Bachelorarbeit

Graph Clustering: A Comparison of Louvain and Leiden

Verfasser

Marc Martin Dollmann

angestrebter akademischer Grad

Bachelor of Science (B.Sc.)

Wien, 2023

Studienkennzahl lt. Studienblatt:	UA 033 521
Fachrichtung:	Informatik - Allgemein
Betreuer:	Dr. Martin Schirneck, B.Sc. M.Sc. Antoine El-Hayek, M.Sc. Maximilian Vötsch, M.Sc.

Contents

Abstract

Graphs can be used to represent many real world situations. In those graphs, the community structures of the real world situations are embedded. To uncover those structures, clustering algorithms are needed. Louvain and Leiden are two common examples of such algorithms. Leiden is an improvement of Louvain and brings with it a number of guarantees on the clusterings it produces. One of them is that the communities Leiden clusters are connected. Leiden produces clusterings of higher modularity scores on all real world graphs examined. On most of them, the difference is significant.

1 Motivation

Many aspects of the real world can be represented by graphs. Road networks with cities as nodes and roads as edges, social media with accounts as nodes and following or "friendship" as edges and brain connectomes ("wiring diagrams") with neurons as nodes and synapses as edges are just three of the many examples [12]. Clustering nodes is a powerful tool that can uncover aspects of the underlying real world situation. For instance, one can group Twitter users by their opinion towards Covid-19 vaccines [18]. Louvain [2] and Leiden [16] are two examples of widely used graph clustering algorithms. Louvain and Leiden are key parts of the `scanpy` [19] python package which can be used to do single cell analysis. Each community in the clustering of cells represents one cell type. The clustering can then be used to determine marker genes for each cell type [19].

In this thesis, I will introduce modularity and Constant Potts Model, two common quality scores for graph clustrings. Then, I will examine closely the Louvain and Leiden algorithms theoretically. Finally, I will apply them to graphs in order to compare the modularity scores of their clusterings.

2 Related Work

Modularity is a quality score of how well a clustering represents the structure of a graph. The graph clustering algorithm Louvain is, at its core, a greedy algorithm trying to maximise the modularity score of the clustering [2]. The Leiden algorithm is an improvement on Louvain which guarantees connectivity of the communities. Leiden works also with the alternative quality measure Constant Potts Model (CPM). Traag et al. use CPM to prove the guarantees Leiden provides [16]. Hence, I will also introduce CPM. This section is organised as follows. First, I will introduce modularity and CPM. Having introduced Modularity, I can then examine Louvain. The shortcomings of Louvain then motivate Leiden. Lastly I will go into the guarantees Leiden provides.

A clustering of the nodes of a graph is a partition of the nodes. One cluster in a clustering, i.e. one of the sets that make up the partition, is also sometimes called a community. Clustering and partition will be used interchangeably, as they are in the papers I am referencing.

2.1 Modularity

I will first define the configuration model for producing random graphs, then modularity. Lastly, I will discuss the Constant Potts Model (CPM) and show some examples.

Definition 2.1 (Configuration model [7])**.** The configuration model constructs a random graph $G = (V, E)$ with a given degree distribution. It works as follows:

1. Let $\mathbf{k} = (k_i)_{i=1,\dots,n} \in \mathbb{N}^n$ be a sequence of natural numbers of length n and even sum
$$\sum_{i=1}^{n} k_i = 2 * m \text{ with } m \in \mathbb{N}.$$
2. Initialise n nodes, $V = \{1, \dots, n\}$.
3. For each $i \in \{1, \dots, n\}$ attach k_i half-edges, or stubs, to node i. Let S be the set of all stubs.
4. Until $S = \emptyset$:
   (a) Consider any stub $s \in S$, choose uniformly at random any other stub $s' \in S \setminus \{s\}$.
   (b) Pair s and s' to form an edge e.
   (c) Add e to the set of edges E.
   (d) Remove s and s' from the set of stubs S.

The resulting graph G is called the *configuration model with degree sequence* $\boldsymbol{k}$*.*

Consider any pair of nodes $x, y \in V$. In the configuration model with degree sequence $\mathbf{k}$, any stub is connected uniformly at random to any other of the $2m - 1$ remaining stubs. Furthermore, there are k_x stubs attached to node x and k_y to y. Let $(A_{x,y})_{x,y \in V}$ denote the adjacency matrix. Hence, the expected number of edges going from x to y in the configuration model is:
$$\mathbb{E}[A_{x,y}] = \frac{k_x \cdot k_y}{2m - 1}.$$

Definition 2.2 (Modularity [3])**.** Let $G = (V, E)$ be a graph, $(A_{x,y})_{x,y \in V}$ its adjacency matrix and $\mathcal{P}$ be a partition of G. Furthermore, $m = |E|$ is the number of edges in G and k_x the degree of a node $x \in V$. The *modularity* of the partition $\mathcal{P}$ is:
$$Q(\mathcal{P}) = \frac{1}{2m} \sum_{C \in \mathcal{P}} \sum_{x,y \in C} \left(A_{x,y} - \frac{k_x \cdot k_y}{2m} \right).$$

Modularity is a quality score that quantifies how well a partition represents the structure of a graph. For each pair of nodes $x, y \in C$ in a cluster C, modularity compares how many edges go from x to y to the expected number of edges going from x to y according to the configuration model with degree sequence $\mathbf{k} = (k_x)_{x \in V}$. Instead of using the expected value as calculated above, modularity ignores the -1 in the denominator. Hence, the expected value that modularity works with is:
$$\mathbb{E}[A_{x,y}] \approx \frac{k_x \cdot k_y}{2m}.$$

Modularity is a scalar in the range $[-\frac{1}{2}, 1]$. It is desirable to maximise modularity. The minimum value of modularity is achieved when all edges are between communities and none are inside any community. For example, consider a bipartite graph $G = (V, E)$, with $V = V_1 \cup V_2$ and $V_1 \cap V_2 = \emptyset$, and $E \subseteq V_1 \times V_2$. The clustering $\mathcal{P} = \{V_1, V_2\}$ of G has modularity $-\frac{1}{2}$. On the other hand, consider the following theorem.

Theorem 2.3 (Connectivity and maximal modularity [3])**.** A partition of maximal modularity does not contain any disconnected communities.

Definition 2.4 (Modularity decision problem [3])**.** Let G be a graph and $q \in \mathbb{R}$ be a number. The *modularity decision problem* asks whether there exists a partition $\mathcal{P}$ with modularity bounded from below by q:

$$Q(\mathcal{P}) \geq q.$$

To see how hard maximising modularity is, consider the modularity decision problem. Brandes, Delling, Gaertler, Görke, Hoefer, Nikolosky and Wagner proved in their 2008 paper "On Modularity Clustering" that the modularity decision problem is $\mathcal{NP}$-complete [3].

Another property of modularity is, that for any Graph $G = (V, E)$, the trivial partition $\mathcal{P} = \{V\}$, where all nodes are in one cluster, has modularity score of 0.

Proof. Let $G = (V, E)$ be a graph of $m = |E|$ edges. Let $\mathcal{P} = \{V\}$ be the trivial partition of G.

$$\begin{aligned} Q(\mathcal{P}) &= \frac{1}{2m} \sum_{C \in \mathcal{P}} \sum_{x,y \in C} \left(A_{x,y} - \frac{k_x \cdot k_y}{2m} \right) \\ &= \frac{1}{2m} \sum_{x,y \in V} \left(A_{x,y} - \frac{k_x \cdot k_y}{2m} \right) \end{aligned}$$

This follows from the fact that there is only one community $C \in \mathcal{P}$ which contains all nodes $C = V$. Moving the summation into the brackets gives:

$$Q(\mathcal{P}) = \frac{1}{2m} \left(\sum_{x,y \in V} A_{x,y} - \sum_{x,y \in V} \frac{k_x \cdot k_y}{2m} \right).$$

The sum over the degrees of all nodes is $\sum_{x \in V} k_x = 2m$. Furthermore, note that the sum of all entries in the adjacency matrix A is $2m$ as each edge is counted twice. Splitting the sum over the degrees into the parts on x and y gives:

$$\begin{aligned} Q(\mathcal{P}) &= \frac{1}{2m} \left(2m - \frac{\sum_{x \in V} k_x \sum_{y \in V} k_y}{2m} \right) \\ &= \frac{1}{2m} \left(2m - \frac{2m \cdot 2m}{2m} \right) \\ &= \frac{2m}{2m} - \left(\frac{2m}{2m} \right)^2 = 1 - 1 = 0. \end{aligned}$$

□

Note that one can also introduce a resolution parameter $\gamma \in [0,1]$ in the definition of modularity. The formula for modularity then becomes:

$$Q(\mathcal{P}) = \frac{1}{2m} \sum_{C \in \mathcal{P}} \sum_{x,y \in C} \left(A_{x,y} - \gamma \cdot \frac{k_x \cdot k_y}{2m} \right).$$

For $\gamma = 1$ this is equal to the previous definition of modularity. Lowering γ reduces how much modularity penalises deviation from the configuration model. For example, when γ is set to 0, the modularity of the trivial partition is not 0 any more, but 1. Hence modularity does not penalise the partition at all for having no community structure.

Modularity can also be calculated for weighted graphs. In this case, m denotes the total weight of all edges, k_x denotes the total weight of all edges incident to x and $A_{x,y}$ denotes the weight of all edges going from x to y. The unweighted modularity of an unweighted graph G, is equal to the weighted modularity if all weights are set to 1 in G.

2.2 Constant Potts Model

The Constant Potts Model (CPM) is, like modularity, a quality measure for a partition $\mathcal{P}$ of a graph G. CPM tries to maximise the number of edges inside communities and minimise number of the edges between communities. I will introduce two definitions of CPM.

Definition 2.5 (Original Constant Potts Model [15])**.** Let $G = (V, E)$ be a graph, A its adjacency matrix, $\mathcal{P}$ be a partition of G and let $\gamma \in [0,1]$. The *Constant Potts Model* Q of the partition $\mathcal{P}$ is:

$$Q(\mathcal{P}) = -\sum_{C \in \mathcal{P}} \left(\sum_{x,y \in C} A_{x,y} - \gamma \cdot |C|^2 \right).$$

I include the original definition (Definition 2.5) as published by Traag, Van Dooren and Nesterov in 2011 [15] for completeness, as their paper is referenced by the Leiden paper [16]. This version of CPM compares the number of edges in a community ($\sum_{x,y \in C} A_{x,y}$) with the possible number of edges ($|C|^2$) multiplied with a threshold $\gamma \in [0,1]$. Here any edge is counted twice, once going from x to y and once from y to x. The underlying model also counts self loops as possible edges. Is is desirable to minimise this version of CPM.

Definition 2.6 (Leiden's Constant Potts Model [16])**.** Let $G = (V, E)$ be a graph, $\mathcal{P}$ be a partition of G and let $\gamma \in [0,1]$. Furthermore, let $E(C)$ denote the number of edges in a community $C \in \mathcal{P}$. The *Constant Potts Model* Q of the partition $\mathcal{P}$ is:

$$Q(\mathcal{P}) = \sum_{C \in \mathcal{P}} E(C) - \gamma \cdot \binom{|C|}{2}.$$

Traag, Waltman and van Eck use a slightly different definition of CPM (Definition 2.6) in their paper on the Leiden algorithm [16]. In the rest of my thesis, I will stay in line with the Leiden paper and use Definition 2.6 for CPM.

There are three changes to the original definition. First, there is no negative sign in front, hence it is desirable to maximise this version of CPM. Second, each edge is only counted once. Third, the underlying model does not count self loops as possible edges.

Contrary to modularity, CPM does not scale its value with a factor of $\frac{1}{m}$. Hence, its value range is not bounded to a constant value, but depends on the number of edges and nodes in the graph. For example, consider a graph $G = (V, \emptyset)$ with $|V| = n$ nodes and no edges. For $\gamma = 1$, the trivial partition $\mathcal{P} = \{V\}$ has CPM score of:

$$Q(\mathcal{P}) = \sum_{C \in \mathcal{P}} E(C) - 1 \cdot \binom{|C|}{2} = 0 - \binom{|V|}{2} = -\frac{n \cdot (n-1)}{2}.$$

Claim 2.7. Let $G = (V, E)$ be a graph and let $E(R, T)$ denote the number of edges going from R to T for any pair of disjoint subsets of nodes $R, T \subseteq V$. If a community $C \in \mathcal{P}$ has a partition $R, T \subset C$, for which it is better to split the community into R and T, i.e. CPM increases by splitting C into R and T. Then, the following holds:

$$\frac{E(R,T)}{|R| \cdot |T|} < \gamma.$$

Proof. Let $G = (V, E)$ be a graph, $\mathcal{P}$ be a partition. Let $C \in \mathcal{P}$ be a community with subsets $R, T \subset C$ s.t. $R \cup T = C$ and $R \cap T = \emptyset$. Consider the case that CPM increases if C is split into R and T, i.e.

$$E(C) - \gamma \binom{|C|}{2} < E(R) - \gamma \binom{|R|}{2} + E(T) - \gamma \binom{|T|}{2}$$

Note that the number of edges in C equals the number of edges in R and T plus the number of edges going from R to T:

$$E(C) = E(R) + E(T) + E(R, T).$$

This implies

$$E(R, T) - \gamma \binom{|C|}{2} < -\gamma \left(\binom{|R|}{2} + \binom{|T|}{2} \right).$$

Also note that because R, T partition C, their sizes sum to:

$$|R| + |T| = |C|.$$

Consider the difference:

$$\begin{aligned}
\gamma\binom{|C|}{2} - \gamma\left(\binom{|R|}{2} + \binom{|T|}{2}\right) &= \gamma\left(\binom{|C|}{2} - \binom{|R|}{2} - \binom{|T|}{2}\right) \\
&= \gamma\frac{1}{2}\left(|C|(|C|-1) - |R|(|R|-1) - |T|(|T|-1)\right) \\
&= \gamma\frac{1}{2}\left(|C|^2 - |C| - |R|^2 + |R| - |T|^2 + |T|\right)\big) \\
&= \gamma\frac{1}{2}\left(|C|^2 - |R|^2 - |T|^2\right)\big) \\
&= \gamma\frac{1}{2}\left((|R| + |T|)^2 - |R|^2 - |T|^2\right)\big) \\
&= \gamma\frac{1}{2}\left(|R|^2 + |T|^2 + 2|R|\cdot|T| - |R|^2 - |T|^2\right)\big) \\
&= \gamma\frac{1}{2}\left(2|R|\cdot|T|\right) \\
&= \gamma\cdot|R|\cdot|T|
\end{aligned}$$

Hence,

$$E(R,T) < \gamma\cdot|R|\cdot|T|$$

which in turn implies the statement

$$\frac{E(R,T)}{|R|\cdot|T|} < \gamma.$$

□

The factor $\gamma \in [0,1]$ works as a threshold between the edge density inside a community and between communities. Claim 2.7 shows that it increases CPM to split a community if there is a partition of it with a edge density lower than γ between the two parts of the partition.

Like modularity, CPM can also be applied to weighted graphs. In this case, $E(C)$ does not denote the number of edges in the community C, but the total weight of the edges in community C. Note that in the weighted case, the proof of Claim 2.7 still holds as $E(C) = E(R) + E(T) + E(R,T)$ is still true in the weighted case.

2.3 Louvain

This section closely follows the work publishing the Louvain algorithm of Blondel, Guillaume, Lambiotte and Lefebvre [2]

Louvain is a graph clustering algorithm trying to maximise the modularity Q of the clustering $\mathcal{P}$. At its core it is a greedy algorithm. It was developed by Blondel, Guillaume, Lambiotte and Lefebvre and is named after the location of its authors, a city in Belgium. It was published in 2008 and was and still is one of the most widely used algorithms in its class.

Louvain takes as input an undirected graph $G = (V, E)$, allowing for self loops and multi-edges. Furthermore, it may take the weights of the edges, if none are given, weights w are assumed to be 1 for all edges: $w(x, y) = 1$ for all $x, y \in V$.

One key advantage of Louvain is that it does not need a number of communities to find as an input. This allows users to apply it to graphs without knowing anything about the structure it contains.

The communities Louvain finds are returned in a hierarchical structure. This allows users to dive deeply into individual communities and recover significant substructures that might not be visible on the highest level.

Louvain can take optional parameters depending on the implementation. One such value is a resolution parameter for calculating the modularity. It may also take a threshold value. If this is not given, Louvain runs until there is no increase in modularity any more. If it is given, Louvain only runs until the change in modularity gets smaller than the threshold. This can limit the diminishing returns of a longer running time.

The general procedure of the algorithm goes as follows:

1. Start with the singleton partition, i.e. each node in its own community.
2. Repeat until no further increase in modularity happens (or the increase falls below the threshold value):
   (a) Modularity optimisation sequence (MOS): look at each node and move it to the community which results in the biggest increase in modularity (this is the greedy part of the algorithm).
   (b) Community aggregation sequence (CAS): create a new graph based on the clustering provided by the previous MOS (this is where the hierarchical structure of the clustering comes from).
3. Return the clustering of the graph.

One iteration of MOS and CAS is called a pass. Each pass works on the output of the previous pass. Passes are made until no increase in modularity happens from one pass to the other. This is equivalent to saying that passes are made until no change in the clustering happens. To see this, note that on the one hand, the algorithm moves a node only if it leads to a strict increase in modularity, hence if the modularity did not increase, there could not have been a movement of a node. On the other hand, if there was no change in the community, the modularity also stays the same.

2.3.1 Modularity Optimisation Sequence

The aim of the modularity optimisation sequence (MOS) is to improve the modularity score of the current clustering of the graph. Roughly, this is done by iterating through all nodes of the graph repeatedly and moving the considered node to the community which brings the largest increase in modularity. A node is not moved if there is no community to move it to that would bring a strict increase in modularity. The MOS stops when an entire check of all nodes happens which does not result in any node movement.

The order in which the nodes are considered can have a big effect on the run time of the algorithm. A priori Louvain has no way to decide on a good or even optimal ordering. Hence, it uses a random ordering.

When looking at a node i, only the communities of i's neighbours are considered, as moving a node to a community from which it is disconnected would not bring an increase in modularity. Furthermore, moving node i from its community C to another community D is split into two steps: First it is moved to an isolated community $C' = \{i\}$ and then it is added to the new community D.

To calculate the change in modularity ΔQ when moving node i from community C to community D a couple of things are needed. Without loss of generality, let the graph be weighted. Otherwise assume all weights to be 1. The sum of the weights of the edges incident to node i is denoted k_i. The sum of the weights of edges going from nodes in community C to node i is denoted by $k_{i,in}(C)$. The sum of the weights of all the edges in a community C, i.e. the edges where both ends are in C, is denoted by $\Sigma_{tot}(C)$. This includes the edges incident to node i if $i \in C$ even when considering to remove i from C. Similarly, it does not include the edges incident to node i when considering moving i to community D while $i \notin D$.

Using this notation, removing node i from a community C gives

$$\Delta Q^- = -\frac{k_{i,in}(C)}{2m} - \Big(\frac{\Sigma_{tot}(C) - k_i}{2m}\Big)^2 + \Big(\frac{\Sigma_{tot}(C)}{2m}\Big)^2 - \Big(\frac{k_i}{2m}\Big)^2$$

which has to be calculated once and adding node i to a community D gives

$$\Delta Q^+ = \frac{k_{i,in}(D)}{2m} - \Big(\frac{\Sigma_{tot}(D) + k_i}{2m}\Big)^2 + \Big(\frac{\Sigma_{tot}(D)}{2m}\Big)^2 + \Big(\frac{k_i}{2m}\Big)^2$$

which has to be calculated for each of the communities of i's neighbours. Hence the change in modularity when moving node i from communities C to D is

$$\Delta Q = \Delta Q^+ - \Delta Q^-.$$

Note that $k_{i,in}(C)$ and $k_{i,in}(D)$ can be computed locally because only the neighbours of i have to be considered, and $\Sigma_{tot}(C)$ and $\Sigma_{tot}(D)$ can be stored and only updated if a node changes communities. Furthermore, k_i and m are constants for the whole running of the algorithm and hence can be computed once and then stored for further use.

A node switches communities only if $\Delta Q^+ > \Delta Q^-$. If they are equal or $\Delta Q^+ < \Delta Q^-$, the node stays in its own community.

Once the algorithm detects that there is no possibility to increase modularity by moving individual nodes, the MOS ends and the community aggregation sequence (CAS) starts. Louvain detects this by checking if an entire run of all nodes has occurred and not a single change happened.

2.3.2 Community Aggregation Sequence

The motivation for the community aggregation sequence (CAS) is as follows. No improvement to the partition $\mathcal{P}$ of the graph G is produced by the MOS can be made by moving individual nodes. Instead, one can still try to increase the modularity by merging entire communities. To allow that a new weighted graph G' is constructed. G' is weighted even if G was unweighted.

Each community $C \in \mathcal{P}$ in the partition produced by the previous MOS becomes a node in the new graph. To each node in the new graph, a self-loop ℓ is added. The weight of the loop, denoted $w(\ell, \ell)$, is set to the sum of the weights of the edges inside the community:

$$w(\ell, \ell) = \sum_{x,y \in C} w(x, y).$$

If G is unweighted, $w(\ell, \ell)$ is twice the number of edges inside the community, as in the sum above, each edge weight is summed twice: once as $w(x, y)$ and once as $w(y, x)$. An edge between two nodes i and j is added, if there are edges in G between the corresponding communities C and D in $\mathcal{P}$. The weight of this edge is the sum of the weights of the edges between C and D if G is weighted, else it is the number of edges between the communities

$$w(i, j) = \sum_{x \in C, y \in D} w(x, y).$$

Note that even if Louvain gets as input an unweighted graph, after the first pass (i.e. one run of MOS and CAS) it will work with a weighted graph. This is not a problem as Louvain treats any unweighted graph as a weighted graph with weight 1 for all edges. Furthermore, note that one does not have to consider the case that there are no edges between two communities, because to Louvain a edge with weight zero is the same as having no edge at all.

2.3.3 Summary

To summarise and visualise how Louvain works, consider Figure 1. It starts out on the input shown in the bottom left. The input is an unweighted graph of 16 nodes. Note that in the two top left graphs, the numbers represent node id's while in the other two graphs they represent edge weights. The first modularity optimisation sequence clusters the nodes in 4 communities. These get aggregated in the CAS.

As there were 4 communities found by the MOS, there are 4 nodes in the new graph. Consider the edge of weight 3 connecting the red and light blue nodes in the new graph. In the original graph there were three edges going from the red community to the light blue community ($(8, 11), (10, 11), (10, 13)$) hence in the new graph the edge going from the red node to the light blue node has weight 3.

Furthermore, there is only one edge inside the light blue community going from node 11 to node 13 and the self loop on the light blue node after the CAS

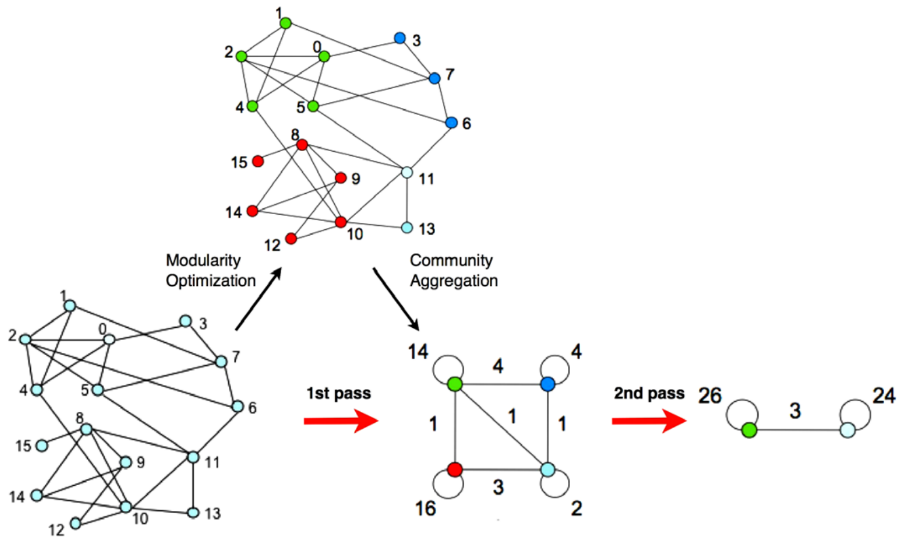


Figure 1: Overview of how Louvain works [2]

has weight 2. This edge gets counted twice, once going from 11 to 13 and once going from 13 to 11.

This is one example where Louvain starts out working on an unweighted graph and after the first pass works with weighted graphs. In the second pass, the green and the dark blue community get merged and the red and the light blue community get merged. This also highlights how there exists a hierarchy of clusterings. On the second level, nodes 10 and 13 are in the same community, on the first they are in different communities.

2.3.4 Discussion

The fact that ΔQ can be computed locally, while modularity is a global property, makes Louvain feasible in the first place. It cannot be highlighted enough how important that is. Every time a node i is considered, for each community of i's neighbours ΔQ has to be calculated, hence at most the degree k_i of i times.

The maximum number of passes the algorithm possibly has to make is the number of nodes the input graph has. To show this, note that the algorithm stops if there is no change between two passes. Hence, in the worst case, at least one node has to be moved. Each Modularity Optimisation Sequence (MOS) starts with the nodes in the singleton partition. At this point, moving one node surely reduces the number of clusters by one. The algorithm starts the first MOS with a community for each node $i \in V$ in the input graph $G = (V, E)$, hence Louvain ends after at most $|V| = n$ passes.

Experiments show that only a low number of passes is needed to achieve stability. Most of the work is done in the first MOS which also takes most of

	Karate	Arxiv	Internet	Web nd.edu	Phone	Web uk-2005	Web WebBase 2001
Nodes/ links	34/77	9k/24k	70k/351k	325k/1M	2.04M/5.4M	39M/783M	118M/1B
CNM	0.38/0 s	0.772/3.6 s	0.692/799 s	0.927/5034 s	—/—	—/—	—/—
PL	0.42/0 s	0.757/3.3 s	0.729/575 s	0.895/6666 s	—/—	—/—	—/—
WT	0.42/0 s	0.761/0.7 s	0.667/62 s	0.898/248 s	0.553/367 s	—/—	—/—
Our algorithm	0.42/0 s	0.813/0 s	0.781/1 s	0.935/3 s	0.76/44 s	0.979/738 s	0.984/152 mn

Figure 2: Summary of the experiments run by Blondel et al. [2]. Rows represent different algorithms, "Our algorithm" is Louvain, the data in the cells is the modularity score / running time. CNM [4], PL [11] and WT [17] are other clustering algorithms. Empty cells mean a running time of more than 24 hours.

the time. Hence, later passes do not add significantly to the run time [2].

Figure 2 summarises the experiments run by Blondel et al [2]. It shows that Louvain can reach higher modularity scores while running several orders of magnitude faster. This enormous speed-up opens up the possibility to cluster Graphs that before were too large and would run for too long. Even 2.5 hours for clustering a graph with 118 million nodes and one billion edges is rather fast compared to the larger than the 24 hour cut-off of the other algorithms. When looking at the `Web nd.edu` graph, Louvain reaches only a marginally higher modularity score than the CNM algorithm, but it is 1678 times faster than CNM.

One issue is the first run of MOS. Here, a lot of time is spent checking if nodes can be moved to other communities, even though there was no change in their neighbourhood. Which is redundant and causes time to be wasted. This issue was later addressed by Leiden [16].

One other of the main issue of Louvain is that it can cluster arbitrarily badly connected communities and even disconnected communities [16]. This can happen because if a community becomes disconnected in the MOS, it cannot split up after a CAS because it gets reduced to a single node. To see how the MOS might create a disconnected community, consider Figure 3. Node 0 is the the connecting node of the two parts in the red community. As this is much strongly connected to the rest of the network, it will be moved to the blue community. Now the red community is disconnected. Louvain cannot split up the red community in two, as it moves only single nodes. Moving any single node of the red community results in a decrease of modularity, hence it will not happen. So the community remains disconnected. This also happens with significant frequency in real world graphs [16]. These two issues led to the development of Leiden.

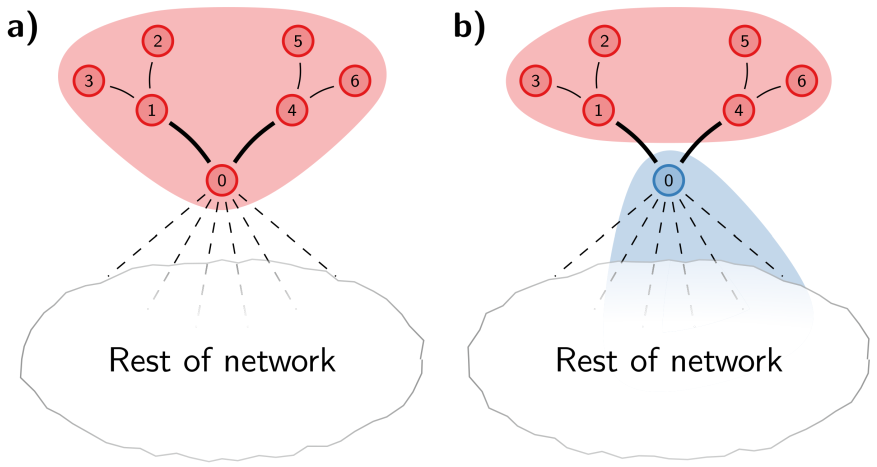


Figure 3: Example of a graph where Louvain clusters a disconnected community. The thickness of the edges represents the weight. The number in the node is their ID. The background shading represents the community nodes belong to [16].

2.4 Leiden

This section follows closely the work of Traag, Waltman and van Eck [16].

Leiden is a graph clustering algorithm trying to maximise the modularity of the partition. It was proposed by Traag, Waltman and van Eck and is, like Louvain, named after the location of its authors. In general, Leiden works quite similarly to Louvain, as it solves the same problem, has similar input/output interfaces, but does that with a few refinements over Louvain, running faster while returning better results.

Leiden takes as input a graph with weighted or unweighted and undirected edges. It allows for self-loops and multi-edges. As Louvain, if the input graph is unweighted, it will assume all edge weights to be one and after the first aggregation sequence it continues working with a weighted graph.

There is one last optional input: an initial partition of the nodes. Louvain always starts with the singleton partition and by default Leiden does so, too. When a partition is given, Leiden starts the first node movement sequence by initialising the nodes in the given partition. This allows to iterate the algorithm, i.e. let it run with the default singleton partition as a start, and feed the output partition back as an input for another run of the algorithm.

Leiden does not require a number of clusters to find as input, allowing it to be used, as Louvain, on graphs without any prior knowledge and no optimisation for the number of clusters searched.

Leiden returns a clustering of the nodes of the input graph. This is different to Louvain, which returns a hierarchy of clusters.

While Leiden works similarly to Louvain, it refines both modularity optimisation sequence (MOS) and community aggregation sequence (CAS). Each sequence gets a little tweak resulting in significant speed and quality improvements. Furthermore, it introduces a third step, the partition refinement step (PRS) in between, refining the partition found before aggregating the graph.

The general structure of the Leiden algorithms is as follows:

1. If there is a partition given as input, start with that, else start with the singleton partition.

2. Repeat until there is no change in the partition from one iteration to the next:

   (a) `MoveNodesFast` (MNF): Each node is moved repeatedly to the neighbouring community which gives the greatest increase in modularity. This is similar to Louvain's modularity optimisation sequence. MNF returns a partition $\mathcal{P}$.

   (b) `RefinePartition` step (RPS): This is where the biggest difference to Louvain comes from. It is an entire new step taking the partition $\mathcal{P}$ produced in MNF and refining it to improve its quality outputting $\mathcal{P}_{\text{refined}}$. In particular, this is the step that makes sure that communities cannot be disconnected. RPS returns another partition $\mathcal{P}_{\text{refined}}$.

   (c) `AggregateGraph` step (AGS): A new graph G' is created based on the refined partition $\mathcal{P}_{\text{refined}}$ returned from the RPS. On top of that, the partition $\mathcal{P}$ from the MNF is turned into a partition for the new graph G'. This will be the starting partition for the next MNF. Overall, this step is similar to the community aggregation sequence of Louvain.

3. Return clustering of the vertices of the input graph.

A pass is one run of the MNF, RPS and AGS. Each pass creates one level in the hierarchy of clusters. This is very similar to Louvain. One big difference is that Louvain then returns the hierarchy of clusters, while Leiden returns a clustering on the nodes of the original graph. Due to the repeated aggregation of the graph Leiden works on, still a hierarchy of clusters arises. To see how Leiden goes from that hierarchy of partitions to one on the original nodes, I need to introduce the flattening operation.

Definition 2.8 (Flattening [16])**.** The *flattening* operation for a set S is defined as

$$\operatorname{flat}(S) = \bigcup_{s \in S} \operatorname{flat}(s)$$

where $\operatorname{flat}(s) = s$ if s is not a set itself. A set that has been flattened is called a *flat set*.

The flattening operation for a partition $\mathcal{P}$ is defined as

$$\operatorname{flat}^*(\mathcal{P}) = \{\operatorname{flat}(C) \mid C \in \mathcal{P}\}$$

the set of the flattened communities. A partition that has been flattened is called a *flat partition.*

Note the subtle difference between the flattening of a set and the flattening of a partition. Consider any set of sets S. The flattening $\text{flat}(S)$ of a set S keeps "extracting" the elements $s \in S$ until it returns non-set elements. On the other hand, the flattening $\text{flat}(\mathcal{P})$ of a partition $\mathcal{P}$ preserves the structure of the partition. In particular, the flattening of a set is a set of elements, while the flattening of a partition is a set of sets.

If you apply the set flattening to a (possibly hierarchical) partition $\mathcal{P}$ of the nodes in a graph $G = (V, E)$, you loose the partition information and get back the set of nodes V:

$$\text{flat}(\mathcal{P}) = V \neq \mathcal{P}_{\text{flat}} = \text{flat}^*(\mathcal{P}).$$

Coming back to Leiden, the flattening operation for partitions is used by Leiden to turn the hierarchical partition of the nodes V of the input graph into a flat partition. This way Leiden can flatten the hierarchy created by its iterative structure and return a partition on the nodes of the input graph.

Next, I will go into the details of the three steps that make up Leiden: `MoveNodesFast`, `RefinePartition` and `AggregateGraph`.

2.4.1 `MoveNodesFast`

The `MoveNodesFast` (MNF) step tries to maximise the modularity of the clustering of the current graph. This step is quite similar to the modularity optimisation sequence of Louvain (see Section 2.3.1). In the MNF, each vertex is considered repeatedly and moved to the neighbouring community which gives the highest increase in modularity. This is done until no further improvement is possible.

The MNF takes as input two objects. The graph $G = (V, E)$ whose partition it tries to optimise, and an initial partition $\mathcal{P}_{in}$. This will be the starting point. As mentioned before, Leiden takes a partition as an input. This partition is used here in the first pass. If Leiden is given no partition, it starts with the singleton partition as does Louvain. In later passes the MNF will start with an input partition which comes from the `AggregateGraph` step, see Section 2.4.3.

The MNF starts by initializing a queue $\mathcal{Q}$ with all nodes $v \in V$ added to it in random order. The MNF removes a node i from $\mathcal{Q}$ and considers the communities of i's neighbours. Then, it calculates the change in modularity ΔQ for each of the possible communities the same way as in Louvain (see Section 2.3.1). Node i is then moved to the community C' that results in the largest positive change in modularity. If there is no community which results in a positive change in modularity, the node is not moved. Lastly, if the node was moved, all its neighbours which are not already in the queue $\mathcal{Q}$, and are not in i's new community C' are added to the queue:

$$\mathcal{Q}.\text{add}(\{j \in V \mid (i, j) \in E, j \notin Q, j \notin C'\}).$$

The MNF keeps checking nodes until the queue $\mathcal{Q}$ is empty. This is when the MNF ends. This cut-off leads to Leiden having less computational steps than Louvain, as it only adds those nodes to the queue for which the movement of node i might result in a better community for them to be in. On the other hand, Louvain keeps repeatedly checking all nodes until not a single movement occurred. So if one node is moved, Louvain will check again at least all other nodes one more time.

2.4.2 `RefinePartition`

The `RefinePartition` step (RPS) takes the partition $\mathcal{P}$ of the graph $G = (V, E)$ as produced by the MNF and refines it. This makes sure the partition follows certain quality aspects, like connectedness of communities. A community C is connected if the subgraph of G induced by C is connected. The RPS presents the biggest change from Louvain because no similar step is part of Louvain.

The MNF starts by initialising the refined partition $\mathcal{P}_{\text{refined}}$ with the singleton partition on the nodes V:

$$\mathcal{P}_{\text{refined}} = \texttt{SingletonPartition}(V).$$

It then iterates through all the communities $C \in \mathcal{P}$ and refines them. As each node is in one and only one of the communities in the partition $\mathcal{P}$, each node will be looked at most once when its community gets refined.

To explain how the refinement of a single community works, I need to introduce the *recursive size* of a set:

Definition 2.9 (Recursive size [16])**.** The *recursive size* of a set S is defined as

$$\|S\| = \sum_{s \in S} \|s\|,$$

where $\|s\| = 1$ if s is not a set itself.

This makes it possible to count the individual elements, i.e. non-set items, in a set of sets. For example, let $S = \{\{x, y\}, \{y, z\}\}$. Then the recursive set size of S is:

$$\begin{aligned} \|S\| &= \|\{\{x, y\}, \{y, z\}\}\| \\ &= \|\{x, y\}\| + \|\{y, z\}\| \\ &= (\|x\| + \|y\|) + (\|y\| + \|z\|) \\ &= (1 + 1) + (1 + 1) \\ &= 4. \end{aligned}$$

Note the difference between the size of the flattening of a set S (see Definition 2.4) and its recursive set size. The flattening of S is

$$\text{flat}(S) = \{x, y, z\}.$$

Hence the size of the flattening is 3 which is lower than the recursive size of S. This happens because the flattening operation produces a set, hence there cannot be duplicate elements. On the other hand, the recursive size counts each element in each of the sets, hence there may be duplicates. The two might be equal if S does not contain any element more than once, as is the case for the hierarchy of partitions as produced by Louvain. Hence, the recursive set size is an upper bound on the size of the flattening of the set:

$$\|S\| \geq |\mathrm{flat}(S)|.$$

Having introduced the recursive set size, I can now explain the refinement of a single community C. It works as follows: Consider only those nodes in the community C that are well connected to the rest of the community. The set of those nodes is denoted R. R is defined as:

$$R = \{i \mid i \in C, w(i, C \setminus \{i\}) \geq \gamma \cdot \|i\| \cdot (\|C\| - \|i\|)\}.$$

I will now explain what each term means. To decide whether a node i is well connected to the rest of its community, i.e. part of R or not, the threshold d for the weight of edges going from i to the rest of C is calculated:

$$d = \gamma \cdot \|i\| \cdot (\|C\| - \|i\|).$$

The underlying assumption is that one unweighted edge corresponds to an edge of weight 1. Here, the recursive set size is needed, as the algorithm might have done already a number of passes, and the nodes of the original input graph are "buried" in a hierarchy of sets. The product of the recursive size of i and the recursive size of C minus the recursive size of i gives the possible number of edges between i and the rest of the community. This product gets scaled with a factor $\gamma \in [0, 1]$ giving d. When discussing the guarantees of Leiden in Section 3 I will go into more detail what this γ means. Then d gets compared to the actual weight of the edges going from i to the rest of C, denoted $w(i, C \setminus \{i\})$. If there is more than or as expected much weight of edges going from i to the rest of the community, it is deemed well-connected and part of R.

The RPS now considers each of the nodes in R one by one. If node i, which is currently considered, is not any more in a singleton community in $\mathcal{P}_{\text{refined}}$ it will be skipped. If it is still in a singleton community, it means it is up for merging with any of the other communities. Consider the set $\mathcal{T}$ of all communities in the refined partition $\mathcal{P}_{\text{refined}}$ which are a subset of C and are well-connected to the rest of C. This is defined similarly, where $w(D, C \setminus D)$ refers to the weight of the edges going from D to the rest of C:

$$\mathcal{T} = \{D \mid D \in \mathcal{P}_{\text{refined}}, D \subseteq C, w(D, C \setminus D) \geq \gamma \cdot \|D\| \cdot (\|C\| - \|D\|)\}.$$

Node i will be merged randomly to any of the communities in $\mathcal{T}$ which results in a non-negative change in modularity ΔQ. The probability for each community in $\mathcal{T}$ is proportional to the increase in modularity. If node i was moved, $\mathcal{P}_{\text{refined}}$ gets updated accordingly.

Note the crucial difference to Louvain and the MNF of Leiden, instead of moving node i to the community with the highest increase of modularity, it is moved randomly to any one of the communities that does not decrease modularity. This is the reason why Leiden is not strictly greedy. I will discuss the implications of this and why this is necessary in Section 3.1.

After all communities are split up and merged back together into well connected parts, the refined partition is returned by the RPS. This triggers the `AggregateGraph` step.

Note that RPS does not move nodes with the freedom of the MNF step. It can only move a node i into communities that are subsets of the community i was part of, in the partition produced by MNF. Hence, RPS can not move nodes to any other community, it only splits up existing communities along lines of weak connectedness, resulting in well connected communities.

2.4.3 `AggregateGraph`

The `AggregateGraph` step (AGS) takes the results of the previous MNF and RPS and turns them into a new graph for the MNF of the next pass to work on. This is quite similar to the Community Aggregation Sequence of Louvain (see Section 2.3.2). The main difference is that AGS does not use the partition of MNF to create the new graph, but the refined partition of RPS.

AGS creates a new graph $G' = (V', E')$ based on the graph $G = (V, E)$ of this previous pass. Each community C in the refined partition $\mathcal{P}$ becomes a node in the new graph:

$$V' = \{C \mid C \in \mathcal{P}_{\text{refined}}\}.$$

To each node $i \in V'$ in the new graph a self loop ℓ is added. The weight of ℓ is twice the total weight of the edges in community $C \in \mathcal{P}_{\text{refined}}$ corresponding to node i. To each pair of nodes $(i, j) \in V'$ an edge gets added. The weight of that edge equals the sum of the weights of the edges between the communities in $\mathcal{P}_{\text{refined}}$ corresponding to nodes i and j.

Lastly, the AGS takes the original partition $\mathcal{P}$ as produced by the MNF and creates a partition $\mathcal{P}'$ for the new graph G'. Each community in $\mathcal{P}'$ corresponds to a community in $\mathcal{P}$. Nodes in V' are in the same community in $\mathcal{P}'$ if their corresponding communities in $\mathcal{P}_{\text{refined}}$ are subsets of the same community in $\mathcal{P}$. Hence, a community C' of the new partition $\mathcal{P}'$ corresponding to the community $C \in \mathcal{P}$ is of the form:

$$C' = \{C_{\text{refined}} \mid C_{\text{refined}} \subset C, C_{\text{refined}} \in \mathcal{P}_{\text{refined}}\}$$

$\mathcal{P}'$ is then used by the MNF of the next pass as the initial partition.

2.4.4 Summary

To summarise how Leiden works consider Figure 4. Level 1 represents the first pass, Level 2 the second pass. Further passes have been omitted. The thickness

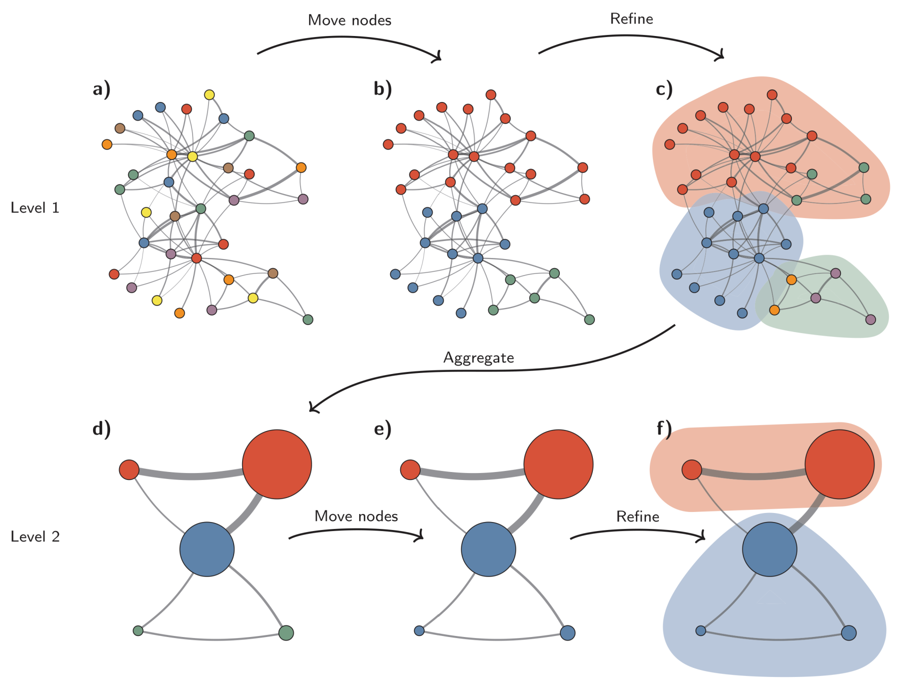


Figure 4: Overview of how Leiden works [16]

of the edges represents their weight, the self loops are represented by the size of the nodes. The input graph is represented in a). Leiden starts with a singleton partition. After the first MNF 3 communities arise as shown in b). The node colour represents the community of the node. Then, the red and the green communities get split up by the RPS into two smaller communities each, while the blue community is not split up by the RPS. The background shading in c) represents the partition as produced by the MNF. Graph c) gets aggregated by the AGS and results in graph d). Note that the two top nodes are both coloured red, i.e. in the same community, and the two bottom nodes are both coloured green, representing the green community from the first MNF. This is then the input graph and partition for the second pass of the algorithm.

2.4.5 Improvements on Louvain

The most obvious drawback of Louvain was that it could create disconnected communities. This gets addressed by Leiden in the `RefinePartition` step (RPS). Badly connected, in particular disconnected, communities get split up into (well-)connected components.

The RPS also introduces a step which is not strictly greedy. When refining a community C, C is first split up completely into the singleton partition. Then, the nodes in the community C get merged. A node considered for merging is not merged greedily to the community that gives the highest increase in modularity. Instead, it is merged randomly to one of the communities that results in a non-negative change in modularity. The probability for each of the communities is proportional to the increase of modularity. Hence, a node is more likely to be merged into a community that gives a higher yield. By avoiding a strictly greedy merging and implementing a non-decreasing modularity merging allows Leiden to reach optimal partitions. I will discuss in detail why this is necessary and how this is achieved in Section 3.

Another advantage of Leiden over Louvain is that Leiden does the movement of nodes more efficiently by reducing the number of times the `MoveNodesFast` (MNF) of Leiden checks if it can move a node. While Louvain keeps checking *all* nodes until it encounters an iteration over all nodes where not a single node moves, Leiden keeps track of the nodes that have to be checked. A node has to be checked again only if its neighbourhood has been modified. Hence, if a node is moved, its neighbours outside its new community have to be checked again. These get added to the queue of nodes to check, while not adding any node that is already in the queue.

Furthermore, after the first pass, the following `MoveNodesFast` (MNF) does not start from the singleton partition, as does Louvain. Instead, it starts from a partition based on the result of the previous MNF. Consider a community which got split up in the RPS, all parts are turned into nodes of the aggregated graph for the next part. These nodes are put in one community. If a community did not get split up in the refinement step, it is put into its own community.

Leiden provides a number of guarantees which Louvain does not. I will discuss this in detail in Section 3.

3 Guarantees of Leiden

Leiden provides a number of guarantees which are summarised in Table 1. I will define and prove these in this section. Furthermore, I will formalize and discuss the implications of the greediness of Louvain and compare that to Leiden. This section follows closely the supplementary information to the paper presenting Leiden by Traag et al. [16].

While Louvain and Leiden are published with modularity as the quality score they try to maximise, Traag et al. provide the proofs of the guarantees based on the Constant Potts Model (CPM). See Section 2.2 for its definitions and discussion. Note that CPM takes a threshold parameter $\gamma \in [0, 1]$ (see Definition 2.6). This is the same γ that the guarantees depend on.

Traag et al. claim that the proofs can be extended to modularity by re-defining the base case of the recursive set size (see Definition 2.4.2). Instead of $\|i\| = 1$ if i is not a set itself, set $\|i\| = k_i$ to the degree k_i of node i.

As I explained before, it is possible to iterate Leiden, i.e. using the output partition from one run of Leiden as input for another run of Leiden. Louvain can be slightly altered to also take an input partition, hence opening the possibility for iterating Louvain. This is done by Traag et al. [16].

Table 1: Overview of guarantees of Louvain and Leiden [16].

When	Guarantee	Louvain	Leiden
Each iteration	γ-separation	✓	✓
	γ-connectivity		✓
Stable iteration	Node optimality	✓	✓
	Subpartition γ-density		✓
Asymptotically stable iteration	Uniform γ-density		✓
	Subset optimality		✓

The guarantees of Leiden and Louvain come at different points in time when iterating the algorithm. When a guarantee occurs at "each iteration", it is true for any output of the algorithms. A "stable iteration" is one where the output partition of the algorithm equals the input it got, i.e. the algorithm did not change the partition

$$\mathcal{P} = \texttt{Leiden}(G, \mathcal{P}).$$

A stable iteration for Leiden does not mean that the next iteration will also be stable. This arises due to the randomness in the `RefinePartition` step. But, there will be a time when Leiden cannot make any further adjustments to a partition (see Lemma 3.1). This is called "asymptotically stable iteration". Any asymptotically stable iteration is also a stable iteration.

Lemma 3.1 ([16])**.** Let $G = (V, E)$ be a graph, $\mathcal{P}_0$ be a flat partition of G and $\mathcal{P}_{t+1} = \texttt{Leiden}(G, \mathcal{P}_t)$ represent the sequence of partitions arising from iterating Leiden. There exists a time τ such that $\mathcal{P}_\tau = \mathcal{P}_t$ for all $t \geq \tau$.

3.1 Greediness

Both Louvain and Leiden move nodes greedily during their node movement parts, the Modularity Optimization Sequence (MOS) for Louvain and `Move NodesFast` (MNF) for Leiden. As Louvain does not move nodes in any other step of the algorithm, it makes it a greedy algorithm. While Leiden also moves nodes greedily in the MNF, it does not move them greedily during the `RefinePartition` step (RPS). To discuss the implications of this let me introduce some notation and definitions.

Notation. For a graph $G = (V, E)$, a node $i \in V$, a partition $\mathcal{P}$ of G, and a community $C \in \mathcal{P}$, let

$$\mathcal{P}(i \mapsto C)$$

denote the partition that results from moving node i to the community C. Note that $\mathcal{P}(i \mapsto C) = \mathcal{P}$ if $i \in C$. Furthermore, let

$$Q(\mathcal{P})$$

denote the quality score Q of a partition $\mathcal{P}$. Lastly, let

$$\Delta Q(P(i \mapsto C))$$

denote the change in quality score arising from moving node i to community C starting from the partition $\mathcal{P}$. Here $C \in \mathcal{P} \cup \{\emptyset\}$ may be any community in $\mathcal{P}$ or an empty community. It means moving i to a new community. This can be generalised to a set $S \subset V$ of nodes: $\mathcal{P}(S \mapsto C)$ and $\Delta Q(P(S \mapsto C))$.

Definition 3.2 (Move sequence [16]). Let $G = (V, E)$ be a graph, and let $\mathcal{P}_0, \dots, \mathcal{P}_\tau$ be a sequence of partitions of G. A sequence of partitions $\mathcal{P}_0, \dots, \mathcal{P}_\tau$ is called a *move sequence* if for each time $t = 0, \dots, \tau - 1$ there exists a node $i \in V$ and a community $C \in \mathcal{P}_t \cup \{\emptyset\}$ such that $\mathcal{P}_{t+1} = \mathcal{P}_t(i \mapsto C)$.

Definition 3.3 (Non-decreasing move sequence [16]). A move sequence $\mathcal{P}_0, \dots, \mathcal{P}_\tau$ is called *non-decreasing* if the quality score Q of the partitions $\mathcal{P}_i$ does not decrease from one step to the next, i.e. for all $t = 0, \dots, \tau - 1$:

$$Q(\mathcal{P}_{t+1}) \geq Q(\mathcal{P}_t).$$

Definition 3.4 (Greedy move sequence [16]). A move sequence $\mathcal{P}_0, \dots, \mathcal{P}_\tau$ is called *greedy* if in each step $t = 0, \dots, \tau - 1$ the node i which is moved, is moved to a community which maximises the change in quality score:

$$\mathcal{P}_{t+1} = \mathcal{P}_t(i \mapsto C) \text{ s.t. } C = \arg \max_{D \in \mathcal{P}_t \cup \emptyset} \Delta Q(P_t(i \mapsto D)).$$

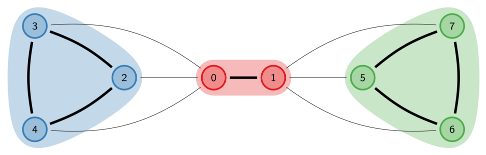
Note that any greedy move sequence is also non-decreasing, as a node can always be left in its own community, resulting in no change in quality score. Also note that whether a move sequence depends on the order of nodes moved. If at time t a different node is moved, the result of the arg max may be different.

Having defined greedy and non-decreasing move sequences, I can now say that each pass of Louvain follows a greedy move sequence. This move sequence starts with the singleton partition and ends with the output partition. On the other hand, each pass of Leiden produces a non-decreasing move sequence. It starts with the input partition, if given, else with the singleton partition. It ends with the output partition.

There are graphs for which a greedy move sequence cannot reach an optimal partition. Figure 5 shows such a graph. Any greedy move sequence reaches the partition shown in a). Greediness always moves nodes 0 and 1 together in one community because the edge between them has more weight than the edges going from 0 to the blue community or from 1 to the green community. Reaching the optimal partition, as shown in b), would require that 0 and 1 are not in the same community. This shows that Louvain cannot guarantee reaching the optimal partition.

On the other hand, Leiden does not produce a greedy move sequence. It produces a non-decreasing move sequence. Theorem 3.5 shows that any optimal partition can be reached by a non-decreasing move sequence. This means that Leiden can always reach an optimal partition. Leiden also does this eventually as I will discuss in Section 3.6. Note that Theorem 3.5 does not imply Leiden reaches an optimal partition in polynomial time.

a)



b)

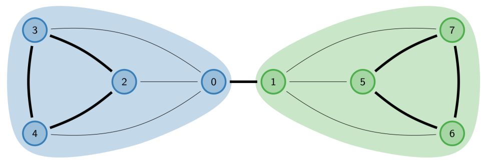


Figure 5: This graph is an example where a greedy move sequence optimising CPM with resolution $\gamma = 1$ cannot reach the optimal partition Subfigure b). The thick edges have weight 3, the thin have weight $\frac{3}{2}$. The number in the nodes are their ID. Subfigure a) shows the partition that is reached by any greedy move sequence and Subfigure b) shows the optimal partition [16].

Theorem 3.5 (Optimal move sequence [16])**.** Let $G = (V, E)$ be a graph, and let $\mathcal{P}^*$ be an optimal partition of G. Then there exists a non-decreasing move sequence $\mathcal{P}_0, \ldots, \mathcal{P}_\tau$ which starts with the singleton partition $\mathcal{P}_0 = \{\{i\} \mid i \in V\}$ and ends with the optimal partition $\mathcal{P}_\tau = \mathcal{P}^*$ and has length $\tau = |V| - |\mathcal{P}^*|$.

Proof. This is proven by Traag et al. in the supplementary information to the Leiden paper [16]. Let $\mathcal{P}^*$ be an optimal partition of a graph $G = (V, E)$ and $\mathcal{P}_0, \ldots, \mathcal{P}_\tau$ be a move sequence starting at the singleton partition

$$\mathcal{P}_0 = \{\{i\} \mid i \in V\}.$$

To see how to construct the move sequence for a single community, consider any $C \in \mathcal{P}^*$. Chose any node $i \in C$ as the starting point. Then move each of the other nodes $j \in C \setminus \{i\}$ into the community of i. Why is this always possible without decreasing CPM? Traag et al. prove this by contradiction.

Assume there is a time $t = 0, \ldots, \tau - 1$ in the move sequence $\mathcal{P}_0, \ldots, \mathcal{P}_\tau$ where there exist only nodes i for which moving i to the current state $C \in \mathcal{P}_{t-1}$ of its optimal community $C^* \in \mathcal{P}^*$ decreases CPM:

$$\Delta Q(\mathcal{P}_{t-1}(i \mapsto C)) < 0. \tag{1}$$

Let $S = C^* \setminus C$ be the nodes that still have to be moved to C to produce the optimal community C^* at time t. Inequality 1 implies that for all $i \in S$ there is less weight of the edges going from i to C than the model of CPM expects it to. By Claim 2.7:

$$w(i, C) < \gamma \cdot \|i\| \cdot \|C\| \, .$$

Remember that

$$\|S\| = \sum_{i \in S} \|i\| .$$

is the definition of the recursive set size. Hence, the sum over all $i \in S$ gives

$$w(S, C) = \sum_{i \in S} w(i, C) < \sum_{i \in S} \gamma \cdot \|i\| \cdot \|C\| = \gamma \cdot \|S\| \cdot \|C\| .$$

On the other hand, by optimality of C^*,

$$w(S, C) \geq \gamma \cdot \|S\| \cdot \|C\| .$$

Otherwise C would already be an optimal community. Hence there is a contradiction, and it is always possible to move a node to its desired community without decreasing CPM.

After $|C| - 1$ steps the community C exists in the current partition and is in its desired form. Construct the entire move sequence by joining the sequences for each of the communities $C \in \mathcal{P}^*$. Hence, this move sequence will start at the singleton partition and end at the optimal partition $\mathcal{P}^*$.

Note that not all nodes have to be moved because the communities don't start out empty, but each community C of the optimal partition $\mathcal{P}^*$ already has a node in it from the start. Hence $|\mathcal{P}^*|$ many nodes do not have to be moved. The rest of the nodes have to be moved once. Hence the length of the move sequence is:

$$\tau = \sum_{C \in \mathcal{P}^*} |C| - 1 = |V| - |\mathcal{P}^*|.$$

□

3.2 γ-Separation

Definition 3.6 (γ-separation [16]). A pair of communities $C, D \in \mathcal{P}$ of a partition $\mathcal{P}$ is *γ-separated* if merging the two communities does not increase the CPM score Q:

$$\Delta Q(\mathcal{P}(C \mapsto D)) \leq 0.$$

Note that $\mathcal{P}(C \mapsto D) = \mathcal{P}(D \mapsto C)$. Hence $\Delta Q(\mathcal{P}(C \mapsto D)) = \Delta Q(\mathcal{P}(D \mapsto C))$. A community $C \in \mathcal{P}$ is γ-separated if C is γ-separated with respect to all $D \in \mathcal{P}$. A partition $\mathcal{P}$ is γ-separated if all communities $C \in \mathcal{P}$ are γ-separated.

The intuition to γ-separation links back to Claim 2.7. Let $C, D \in \mathcal{P}$ be two γ-separated communities. Hence, merging them would not increase the modularity. By Claim 2.7 this means that

$$\frac{E(C, D)}{|C| \cdot |D|} \leq \gamma.$$

Hence the density of nodes between the two communities is lower than γ.

Theorem 3.7. Let $G = (V, E)$ be a graph and $\mathcal{P}_t$ be a flat partition of G. Then $\mathcal{P}_{t+1} = \texttt{Leiden}(G, \mathcal{P}_t)$ is γ-separated.

This theorem is proven by Traag et al. [16], I will explain the main idea behind it. Recall that Leiden aggregates the graph it works on in each pass. It stops when the `MoveNodesFast` (MNF) does not move any nodes. Hence, no increase in CPM can be achieved by moving a node in the graph at the current level of aggregation. A node in the aggregate graph represents a community in the output partition. Hence the previous statement is equivalent to saying merging two communities does not increase, i.e. they are γ-separated.

3.3 γ-Connectivity

Definition 3.8 (γ-connectivity [16])**.** A set of nodes S, subset of a community $S \subseteq C \in \mathcal{P}$, is γ-*connected* if $\|S\| = 1$ or S can be partitioned into two sets R and T such that $E(R, T) \geq \gamma \cdot \|R\| \cdot \|T\|$ and R and T are γ-connected.
A community $C \in \mathcal{P}$ is γ-connected if $S = C$ is γ-connected.
A partition $\mathcal{P}$ is γ-connected if all communities $C \in \mathcal{P}$ are γ-connected.

Due to the recursive nature of the definition of γ-connectivity it is not straight forward to give an intuition for it. In a way, it is the opposite of γ-separation. γ-separation requires for a pair of communities, i.e. a disjoint pair of sets of nodes, that

$$\frac{E(C, D)}{\|C\| \cdot \|D\|} \leq \gamma.$$

γ-connectivity requires for a disjoint part of sets of nodes that

$$\frac{E(C, D)}{\|C\| \cdot \|D\|} \geq \gamma.$$

Hence, for a γ-connected community C that splits into R and T. The converse of Claim 2.7 implies that CPM does not increase if community C is split into R and T.

Consider the extreme case of $\gamma = 0$. Any subset S of nodes of any graph is 0-connected. This holds as $E(R, T) \geq 0$ is a non-negative function. Hence, any partition R, T of S will have 0 or more edges going from R to T.

On the other hand, any complete graph K_n is 1-connected. This holds as any partition into set R, T of K_n has the maximum number of edges between them, hence $E(R, T) = \|R\| \, \|T\|$.

To show the limitations of γ-connectivity let me introduce dumbbell graphs.

Definition 3.9 (Dumbbell graph)**.** Let $n \in \mathbb{N}$ be a natural number. A *dumbbell graph* D_n is a graph that consists of two complete graphs K_n of n nodes that are connected by a single edge.

Claim 3.10 (Dumbbell graph is $\frac{1}{3}$-connected)**.** Let $n \in \mathbb{N}$. Let D_n be a dumbbell graph, then D_n is $\frac{1}{3}$-connected.

Proof. I will do this proof by induction on n, the size of the complete graphs K_n that make up D_n. Let $n = 1$. D_1 is a graph with two nodes and edge between them, hence a complete graph of size 2. It can be partitioned into two subsets R and T each having one node in them, and one edge between them. Hence,

$$E(R, T) = 1 \geq \frac{1}{3} = \frac{1}{3} \cdot 1 \cdot 1 = \gamma \left\|R\right\| \left\|T\right\|$$

This shows that D_1 is $\frac{1}{3}$-connected. In fact it is 1-connected.

Let $n \geq 2$ and assume D_{n-1} is $\frac{1}{3}$-connected. Let x, y be the two nodes that are the endpoints of the edges that connects the two K_n that make up D_n.

Set R to one node from one of the K_n, but neither x nor y. Set T to the rest of the nodes. R has $n-1$ edges going to the rest of the K_n it was taken from, and none to the other. Hence

$$E(R, T) = n - 1 \geq \frac{2}{3}n - \frac{1}{3} = \frac{1}{3} \cdot 1 \cdot (2n - 1) = \gamma \left\|R\right\| \left\|T\right\|.$$

So there is a partition of D_n into R, T where $\|R\| = 1$ and $E(R, T) \geq \gamma \cdot \|R\| \cdot \|T\|$. It remains to show that T is $\frac{1}{3}$-connected.

T is made up of one K_n and one K_{n-1}. Set R' to one node from the K_n which is neither x nor y. Set T' to the rest of nodes. Note that $T' = D_{n-1}$. Again, there are $n-1$ edges going from the node in R to the rest of the K_n and none to the K_{n-1}. Hence,

$$E(R', T') = n - 1 \geq \frac{2}{3}n - \frac{2}{3} = \frac{1}{3} \cdot 1 \cdot (2n - 2) = \gamma \left\|R\right\|' \left\|T\right\|'.$$

So there is a partition of T into two subsets R', T' where the edge threshold is met, $\|R\| = 1$ and by induction assumption $T' = D_{n-1}$ is $\frac{1}{3}$-connected. Hence, T is $\frac{1}{3}$-connected. Hence, D_n is $\frac{1}{3}$-connected. ☐

Claim 3.10 shows an extreme case of γ-connectivity where there is only one partition of the set of nodes with high enough edge density between them. Especially the edge density between the two K_n that make up a dumbbell graph can be arbitrarily low (namely $\frac{1}{n^2}$) while it is still $\frac{1}{3}$-connected.

Theorem 3.11 (γ-connectivity [16])**.** Let $G = (V, E)$ be a graph, let $\mathcal{P}$ be a flat partition of G, and let $\mathcal{P}' = \texttt{Leiden}(G, \mathcal{P})$ be the output of the Leiden algorithm. Then $\mathcal{P}'$ is γ-connected.

Traag et al. prove this inductively over ℓ, the pass number. They show that each community in the partition as produced by the pass is γ-connected. The part of Leiden that ensures γ-connectivity is the `RefinePartition` step (RPS). The RPS makes sure that only well-connected communities are produced. This also formalizes what "well-connected" means: γ-connected.

3.4 Subpartition γ-Density

Definition 3.12 (Subpartition γ-density [16])**.** A set of nodes S, subset of a community $S \subseteq C \in \mathcal{P}$, is *subpartition γ-dense* if the following two conditions are satisfied:

1. Moving S to its own community does not increase the CPM score:
$$\Delta Q(\mathcal{P}(S \mapsto \emptyset)) \leq 0,$$
2. $\|S\| = 1$ or S can be partitioned into two sets R and T such that $E(R, T) \geq \gamma \cdot \|R\| \cdot \|T\|$ and R and T are subpartition γ-dense.

A community $C \in \mathcal{P}$ is subpartition γ-dense if $S = C$ is subpartition γ-dense. A partition $\mathcal{P}$ is subpartition γ-dense if all communities $C \in \mathcal{P}$ are subpartition γ-dense.

Subpartition γ-density is an expansion on γ-connectivity. When partitioning a community C into the two sets R and T, splitting the community C into R and T must not increase the quality function. Furthermore, this must also hold recursively for R and T. This shows how subpartition γ-density directly implies γ-connectivity. Next, I will explore if the converse holds as well

Consider a γ-connected community C. When splitting C into subsets R and T as specified for γ-connectivity, Claim 2.7 implies that moving R or T to their own community does not decrease CPM. Hence Condition 1 of the definition of subpartition γ-density is satisfied. Due to the recursive nature of subpartition γ-density, this must also hold for the partitions of R and T. γ-connectedness of C implies γ-connectedness of R as well, hence there is a partition of R into R' and T' with
$$E(R', T') \geq \gamma \, \|R'\| \, \|T'\| \, .$$
This does not imply that moving R' or T' out of C into their own communities does not decrease CPM. Hence, this shows that while it might seem like γ-connectivity implies subpartition γ-density, it does not.

Theorem 3.13 (γ-connectivity [16])**.** Let $G = (V, E)$ be a graph, and $\mathcal{P}$ be a flat partition of G. Let $\mathcal{P}' = \texttt{Leiden}(G, \mathcal{P})$ be the output of Leiden for G and $\mathcal{P}$. If this is a stable iteration, i.e. $\mathcal{P}' = \mathcal{P}$, then $\mathcal{P}'$ is subpartition γ-dense.

Traag et al. prove this by induction on the pass level ℓ. The RPS then makes sure that each community is subpartition γ-dense. The recursiveness arises from the number of passes Leiden does.

Note that subpartition γ-density implies that moving any node to a new community does not increase CPM. This is not the same as node optimality, as discussed below.

3.5 Node optimality

Definition 3.14 (Node Optimality [16])**.** A community $C \in \mathcal{P}$ is *node optimal* if moving any node $i \in C$ to any community $D \in \mathcal{P}$ does not increase the quality score Q:

$$\Delta Q(\mathcal{P}(i \mapsto D)) \leq 0.$$

A partition $\mathcal{P}$ is node optimal if all communities $C \in \mathcal{P}$ are node optimal.

Theorem 3.15 (Node optimality [16])**.** *Let $G = (V, E)$ be a graph, and $\mathcal{P}$ be a flat partition of G. Let $\mathcal{P}' =$ `Leiden`$(G, \mathcal{P})$ be the output of Leiden for G and $\mathcal{P}$. If this is a stable iteration, i.e. $\mathcal{P}' = \mathcal{P}$, then $\mathcal{P}'$ is node optimal.*

Traag et al. prove this by contradiction. Suppose there is a node which can be moved to a better community, i.e. moving it there increases CPM. Then `MoveNodesFast` does so. This is a contradiction to the fact that $\mathcal{P}$ is a stable partition, i.e. the algorithm does not do a movement of nodes. Note that this argument also applies to Louvain.

3.6 Subset Optimality

Definition 3.16 (Subset optimality [16])**.** A community $C \in \mathcal{P}$ is *subset optimal* if moving any subset $S \subseteq C$ of C to a different community $D \in \mathcal{P}$ or a new community $D = \emptyset$ does not increase the quality score Q:

$$\Delta Q(\mathcal{P}(S \mapsto D)) \leq 0.$$

A partition $\mathcal{P}$ is subset optimal if all communities $C \in \mathcal{P}$ are subset optimal.

Definition 3.17. Let $G = (V, E)$ be a graph, $\mathcal{P}_0$ be a flat partition of G and $(\mathcal{P}_t)_{t \in \mathbb{N}}$ be a sequence of partitions defined recursively:

$$\mathcal{P}_{t+1} = \texttt{Leiden}(G, \mathcal{P}_t).$$

This represent the sequence of partitions arising from iterating Leiden. A partition $\mathcal{P}_\tau$ is called *asymptotically stable* if $\mathcal{P}_t = \mathcal{P}_\tau$ for all $t \geq \tau$, i.e. τ is an asymptotically stable iteration of Leiden.

Theorem 3.18 (Subset optimality [16])**.** *Let G be a graph, and let $\mathcal{P}$ be a flat partition of G. If $\mathcal{P}$ is asymptotically stable, then $\mathcal{P}$ is subset optimal.*

As stated above, note that subset optimality is not the same as optimality. Subset optimality is the closest guarantee Leiden has to optimality. Traag et al. prove this by contradiction. Let $\mathcal{P}$ asymptotically stable. If $\mathcal{P}$ were not subset optimal, there would be a move sequence for the RPS to change the partition. The probability of the RPS following that move sequence is non-zero. Hence, the would eventually do so with probability 1. Here the probabilistic part of Leiden is essential.

3.7 Uniform γ-Density

Definition 3.19 (Uniform γ-density [16])**.** A community $C \in \mathcal{P}$ is *uniformly γ-dense* if moving any subset $S \subseteq C$ to a new community does not increase the quality score Q:

$$\Delta Q(\mathcal{P}(S \mapsto \emptyset)) \leq 0.$$

A partition $\mathcal{P}$ is uniformly γ-dense if all communities $C \in \mathcal{P}$ are uniformly γ-dense.

Uniform γ-density is a restriction on subset optimality where a subset of a community can only be moved to a new community, instead of any other community. Hence, if a partition is subset optimal, it is also uniformly γ-dense. Traag et al. present uniform γ-density as a corollary to subset optimality (see Theorem 3.18).

3.8 Summary

Let me summarise the guarantees of Leiden and list again which implies which. Note that subset optimality is not the same as optimality. Optimality implies subset optimality, but not the other way round. Subset optimality is the strongest of all the guarantees, it implies γ-separation, node optimality and uniform γ-density. Uniform γ-density in turn implies subpartition γ-density, which implies γ-connectivity which implies connectivity, if $\gamma > 0$. It is not enough to only prove subset optimality, as it occurs only in asymptotically stable iterations, and the other guarantees (except uniform γ-density) occur before that, at different stages.

4 Design

The purpose of my software is to automate the process of clustering graphs using Louvain and Leiden and in order to compare the two algorithms. This process starts with an input graph file and produces a clustering of this graph visualised in a plot, or modularity scores of multiple clusterings shown in a chart. At the press of a button the software should do all the work and export the desired data, graphs and charts to predefined locations. In this section, I will explain how the design for my software arose and what thoughts and ideas went into creating it.

In general, I followed an iterative design approach, starting from a rough sketch of the software with pen and paper. I successively refined the design as I worked out the necessary features for the software. The final design of the software is shown as a UML2.0 class diagram in Figure 6.

The first step was to get an overview of possible inputs and decide which graphs I wanted to work with. For that I went to an online graph repository "Network Repository" [12] and selected 20 graphs. My aim was to cover sizes, i.e. number of nodes and edges, of various orders of magnitude, and cover several fields of study where the graphs come from. Furthermore, some graphs have

Table 2: Name, number of nodes and edges of graphs clustered as determined by my code. Except graph 5 [12] which was too large for my computing resources.

	Graph	Number of Nodes	Number of Edges
1	fly	1781	33641
2	cat	65	1139
3	mouse-1	29	44
4	mouse-2	193	214
5	bnu	696300	14320000
6	gene-cx	4413	108818
7	gene-ht	2570	13691
8	gene-lc	4227	39484
9	email-eu	32430	54397
10	email-tarragona	1133	5451
11	power-bus	1138	2596
12	power-usa	4941	6594
13	matrix-11	87804	2565054
14	matrix-13	94893	3260967
15	ship-1	140385	1707759
16	ship-5	179104	2200076
17	youtube	1134890	2987624
18	dblp	317080	1049866
19	wikipedia	1791489	28511807
20	email-gt	1005	25571

weighted edges while some have unweighted edges. Table 2 shows the sizes of those graphs as determined by my code, the names were chosen by me. Note that Graph 5 was too large to be processed by my computing resources (Intel Core i7-7700HQ a quad-code running at 2.8 GHz with 16 GB of RAM running Windows 10 with WSL Ubuntu) and thus the numbers are as reported on the Network Repository [12] and not as determined by my code.

After having selected the graphs to cluster, the next step is to decide how to parse their data. As the graphs from the network repository come in different file formats, a core part of the pipeline is designing a set of parsers for each file type. To facilitate code reuse, the design follows the Template Method design pattern [13]. All common code is implemented in the abstract the `Parser` base class that provides the interface to parse a given graph file. Each concrete parser inherits from that (see Figure 6) and only the `parse` method has to be implemented. This allows my implementation to follow the "Don't repeat yourself" (DRY) software engineering best practice. The rough procedure of each of the parsers is the same. They have to open the file, read the data (this is where they differ), produce the graph object and close the file. The Template Method Pattern lets me keep the parts of the algorithm that stay the same, leaving only the `parse` method that varies depending on the file format.

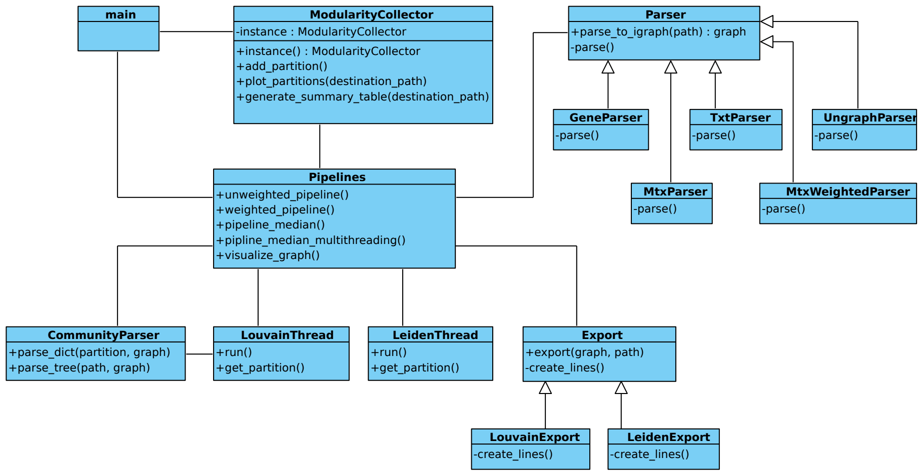


Figure 6: UML2.0 class diagram showing the design of my software.

The `Pipelines` class is the central piece of my software. This is where the different experiments are defined. A pipeline gets a graph to cluster, parses it with the appropriate parsers. It then clusters the graph using Louvain and Leiden. Finally it returns the graph and the two partitions, as produced by the algorithms, which can then be fed to the `visualization_pipeline` to plot the graphs and show the communities as node colourings. The pipelines offer the automation I set out to reach.

5 Implementation

The software is implemented in Python 3. The main reason for choosing Python is its versatility and readability. Furthermore, Traag et al. published an implementation of the Leiden algorithm as a Python package [14]. While Blondel et al. [2] published an implementation of the Louvain algorithm in C++, this implementation ended up not working as expected, returning the trivial partition for the largest graphs. While interfacing C++ form Python is possible, in the end I opted to use the Python implementation of Louvain by Aynaud [1].

To run the software, make sure you have a Python interpreter version `3.8.10` or higher and the following list of packages installed with the given version:

- `python-louvain 0.16` [1]
- `leidenalg 0.9.1` [14]
- `igraph 0.10.2` [5]
- `matplotlib 3.6.2` [8]
- `networkx 2.8.8` [6]

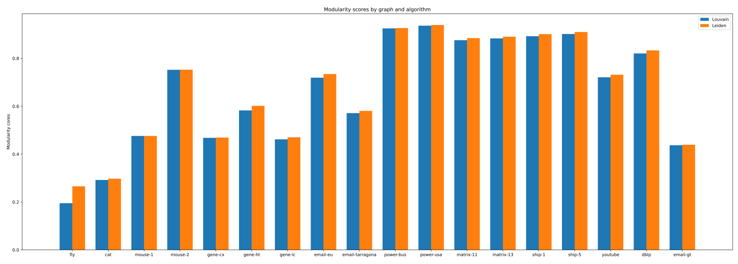


Figure 7: Modularity scores of Louvain (blue) and Leiden (orange) for selected graphs. The graphs from left to right: `fly`, `cat`, `mouse-1`, `mouse-2`, `gene-cx`, `gene-ht`, `gene-lc`, `email-eu`, `email-tarragona`, `power-bus`, `power-usa`, `matrix-11`, `matrix-13`, `ship-1`, `ship-5`, `youtube`, `dblp`, `email-gt`.

- `pandas 1.5.2` [10]

It is a matter of executing the command `python3 main.py` and the software starts clustering the graphs. In the `graph_settings.py` file, I specify the details for each graph to cluster. Change the details here if needed.

Note that the Leiden algorithm as implemented in the `leidenalg` package [14] is slightly different compared to the version that was published by Traag et al. [16]. `leidenalg` allows the choice of other quality scores to try to maximise, not only modularity and CPM. To achieve that, the `RefinePartition` step is slightly different. When merging subsets of communities, `leidenalg` does not check if the subset in question is sufficiently well connected to the rest of the community. Hence, subpartition γ-density is not guaranteed by this implementation. The other guarantees still hold [14].

6 Discussion

To compare the modularity scores achieved by Louvain and Leiden, consider Figure 7. For each of the 19 graphs I worked with, it shows the modularity score of the clustering as produced by Louvain and Leiden. Figure 7 shows that Leiden consistently outperforms Louvain. Only for graphs `mouse-1`, `mouse-2` and `gene-cx` Louvain comes very close to Leiden. In all the other cases, Leiden clearly outperforms Louvain. This confirms the findings of Traag et al. [16].

The graph where Leiden achieves the biggest improvement on Louvain is the `fly` graph. Figure 8 (Appendix B) shows the partition by Louvain and Figure 9 shows the partition by Leiden. The colours of the nodes represent the communities they belong to. Leiden achieves a modularity of 0.2660 while Louvain achieves a modularity of 0.2043. The position of each node is the same in both plots. For example, the three orange nodes in the top right corner of the Louvain plot represent the same nodes as the three dark blue nodes in the top right corner of the Leiden plot.

On limitation of my implementation is its inflexibility. If I were to do this project again, I would use Jupyter notebooks [9]. The main bottleneck I encountered was the amount of time needed to cluster the graphs. Every time my program terminates, it discards the clusterings. Hence, when developing and debugging, I would cluster the same graphs over and over again. Jupyter notebooks would have allowed me to keep the clusterings and only re-run a part of the code. One thing I would keep the same is the design of the parsers.

What stood out to me was how readily available graphs are on the internet. The Network Repository hosts over 3000 graphs from more than 30 areas. Biological, economical and citation networks are three examples [12]. On the other hand, what was really difficult to find, was how these graphs were created, i.e. what real world situation they portray, what the nodes represent, and how edges are created or weighted.

One issue I had to deal with was selecting colours for the different communities. An upper bound on how many clusters Louvain or Leiden can find for a graph $G = (V, E)$ is $|V|$. In graphs I looked at, only a low number of clusters were present. Yet, finding 21 easily distinguishable colours, as it was necessary for the visualisation of the `fly` Leiden clustering (see Figure 9 in Appendix B), was not trivial. The largest colour palette of the `matplotlib` package [8] has only 20 colours. My solution is to combine multiple colour palettes. To my regret, the colour palettes I used are not colour blind friendly.

7 Conclusion and Future Work

There are many possible ways to cluster graphs. Louvain and Leiden are two of them. When comparing them on real world graphs, Leiden achieves significantly higher modularity scores on most of them. Furthermore, Leiden also provides more guarantees for the clustering it produces than Louvain. γ-separation and γ-connectivity are guaranteed for each clustering produced by Leiden. Node optimality is guaranteed for stable iterations and subset optimality is guaranteed for asymptotically stable iterations.

When iterating Leiden it is not possible to tell when an asymptotically stable iteration is reached, but it is clear when a stable iteration is reached. One open question for future work is how often Leiden has to be iterated to reach such a stable partition on real world graphs. Moreover, whether the run time of later iterations of Leiden is as long as the first iteration of Leiden remains another open questions.

The modularity score of the trivial partition, i.e. all nodes of the graph in one community, is zero. Hence, the trivial partition is highly undesirable for Louvain and Leiden. What happens when the desired result is one community containing all nodes. How would Louvain and Leiden cluster a real-world graph which represents only one community? Would they uncover substructures? These questions represent further alleys to be explored in future work.

A Notation

Unless otherwise stated, this is the notation used in my thesis.

- $G = (V, E)$ is a graph with nodes V and edges E
- $n = |V|$ is the number of nodes
- $m = |E|$ is the number of edges
- Q is the quality function modularity (unless explicitly stated otherwise)
- ΔQ is the change in Q
- k_i is the sum of weights of edges incident on node $i \in V$ for weighted graphs, else it is the degree of node i
- $k_{i,in} = k_{i,in}(C)$ is the sum of weights of edges of community C incident on node $i \in V$ for weighted graphs, else it is the number of edges of community C incident on node $i \in V$
- $\mathcal{P}$ denotes a partition
- $C \in \mathcal{P}$ is a community
- $w(e)$ is the weight of an edge $e \in E$, also denoted $w(x, y)$ for pairs of nodes $x, y \in V$
- $\Sigma_{in} = \Sigma_{in}(C) = \sum_{x,y \in C} w(x, y)$ is the sum of the weights of edges inside of the community C
- A pass in the algorithm Louvain is one run of the modularity optimisation sequence MOS and community aggregation sequence CAS
- A pass in the algorithm Leiden is one run of the `MoveNodesFast`, `RefinePartition` and `AggregateGraph` steps.

B Visualisation of graph clustering

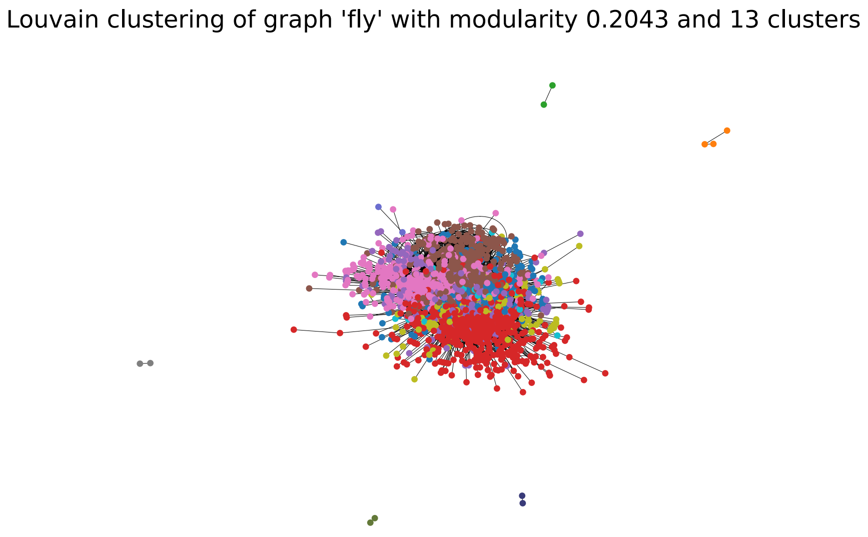


Figure 8: Louvain clustering of graph “fly”. The colours of the nodes represent their community belonging.

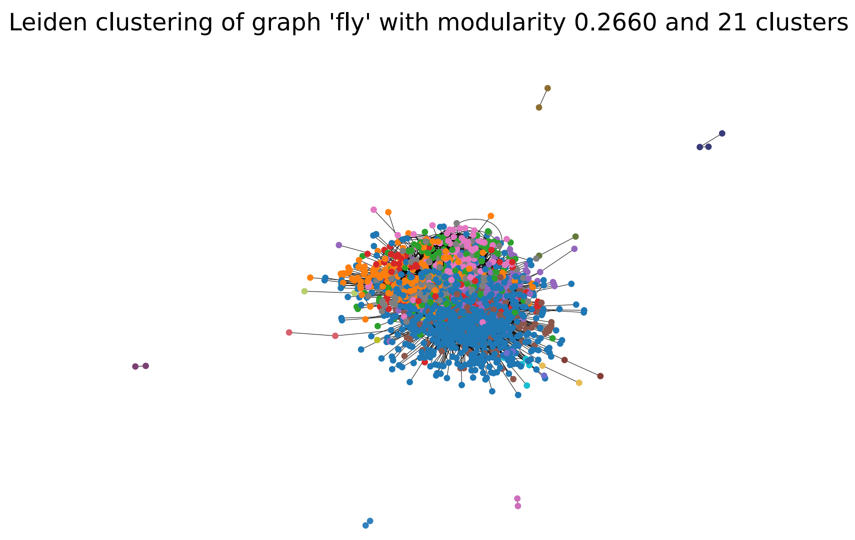


Figure 9: Leiden clustering of graph "fly". The colours of the nodes represent their community belonging.

References

[1] Aynaud, T. python-louvain 0.16: Louvain algorithm for community detection. `https://github.com/taynaud/python-louvain`, 2020.

[2] Blondel, V. D., Guillaume, J.-L., Lambiotte, R., and Lefebvre, E. Fast unfolding of communities in large networks. P10008. Publisher: IOP Publishing.

[3] Brandes, U., Delling, D., Gaertler, M., Gorke, R., Hoefer, M., Nikoloski, Z., and Wagner, D. On modularity clustering. 172–188. Conference Name: IEEE Transactions on Knowledge and Data Engineering.

[4] Clauset, A., Newman, M. E. J., and Moore, C. Finding community structure in very large networks. 066111. Publisher: American Physical Society.

[5] Csardi, G., and Nepusz, T. The igraph software package for complex network research, 2006.

[6] Hagberg, A. A., Schult, D. A., and Swart, P. J. Exploring network structure, dynamics, and function using networkx. In *Proceedings of the 7th Python in Science Conference* (Pasadena, CA USA, 2008), G. Varoquaux, T. Vaught, and J. Millman, Eds., pp. 11 – 15.

[7] Hofstad, R. v. d. *Random Graphs and Complex Networks.* Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press.

[8] Hunter, J. D. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering 9*, 3 (2007), 90–95.

[9] Kluyver, T., Ragan-Kelley, B., Pérez, F., Granger, B., Bussonnier, M., Frederic, J., Kelley, K., Hamrick, J., Grout, J., Corlay, S., Ivanov, P., Avila, D., Abdalla, S., and Willing, C. Jupyter notebooks – a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas* (2016), F. Loizides and B. Schmidt, Eds., IOS Press, pp. 87 – 90.

[10] McKinney, W. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference* (2010), S. van der Walt and J. Millman, Eds., Proceedings of the Python in Science Conference, SciPy, pp. 56–61.

[11] Pons, P., and Latapy, M. Computing communities in large networks using random walks. In *Computer and Information Sciences - ISCIS 2005*, p. Yolum, T. Güngör, F. Gürgen, and C. Özturan, Eds., Lecture Notes in Computer Science, Springer, pp. 284–293.

[12] Rossi, R. A., and Ahmed, N. K. The network data repository with interactive graph analytics and visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence* (2015).

[13] Shvets, A. Refactory guru: Design patterns. `https://refactoring.guru/design-patterns`, 2021.

[14] Traag, V. leidenalg 0.9.1. https://github.com/vtraag/leidenalg, 2022.

[15] Traag, V. A., Van Dooren, P., and Nesterov, Y. Narrow scope for resolution-limit-free community detection. 016114.

[16] Traag, V. A., Waltman, L., and van Eck, N. J. From louvain to leiden: guaranteeing well-connected communities. 5233. Number: 1 Publisher: Nature Publishing Group.

[17] Wakita, K., and Tsurumi, T. Finding community structure in mega-scale social networks: [extended abstract]. In *Proceedings of the 16th international conference on World Wide Web*, WWW '07, Association for Computing Machinery, pp. 1275–1276.

[18] WANG, G., AND KWOK, S. W. H. Using k-means clustering method with doc2vec to understand the twitter users' opinions on COVID-19 vaccination. In *2021 IEEE EMBS International Conference on Biomedical and Health Informatics (BHI)*, pp. 1–4. ISSN: 2641-3604.

[19] WOLF, F. A., ANGERER, P., AND THEIS, F. J. SCANPY: large-scale single-cell gene expression data analysis. 15.