusters’ densities – density of the sparser is divided by the density of the denser cluster. This further encourages the merging of clusters with similar densities.

The whole improved formula [23] looks as follows:

\[
\text{Sim}(C_i, C_j) = \frac{|C_{i,j}|}{\min(|C_i|, |C_j|)} \times \left( \frac{CL(C_{i,j})}{|C_{i,j}|} \times \frac{CL(C_i)}{|C_i|} \times \frac{|C_i|}{\sum_{i \neq k} |C_k|} \times \frac{CL(C_j)}{|C_j|} \right)^\alpha \times \frac{\min(CL(C_i), CL(C_j))}{\max(CL(C_i), CL(C_j))}
\]

where:

- \(|C_{i,j}|, |C_i|, |C_j|\) are interconnectivity properties – number of edges between and inside clusters.
- \(CL(C_{i,j}), CL(C_i), CL(C_j)\) denote closeness properties – average weights of all edges between and inside clusters.
2.4. Merging

\( \alpha \) is a user specified closeness parameter, the same as in the previous similarity measure.

2.4.5 Clusters formed by single item

Common problem of all similarity functions in the Chameleon algorithm are clusters formed by single item. We cannot determine such cluster’s properties because it cannot be bisected and contains no edges – is it infinitely dense or not at all? That is why, single-item clusters can cause trouble during the clustering process.

The first solution to the problem is obvious – not to have such clusters in the first place. All bisection algorithms try to produce equal-sized partitions but they cannot guarantee that these clusters will not occur, especially because of the existence of disconnected clusters described in 2.3.1.1.

Therefore, every similarity function in our implementation gives much higher similarity to pairs of clusters in which single-item cluster is detected. This way, they are joined with the neighboring clusters in the early stages of the merging process and do not cause problems later on.
In this chapter, we describe the implementation of methods introduced in the Chameleon analysis.

The whole Clueminer clustering platform is written in Java. We also chose this language for our implementation, mainly to avoid problems during the integration and to be able to fully use all Clueminer’s modules.

3.1 Chameleon module

Chameleon algorithm is implemented as a separate module in the Clueminer clustering platform. The most important part of this module is the Chameleon class which extends Clueminer’s AbstractClusteringAlgorithm class and implements AgglomerativeClustering interface. Therefore, it has to implement cluster(Dataset) and hierarchy(Dataset) methods which return clustering and hierarchical result, respectively. The module also contains classes which implement more specific parts of the algorithm, such as Merger.

Some parts of Chameleon, for example graph interface or partitioning, can be useful on its own or in other clustering algorithms. That is why we split these independent pieces into separate modules. Model of important Chameleon modules and the dependencies between them is shown in figure 3.1.

Class diagram describing relationships between the most important classes can be seen in figure 3.2.

3.2 Graph interface

Since Chameleon uses graph as an internal representation of the data throughout the whole process, we had to implement representation of the graph itself. This is done in the AdjMatrixGraph class which implements Clueminer’s graph interface.
3. **Chameleon implementation**

Figure 3.1: A diagram of the most important modules and the dependencies between them.

Figure 3.2: A diagram of classes directly related to Chameleon.
Nodes and edges of the graph are stored in the form of an adjacency matrix. The class offers all important graph manipulation operations like adding and removing nodes and edges, getting all neighbors of a node, determining if the two given nodes are connected by an edge, etc.

### 3.3 k-NN

K-NN graph is built using the KNN class. The class contains two public functions `getNeighborArray(dataset)` and `getNeighborGraph(dataset)` which find $k$ nearest neighbors of each item and return the result either in the form of a two-dimensional array or as a graph. The number of neighbors and the distance measure between items can be specified via setter methods.

The calculation of distance between two items can be computationally intensive, especially for high-dimensional datasets. Therefore we compute the distances between all pairs of items before the k-NN algorithm itself and store them in the two-dimensional cache. We can then determine the distance between any two items in $O(1)$ time, regardless of the dataset dimension.

### 3.4 Partitioning

Due to the existence of many possible partitioning methods, we created a common partitioning interface with an abstract method `partition(maxPartitionSize, graph)`. Parameter `graph` is the graph we want to partition and `maxPartitionSize` specifies the maximum number of items which can be in a single partition. The function returns list of clusters where clusters are represented by a list of nodes belonging to the same cluster. We also created an interface for bisection with an abstract method `bisect(Graph)`. We provide only one significant class implementing the Partitioning interface – RecursiveBisection. We also experimented with other partitioning methods which try to partition the graph all at once but none of them brought any useful results. An example of such method is the P2Ppartitioning class which implements the algorithm described in [24].

`RecursiveBisection` recursively applies a bisection algorithm until the partitions are smaller than the `maxPartitionSize` parameter. Therefore, a bisection algorithm has to be specified before the start of the partitioning process.

Available implementations of bisection algorithms are: KernighanLin, FiducciaMattheyses and SpectralBisection. The structure and properties of these algorithms are described in 2.3.1.

In the end of the RecursiveBisection, the partitioning result is given to the FloodFill class which splits any disconnected partitions into separate clusters. This prevents the partitioning from returning any disconnected clusters, regardless of the specified bisection algorithm.
3. Chameleon Implementation

3.5 Merging

In the Chameleon analysis 2.4, we described several merging strategies with many common features. Therefore, we designed an abstract class `Merger` which implements common methods for computing internal and external cluster properties. The external and internal properties are then used to compute relative closeness and interconnectivity between all clusters.

3.5.1 Internal properties

We use class `Cluster` to represent a single cluster. The class contains cluster’s graph and it is able to compute internal properties of the cluster either by bisecting the graph or by iterating through all of its edges.

3.5.2 External properties

The external properties are stored in a two-dimensional matrix. Computation of external properties is done in \( O(E) \) time where \( E \) is the number of edges in the graph created by k-NN algorithm. Going through all edges is all we need because the edge either connects two clusters and we use the edge weight to update information in the property matrix or it connects items inside a cluster in which case we do nothing.

3.5.3 Merging strategies

Particular child classes of `Merger` are `ThresholdMerger` and `PairMerger`. The `ThresholdMerger` and `PairMerger` differ not only in the implemented merging strategy but also in the way they compute cluster properties after each merging step.

3.5.3.1 Threshold merger

Algorithm in `ThresholdMerger` basically starts over after every merge. In every step, usually more than one pair of clusters is merged together, many new clusters arise and properties of the new clusters have to be computed. Because the structure of clusters is significantly changed after every step there is no point in gradually updating the values – it is easier to recompute everything from the beginning.

3.5.3.2 Function-defined merger

`Pairmerger`, on the other hand, merges only one pair of clusters in each step. That is why it would be very ineffective to recompute properties of all clusters after every single merge. We only compute internal properties of the newly created cluster and update changed external properties in the matrix with external properties.
3.6. Chameleon class

**Merging time complexity** Having eliminated recomputation of cluster properties, speed of the function-defined merger mostly depends on how quickly we can find the most similar cluster pair. The naive approach is to look through all cluster pairs in each step. Finding the most similar pair then takes $O(m^2)$ steps with $m$ being the number of clusters at the beginning. The merging of all clusters takes $O(m)$, therefore the overall time complexity of this approach is $O(m^3)$.

To speed the process up, we use a priority queue to store similarities between all cluster pairs. Finding the most similar pair in the queue takes $O(1)$ time, removing the top element and adding $O(\log(n))$ where $n$ is the number of pairs in the queue. Overall time complexity of this approach is $O(m^2 \log(m))$. We have to merge $m$ clusters and during each merge, compute similarities of the newly created cluster with the remaining $m$ clusters and add them into the queue.

**Standard and improved similarity** As mentioned in the analysis, we have two functions which determine the similarity between clusters. For this reason, the `Pairmerger` class is abstract and is further extended by `StandardSimilarity` and `ImprovedSimilarity` instantiable classes. The classes have its own implementation of the similarity function and they also differ in the way they merge the clusters.

In `ImprovedSimilarity`, we can instantly set the new cluster’s internal properties during the merge. Because the properties are computed from sum of weights of edges and we know both the sums of the clusters to be merged and the sum of edges between them (from external properties), we can easily determine the new properties via weighted average. This saves us the computation which would normally take $O(E)$ time.

This cannot be done in the `StandardSimilarity` class because there the properties are determined via bisection and we cannot predict how the bisection of the new cluster will look based on the properties of the old clusters. Therefore, each new cluster has to be bisected.

3.6 Chameleon class

The Chameleon class serves as an interface for the Chameleon algorithm. Its methods `cluster(Dataset)` and `hierarchy(Dataset)` combine the previously described classes in a simple pipeline. Specifically, the data are initially sent to the `KNN` class which builds the graph. Then, the graph is partitioned through any function implementing the partitioning interface. Finally, the partitions are merged in the `Merger` class.

The class is marked as a clustering service provider, therefore it shows everywhere in Clueminer where clustering algorithms are requested.
3. Chameleon implementation

3.7 Algorithm parameters

There are a total of 7 parameters which affect different parts of the Chameleon clustering process. The parameters are described in the following list:

\( k \)

This parameter specifies the number of neighbors during graph construction in the k-NN algorithm. It is a very important parameter because the rest of the algorithm performs all its operation on this graph. Unfortunately, the optimal value of \( k \) is different for each dataset.

When \( k \) is too low and the constructed graph is therefore too sparse, the graph does not contain enough information to determine where the clusters are. The parameter also cannot be too high – we do not want to create a graph where every node is connected with almost every other because everything would look like a single cluster.

The default value is set to \( 2 \times \log_2 n \) which creates enough but not too many edges (\( n \) is the dataset size).

maxPartitionSize

Maximum size of a single partition at the end of the partitioning phase. If too high, there is a danger that the partitioning will not split different clusters. This error cannot be undone in the merging phase because there the clusters are only merged together. Therefore, it is better to create smaller partitions and merge them during the merging. However, the partition size cannot be too low because then it would be impossible to properly determine cluster’s internal properties. The default maxPartitionSize is set to 10.

distanceMeasure

This parameter specifies which distance measure to determine distance between items in the dataset should be used. The default distance function is the Euclidean distance.

bisection

Specifies the bisection algorithm used during recursive bisection and determining internal cluster properties. The default algorithm is Fiduccia-Mattheyses.

closenessPriority

Determines whether the algorithm should give higher importance to closeness or interconnectivity of the clusters. The default value is set to 2 – closeness is slightly more important than interconnectivity.

similarityMeasure

This parameter specifies which similarity function should be used during
the function-defined merging. Available similarity measures are standard and improved, the default being the improved similarity.

**internalEvaluator**
This parameter tells which measure to use during cluster evaluation when we try to find the best cut-off. Internal evaluators are further described in the next section.

The option to use user-specified threshold during the merging phase is deliberately not included. Its sensitivity to user-specified parameters described in 2.4.4.1 makes it unusable for any real data.

## 3.8 Clueminser integration

### 3.8.1 Used modules
Integration into the Clueminser platform and use of its modules is vital for our implementation of Chameleon. The most important used modules are:

**Clustering API**
By implementing Clueminser’s clustering application programming interface (API) and returning clustering results in the required form, we can easily integrate Chameleon between other clustering algorithms. Chameleon can then be used in any module which requires clustering.

**Distance measure**
Determining distance between items is very important throughout the whole clustering process. Clueminser’s **clustering-dist** module offers efficient implementations of many distance measures, such as **Euclidean distance**, **cosine similarity**, **Manhattan distance** and many more.

**Dataset interface**
The **dataset-api** module takes care of all data manipulation like loading the data from a file. The data are given to a clustering algorithm in the form of the **Dataset** class from which we can easily access all of the input items and its attributes.

**Hierarchical result**
The **HierarchicalResult** class stores hierarchical clustering result in a tree which is built during the hierarchical clustering process. The tree can be rendered using the **dendrogram** module. We can also apply a cut-off function to the tree which finds the best clustering result and cuts the tree at the corresponding level.

Clueminser’s original clustering tree could only store one item in every leaf. This was enough for hierarchical clustering methods which start the
merging process on clusters created by individual items. However, since Chameleon first partitions the data into subclusters and then merges these subclusters, we had to design an extra class called DClusterLeaf which extends the original DLeaf and is able to store groups of items. We also modified all important methods of HierarchicalResult to work correctly with this new type of leaf.

**Dendrogram**

The dendrogram module is able to render any hierarchical result in the form of a dendrogram. As in the case of HierarchicalResult class, we had to modify some of dendrogram’s parts to allow rendering of trees where leaves contain whole clusters. In figure 3.3a we can see how the tree with leaves representing clusters is drawn.

**Cut-off**

To find an optimal cut of a dendrogram, Clueminer offers many cut-off strategies and cluster evaluation measures. Cluster evaluators measure different cluster properties to determine how good the produced clusters are. Cut-off strategies then use these evaluations to cut the dendrogram at the best possible level.

Clueminer offers almost 30 different clustering evaluators which compute everything from cluster connectivities and distances to the silhouette score which determines if the objects lie well within their clusters [25].

**Matrix**

In spectral bisection, we have to compute eigenvectors of graph’s Laplacian matrix. This computation is already implemented in the math module, specifically in the JMatrix class. Therefore, we used this class to compute the desired Fiedler vector which saved us a lot of time and effort we would have spent implementing the eigenvector decomposition.

### 3.8.2 Chameleon user interface

To run Chameleon from Clueminer, we designed a dialogue which enables to set all important algorithm parameters. The dialog is shown in figure 3.3b. When user confirms the dialog, the parameters are given to an instance of the Chameleon class and its hierarchy() method is called.

### 3.9 Time complexity

The overall time complexity of our Chameleon implementation is determined by complexities of its key parts which are described in the rest of this chapter.
3.9. Time complexity

3.9.1 k-NN complexity

Time complexity of our k-NN algorithm is $O(n^2)$. The algorithm has to visit all neighbors of every item to find the nearests.

The computational time can be improved with complex data structures, for example kd-trees [26]. Search for nearest neighbors of one item then takes $O(\log n)$ instead of $O(n^2)$. Unfortunately, the speed improvement decreases with growing dimensionality and in high dimensional spaces, kd-tree has the same performance as a simple linear search.

3.9.2 Partitioning complexity

The amount of time required by the partitioning depends on computational time of a bisection algorithm because we use recursive bisection to partition the graph. As we showed in 2.3.1.2, time complexity of Fiduccia-Mattheyses bisection is $O(n^2)$. Spectral partitioning, with its speed of $O(n^3)$, is applicable only on the smallest datasets, therefore we do not include it in the rest of our time complexity analysis.

The amount of time needed to bisect all graphs at a single level of recursion is always $O(n)$ – the graphs are being divided but the number of nodes throughout the graphs remains the same. Depth of the recursion depends on the desired number of partitions. To obtain $m$ partitions of size $n/m$, we need to recursively bisect the graph $\log(m)$ times. Therefore, the overall time complexity of the recursive bisection is $O(n(\log(m)))$.

\footnote{Since we work with sparse graph constructed by k-NN, number of nodes $O(N) = O(E)$ and the number of input items $n$ corresponds to the number of nodes.}
3. CHAMELEON IMPLEMENTATION

3.9.3 Merging complexity

Complexity of merging algorithms was already discussed in 3.5.3.2. We concluded that with the help of a priority queue, we can find the most similar cluster pair in $O(\log(m))$. The whole merging then takes $O(m^2 \log(m))$.

We also showed that in case of a standard similarity measure, we need to bisect the graph at each step. Since the bisection is done in $O(n)$ time, this brings an extra $O(mn)$ complexity throughout the merge of all clusters. No such computation is needed in the improved similarity.

3.9.4 Overall complexity

To sum it all up, the overall time complexity of the Chameleon algorithm with Fiduccia-Mattheyses bisection, improved similarity measure and graph constructed by k-NN algorithm is $O(n^2 + n\log(m) + m^2 \log(m))$. 
Results and comparisons

In this chapter, we compare the performance of various algorithms used in this work and present the results achieved by our implementation of the Chameleon algorithm.

4.1 Comparison of bisection algorithms

In this section, we compare the quality and speed of Fiduccia-Mattheyses and spectral bisection. Kernighna-Lin, which is also implemented, is basically just a slower version of Fiduccia-Mattheyses, therefore we do not present its results.

In figure 4.2 we see computational times of the algorithms. Comparison of resulting bisections is provided in figure 4.1.

The results in 4.1 and many other conducted experiments showed that the quality of spectral bisection and iterative Fiduccia-Mattheyses is comparable. However, the time complexity is not, therefore the use of spectral bisection makes sense only on small datasets.

The duration of a single F-M iteration is also shown in the figure 4.2 but in 4.1, we can see that one iteration is not enough to guarantee a good bisection. The process of iterative improvement is presented in figure 4.3. It shows that

![Figure 4.1: Comparison of bisection algorithms on dataset with 1250 items.](image)

(a) Spectral bisection  (b) Iterative F-M  (c) Single F-M
4. Results and comparisons

4.1 Limiting the number of iterations in F-M

Iterative F-M has one drawback – we cannot predict how many iterations will be done until the result stops improving. Usually, the number is not very high, it ends in about 10 or 20 steps but even that can be a problem on large datasets where every iteration takes a long time. That is why we included an option to set the maximum number of iterations. As shown in Figure 4.3, the most important swaps happen during the first few iterations, therefore the
4.2 Comparison of cluster similarity measures

We implemented two cluster similarity functions – standard and improved similarity measure which were described in 2.4.4.2. We tested both measures on various datasets and the experiments showed that the improved similarity is better than the original approach.

Its advantage lies in additional comparison of cluster densities. This leads to better merging of clusters with similar structure and as a results, the produced dendrogram is much more balanced. An example is shown in figure 4.4.

4.3 Results

4.3.1 Two-dimensional datasets

In this part, we present the results achieved by our implementation of the Chameleon algorithm. The tested datasets contain clusters of different shapes, sizes and densities. Moreover, almost all of the datasets are full of noise. All of this properties make these datasets quite difficult to cluster, most clustering algorithms are not able to find correct clusters in these examples.

We chose two-dimensional datasets because there we can easily visualize and evaluate the quality of the discovered clusters. However, Chameleon can be applied to any dataset where similarity between items can be computed (or is given).

The results are shown in figure 4.5. We used the same parameters for all inputs – default value of $k$ ($2 \times \log_2(n)$), default $maxPartitionSize$, Fiduccia-Mattheyses bisection and improved similarity measure with $closenessPriority$ set to 2.

Chameleon returns the result as a dendrogram of merges. To obtain a result which assigns all items to clusters, the dendrogram has to be cut at some level. In case of the presented two-dimensional datasets, we cut the dendrogram manually to get the best result. Normally, the cut is automatically determined with the help of a cluster evaluator measure but here we want to demonstrate Chameleon’s ability to correctly identify the clusters at some point during the merging. Determining where that point is is another task.
4. **Results and comparisons**

(a) Improved similarity  

(b) Standard similarity

Figure 4.4: Comparison of standard and improved similarity measures on a dataset with a lot of noise (Three ring clusters dataset shown in figure 4.5). The dataset contains three clusters which were correctly identified by the improved similarity but not by the standard. Dendrogram of the improved similarity measure is also much more balanced.
4.3. Results

As we can see, our implementation correctly identifies clusters in all of the datasets, even clusters located inside other clusters. Also, noise and paths between clusters do not cause merging of separate clusters which is important because real data usually contain a lot of noise.

4.3.2 Datasets with more dimensions

We also tested the implementation on datasets with more than two dimensions to evaluate Chameleon’s ability to cluster multi-dimensional data. In this case, we used a hill climbing cut-off strategy with SD index evaluator to determine the optimal cut-off because in more dimensions, results cannot be easily visually evaluated.

First, we ran clustering on an artificial 32-dimensional dataset with 16 well-separated Gaussian clusters. Chameleon was able to find all of the clusters without any mistakes. Dendrogram visualising the hierarchical result can be seen in the appendix in figure B.4.

Next, we tested several well known and widely used real world datasets – Iris, Ecoli and Thyroid Disease. All of them can be found for instance in the UCI Machine Learning Repository at http://archive.ics.uci.edu/ml/.

What these datasets have in common is that the clusters of different classes overlap and cannot be completely separated without the external knowledge of class labels. However, Chameleon was still able to find several clusters which roughly match the class labels of the data. The discovered clusters are shown in the form of a scatterplot matrix in the appendix in figures B.1, B.2 and B.3.

4.3.3 Comparison with the reference

In figure 4.6, we compare our results with the results presented in the article about Chameleon [1]. The results differ only in a few details. On the first dataset (t7.10k.dat), the original result contains two clusters with outliers, whereas our implementation assigned every point to one of the genuine clusters and the number of clusters found therefore matches the number of genuine clusters. In the second (t8.8k.dat), the only difference is that we split one of the clusters into two at its narrowest point.

4.3.3.1 Absence of the reference implementation

We could not compare the results with the referential implementation because the implementation is not available. On the page where Chameleon is presented [27], it is mentioned that Chameleon’s partial implementation can be found in the Cluto software package. We installed Chuto, looked through the program and its documentation and found that only the first two phases of Chameleon (graph construction and partitioning) are implemented. Chuto offers several similarity measures which can be used to hierarchically cluster
4. Results and comparisons

(a) Six clusters with a lot of noise
(b) George dataset
(c) Cluster inside another cluster
(d) Three ring clusters
(e) Gaussian clusters
(f) Aggregation dataset

Figure 4.5: Results on various datasets with default parameters.
4.3. Results

(a) Original result on t7.10k.dat
(b) Our result on t7.10k.dat

(c) Original result on t8.8k.dat
(d) Our result on t8.8k.dat

Figure 4.6: Comparison with the results presented in the original article about Chameleon [1]. On t7.10k.dat, original implementation found eleven clusters, two of them are formed by noise. Our implementation assigned every point to one of the genuine clusters, the number of identified clusters therefore matches the number of genuine clusters. On t8.8k.dat, the only difference between the results is that our algorithm split one of the clusters into two at its narrowest point.

the data, however, none of the functions compares internal and external close-ness and connectivity the way Chameleon does. Therefore, we only compared our results with the ones shown in the original article describing Chameleon [1].
Possible parallelization

To improve Chameleon’s execution speed, we consider which parts of the algorithm could be parallelized.

5.1 k-NN

Parallelization of k-NN graph construction is quite straightforward. We look through all neighbors of every item to find the nearests. Since the items are independent, different threads can simultaneously search for neighbors of different items.

5.2 Partitioning

Parallelization of recursive bisection is possible by bisecting more graphs at once. At the beginning of recursive bisection, graph is bisected into two parts. Both of the parts are then bisected and the same process is applied to the results of this bisection until the stopping condition is reached. We can run bisections on the intermediate graphs in parallel because they are completely independent. This means that in the first step, we can run only one bisection at once, then two, four in the third level of recursion, etc.

Parallelization of the bisection itself can significantly improve otherwise too slow spectral bisection, especially the computation of eigenvectors and eigenvalues. There are many algorithms available for parallel eigenvector computation, for instance the algorithm designed by J. J. Dongarra and D. C. Sorensen in [28].

On the other hand, our fastest bisection algorithm, Fiduccia-Mattheyses, is not suitable for parallelization. Its process is based on sequential switching of nodes where every step depends on the previous. We can parallelize only the initial computation of node differences which is relatively time-consuming but occurs only once at the beginning of the algorithm.
5. **Possible parallelization**

5.3 **Merging**

Merging cannot be easily parallelized as a whole because the resulting dendrogram is a sequentially built hierarchical structure. However, many sub-parts of the merging process can be sped up by parallelization.

At the beginning, before the merging itself starts, internal and external cluster properties have to be computed. Both of these operations can be parallelized quite easily. Clusters are independent, therefore during computation of internal properties, we can compute properties of more clusters simultaneously. External properties, as mentioned in 3.5.2, are computed via iterating over all edges of the original graph created by k-NN. We can therefore iterate through more than one edge at once.

Another suitable part is the update of external properties. After each merge of two clusters we get a new one and properties between the new cluster and all others have to be computed. This computation can be parallelized because external properties are independent of each other.

5.4 **Conclusion**

To conclude, k-NN and spectral bisection could be significantly sped up by parallelization. Important parts of the merging phase would also benefit from the potential parallelization and only minor portion of Fiduccia-Mattheyses can be run in parallel.
In this thesis, we analyzed and implemented the Chameleon clustering algorithm. The implementation consists of:

- Creating a method for building graphs based on k-nearest neighbors algorithm.
- Implementing a variety of bisection algorithms which are used to recursively partition the graph.
- Hierarchical merging of clusters based on their similarity.
- Linking all of these parts together to create a compact algorithm.

The whole implementation is based on a theoretical description by Kumar and colleagues [1] but we made several changes to improve the algorithm. Most importantly, we incorporated an additional merging function from [23] which outperforms the originally described.

Our work, especially the clustering results, showed that Chameleon is in fact a robust, high-quality clustering algorithm capable of discovering clusters of different shapes, sizes and densities.

Important aspect of our work was also the integration of Chameleon into the Clueminer clustering platform. This integration was successful, we created a graphical user interface which enables to modify Chameleon’s parameters and run the algorithm. Furthermore, all of Chameleon’s clustering outputs can be visualized via Clueminer’s modules such as dendrogram or scatter plot.

Since Clueminer is a free software, anyone can use our implementation of Chameleon for data clustering.
Bibliography


Bibliography


Acronyms

API  Application programming interface
ECL  External closeness
EIC  External interconnectivity
F-M  Fiduccia-Mattheyses
ICL  Internal closeness
IIC  Internal interconnectivity
K-L  Kernighan-Lin
k-NN k-Nearest Neighbors
MST  Minimum spanning tree
RCL  Relative closeness
RIC  Relative interconnectivity
In this chapter, we provide visualizations of results produced by Chameleon on several multi-dimensional datasets.

Figure B.1: Scatterplot matrix of a Chameleon clustering result on the Iris dataset – two clusters discovered.
B. Results on multi-dimensional datasets

Figure B.2: Scatterplot matrix of a Chameleon clustering result on the Ecoli dataset – three clusters discovered.

Figure B.3: Scatterplot matrix of a Chameleon clustering result on the Thyroid Disease dataset – five clusters discovered.
Figure B.4: Visualization of a dendrogram created by Chameleon on a dataset with 32 dimensions and 16 separated clusters. The columns on the right represent clusters discovered by the algorithm and the labels of items. As we can see, the discovered clusters match the labels.
Contents of the enclosed CD

- readme.txt ....................... the file with CD contents description
- Clueminier .......................... Clueminier clustering platform
  __source.................................. Clueminier source files
  __exe...................................... Clueminier binaries
- Thesis ................................. source codes of this thesis
  __pictures............................... images used in the thesis
  __plots................................. source codes of the plots used in the thesis
  __BP_Búna_Tomáš_2015.tex ........... \LaTeX{} source code of the thesis
- thesis.pdf ......................... the thesis text in PDF format