University of Bergen
Department of Informatics

Overlapping Community Detection using Cluster Editing with Vertex Splitting

Author: Gard Askeland
Supervisor: Pål Grønås Drange, *Co-supervisor:* Ahmad Hemmati

UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

November 2022

Abstract

The problem Cluster Editing with Vertex Splitting models the task of overlapping community detection by yielding a cluster graph wherein a vertex of the input graph may be split so that it is present in several clusters. Cluster Editing with Vertex Splitting is previously proved to be in FPT with a quadratic size kernel. In this thesis we apply techniques from the fields of metaheuristics and hyperheuristics in order to develop heuristics for the problem. These heuristics prove to be robust, to scale linearly with a factor of $2.1 \cdot 10^{-3}$ per additional edge in the input graph and to enable the study of high-quality solutions of CEVS for much larger graphs than what is possible with currently known exact methods. Furthermore, CEVS-score is introduced as a ground truth independent measure for comparing solutions to the overlapping community detection task, and communities found by the CEVS-heuristics are compared to communities found by other algorithms solving the overlapping community detection task with results favoring the CEVS-heuristics.

Acknowledgements

I wish to thank my supervisors Pål Grønås Drange and Ahmad Hemmati for all the good advice, and Jakob Kallestad for inspiration, discussion and gaming sessions contributing to the thesis. Also thanks to my family and friends for all the love and support they have given me. Special thanks to Anders and Johan Magnus for being wonderful flatmates.

Gard Askeland
21.11.2022

Contents

Chapter 1

Introduction

Along with the growth of computational power and the subsequently improved capabilities of gathering data, the scientific community has taken a keen interest in graph networks. One much studied property of such graphs is their community structure, leading to development of algorithms for community detection. The task of community detection on graphs has a wide range of applications, for example in biology, social science and data mining [69, 56]. Most attention has been given to the problem of disjoint community detection, where each vertex of a graph is assigned only one community. In the last twenty years, the more complicated problem of overlapping community detection, where one vertex can be in several communities, has also been examined. Arguably, overlapping community detection represents a more realistic scenario in many cases, for example in social networks where most people can be said to be part of several communities, for example one family community and one work community [50].

Although the idea of a community in a network is intuitive, the work on community detection has so far not yielded an agreed-upon best way to define communities in networks. Instead, researchers in the field have explored a wide range of approaches using both statistical methods and decision problems to define communities with desirable properties, resulting in quality measures that are prevalent in the literature [69, 48, 50]. Accordingly, benchmarking experiments have been performed in order to compare community detection algorithms on these measures [69, 56].

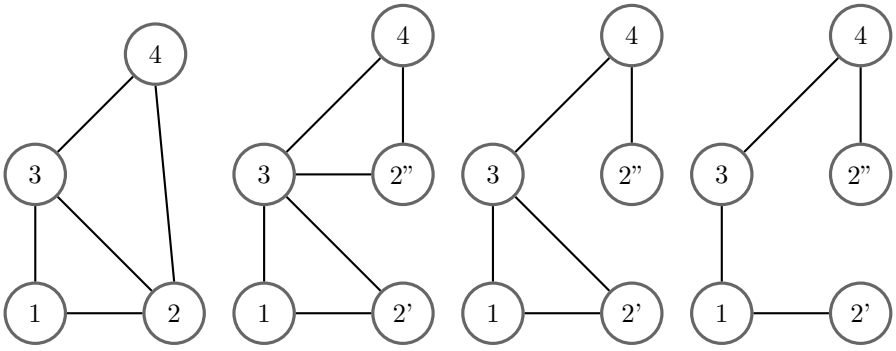
In this thesis we examine a suggested approach to the overlapping community detection task using the decision problem Cluster Editing with Vertex Splitting (CEVS) [3]. The problem is a variation on the much-examined Cluster Editing problem, and both Cluster Editing and CEVS belong to the category of algorithmic problem known as graph modification problems [43, 23]. CEVS implicitly yields a defined least cost for each graph G, which is the lowest integer k for which an input of G and k to the decision problem gives a positive answer. A not necessarily unique community structure corresponding to such a lowest k is then interpreted to be a community structure of highest possible quality for the input graph G. The problem defines four operations

named do nothing, edge addition, edge deletion and vertex split, where the latter removes one vertex and adds two vertices to the graph so that the union of the neighborhoods of the new vertices equals the neighborhood of the removed vertex. A sequence of k of the four kinds of operations applied to the input graph G that yields a cluster graph confirms that the instance of (G, k) is a yes-instance, where a cluster graph is a graph in which every connected component is a clique we call a cluster. Cluster Editing is different from CEVS only in that there is no split operation, this making Cluster Editing model disjoint community detection rather than overlapping community detection. When stating this, we disregard the "do nothing" operator which is included in the definition by Abu-Khzam et al. [3] for convenience with regard to proofs and can be ignored in practice.

In the sequence of operations applied to G we may split vertices that are created by the split operator, this creating new vertices that we may split again and so on. This implies a series of splits, all originating from one vertex split. In Section 2.2 we define the first vertex being split in this series to be the original ancestor of the other vertices in the series, with the other being descendant vertices of the original ancestor. Each vertex in cluster graph G' created by applying the sequence of operations to G is either a descendant vertex or a vertex that has not been split. Let S_v be the set of all vertices in G' that have $v \in V(G)$ as original ancestor. CEVS models a solution to the overlapping community task as such: Each cluster in G' models a community, and if a cluster contains a vertex of S_v, the community corresponding to the cluster contains v. Also, $|S_v|$ gives the number of overlaps for vertex v. If $v \in V(G)$ is in $V(G')$ as well, v has not been split and is in one cluster and community only.

There are two variations of the split operator, known as inclusive and exclusive vertex split, where the exclusive split adds the restriction of the two new vertices having disjoint neighborhoods. Both splits are reasonable approaches to modelling overlapping community detection, but with slightly different consequences. We may argue that inclusive vertex split is preferred to exclusive vertex split from studying Figure 1.1. Here it may make sense for vertex 3 to belong to two clusters including vertices $\{2'', 4\}$ and $\{1, 2\}$ respectively, and if we use an exclusive vertex split as shown in graph c in the figure, we need to add an edge between 2" and 3 to make these clusters, while the inclusive vertex split can duplicate the edge between 2 and 3 and therefore skip this edge addition. CEVS with inclusive vertex split is the main focus in this thesis, and CEVS can be assumed to refer to the problem defined with inclusive vertex split unless otherwise is stated. Since splits can be assumed to be executed after all edge additions (see Section 2.6), we can assume that the two vertices resulting from a split ends up in different clusters and that two vertices originating from the same split operation never have an edge to each other.

The exclusive vertex split was first introduced as an operation on graphs by Eades and Mendonça [18] with the purpose of improving graph drawings by removing a vertex, adding two new vertices and distributing the set of edges incident to the removed vertex between the two new vertices (equivalent to exclusive vertex split). Then Figueiredo and Mendonça proved that the following



(a) Original graph (b) Valid split (c) Valid split (d) Invalid split

Figure 1.1: Example showing different ways to apply the splitting operation of CEVS to a single vertex.

problem using the same splitting operation is NP-complete [20]:

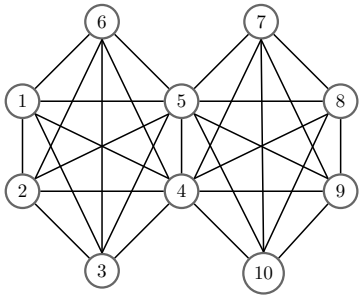
Splitting number

Input:	Graph G and nonnegative integer k
Parameter:	k
Question:	Decide if it is possible to obtain a planar graph by applying at most k splitting operations to graph G

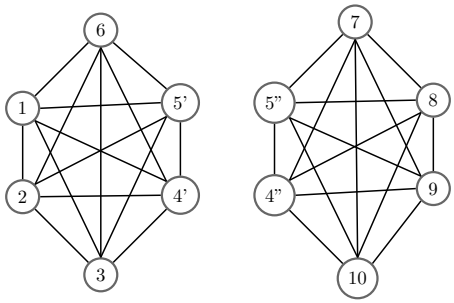
Planar graphs [5] were targeted as they are easy to make clear visualizations of. To our knowledge this is the only other area of interest than CEVS in which the splitting operation has been applied.

CEVS was introduced and proved to be in FPT with a quadratic size kernel by Abu-Khzam et al [3]. It is not known whether the problem is NP-hard [3]. Given the NP-completeness of Cluster Editing [29], it is reasonable to assume that CEVS is NP-hard, motivating the study of heuristics for the problem. Heuristic approaches are frequently preferred over exact algorithms for intractable problems on large instances because of their simplicity and robustness [31], and examining solutions provided by heuristics may inform the study of the complexity of the problem. It is noted by Abu-Khzam et al. [3] that it is not obvious how to prove NP-hardness of CEVS by reducing Cluster Editing to CEVS. Abu-Khzam et al. [2] give a heuristic for the variation on CEVS with exclusive vertex split.

In this thesis we attempt a heuristic approach to solve CEVS, with the goal of finding presumably optimal and near-optimal solutions to larger input graphs than what the currently known exact methods allow. We examine a range of approaches to heuristics, using Adaptive Large Neighborhood Search from the field of metaheuristics, the recently introduced Deep Reinforcement Learning Hyperheuristic [36] and a preprocessing on the input derived from Lemma 8 in Abu-Khzam et al. [3]. Eventually, the solutions found by the heuristics are compared to results found by other algorithms solving the overlapping community



(a) Input graph



(b) Cluster graph of solution

Figure 1.2: Optimal solution for a graph by using two splits

detection task on canonical measures in the literature.

1.1 Community detection

Community detection endeavours to identify groups of vertices called communities in graphs. In the literature there is general consensus that in each community there should be more connections (edges) between vertices in the community than there are connections to vertices not in the community [16, 69]. However, there is as yet no agreed-upon definition of a community [16]. Initially the task of disjoint community detection was given most attention, while in the last twenty years, overlapping community detection has received a great amount of attention also [69].

Several quality measures for evaluating communities assigned to graphs have been introduced, and a few are pervasive in the literature on overlapping communities.

One such measure is Overlapping Normalized Mutual Information (ONMI), introduced by Lancichinetti, Fortunato and Kertész [40] and based on Newman's modularity for evaluating the quality of disjoint communities [49]. The version of ONMI used in this thesis is one with a slightly different normalization, as described by McDaid, Greene and Hurley [48]. ONMI calculates a score for

goodness of overlapping communities by comparing the clustering to a ground truth for the graph the communities apply to. The score given is between 0 and 1 where a score closer to 1 indicates a closer match between the clustering and the ground truth.

Another measure called extended modularity (EQ) [50] instead compares the communities to a null-model, which is a graph where the probability of any edge being present between vertex u and v is given based on the degrees of u and v in the original graph. More edges being present in a limited area of the graph than in the null model contributes to a larger EQ and is interpreted to indicate modularity in the area. Thus, EQ provides an evaluation of the quality of communities regardless of whether a ground truth for the graph examined is available. Just like ONMI, this measure is given between 0 and 1. Here 1 indicates optimal modularity.

In this thesis both ONMI and EQ will be used to evaluate clusters found by the metaheuristic algorithms. Other measures from the literature that are not covered here include F1-score [21], omega-score [12], BCubed [4] and "within cluster average distance" [2].

Many algorithms for finding good communities have been suggested. Some researchers have approached the problem using node-clustering, edge-clustering and combinations of these two techniques [16, 56]. Other approaches use matrix factorization [73], graph partitioning [44], greedy local optimization [40] and edge betweenness [24]. Cluster Editing and CEVS are exact approaches formulated as decision problems, aimed at solving the disjoint and overlapping community detection tasks respectively.

With the large amount of quality measures and algorithms available, there is a vast diversity in approaches to solve, define and assess results for the one task of overlapping community detection. Several papers argue that there is not necessarily a best way to do the task, and that the results of each algorithm owe more to the algorithm itself rather than some innate property of overlapping communities in undirected graphs [69, 54]. Moreover, Peel, Larremore and Clauset [54] prove and emphasize that the no free lunch theorem applies to algorithms for community detection tasks, so that there is no best algorithm for the task, but possibly classes of algorithms that work better for data structured in certain ways. From this perspective, developing new approaches to overlapping community detection may result in methods that gives better detection for some classes of data, motivating the study of CEVS.

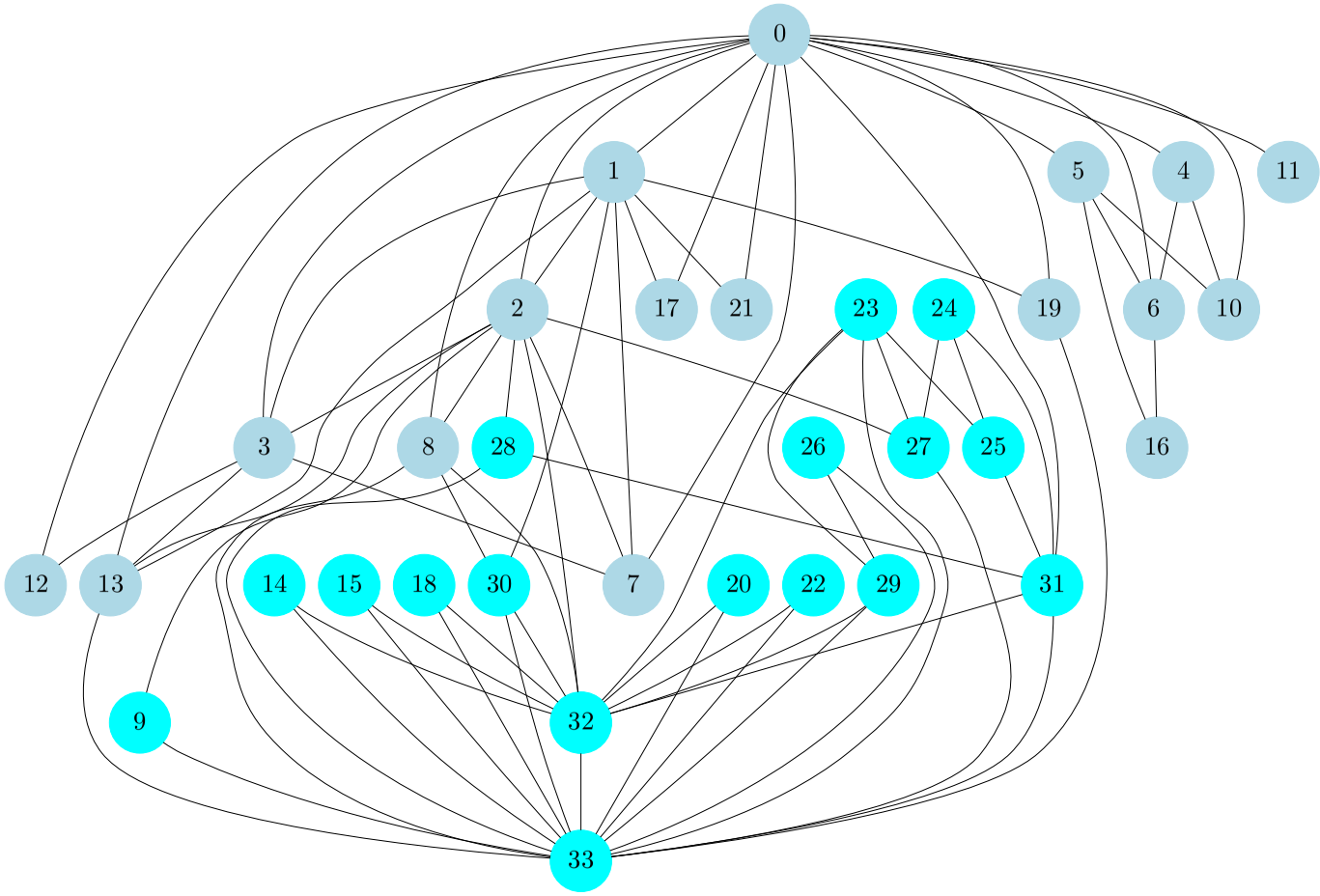


Figure 1.3: Visualization of the original karate graph [74], included for comparison to the visualizations of solutions of CEVS for the karate graph on the next pages. The vertices are colored according to class membership in the ground truth. This visualization was made using Graphviz [19].

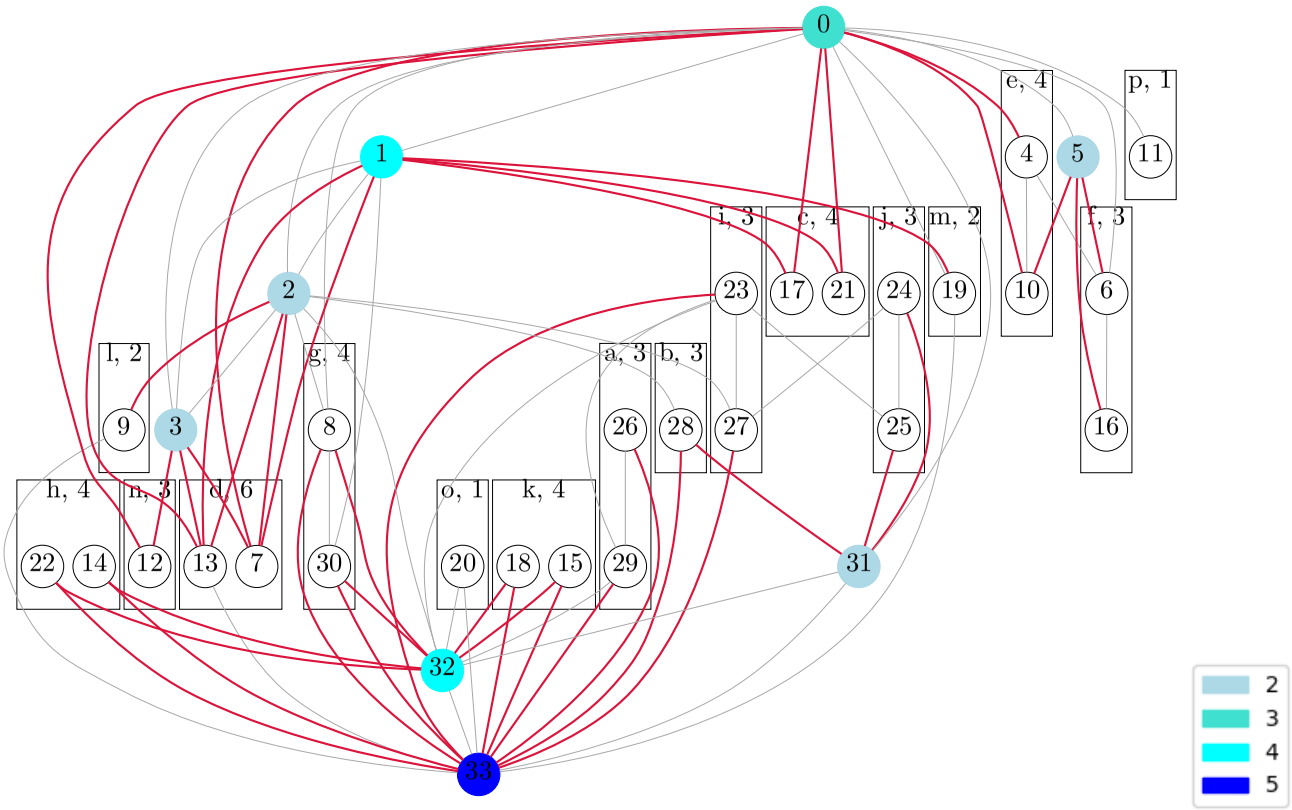


Figure 1.4: Visualization of a minimum solution of CEVS found by execution of AHC on the karate graph. All edges included are those occurring in the original graph. Overlapping vertices are drawn outside the clusters, colored according to the number of clusters they occur in and having thick, red edges to vertices they share a cluster with. Each cluster is labeled (a, k) where a is a unique cluster label and k is the number of vertices in total in the cluster. The cost of the solution is 43, which consists of 22 edge deletions, 5 edge additions and 16 vertex splittings. This visualization was made using Graphviz [19].

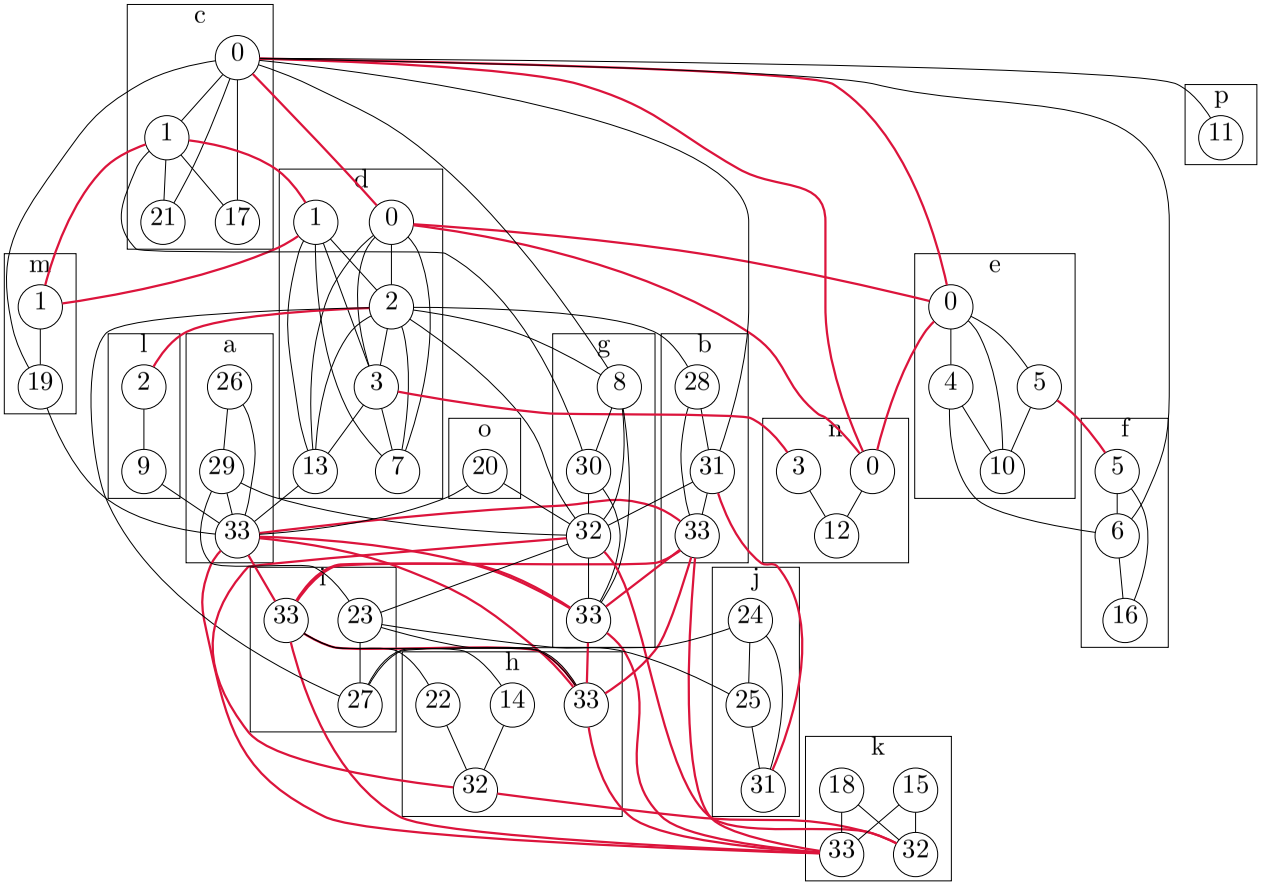


Figure 1.5: Alternative visualization of a minimum solution of CEVS for the karate graph, the same solution as in Figure 1.4. Here the clusters contain the same vertices as in the cluster graph corresponding to the solution, and the black edges are unique edges of the input graph. Split vertices representing the same original vertex have thick, red edges to one another. This visualization was made using Graphviz [19].

1.2 Cluster editing

As pointed out previously, CEVS is a variation on the problem of Cluster Editing. It is formulated as such:

Cluster Editing

Input:	Graph G and nonnegative integer k
Parameter:	k
Question:	Decide if it is possible to obtain a cluster graph, a graph where each component is a complete graph, by modifying graph G by removing and adding at most k edges.

The problem is NP-complete [67] and has received much attention from the field of parameterized complexity, yielding a linear kernel of at most 4k vertices [29]. Fomin et al. [22] give a lower bound depending on the exponential time hypothesis[1] [33] of $2^{o(\sqrt{pk})} \cdot n^{O(1)}$, where p is the maximum number of cliques in the resulting cluster graph and $p = O(k^\sigma)$ where $0 < \sigma < 1$.

Furthermore, Cluster Editing was the problem chosen for the PACE-challenge of 2021, a yearly competition wherein researchers in the field of algorithms compete to implement both exact and heuristic solvers for a specific problem [37]. In the heuristic track the best solvers for Cluster Editing used various metaheuristics, and this work has informed the approach taken for developing heuristics for CEVS in this thesis, for example by using Adaptive Large Neighborhood Search [60] and operators inspired by label propagation [28, 58].

Cluster Editing is applicable to the problem of correlation clustering [7]. In this problem, a graph G with edges labeled either plus or minus is given, and the goal is to partition the graph into clusters so that the partition either maximizes the number of plus-labeled edges between vertices in the same cluster or the number of minus-labeled edges between vertices in different clusters. Correlation clustering is motivated by the practical machine learning problem where one is given a set of n documents and their pairwise correlation defined by a classifier and wants to partition the documents into categories that agree with the classifier as much as possible [7]. An interpretation of the vertex splitting operation in CEVS applied to correlation clustering is that the splitting operation models identification of two erroneously merged documents and models their separation, where the documents for example could have been merged by mistake while hashing documents. Another interpretation is that a document that belongs to several categories can be split so that it can be in several clusters, each representing a category.

Additionally, Cluster Editing has found applications in computational biology [10]. The literature in the field frequently utilizes Weighted Cluster Editing, a version of Cluster Editing where the edges are weighted by the

[1] Informally: There exists no $2^{o(n)}$ algorithm for 3-SAT and no $O^*((2-\epsilon)^n)$ algorithm for CNF-SAT [22]. See Cygan et al. [15] for a closer explanation.

correlation of vertices, with the vertices representing genes, proteins or other biological entities. For more information on applications of Cluster Editing in biology, see Böcker and Baumbach [9]. In some problems in computational biology, for example the study of metabolic pathways, there is a high degree of interconnection and therefore overlap between clusters [71], which suggests CEVS may be applicable.

1.3 Metaheuristics and hyperheuristics

In a recent survey on metaheuristics by Dökeroglu et al. [17], the term metaheuristics is said to describe "higher level heuristics that are proposed for the solution of a wide range of optimization problems". Thus, such heuristics work as guidelines for making heuristics for solving different optimization problems. Techniques from the field are particularly useful to solve NP-hard problems [68], frequently finding optimal solutions faster than exact algorithms and in cases where exact algorithms are impractical yielding the best known solutions to problem instances [30, 70, 17].

Since metaheuristics have been applied to Cluster Editing previously, it is reasonable to assume that techniques from the field can be applied to the relatively similar problem of CEVS. In this thesis the metaheuristics Uniform Random Agent (URA) and Adaptive Large Neighborhood Search (ALNS) are applied to CEVS. Our implementation of ALNS for solving CEVS will be referred to as *AHC*, an abbreviation of *ALNS-based heuristic for solving CEVS*. Note that both URA and ALNS are similar to simulated annealing [38], and that simulated annealing therefore is not considered in much depth in this thesis.

A hyperheuristic is "a heuristic search method that seeks to guide the selection or generation process of heuristics in order to more efficiently solve combinatorial optimization problems" [36]. The field overlaps with metaheuristics, with for example the metaheuristic ALNS being a selection-based hyperheuristic. In this thesis the hyperheuristic Deep Reinforcement Learning Heuristic (DRLH) described and implemented by Kallestad, Hemmati and Hasibi [36] will be compared to URA and ALNS on their performance in terms of running time and objective when applied to CEVS.

1.4 Parallelization

With the stagnation of performance improvement for sequential processors, difficult computational tasks have moved on to the realm of several processes and parallelization [64]. Modern processors have several threads and make parallelization of everyday computing tasks viable, while the emergence of big data urges the development of algorithms that use parallelization in order to enable analysis of larger data sets in less time [64]. In this thesis the possibilities and challenges of parallelizing AHC are examined, and experiments are run on a parallel implementation of AHC using shared-memory parallel computing with

OpenMP [51]. The results show that the attempted approach to parallelization yields only a modest speedup.

1.5 Organization of thesis

The thesis is divided into seven chapters. In Chapter 2 we define the problem CEVS and terminology for presenting the material in the thesis, and we introduce the different meta- and hyperheuristics we use for solving CEVS. Then, we present an overview of the implementations of heuristics to solve CEVS in Chapter 3. We give a branching algorithm to solve CEVS exactly parameterized by maximum vertex degree in Chapter 4, and in Chapter 5 we give a conjecture about the possibility of finding exact solutions of CEVS on complete bipartite graphs. Chapter 6 presents results showing robustness of AHC, compares the performance of ALNS on CEVS to the performance of URA and DRLH, examines scaling and parallelization of AHC and compares results obtained by AHC to results obtained by other community detection algorithms. Finally, a conclusion summarizing the thesis is found in Chapter 7.

Chapter 2

Preliminaries

2.1 General definitions

The following assumes familiarity with the subject of graphs. All graphs examined in the thesis can be assumed to be simple graphs unless otherwise is stated. For a simple graph $G = (V, E)$, let $V(G)$ indicate the vertex set of the graph and let $E(G)$ indicate the edge set. Also, $|V(G)| = n$ and $|E(G)| = m$.

Let the *open neighborhood* of a vertex u in a graph G be $N(u) = \{v \mid uv \in E(G)\}$, and let $\deg(u) = |N(u)|$. The *closed neighborhood* of vertex u is defined as $N[u] = N(u) \cup \{u\}$. For a set S let the open neighborhood of S be $N(S) = \bigcup_{v \in S} N(v) \setminus S$, and let the closed neighborhood of S be $N[S] = N(S) \cup S$. Also, let a *neighborhood set* of a vertex u be a set S so that $u \notin S$ and for some $v \in N(u)$, $v \in S$. We let $\Delta(G)$ denote the maximum degree of any vertex in a graph G.

Let an *induced subgraph* $G'(S)$ of a graph G where $S \subseteq V(G)$ be a graph so that $V(G'(S)) = S$ and $E(G'(S)) = \{uv \mid u, v \in S, uv \in E(G)\}$. A *clique* is an induced subgraph $G'(S)$ on a set $S \in V(G)$ such that $G'(S)$ is a complete graph, which means every vertex in $G'(S)$ has an edge to every other vertex in $G'(S)$. Then a *cluster graph* is a graph in which every connected component is a clique, and we may refer to such a clique as a *cluster*. Let an *induced path* in graph G be a path in G that is an induced subgraph of G, emphasizing that vertices not adjacent in the path do not have an edge to each other. Then, let any induced path on three vertices in a graph be called a P_3. A graph is a cluster graph if and only if it has no P_3 [59].

The notation $O^*(f(n, k))$ will be used to denote asymptotic running time for parameterized algorithms without polynomial factors and addends.

2.2 Problem definition

We first formulate the problem as a decision problem, similar to the definition given by Abu-Khzam et al. [3]:

Cluster Editing with Vertex Splitting

Input: Graph G and integer k
Parameter: k
Question: Decide if there is a cluster graph G' so that G can be changed into G' by applying a sequence of operations $e_1, e_2, ..., e_k$ to G. A new graph is created each time an operator is applied, and this yields the graph sequence $G_0, G_1, G_2, ..., G_k$ corresponding to the sequence of operations, where $G = G_0$ and $G' = G_k$. Let $1 \leq i \leq k$. Each operation e_i is one of the following:

1. do nothing
2. add an edge to $E(G_{i-1})$
3. delete an edge from $E(G_{i-1})$
4. split $v \in V(G_{i-1})$ by deleting v and adding two new vertices v_1, v_2 with edges so that $N(v_1) \cup N(v_2) = N(v)$.

Note that for the split operation, the neighborhoods of the added vertices do not necessarily need to be disjoint. This makes the split an *inclusive vertex split*. As previously mentioned, an alternative approach to CEVS uses an *exclusive vertex split* instead, such that $N(v_1)$ and $N(v_2)$ are a 2-partitioning of $N(v)$.

In this thesis we introduce the following new formulation of CEVS that emphasizes optimization by minimizing k.

CEVSopt

Input: Graph G and objective k
Question: Minimize k so that there is a sequence of operations $e_1, e_2, ..., e_k$ that changes G into a cluster graph G', where each operation in the sequence is one of the operations allowed in CEVS.

The split operation yields a tree structure of vertices that are added and removed by the split operation, and we introduce terminology to describe this tree structure. For the split operation, let v_1 and v_2 be *descendants* of v. Furthermore, any vertices added to the graph by splitting descendants of v are also descendants of v, and v is the *original ancestor* of its descendants.

In Abu-Khzam et al. [3] the following lemma is given, here somewhat modified to fit the definitions used in this thesis:

Lemma 1. *[3, Lemma 7] For a graph $G = (V, E)$ there is a computable bijection between pairs of cluster graphs $G' = (V', E')$ and equivalence classes of sequences*

of operations. A minimum-length sequence corresponding to G' can be computed in $O((|V'|-|V|)\Delta(G) + |V|+|E|+|V'|+|E'|)$ time.

Equivalence classes of sequences of operators are defined so that when two sequences applied to a graph G yield the same graph G', the sequences are considered equivalent. The lemma implies that there is a computable minimum sequence ϕ of operations from CEVS that can be applied to G in order to obtain G'.

Let a *solution* to CEVS for graph G be a family of sets F so that F contains one set S_C for every connected component C in cluster graph G'. Here S_C contains the original ancestors of every vertex in C as determined by the minimum-length sequence ϕ of operators applied to G in order to obtain G'. Then let the length of ϕ be the *CEVS-score* of the solution. Note that giving a solution F for a graph G thus implies a unique CEVS-score for the solution.

Each set in F can be interpreted as a community of the original graph G. Let a solution to the community detection task on graph G, as found by any algorithm, be given by a family of sets F that is a solution to CEVS.

2.3 Parameterized complexity

The three following definitions are adapted from Cygan et al. [15].

Definition 1. *A parameterized problem is a language $L \subseteq \Sigma^* \times \mathbb{N}$, where Σ is a fixed, finite alphabet.*

For Cluster editing with Vertex Splitting, $L' \in \Sigma^*$ is the encoding of a simple graph and the parameter $k \in \mathbb{N}$ is the editing cost. If a pair $(L', k) \in \Sigma^* \times \mathbb{N}$, the graph of the problem instance (L', k) can be made into a cluster graph by using at most k of the allowed operations.

Definition 2. *A* parameterized *problem $L \subseteq \Sigma^* \times \mathbb{N}$ is called* fixed-parameter tractable *(FPT) if there exists an algorithm $\mathcal{A}$, a computable function $f : \mathbb{N} \to \mathbb{N}$ and a constant c such that, given $(x, k) \in \Sigma^* \times \mathbb{N}$, the algorithm $\mathcal{A}$ correctly decides whether $(x, k) \in L$ in time bounded by $f(k) \cdot |(x, k)|^c$. The complexity class containing all fixed-parameter tractable problems is called FPT.*

Definition 3. *A* kernelization algorithm*, or simply a* kernel*, for a parameterized problem Q is an algorithm $\mathcal{A}$ that, given an instance (I, k) of Q, works in polynomial time and returns an equivalent instance (I', k') of Q. It is required that there is a computable function $g : \mathbb{N} \to \mathbb{N}$ so that $|I'|+k' \leq g(k)$.*

Frequently the instance (I', k') is referred to as the *reduced* instance, and the kernel $\mathcal{A}$ as a *reduction*. Abu-Khzam et al. [3] give a polynomial time reduction for CEVS.

Theorem 1. *[15, Lemma 2.2] A decidable problem admits a kernel if and only if it is fixed-parameter tractable.*

Since a kernel was given for CEVS by Abu-Khzam et al. [3], CEVS is in FPT.

2.4 Parallel algorithms

According to Sipser [68], an algorithm is informally "a collection of simple instructions to carry out some task". A formal definition is given by the Church-Turing thesis where algorithms are equated to algorithms that can be run on a Turing machine [68]. Let a sequential algorithm be an algorithm whose instructions are executed in a specific order, and let a processing element or a thread be a unit than can run a sequential algorithm. Then a parallel algorithm is an algorithm that is designed to run on more than one processing elements simultaneously.

Let running time be the time in seconds that a program takes to execute when run once on a computer. Let p be the number of processing elements or threads that are used by algorithms and programs. Thus, p may be used in an expression describing the running time of an algorithm. For algorithm $\mathcal{A}$, let $t_{seq(\mathcal{A})}$ denote the sequential running time of $\mathcal{A}$ and let $t_{par(\mathcal{A},p)}$ denote the parallel running time of $\mathcal{A}$ with p processing elements. Then the speedup of $\mathcal{A}$ executed on p processing elements is defined as:

$$s(\mathcal{A}, p) = \frac{t_{seq(\mathcal{A})}}{t_{par(\mathcal{A},p)}}$$

2.5 Metaheuristics and hyperheuristics

2.5.1 Adaptive Large Neighborhood Search (ALNS)

Adaptive Large Neighborhood Search is a metaheuristic that was initially introduced by Ropke and Pisinger [60] in 2006, then applied to the problem of Pickup and Delivery Problem with Time Windows (PDPTW). A neighbor solution to a solution for a problem is another solution for the problem which is marginally different, and the neighbor solutions of a solution constitute the solution's neighborhood. The metaheuristic assumes a pool of heuristics or *operators* used to modify a solution, where at least some of the operators search in a large pool of neighbor solutions, justifying the word large in the name of ALNS. The algorithm works by picking an operator from the pool in each iteration, and is finished after having executed an integer $i_{\max}$ iterations. Each operator is assigned a weight that is adjusted intermittently depending on the performance of the operator, and the weights determine the probabilities of the operators to be chosen in an iteration. The weight adjustment makes ALNS adaptive, as it can adapt the probability distribution of its operator choices depending on the effectiveness of operators on a given data set.

The metaheuristic specifies that a search occurs in segments of a given number z iterations, where the weights of each operator are adjusted after each segment. Let $w_1, w_2, ..., w_p$ be the weights of the p operators $O_1, O_2, ..., O_p$ in the pool and let $\sigma_1, \sigma_2, ..., \sigma_p$ be the scores given to the operators in a given segment. O_j is scored by adding to σ_j each time operator O_j is chosen in an iteration, where the scoring may depend on any given criteria as specified in a

reward function. At the end of the segment the score is updated as such:

$$w_j \leftarrow w_j \cdot (1 - r) + r \cdot \frac{\sigma_j}{\theta} \tag{2.1}$$

Here r is a real number between 0 and 1 giving the rate of change and θ is $\sum_{j=1}^{j=p} \sigma_j$ for the segment.

Additionally, there is an acceptance criteria deciding whether a new solution found by an operator should be accepted. An example of a criteria that can be used for ALNS is the one described in the subchapter on simulated annealing in Kleinberg and Tardos [38], where better solutions are always accepted while worse solutions are accepted with a certain probability. This is the acceptance criteria that is used for ALNS in this thesis, and it is explained in the following.

Let s be the old solution and s' be the new one. If the cost of s' is less than the cost of s, we accept s'. Else, s' is accepted with probability

$$g(s, s', T) = e^{-(f(s')-f(s))/T} \tag{2.2}$$

where $f(s)$ gives the cost of solution s and T is a temperature that decreases over the execution of the search. Note that this means that it is less likely to accept a solution s' the larger its cost is compared to s and that it is less likely to accept a worse solution the closer the search is to its final iteration.

In the ALNS implementation for CEVS in this thesis the above blueprint for ALNS has been used since it is not specific to PDPTW and can be adapted to other optimization problems. However, not all aspects of ALNS for PDPTW are reasonable to bring along, specifically the split of operators into insertion and removal heuristics so that each operator picks one heuristic from each category. This division is not meaningful for CEVS since vertices can be in several sets in the solution representation at the same time, so that insertion without removal and vice versa is possible.

2.5.2 Uniform Random Agent (URA)

Uniform Random Agent is a metaheuristic that picks operators from a pool each iteration over a number of iterations and therefore can replace ALNS in an algorithm. The operators are chosen by uniform randomness and so that it is equally likely that any operator is picked in any iteration. URA provides a baseline to compare ALNS and DRLH against, showing whether the adaptivity of ALNS and the training of DRLH lead to better results than what one would get with random choices of operators.

2.5.3 Deep reinforcement learning hyperheuristic (DRLH)

DRLH was first introduced by Kallestad, Hemmati and Hasibi [36] and provides a general hyperheuristic selection framework to be used for choosing heuristics to solve combinatorial optimization problems, effectively being an alternative to ALNS. The framework uses deep reinforcement learning to make choices

depending on an observed state and what it has learned from training on data, referred to as its policy, and it proved to find better results than ALNS on four different combinatorial optimization problems, with its performance increasing compared to ALNS for larger problem sizes. Also, its performance did not worsen with an increasing number of available operators as is observed to occur with ALNS. In this thesis DRLH is once again compared to ALNS, but on the graph modification problem CEVS instead of the scheduling and routing problems studied by Kallestad, Hemmati and Hasibi [36].

A closer explanation of the relation of DRLH to the pool of operators is warranted. In reinforcement learning there is an *agent*, for example DLRH, that acts on an environment, which in the case of this thesis is the metaheuristic framework detailed in the implementation section. The environment will at a given time t have a state s_t that is specificically designed to be used by the agent and is separate from for example a solution state. The agent observes this state and uses its *policy*, a probability distribution $\pi(a|s_t)$, to decide which action a to take. The environment then gives a reward to the agent depending on the quality of the action taken, which is determined by a reward function. The agent uses this to update its policy with the goal of optimizing reward granted for each action taken. The policy is usually not updated for every action taken, but at the end of a series of actions called an episode. The agent may in addition to the reward function utilize a value function that takes expected future reward resulting from action a into account, so that the agent uses input from both the reward function and the value function when optimizing its policy. For DRLH the policy is a type of neural network called a multilayer perceptron (MLP) [27] that is optimized using a policy gradient method known as proximal policy optimization [65]. The optimal policy is found by adjusting the parameters of the neural network.

Thus, in each iteration of a solver using DRLH, the DRLH agent makes a decision about what operator to use by using its policy that considers the current state, obtains feedback from a reward function and a value function and stores this information, which in time is used to update the policy at the end of an episode. Note that the DRLH agent does not consider the data instance and current solution when making decisions and updating its policy unless these in some capacity are included in the state. In our implementation of DRLH for CEVS, we use the objective of the current solution only indirectly (see Section 3.5).

From the above it is clear that DRLH takes the same role as ALNS in a solver, similarly making decisions and receiving feedback. However, DRLH needs training data while ALNS can be used directly on the data instance one aims to solve, meaning that using DRLH demands a larger amount of available data sets than using ALNS. A benefit of DRLH is that it can be used for discovering good operators among a possibly large pool of implemented operators. This may speed up the process of figuring out which operators are effective and therefore should be included in the operator pool of the final algorithm [35].

When referring to an implementation of state we use the term *state representation*.

2.6 Metaheuristic framework

In order to solve CEVS, the above methods from the fields of metaheuristics and hyperheuristics are applied. To use these methods, a general framework that can be used together with ALNS, URA and DRLH is needed. This framework should work so that changes can be applied to solutions that have a corresponding objective given by an objective function. The meta- and hyperheuristics then chooses which changes to apply to the solution representation with the goal of obtaining a solution of minimal objective. A solution can be said to be *feasible* if it is a valid solution to the problem and *infeasible* if it is not, where the definition of a valid solution is entirely dependent on the definition of the problem. In the following a solution representation and an objective function for CEVSOPT is given.

As solution representation we use a family F of subsets where $F \subseteq 2^V$, the power set of the vertices of G. Then a set $S \in F$ corresponds to a component in the cluster graph G', just as explained in Section 2.2. A solution is feasible as long as $\bigcup_{S \in F} S = V(G)$, since it is a consequence of the definition of CEVS that for every vertex in $V(G)$, either itself or one of its descendants has to be present in at least one clique in the cluster graph implied by the solution.

Now for the objective function. Abu Khzam et al. [3, Theorem 1] prove that any sequence of operations can be rewritten to another edit sequence of equal or smaller length without "do nothing" operations so that all edge additions are done before all edge deletions and all edge deletions are done before all vertex splittings. This suggests an algorithm for the objective function for a solution F that can be calculated as follows:

1. Initialise cost variable $c = 0$.
2. Edges are added as such: For each $u \in V(G)$, iterate over all vertices v so that $u, v \in S$ for some $S \in F$. If $uv \notin E(G)$, add 1 to c. We make sure not to count any edge twice.
3. Edges are deleted as such: For each vertex $u \in V(G)$, if there is an edge $uv \in E(G)$ and there is no $S \in F$ so that $u, v \in S$, add 1 to c. We make sure not to count any edge twice.
4. Vertices are split as such: For each $u \in V(G)$ we find the number d_u of sets in F that contain u, that is $d_u = |\{S \mid u \in S, S \in F\}|$. Then add $d_u - 1$ to c as this is the number of vertex splitting operations that is required for vertex u.
5. Return c.

2.7 Terminology for implementation details

Now for terminology that is specific to the implementation covered in this thesis. The solution representation is stored in a *solution object* together with other

datastructures that are convenient for developing well-performing heuristics. A change to the solution representation is called an *action*. In order to find actions to execute we use *operators*, which are functions that use the information encoded in the solution object to find and store actions and make changes to the solution object. The *agent* is the higher-level program choosing which operators to call, e. g. when a metaheuristic like ALNS decides to apply an operator to the solution object, the metaheuristic is the agent.

Furthermore, the *cost* of an operator is the change in objective function after applying the action found by an operator to the solution object. An operator may return an integer giving the cost of an action that is ready to be executed, and a lower cost gives a better objective function after applying the action. Note that the cost is negative if an action decreases the result of the objective function, zero if the result stays the same and positive if the result increases.

A vertex is said to be *touched* in an iteration if it is added to a set $S \in F$ or removed from a set $S \in F$, and a set S is said to be touched in an iteration if there is a change in elements of the set. When a vertex or set has not been touched over a number of iterations it is said to be *untouched*.

Chapter 3

Implementation of heuristics

Recall that we refer to the heuristic algorithm solving CEVS using ALNS as ALNS-based Heuristic for CEVS, abbreviated as AHC. The implementation that uses a deep reinforcement learning agent rather than the adaptive agent of ALNS is referred to as Deep Reinforcement Learning Hyperheuristic for CEVS, or DHC.

In this chapter we discuss the implementation details of AHC and DHC. The implementation of AHC used for the experiments in Chapter 6 is available at GitHub [6].

3.1 Datastructures

All of the following datastructures are stored in the solution object in the implementation.

Red-black search trees [66] are used to build a couple of the datastructures. This datastructure is similar to binary search trees [66] in that it consists of nodes that have a key and a value, or a value that works as a key, and that the nodes are placed in the tree so that one can traverse it by moving from parent-node to child-node recursively or iteratively. However, the tree is self-balancing so that we avoid some branches of the tree being significantly longer than others, this giving $O(\log(n))$ complexity for insertion, deletion and query of the datastructure rather than the average $O(\log(n))$ for these operations for binary search trees. If the nodes in the red-black search tree consist of key-value pairs we call the datastructure a red-black search tree map.

The datastructure storing F, the family of sets used as solution representation, is a red-black search tree map where each node in the tree has an integer as its key and a red-black search tree as its value. These red-black search trees as values act as sets and have integer values indicating vertices of the graph G as nodes. Thus, the outer tree map corresponds to F, and the inner trees corresponds to sets $S \in F$. This datastructure for F was chosen in order to enable flexible exploration of different strategies and operators during development, in particular

with regard to enable efficient addition of sets to F and deletion of sets from F and enabling iteration of the sets of F in order. Using sets implemented with hash tables [66] instead of red-black search trees could likely give shorter running times, but would prohibit in-order iteration of the sets of F. In the case where $|F|$ is fixed, which is not further examined in this thesis, an array could be used for F. Since arrays can be accessed in $O(1)$ time rather than the required $O(log(n))$ for red-black search trees and since arrays benefit more from caching than red-black search trees because of related data being closer in memory, solving the problem for fixed $|F|$ is likely more efficient than for non-fixed $|F|$ in practice. The implementations of red-black search tree and red-black search tree map we use for implementing F are the set and map containers in the C++ standard library [34].

In addition to F, there are other datastructures that are derived from F and are updated when F are updated. One of these is a set F' where $|F'|= n$ with one set for each vertex in G so that each set $S_u \in F$, $1 \leq u \leq n$, contains the keys of the sets in F that contain vertex u. This enables us to obtain a list of the sets of F than contain vertex u with time complexity $O(\log(n))$, instead of in time $\Omega(n)$ which one would get by looking through the entirety of F every time. The set F' is implemented with nested red-black search trees in the same way as F.

Another datastructure derived from F is a co-occurrence matrix. This is implemented as a red-black search tree map with a pair of integer indices as key and an integer as value of each node, but it is accessed though an interface that gives it the functionality of a matrix. Each entry (i, j) in the matrix contains a non-negative integer that corresponds to the number of sets of F which contains both i and j. The operators frequently have to check if vertices i and j appear together in some set in the solution when calculating the cost of an action. For example, we often check if there is an edge between i and j in the cluster graph encoded in the solution prior to executing the action. Then if i and j did have such an edge in the solution and we add vertex i to a set with vertex j, we know not to include the addition of the edge ij in the cost of the action. This datastructure enables retrieving this information in $O(\log(n))$ ($O(1)$ if implemented with a matrix) instead of checking if j is in any of the sets i are in, which is $O(n \cdot |F|)$.

Finally, segment trees [32] are used to store which vertices and sets have been touched in an iteration. This information is used by operators that act on vertices and sets that have not been touched in a certain number of iterations (with at most one operator executed per iteration). Specifically, the segment trees enable $O(\log(t))$ time queries of which vertices and sets have been moved in the last k iterations, where integer t is the total number of iterations of the algorithm.

Note that the datastructures of the solution object may be modified even though no action is executed. For example, some of the operators touch sets and vertices during calculation, causing the segment trees to be updated. Furthermore, some operators, in particular the intermittent scan operators introduced below, have assigned datastructures that are modified when the operators are called

regardless of whether the actions the operators find are executed.

3.2 Operators

We define two types of operators to be used by AHC: Suggestive and assertive operators.

The suggestive operators use the graph and the datastructures in the solution object to find an action to apply to the solution. Then the operators store the action in the solution object without executing it and return the cost if the action is executed. Thus, the program calling the operator can decide whether or not it wants to execute the action depending on the cost, for example according to an acceptance criteria. When suggestive operators are called they may not find a valid action. To inform the program calling the operator about this, the operator returns an integer implementation of the optional class template from C++, which we use so that it contains no integer when no valid action is found.

The assertive operators find an action and return the cost of executing it just as the suggestive operators, but differ in that they also execute the action before returning. Thus, the program calling the operator cannot choose whether or not to execute the action found by the operator after the operator call.

Several of the operators of both types work by picking a vertex, set or vertex-set pair that has been deemed good from a sorted array and then finding an action to do for the chosen vertex, set or vertex-set pair. Let such operators be called *intermittent scan operators*. Algorithms 1 and 2 show how a suggestive version of such an operator works when targeting vertices, and we give a corresponding textual explanation in the following paragraph.

The first time the operator is called, the variable `refillCounter` is set to a positive integer r, where r decreases by 1 each time the operator is called, and `goodVertices` is filled with vertices sorted by the cost of the best possible valid action for each vertex. Until `refillCounter` reaches 0, the program picks the first vertex in `goodVertices` that has not been picked since the last refill. The action the operator chooses for the vertex is not necessarily the same as the one giving the cost of the vertex in `goodVertices` because of changes to the solution object. When `refillCounter` reaches 0 or `goodVertices` is empty, `refillCounter` is reset and `goodVertices` is refilled in the same way it was initialized. By using this refill pattern in the operators we utilize that it is likely to find several good actions to execute when searching the whole graph and that this information is useful for executing more than one action, so that the next r times the operator is called we benefit from picking actions from the sorted array instead of expensively searching the entire graph each time. Note that the pseudocode given in Algorithms 1 and 2 display programming patterns and that the implementation of the function `findBestActionOnVertex` varies between operators the patterns are applied to, so that pseudocode for this function is not given.

Following are descriptions of the different operators. The ones listed under

Algorithm 1 Pseudocode for suggestive intermittent scan operator

```
if refillCounter = 0 || goodVertices.size = 0 then
   refillGoodVertices()
end if
if goodVertices is empty then
   return None
end if
u = goodVertices.pop()
cost, operation = findBestActionOnVertex(u)
store operation
refillCounter -= 1
return cost
```

Algorithm 2 Pseudocode for `refillGoodVertices()`

```
reset array goodVertices
for u=0; u < n; u++ do
   result = findBestActionOnVertex(u)
   if result is None then
      continue
   end if
   append result to goodVertices
end for
sort goodVertices by cost of actions
reset refillCounter
```

suggestive and assertive operators were deemed of high enough quality to be used in the final version of AHC, while some operators that did not perform well even though their ideas are intuitive are included in Section 3.2.3. Any operator that checks which vertices or sets have not been touched in the last k iterations uses the segment trees in the solution object to obtain this information.

3.2.1 Suggestive operators

- **Add vertex to set:** The action of this operator is to add a vertex u to a set S_i. It is an intermittent scan operator that stores good pairs (u, S_i) to act with during the intermittent scan and picks from these in order of increasing cost.

- **Add vertex to untouched set:** Chooses a set that has not been changed in the last $5n$ iterations and adds a vertex to it in the same manner as **Add vertex to set**.

- **Move vertex:** Uses intermittent scan. During the scan, for every vertex u it finds the lowest-cost pair of sets (S_1, S_2) so that u can be taken from one set and put into the other. Putting the vertex into an empty set is considered, with this adding a set to F if executed. Otherwise, pairs of sets so that $N[S_1] \cap S_2 = \emptyset$ are not considered. The operator only stores the minimum cost of moving the vertex between any pair (S_1, S_2) for every vertex, and when the vertex is chosen for an action, the attempted action uses the current lowest-cost pair of sets, which may not be the same pair of sets found for the vertex during the scan. This operator is inspired by the label propagation operator used by Gottenbüren et al. [28, 58]

- **Move untouched vertex:** Chooses a random vertex from the vertices that have not been moved in the last $5n$ iterations. The vertex is moved in the same way as for **Move vertex**.

- **Remove 3 then add 3:** This operator chooses a random P_3 in the graph, removes the vertices in the P_3 and then greedily inserts them back in the same order as they were removed, each vertex inserted into the current minimum-cost set for the vertex to be inserted into. Only neighborhood sets of the vertices and their original sets are considered. Recall that a graph is a cluster graph if and only if it is has no P_3, with this motivating the creation of the operator.

3.2.2 Assertive operators

- **Add vertex to neighbors:** This intermittent scan operator adds a chosen vertex u to each of the neighbor sets of u if the change in cost is not positive when adding. Let $\text{add}(u, S_i)$ be an action where vertex u is added to the set S_i. Then the operator does the following set of actions:

$$\Big\{\text{add}(u, S_i) \mid \text{cost}(\text{add}(u, S_i)) \leq 0, S_i \in [S_j \mid v \in S_j \wedge v \in N(u)]\Big\}.$$

- **Add untouched vertices to neighbors:** This operator picks a vertex that has not been moved in the last $5n$ iterations and adds it to sets in the same manner as **Add vertex to neighbors**.

- **Remove vertex:** This is an intermittent scan operator that picks a vertex u and removes it from as many sets as possible so that the cost for removing any vertex by itself is not positive. The sets are examined in label order. If all vertices are removed from a set, the set is deleted from F.

3.2.3 Operators with poor performance

- **Merge:** This suggestive operator checks the cost of merging any pair of sets $S_i, S_j \in F$ and puts these results in a priority queue sorted by cost. Each time the operator is called it checks which sets in F have been changed and updates the priority queue with this information, before picking a pair with low cost to merge. To check which sets have been changed since the last time the operator was called, a segment tree storing which sets are changed in which iterations is used. The operator was chosen few times by the adaptive agent when used together with other operators, and performance of AHC improved when removing this operator with regard to both running time and objective. It is worth noting that this operator is computationally much more expensive than the simpler operators adding, moving and removing single vertices while its performance is worse.

- **Split:** The first time this suggestive operator is called, the *inner cost* defined as

$$\frac{|[uv \mid uv \notin E(G), u, v \in S_i]|}{|S_i|}$$

  is calculated for each set S_i. For later calls of the operator the inner cost is updated for each set that has been changed since the last time the operator was called, where checking which sets have changed is done by using the same segment tree as was used for the merge operator. The information is put in a priority queue sorted by inner cost so that a set which contains many pair of vertices without edges in $E(G)$ can be picked. The operator splits this set using the Karger's algorithm, which is a randomized algorithm that gives the minimum cut in polynomial time with high probability [38]. As for merge, the running time and performance of AHC proved to be worse when including this operator, and the operator was assigned little weight by the adaptive agent. Therefore, this type of operator was excluded from the final version of AHC.

3.2.4 Selection heuristics

An important part of every operator is its choice of which parts of the graph to act on. In the final version of AHC, selection heuristics like intermittent scan and picking vertices that have been untouched for a while are used, as these have been judged to yield the best performance of the selection heuristics that were tried. Other selection heuristics that were implemented but later abandoned are briefly described in this section.

One way of picking sets to change is to look at statistical properties of the sets. An example is the inner cost used by **Split** as explained previously, as well as a measure counting outgoing edges from vertices of a set used by **Merge**. Generally this kind of measure turned out be computationally expensive compared to using the cost of executing an action on for example a vertex as a measure, where the latter approach is both simpler and more accurate with regard to the problem at hand.

Other attempted selection heuristics acted on several vertices for the same call of the operator. This selection heuristic is similar to the one used for the label propagation operator in the heuristic for Cluster Editing by Göttenbüren et al. [28], inspiring an early version of **Move vertex**. In this version, each vertex in the graph is moved to its best position in a neighbor set, with the vertices being moved in a random order. Compared to operators based on intermittent scan, this selection heuristic resulted in much more expensive operators that yielded similar or worse performance in terms of objective. Thus, intermittent scan was used instead in all cases in the final version of AHC.

For some of the operators, weighted randomness was used to select which of the best vertices or sets to act on. For example when choosing which vertex to move in a variation on **Move vertex**, a logarithmic function is used to assign 50% probability of picking the vertex of least cost, 25% probability to the second vertex and so on up to limit. For array length l, max allowed index i, random number r and logarithm base b, the weighted random function can be written as such:

$$\max(0, k - \lfloor \log_b((r \mod b^{\min(i,l)}) + 1) \rfloor - 1) \tag{3.1}$$

Note that adjusting the logarithm base affects the probability with which indices are chosen. The case where there is 50% chance to pick the first index, 25% to pick the second and so on uses $b = 2$.

3.2.5 Parallelization of operators

Some of the operators have been parallelized in order to examine the viability of parallelizing AHC. The pattern used for every parallelized operator distributes computational tasks done for every vertex of the graph between the available threads and uses prefix sum to merge the results into an array, similar to the prefix sum used in shared-memory BFS [64]. This pattern is applied to the scan of intermittent scan operators and the check of whether a vertex has been

used in the last integer i iterations, a check which is done in operators moving untouched vertices.

When dividing computation between all the vertices of the graph one could expect a speedup of p where p is the number of threads used by the algorithm. However, several factor prohibit this. One of these is that AHC uses datastructures that are not optimized for parallelization, like red-black search trees, which use links between nodes in the trees and thus cause elements closely located in the datastructure to not necessarily be closely located in memory. This results in poor utilization of cache, which is a well-known issue that may occur when writing parallel programs [64]. Usually one gets larger speedup when one is able to use arrays as datastructures, which is troublesome for CEVS where we want to be able to add and delete sets labeled with specific keys. An algorithm for CEVS with fixed cardinality of F could utilize arrays more and thus be more amenable to parallelization. Another factor is that parallelization is initiated each time the parallel part of an operator is reached, where both initializing new threads and synchronizing them once they are done are expensive procedures. Working around the frequent initiation would require more synchronization and threads that are inactive for large amounts of time, which makes this not seem like a viable approach. Finally, significant parts of the algorithm run sequentially and can therefore not be sped up with parallelization. As shown in the results, the best speedup achieved using the approach to parallelization described above is somewhat less than 2.

3.3 Reduction using critical cliques

We now consider using Lemma 8 from Abu-Khzam et al. [3] in order to reduce the size of the graph before applying metaheuristics. This lemma uses the definition of a critical clique graph $CC(G)$, a graph which can be made from every graph G so that each vertex in $CC(G)$ is derived from vertices of G that have the same closed neighborhood in G. Then, the vertices that share closed neighborhoods are said to form a critical clique and are mapped to a single common vertex in $CC(G)$. Let any set of vertices that share a closed neighborhood in G be called $S(v)$ for $v \in CC(G)$. Two vertices $u, v \in CC(G)$ have an edge if some vertices $w \in S(u)$ and $w' \in S(v)$ have an edge to each other in G, and because of the shared closed neighborhoods then every vertex in $S(u)$ and $S(v)$ have edges to one another in $E(G)$. It is proved by Lin, Jiang and Kearney [42] that each vertex of $V(G)$ is present in only one critical clique.

The lemma states the following:

Lemma 2. *[3, Lemma 8] Any solution $F = (S_1...S_l)$ to CEVS for a graph G that minimizes k will always satisfy the following property: for any $u \in G$, $v \in CC(G)$, $u \in S(v)$ and for any $S_i \in F$, either $S(v) \subseteq S_i$ or $S(v) \cap S_i = \emptyset$.*

Thus, every optimal solution of CEVS on G consists only of sets that are unions of sets corresponding to vertices of $CC(G)$.

Graph	Running time of reduction	Average running time of AHC
FARZ_test_0	0.245	21.779
FARZ_test_1	0.236	130.901
FARZ_test_2	0.267	122.954
FARZ_test_3	0.209	3.638
FARZ_test_4	0.269	195.076
FARZ_test_5	0.256	163.185
FARZ_test_6	0.228	28.315
FARZ_test_7	0.236	96.665
FARZ_test_8	0.237	63.972
FARZ_test_9	0.233	43.625

Table 3.1: Experiments comparing the running time for the reduction using critical cliques and the average running time of AHC. Time is given in seconds. Run on Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz.

Abu-Khzam et al. [3] use this fact to give a kernel, but since we here consider heuristics, a simpler application is possible. Since every vertex of $CC(G)$ is a subset of a set in an optimal solution of CEVS, we can execute the heuristics on $CC(G)$ instead of G. In order to make this work the vertices of $V(CC(G))$ are given the weight of the number of vertices in $V(G)$ with equal closed neighborhoods they correspond to, and the edges uv are given the weight of $|S(u)|\cdot|S(v)|$, corresponding to the number of edges between the vertices of $S(u)$ and $S(v)$ in $E(G)$. Then the operators can be applied to $CC(G)$ like they are applied to G, with the only difference being that it costs the weight of a vertex to split the vertex and the weight of an edge to add or remove the edge.

This reduction gives a new graph with fewer or the same number of vertices and edges. Guo [29] shows that a critical clique graph for input graph G can be found in $O(n + m)$ time, suggesting that the running time of the reduction is low compared to the running time of AHC, which experimental results in Table 3.1 confirm. Since the operators use the same number of calculations on an undirected graph as on a vertex- and edge-weighted graph, as weights of edges and vertices simply are relabelled from 1 to other integers, this reduction could enable more efficient execution of the operators by virtue of the operators being run on a smaller graph. In this thesis we run an experiment comparing the performance of AHC using this reduction to AHC not using this reduction.

3.4 ALNS

The versions of AHC used for experiments in the results in Chapter 6 use these eight operators: **Add vertex to neighbors**, **Add untouched vertices to neighbors**, **Move vertex**, **Move vertex** with a weighted randomness selection heuristic used to pick from the array of `goodVertices`, **Move untouched vertices**, **Remove vertex**, **Add vertex to set** and **Remove 3 then add 3**.

The adaptive agents use a segment size of $z = 300$ iterations between weight changes with a rate of change $r = 0.5$.

Now for the acceptance criteria we have used (see Section 2.5.1). We use the function $T_i = T_{i-1} \cdot \alpha$ for some number $0 < \alpha < 1$ to find temperature T_i of acceptance criteria $g(s', s_i, T_i)$. Here s_i is the solution that an operator is executed on in iteration i and s' is a solution that may be accepted, resulting from an action found by an operator executed in iteration i. Variable α is set once over the execution of the algorithm, using a warm-up sequence. This is a sequence of 1000 iterations where any action with positive cost is accepted with 0.8 probability and the average cost of actions with positive cost is calculated. The average cost is used to set T and α so that there is 0.8 probability that an action with positive cost is accepted in the first iteration after the warm-up sequence, and so that $T_{i_{\max}} = T_{\text{end}}$ for some end temperature T_{end} in the last iteration of the algorithm execution, giving a negligible probability of accepting an action with positive cost. Written with formulas, the temperature T_{start} is set to

$$T_{\text{start}} = -C_w \cdot \frac{1}{\ln(0.8)}$$

where C_w is the average difference between new worse solutions and previous solutions calculated during the warm-up sequence. Note that substituting C_w for $f(s') - f(s)$ and T_{start} for T in the acceptance probability formula in Equation 2.2 gives a probability of 0.8 of accepting a new solution that has an objective that is C_w more than the previous solution. Then, we derive from

$$T_{\text{end}} = \alpha^{i_{\max} - i_{\text{start}}} \cdot T_{\text{start}}$$

that α should be set to

$$\alpha = \frac{T_{\text{end}}}{T_{\text{start}}}^{\frac{1}{i_{\max} - i_{\text{start}}}}$$

where i_{start} is the first iteration after the warm-up sequence and $i_{\max}$ is the final iteration.

For some versions of AHC we have used a slightly unorthodox reward function. An operator is awarded 7 points if it finds a solution that has not been found previously in the algorithm execution, 2 points if the solution it finds is better than the previous solution and 5 points if it finds a new best solution. Here each reward condition is evaluated independently, so that one solution may gain $7 + 2 + 5 = 14$ points if it fulfills all conditions. Let this function be named the 725-function. A more conventional approach would be a function that gives a small amount of points for finding a new solution, some more points for finding a solution that is better than the last and the most points for finding a new best solution, which for example was done by Hemmati et al. [30]. Let the 123-function be a function that gives respectively 1, 2 and 3 points for each

Graph	Avg objective 725	Avg objective 123	Running time 725	Running time 123
val_10.txt	561.2	561	40.26	33.45
val_11.txt	1483.2	1486.4	153.45	121.50
val_12.txt	2383.8	2384.2	187.7	157.25
val_13.txt	1010.2	1012	40.03	32.88
val_14.txt	1811.8	1813	127.12	113.314
val_15.txt	1066.2	1069	40.67	33.80
val_16.txt	303.4	303.2	22.95	17.91
val_17.txt	498.4	498.6	13.77	11.66
val_18.txt	1052.6	1055.6	27.16	21.26
val_19.txt	682.6	683.8	34.79	26.63

Table 3.2: Comparing reward functions used with ALNS. The two left columns show average CEVS-score of the found solutions, and the two right columns show the corresponding average running time in seconds. Run on AMD Ryzen 5 3600.

725-function	123-function
$5.491 \cdot 10^{-3}$	$4.309 \cdot 10^{-3}$

Table 3.3: Percent average improvement versus URA for ALNS with different reward functions.

scoring criteria in the previous sentence, but gives score for satisfying the criteria with highest assigned score only.

Experiments show that choosing the 725-reward function over the 123-function yields marginally better results, although it requires a slightly longer running time (see Tables 3.2 and 3.3). Note that the variation in running time can be attributed to the fact that some operators are more time consuming than others (see Section 6.5). As the 725-reward function is more diversifying than the 123-function, with more reward given for a new solution, this suggests that emphasizing diversification leads to better performance for this ALNS-implementation and the pool of operators that is used. However, using the 123-function and increasing the number of iterations is also a reasonable approach. In the interest of keeping to a fixed number of iterations for comparison, the 725-function is used for the experiments in the Section 6 unless otherwise is stated as this is the function that gave the best results in terms of CEVS-score for AHC with 100'000 iterations in these preliminary experiments.

3.5 DRLH

DHC uses an acceptance criteria similar to that described for AHC, with a warm-up sequence of 1000 iterations. Two types of reward functions have been tried, this being the same two functions used for AHC. We run the algorithm with 100'000 iterations, and the agents are trained on 100, 1000 and exclusively for DHC with 123-function 2000 training sets. The state used by the agent stores

the following properties:

- **Reduced distance:** The change in cost after an iteration.
- **Distance from minimum:** The difference in cost from the current solution to the minimum-cost observed solution over the course of the run.
- **No improvement:** Stores the number of iterations since last iteration wherein the cost of the solution improved.
- **Index step:** Stores the iteration number i, where $0 \leq i \leq i_{\max} = 100'000$.
- **Was changed:** Boolean indicating whether the solution changed in the last iteration.
- **Unseen:** Boolean indicating whether the current solution has been seen in a previous iteration, using hashes of solutions to compute this.

The chosen properties are a subset of those in the state representation used by Kallestad, Hemmati and Hasibi [36], and the above is a suggested state representation in the DRLH framework that they have developed, which is used for DHC.

The preceding configurations for DRLH were chosen in order to present results representative of what DHC achieved during experiments and highlight issues with DHC. We take this approach as the results yielded by DHC were subpar compared to those obtained by AHC.

Chapter 4

Algorithms to solve CEVS parameterized by maximum vertex degree

Recall that a graph without any P_3 is a cluster graph, so that eliminating every P_3 from a graph yields a cluster graph and therefore a solution to CEVS. The following algorithms aim to solve CEVS for a graph by finding P_3s in the graph and eliminating them by branching on the possibilities of adding an edge, deleting one of two edges or splitting the middle vertex in a number of ways limited by the maximum vertex degree of any vertex in the input graph.[1] We give algorithms for both CEVS with inclusive split and CEVS with exclusive split, with running time parameterized by maximum length of sequence of operations k and maximum vertex degree of any vertex in the input graph d.

Consider the case of inclusive vertex split. For each of the two vertices created, there are at less than

$$1 + \binom{d+k}{1} + \binom{d+k}{2} + \cdots + \binom{d+k}{d+k} = (1 + (d+k))^{(d+k)}$$

choices of edges adjacent to the deleted vertex to include. The maximum vertex degree in a branching step is $d + k$ since the splitting operation can duplicate edges, adding one to the degree of a vertex in the neighborhood of both the new vertices, and also since the addition operation adds edges. Thus, there are $O((d+k)^{2(d+k)})$ possible splits. Since it is possible to find a P_3 in polynomial time[2] and since one can only branch k times while using one of $3+O((d+k)^{2(d+k)})$ operations for each branch, this results in an $O^*((d+k)^{2(d+k)k})$ exact algorithm.

A better upper bound running time is obtained for CEVS with exclusive split since there are fewer cases to consider when splitting. Specifically, represent

[1] See Cygan et al. [15] for an overview of the topic of branching algorithms.

[2] For every vertex in the graph, check every pair of adjacent vertices and see if they have an edge to each other.

Graph	Vertices	Edges	AHC best	Optimal	AHC running time	Exact running time
gr_0.txt	10	13	4	4	0.093	3.189
gr_1.txt	8	11	6	6	0.163	578.0
gr_2.txt	8	19	2	2	0.071	1.82
gr_3.txt	8	7	4	4	0.132	0.232
gr_4.txt	8	13	5	5	0.161	1996.6

Table 4.1: Comparison of AHC and the exact branching algorithm finding optimal solutions for CEVS on small graphs. Run on Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz. Running time is given in seconds. *AHC best* and *optimal* refer to the objective of CEVSOPT.

each splitting operation of a vertex with degree d' as a binary string of length d' where 0 corresponds to edges belonging to the first new vertex and 1 corresponds to edges belonging to the second new vertex. The vertex degree of any vertex may also here increase to $d + k$ because of edge addition. Clearly, this gives us an $O^*(2^{(d+k)k})$ algorithm.

The branching algorithm for CEVS with inclusive split is used to find optimal solutions of CEVS on set of small graphs we have created by hand. Table 4.1 shows that AHC uses less time than an implementation of this exact algorithm to find optimal solutions for these graphs. Note that the time given for the branching algorithm sums the time the algorithm uses for several inputs of k, for example for $k = 4$ (returning NO) and $k = 5$ (returning YES) when the optimal solution is 5.

Chapter 5

Solving CEVS on complete bipartite graphs with formula

Complete bipartite graphs present a challenge for non-bipartite clustering algorithms like CEVS since they are dense[1], leading to slower execution times, and since there are edges only between the two bipartite sets and not between vertices within the sets, making clustering algorithms yield results that do no match the bipartite sets. We also note that a graph with community structure should have many cycles of length three, e. g. three people knowing each other in a social network, while bipartite graphs have no odd cycles [5]. Investigating algorithms to solve CEVS on bipartite graph are interesting since it could inform the discovery of a crown decomposition for CEVS, which would yield a linear size kernel [15].

By running AHC on complete bipartite graphs we found that the minimum solutions showed a pattern which we describe in the following and have named c-d-form. In these solutions each edge has both endpoints in the same set of F so that there is no edge deletion operation in the minimal sequence of operations corresponding to the solution. Also, the vertices of each bipartite set are split in groups of equal size with an error of one, these groups being present in one or several sets of F and never together with any other group from the same bipartite set. The integers c and d give how many groups the two bipartite sets respectively should be split between. See Figure 5.3 for a visualization. We proceed with a larger degree of formalism.

Definition 4 (c-d-form)**.** *A solution of CEVS $F \subseteq 2^V$ for a complete bipartite graph $G = (V, E)$ with bipartition (A, B), $A, B \subseteq V$, $|A| \leqslant |B|$ is on c-d-form if $|F| = cd$ for integers $1 \leqslant c \leqslant |A|$ and $1 \leqslant d \leqslant |B|$ and each vertex of A and B is*

[1] I. e. many edges per vertex.

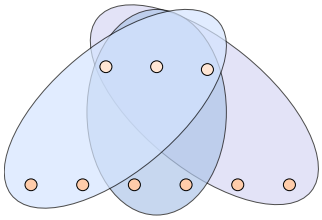


Figure 5.1: $K_{3,6},\ (c,d) = (1,3)$

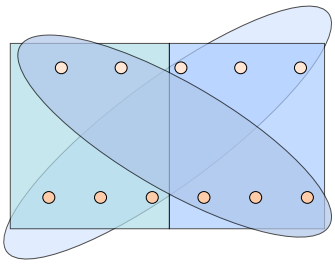


Figure 5.2: $K_{5,6},\ (c,d) = (2,2)$

Figure 5.3: Visualizations of solutions on c-d-form. The upper vertices and lower vertices correspond to sets A and B of the complete bipartite graphs respectively. We have not drawn the edges since they are the same as those in the complete bipartite input graph and would clutter the drawings. The ellipses and rectangles indicate which vertices belong to the same sets.

in exactly d and c of the sets of F respectively. Furthermore, for any two sets $S_1, S_2 \subseteq F$, $|S_1 \cap A| - |S_2 \cap A| \leqslant 1$ and $|S_1 \cap B| - |S_2 \cap B| \leqslant 1$.

Let a group be a set of vertices either exclusively from A or exclusively from B so that a vertex u in a group co-occurs with another vertex v from the same bipartite set in a cluster in the resulting cluster graph if and only if u and v are in the same group. The integers c and d correspond to how many groups the vertices in A and B respectively are distributed between. Since each set of F has the same cardinality within an error of 1, either $\lfloor |A|/c \rfloor$ or $\lceil |A|/c \rceil$ vertices in each set are from A and $\lfloor |B|/d \rfloor$ or $\lceil |B|/d \rceil$ vertices in each set are from B. A consequence of the definition is that each vertex in A is split $d-1$ times, and each vertex in B is split $c-1$ times.

Conjecture 1. *The following formula gives the minimal objective of* CEVSOPT *on complete bipartite graphs with bipartition (A,B) where $|A| = s > 1, |B| = t > 1, |A| \leq |B|$:*

$$\min_{1 \leq c \leq s, 1 \leq d \leq t} h(c,d) \tag{5.1}$$

where c and d are integers such that $1 \leq c \leq s$ and $1 \leq d \leq t$, and h is defined as such:

$$\begin{aligned} h(c,d) = {} & s(d-1) + t(c-1) + (s \pmod c)\binom{s/c}{2} \\ & + (c - (s \pmod c))\binom{s/c - 1}{2} + (t \pmod d)\binom{t/d}{2} \\ & + (d - (t \pmod d))\binom{t/d - 1}{2} \end{aligned} \tag{5.2}$$

Also, $h(c,d)$ returns the objective of a solution on c-d-form.

Conjecture 1 gives the cost of splitting vertices so that they can be in several sets in F and adding edges between all vertices in the same group for every group. The first addend of Equation 5.2 gives the cost of splitting the s vertices of A $(d-1)$ times so that each of the vertices can share presence in a set of F with every vertex of B, the s vertices of A being present in d sets each. The second addend gives the converse for the t vertices of B. Next, note that for vertices in a group C, we need to add $\binom{|C|}{2}$ edges between the vertices of C for C to be part of cluster in the resulting cluster graph. This explains the four last addends of Equation 5.2, with the two first corresponding to adding edges between vertices in A and the two last corresponding to adding edges between vertices in B. We need to split the calculations for each bipartite set into two addends since c and d do not necessarily divide s and t respectively, yielding groups with cardinalities different from each other by at most 1.

An experiment were conducted where AHC was run five times on all complete bipartite graphs where the cardinality of each bipartite set was at least 2 and at most 19. The best solutions found had the same objective as that predicted by conjecture 1 for all but the following set cardinalities $(|A|, |B|)$ for bipartite sets A and B: (12, 18), (13, 13), (14, 14), (14, 15), (14, 19), (15, 15), (15, 18), (16, 17), (16, 18), (16, 19), (17, 17), (17, 18), (17, 19), for which the least objective found by AHC was slightly larger. This suggests that Conjecture 1 is correct, and that AHC does not find these presumably optimal solutions on larger graphs. Note that AHC is optimized with regard to solving CEVS for graphs with data in clusters and therefore should not be expected to perform as well when applied to bipartite graphs.

We believe that Conjecture 1 can be proved by induction from base step $K_{2,2}$, and we leave this as an open problem.

Chapter 6

Results

6.1 Data sets used in experiments

The experiments in this section are run on both artificial and real-world data sets. We used the benchmark generators LFR [40] and FARZ [57] to generate artificial graphs with given attributes like size in terms of vertices and edges and maximum vertex degree, where both generators provide ground truth communities with each generated graph. The real-world data sets have been downloaded from the online networks catalog Netzchleuder [55]. These real-world data sets are used with no preprocessing with exception of eu_airlines, which has been converted from a directed graph to an undirected graph by considering each directed edge as an undirected edge. We run most of the experiments on a data set we name the test data set. This data set consists of 9 real-world graphs and 10 artificial graphs generated using FARZ, and the names we have given these graphs can be read in Table 6.1.[1]

6.2 Robustness of AHC

Initially we examine the robustness of AHC by running the algorithm 5 times on the graphs in the test data set. The results in Table 6.1 show that AHC finds the same objective or objectives only slightly larger every time on the smaller graphs, and that the distance between best and average objective increases with the size of the graph in terms of vertices and edges. Also, the average objective is never more than a percent off from the best objective, suggesting AHC is robust when it comes to finding high quality solutions of CEVS. The fact that the same best objective is found every time for some of the smaller graphs suggests that these

[1]The names of the real-world graphs and corresponding references are listed here: cs_department [45], eu_airlines [11], facebook_friends [46], football [25], game_thrones [8], jazz_collab [26], karate78 [74], law_firm [41], revolution [39], social_circles [47] and lastfm_asia [63].

Graph name	Vertices	Edges	Best objective	Avg. objective	Avg. time
cs_department	61	353	140	140.2	9.04334
eu_airlines	450	2953	1712	1715.8	90.6118
facebook_friends	362	1988	759	764.6	31.5495
football	115	613	268	268.6	6.14791
game_thrones	107	352	174	174	12.7105
jazz_collab	198	2742	618	620.2	62.7093
karate78	34	78	43	43	2.71692
law_firm	71	1008	406	408	37.1806
revolution	141	160	150	150	21.9954
FARZ_test_0	405	1302	846	846.4	21.7785
FARZ_test_1	258	3296	1864	1868.8	130.901
FARZ_test_2	313	5507	1936	1942.4	122.954
FARZ_test_3	36	186	65	65	3.63799
FARZ_test_4	243	5524	2181	2186.8	195.076
FARZ_test_5	250	5577	1827	1833.4	163.185
FARZ_test_6	444	1651	1160	1162	28.3153
FARZ_test_7	295	3056	1833	1835.4	96.6649
FARZ_test_8	380	2556	1778	1779.4	63.9715
FARZ_test_9	411	2024	1403	1406	43.6251

Table 6.1: Demonstrating the robustness of AHC when run 100'000 iterations 5 times. Run on AMD Ryzen 5 3600. Running time is given in seconds.

Graph name	Vertices	Edges	Best objective	Avg. objective	Avg. time
social_circles	4039	88234	33347	33459	1134.86
lastfm_asia	7624	27806	18746	18792.8	575.82

Table 6.2: Demonstrating the robustness of AHC when run 100'000 iterations 5 times on a couple of larger graphs. Run on Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz. Running time is given in seconds.

Graph name	URA	ALNS	ALNS with CC
cs_department	140.2	140.2	140.8
eu_airlines	1713.4	1715.8	1717.8
facebook_friends	764.6	764.6	766.4
football	268	268.6	268.6
game_thrones	174	174	174
jazz_collab	621.2	620.2	619
karate78	43	43	43
law_firm	413	408	408.8
revolution	150	150	150
FARZ_test_0	847	846.4	847.8
FARZ_test_1	1878	1868.8	1866
FARZ_test_2	1986.4	1942.4	1942.4
FARZ_test_3	65	65	65
FARZ_test_4	2223.6	2186.8	2185.4
FARZ_test_5	1882.8	1833.4	1833
FARZ_test_6	1162.2	1162	1162
FARZ_test_7	1846.4	1835.4	1834.2
FARZ_test_8	1785.6	1779.4	1781.6
FARZ_test_9	1405.8	1406	1405.4

Table 6.3: Average objective over 5 runs using 100'000 iterations with different agents. Run on AMD Ryzen 5 3600.

objectives may be optimal. We also run the experiment on a couple of larger graphs to demonstrate that AHC is robust for these as well (see Table 6.2).

6.3 Compare URA, ALNS and DRLH

Then we compare the performance of the ALNS agent with URA. Thus, URA provides a baseline that can be used to evaluate the performance of the ALNS agent. The results in Tables 6.4 and 6.5 show that ALNS give slightly better results than URA, with an average of $5 \cdot 10^{-3}$ percent improvement in objective. However, the running time is in the worst case about 17% longer for ALNS than for URA, which likely can be attributed to URA picking computationally cheaper operators more frequently than ALNS. For the graphs with largest CEVS-score, ALNS gives better results than URA, suggesting that the difference in performance between ALNS and URA increases for larger data sets.

Furthermore, the performance of ALNS with and without using the critical clique reduction is examined. ALNS with reduction obtains a marginally worse objective than ALNS without the reduction, and using the reduction increases the running time for every graph in the test data set. This suggests that the reduction is not useful in general, although the poor results may be attributed to few occurrences of critical cliques in the graphs in the test data set.

Graph name	URA	ALNS	ALNS with CC
cs_department	7.924 18	9.043 34	9.403 27
eu_airlines	76.0735	90.6118	100.646
facebook_friends	23.2923	31.5495	31.8812
football	7.067 75	6.147 91	6.893 47
game_thrones	9.822 84	12.7105	13.9336
jazz_collab	45.8752	62.7093	68.5939
karate78	3.134 96	2.716 92	3.196 35
law_firm	35.6836	37.1806	42.3237
revolution	18.8828	21.9954	26.0216
FARZ_test_0	15.3862	21.7785	26.439
FARZ_test_1	110.28	130.901	149.161
FARZ_test_2	99.4255	122.954	127.255
FARZ_test_3	5.101 04	3.637 99	4.806 17
FARZ_test_4	183.816	195.076	198.036
FARZ_test_5	139.6	163.185	167.498
FARZ_test_6	20.8094	28.3153	31.1387
FARZ_test_7	83.9227	96.6649	112.404
FARZ_test_8	50.528	63.9715	74.4324
FARZ_test_9	33.0601	43.6251	45.6797

Table 6.4: Average running time over 5 runs using 100'000 iterations of different metaheuristics. Run on AMD Ryzen 5 3600.

ALNS		ALNS with CC	
Objective	Running time	Objective	Running time
$4.741 \cdot 10^{-3}$	-0.1669	$4.358 \cdot 10^{-3}$	-0.2953

Table 6.5: Average improvement in objective and running time vs URA for ALNS with and without utilizing the critical clique graph. Observe that the average objective is better while the average running is worse for both programs compared to URA.

DHC yielded overall worse results than both AHC and URA when run on the artificial graphs in the test data set. Table 6.6 shows that the results became worse with a larger training set for the 725-reward function, suggesting that the reinforcement learning agent is able to abuse the rules of DRLH so that it obtains reward for choosing actions that do not give improvement in objective for CEVS. We see some of the same for the 123-reward function, although this does better for 1000 training sets because it uses more expensive operators (like **Remove 3 then add 3**), indicated by the comparatively larger running time.

A possible explanation for the poor performance of DRLH could be that the operators in the pool are complex and involve too much randomness and storing of future operations for DRLH to efficiently evaluate when it is reasonable to use an operator. The most successful operators for the pick-up and delivery problems examined by Kallestad, Hemmati and Hasibi [36] were heuristics that removed a part of the solution and inserted it somewhere else, trying all possibilities of removal and insertion and picking the best or one of the best possibilities. This is a direct and simplistic approach compared to intermittent scan, which most of the operators used for CEVS rely on and which performed well together with both ALNS and URA. Future work could try to use a different pool of simpler operators for solving CEVS and compare the performance of ALNS and DRLH when using this pool.

Another explanation could be the fact that CEVS is a less structured problem than pick-up and delivery problems. Whereas a solution to CEVS for a graph G only has the constraint of being a family of sets F so that each vertex in $V(G)$ is in some set in F, the pick-up and delivery problems consists of many constrains like vehicles with limited capacity, packets of a certain size that can only be delivered by certain vehicles, time-windows for delivery, distance between delivery destinations and more. These constraints could be beneficial for DRLH and make it so that DRLH performs better than ALNS for these problems, while the lack of constraints in CEVS makes DRLH perform worse for the problem than ALNS does.

Graph	725, t=100	725, t=1000	123, t=100	123, t=1000	123, t=2000
FARZ_test_0	851.4	874.2	856.2	875.6	883.2
FARZ_test_1	1892.8	1928.8	1929.4	1880.6	1927.4
FARZ_test_2	1972	2030	2020	2076.8	2020
FARZ_test_3	65	65	65	65	66
FARZ_test_4	2214.6	2259	2278.8	2201.4	2241.6
FARZ_test_5	1869	1915.4	1919.2	1844.2	1902.8
FARZ_test_6	1173	1207.8	1183	1219.6	1211.4
FARZ_test_7	1859.8	1893.6	1911	1847.4	1901.8
FARZ_test_8	1803.6	1836.4	1857.6	1824.6	1854
FARZ_test_9	1420.8	1455.8	1457.8	1447.4	1469.4

Table 6.6: Average objectives of DRLH run 5 times with 100'000 iterations, run on AMD Ryzen 5 3600. The variable t gives the size of the training set.

Graph	725, t=100	725, t=1000	123, t=100	123, t=1000	123, t=2000
FARZ_test_0	191.58	197.94	181.18	363.65	194.08
FARZ_test_1	116.00	173.76	84.80	114.18	145.89
FARZ_test_2	102.36	156.87	61.85	628.45	110.23
FARZ_test_3	36.71	37.98	35.15	45.16	37.47
FARZ_test_4	144.67	258.36	97.67	153.78	184.78
FARZ_test_5	117.98	189.91	77.11	127.06	135.13
FARZ_test_6	67.74	73.67	53.33	577.05	68.16
FARZ_test_7	103.39	132.72	65.14	93.61	122.47
FARZ_test_8	89.39	106.91	54.64	255.48	99.50
FARZ_test_9	77.82	89.70	55.25	237.52	85.02

Table 6.7: Average running time of DRLH run 5 times with 100'000 iterations, run on AMD Ryzen 5 3600. The variable t gives the size of the training set.

6.4 Running time scaling

Calculating the theoretical worst-case running time of AHC is a difficult exercise because of the non-determinism and randomness of the execution. Neither is it particularly useful as the agent in AHC is unlikely to pick the most expensive operators every time. A more fruitful and informative approach is to investigate how the running time scales for graphs of different sizes and attributes.

Before executing scaling experiments it is prudent to examine which features of a graph that are reasonable parameters to scale. In order to achieve this, we execute experiments with AHC with 10'000 iterations run 5 times on 117 graphs. The number of vertices, the number of edges, the number of P_3s, the difference between best and average solution and the average running time are recorded, and we make a covariance matrix included in Figure 6.1 using these results. From the matrix it is clear that the running time correlates strongest with the number of edges in the graph, and almost as strong with the number of P_3s. From these results we conclude that the number of edges is the best parameter to use to scale graphs for the scaling experiments, both because of the high correlation with running time and because it is a simpler measure than the possible alternative of amount of P_3s.

Accordingly, AHC is run on graphs with increasing amounts of edges and similar amounts of vertices in proportion to the number of edges. Figure 6.2 indicates running time growth linear in the number of edges in the graph with a growth rate of approximately $2.1 \cdot 10^{-3}$ per added edge, or one second added per 500 edges. This is promising with regard to executing the algorithm on larger graphs than those we have examined in this thesis.

Furthermore, we investigate the scaling of running time of AHC with maximum vertex degree. Here the LFR benchmark generator is used to generate

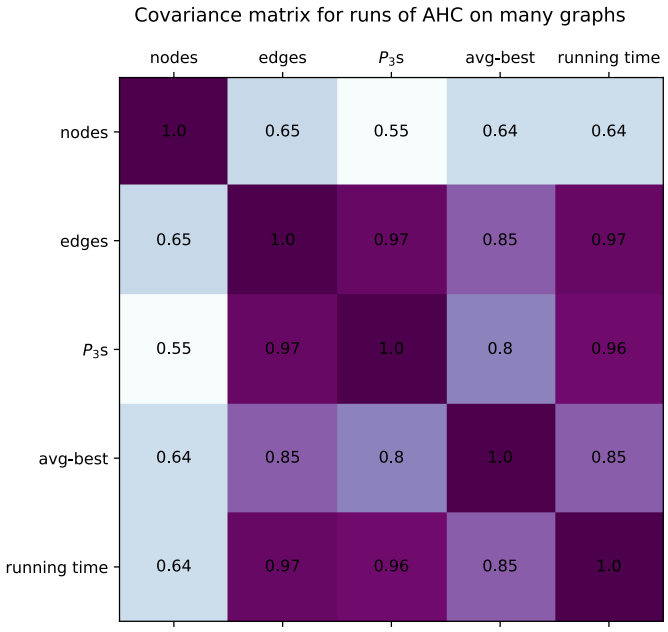


Figure 6.1: Covariance matrix derived from running AHC with 10'000 iterations 5 times on 117 graphs.

artificial graphs with increasing allowed maximum degree of vertices and otherwise similar attributes. The results are visualized in Figure 6.3 and Table 6.9 and suggest that the running time of AHC scales with the maximum allowed vertex degree in the graphs. Included in Figure 6.3 is the average running time plotted against the number of edges, which shows that scaling of running time with number of edges in the graph is less obvious than scaling with maximum degree.

6.5 Running time of operators

We include this subsection in order to give an impression of the variation in running time between the operators, and the distribution of the running times of a single operator when it is executed several times during a run of AHC. Figure 6.4 and Table 6.10 show statistics related to the operators gathered over the course of 1 run with 100'000 iterations of AHC on the graph jazz_collab. The boxplot in Figure 6.4 shows that all operators have time-consuming outliers. **Remove 3 then add 3** has the largest outliers, which likely occurs when a P_3 with vertices with many neighborhood sets are chosen. Also, this operator has

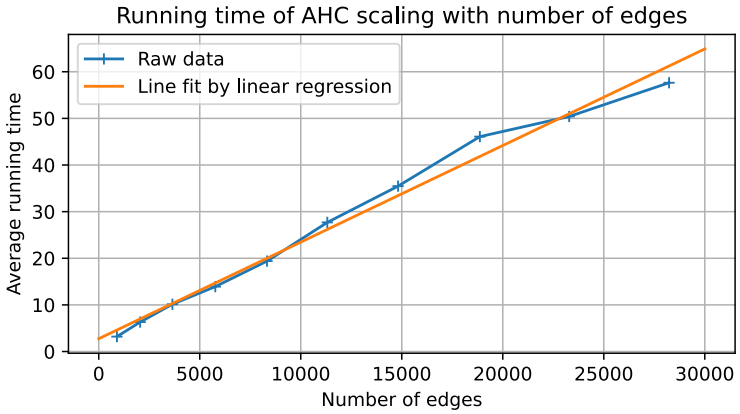


Figure 6.2: Results of scaling experiments of AHC with 10'000 iterations run 5 times on Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz, 7.9.2022

the largest median and average, indicating that this is the most time-consuming operator in general. For the intermittent scan operators (the five furthest to the left in Figure 6.4) the outliers can be explained by the refill-sequences where scans of the entire graph are performed, while most executions of these operators only consider actions for a single vertex or set. All operators have a set of outliers right above $Q3 + 1.5 \cdot \text{IQR}$[2], suggesting that some procedures common to all operators make the operators slower in relatively few cases.

[2]Q3 is the value at the top of the boxes in the plot, so that 75% of the data points have values below. Q1 is the value at the bottom of the boxes with 25% of the data points below. Then IQR (Interquartile range) is (Q3 − Q1), and the whiskers of the boxplot are drawn at the values $Q3 + 1.5 \cdot \text{IQR}$ and $Q1 - 1.5 \cdot \text{IQR}$ respectively.

Graphs	Edges	Average running time
FARZ_scale_0	900	3.19937
FARZ_scale_1	2051	6.32058
FARZ_scale_2	3647	10.1416
FARZ_scale_3	5769	13.919
FARZ_scale_4	8330	19.3919
FARZ_scale_5	11315	27.6948
FARZ_scale_6	14818	35.4722
FARZ_scale_7	18866	46.088
FARZ_scale_8	23288	50.4095
FARZ_scale_9	28232	57.6458

Table 6.8: Results of scaling experiments of AHC with 10'000 iterations run 5 times on Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz, 7.9.2022. Corresponds to Figure 6.2.

Graphs	Edges	Maximum degree	Average running time
LFR_scale_maxdeg_0	4500	30	17.967
LFR_scale_maxdeg_1	4483	35	19.5749
LFR_scale_maxdeg_2	4557	40	14.1715
LFR_scale_maxdeg_3	4521	45	17.6336
LFR_scale_maxdeg_4	4449	50	18.2918
LFR_scale_maxdeg_5	4423	55	16.1843
LFR_scale_maxdeg_6	4456	60	19.0757
LFR_scale_maxdeg_7	4407	65	20.3491
LFR_scale_maxdeg_8	4362	70	30.7944
LFR_scale_maxdeg_9	4486	75	31.4222
LFR_scale_maxdeg_10	4894	80	32.5607
LFR_scale_maxdeg_11	4472	85	31.6595
LFR_scale_maxdeg_12	4461	90	31.0689
LFR_scale_maxdeg_13	4477	95	31.0299
LFR_scale_maxdeg_14	4319	100	34.0613
LFR_scale_maxdeg_15	4571	105	31.3374
LFR_scale_maxdeg_16	4505	110	52.7346
LFR_scale_maxdeg_17	4530	115	39.7545
LFR_scale_maxdeg_18	4404	120	37.7999
LFR_scale_maxdeg_19	4258	125	36.9591
LFR_scale_maxdeg_20	4560	130	47.3889
LFR_scale_maxdeg_21	4260	135	33.5064
LFR_scale_maxdeg_22	4479	140	45.704
LFR_scale_maxdeg_23	4361	145	40.486

Table 6.9: Results of scaling experiments of AHC with 30'000 iterations run 5 times on Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz, 7.9.2022. The number of vertices (300) is the same for all graphs, and the average degree of vertices (30) is approximately the same. Corresponds to Figure 6.3.

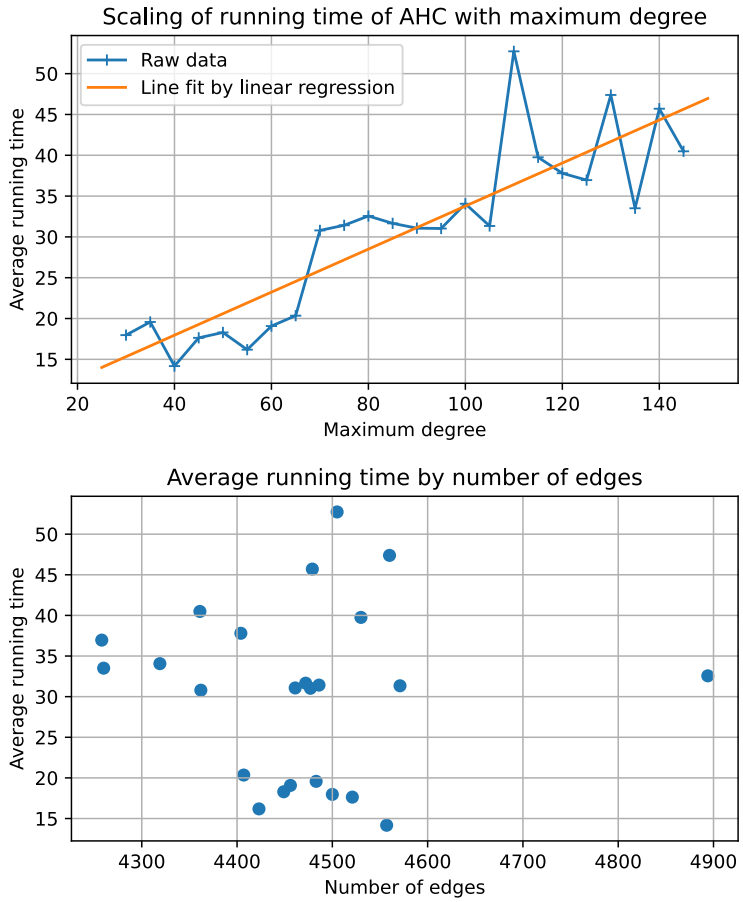


Figure 6.3: Results of scaling experiments of AHC with 30'000 iterations run 5 times on Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz, 7.9.2022. The number of vertices (300) is the same for all graphs. The above plot demonstrates scaling with maximum degree in the graph, while the bottom plot in comparison to the above plot indicates that increase in running time is better explained by increase of maximum degree than increase of number of edges.

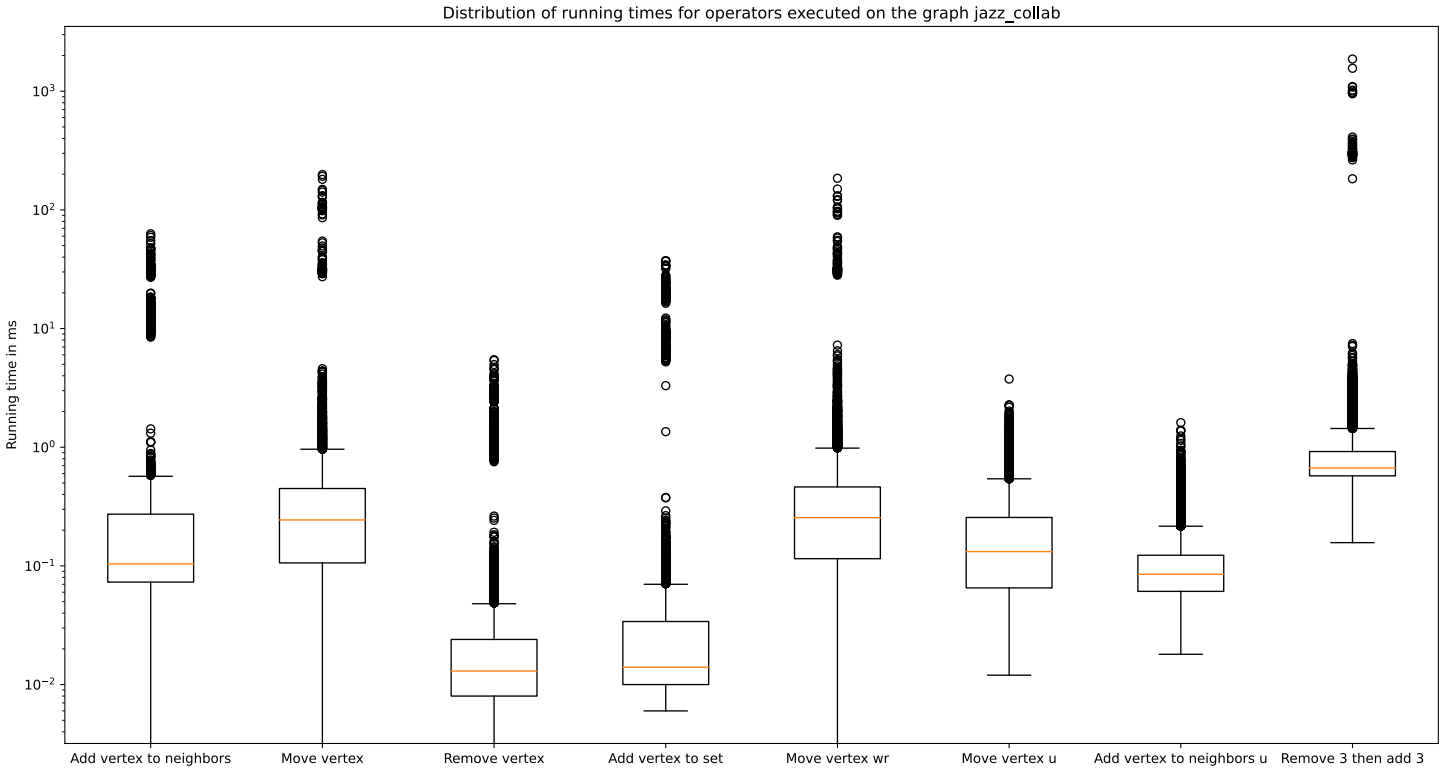


Figure 6.4: Boxplot showing the distrubution of the running times of the operators of AHC over 1 run with 100'000 iterations of AHC on the graph jazz_collab. Note that the y-axis is logarithmic.

Operator	Min	Max	Median	Average	Picked
Add vertex to neighbors	0	62.877	0.104	2.021	16618
Move vertex	0	198.611	0.244	0.983	9347
Remove vertex	0	5.469	0.013	0.103	17900
Add vertex to set	0.006	37.341	0.014	0.396	10557
Move vertex wr	0	184.743	0.255	0.888	8567
Move vertex u	0.012	3.755	0.132	0.208	7878
Add vertex to neighbors u	0.018	1.613	0.085	0.118	14872
Remove 3 then add 3	0.157	1868.3	0.668	2.796	14261

Table 6.10: This table corresponds to Figure 6.4. Shows statistics for the running times of the operators of AHC during 1 run with 100'000 iterations of AHC on the graph jazz_collab. The middle four columns display statistical measures of the running time of the operators given in milliseconds, and the column *Picked* shows the amount of times each operator was picked by the adaptive agent over the course of the run. Weighted randomness is abbreviated as wr.

6.6 Parallelization of operators

In these experiments we use a version of AHC with 6 parallelized operators and no non-parallelized operators. These operators are **Move vertex**, **Move untouched vertex**, **Add vertex to set**, **Add vertex to neighbors**, **Remove vertex** and **Add untouched vertices to neighbors**. AHC is run five times on three different graphs for a varying number of processors on the shared-memory parallel computer Brake[3] owned by the University of Bergen. The running times and the speedup are plotted in Figure 6.5, showing that we achieve a speedup of about 1.5 for the examined graphs, except for one run on FARZ_200_40 which yields a speedup of 1.8. This demonstrates that parallelization is applicable to heuristics for solving CEVS.

In Figure 6.6 we have plotted the speedup of the operators calculated using the average running time used each time an operator is called during a run. These results show that certain operators benefit more from parallelization than others.

6.7 Properties of solution communities with comparison to other algorithms

In this section we aim to examine the performance and solution characteristics of AHC compared to other algorithms also solving the task of overlapping community detection. The algorithms chosen for comparison are BIGCLAM [73, 62], Demon [13, 14, 61], SLPA [72, 1] and CFinder [52, 53]. These were picked because they have previously been used for such comparisons [69] and because implementations are available online. As suggested by Vieria et al. [69],

[3]Consists of 80 Intel® Xeon® CPU E7-4850 2.00GHz processors.

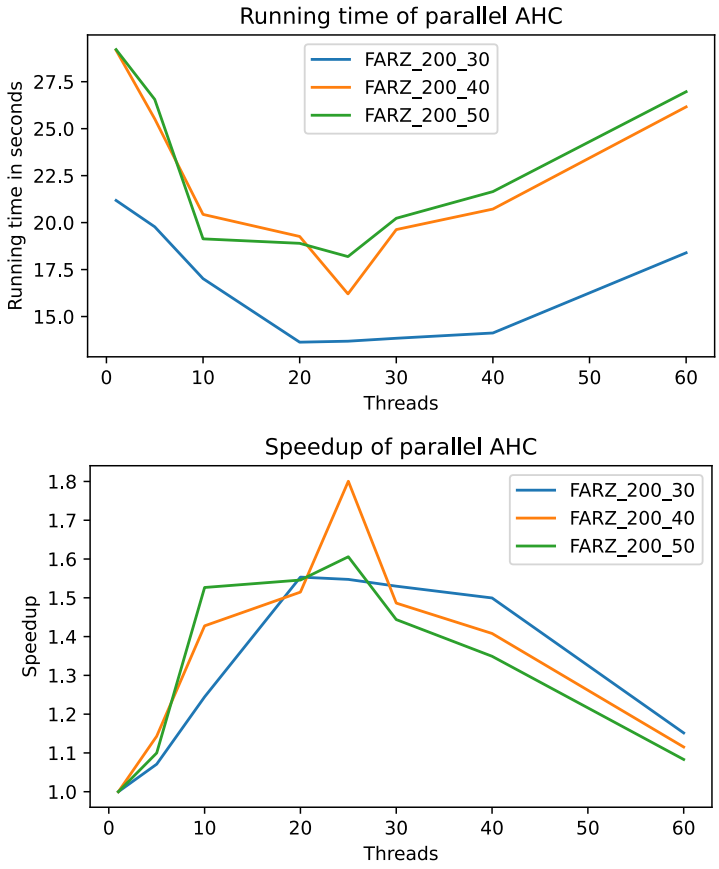


Figure 6.5: Running time and speedup for parallel AHC with different amount of threads.

optimizations towards an objective function yields an implicit assumption about "the characterization of an overlapping community structure". This implies that each algorithm examined here, including AHC, yields solutions with certain characteristics. By evaluating the solutions using ONMI, EQ and CEVS-score, we attempt to discover such characteristics.

The running times of the algorithms on the nine real-world graphs in the test data set are plotted in Table 6.11. It is clear that AHC is significantly slower than the other algorithms, except CFinder which for larger graphs may be even slower than AHC.

One characteristic in which solutions found by different algorithms are different is in the distribution of the sizes of communities. In Figure 6.7 we plot this distribution for solutions of highest EQ found on several graphs by different algorithms. AHC frequently yields more communities and a higher number of small communities compared to the other algorithms, whose solutions overall

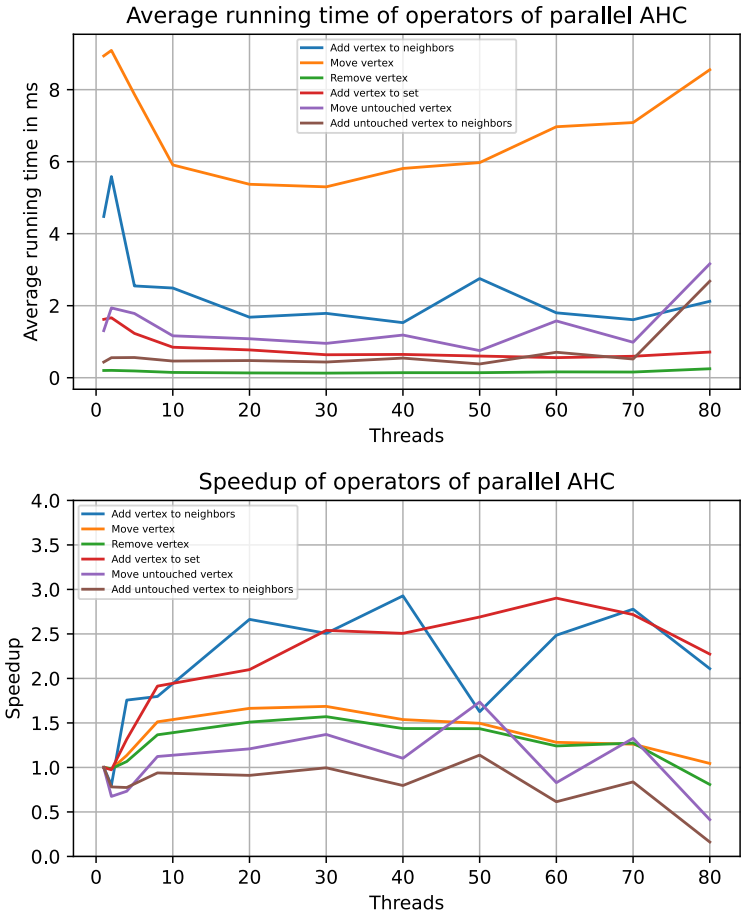


Figure 6.6: Average running time in milliseconds and speedup of parallelized operators of AHC using a varying amount of threads, run on the graph FARZ_200_50.

Graph	AHC	BIGCLAM	Demon	SLPA	CFinder
cs_department	9.043	0.203	0.051	0.067	0.085
eu_airlines	90.612	1.476	0.659	0.615	177.222
facebook_friends	31.550	1.119	0.362	0.377	1.624
football	6.148	0.347	0.073	0.116	0.080
game_thrones	12.711	0.319	0.045	0.072	0.051
jazz_collab	62.709	0.635	0.665	0.494	9.076
karate78	2.717	0.101	0.009	0.017	0.018
law_firm	37.181	0.230	0.216	0.167	5.435
revolution	21.995	0.413	0.019	0.044	0.009

Table 6.11: Average running times in seconds of the algorithms of AHC and algorithms used for comparison. For CFinder one run finds several solutions, so the time one run takes is reported here instead of the average running time.

have a few larger communities and some smaller. We exclude CFinder in these plots since CFinder does not put every vertex in a community, so that the definition of community size distribution for solutions found by CFinder are different and therefore would be misleading to include in the plots.

In order to examine the solutions with regard to CEVS-score and EQ, these measures are plotted against each other for solutions for different real-world graphs. Each point in Figure 6.8 corresponds to a solution found by one of the algorithms. Solutions that are farther up and left in the plot may be considered to be better, as they have lower CEVS-score and higher EQ. For AHC, two solutions found by two runs of the algorithms are plotted, and for BIGCLAM, Demon and SLPA the solutions come from runs with different parameters. CFinder finds several solutions for each run, and these solutions are used.

In general it looks like the solutions of highest EQ have middling CEVS-score when considering all plotted solutions, and that the solutions with best CEVS-score, not surprisingly found by AHC since it uses this as objective for optimization, have a middling EQ. Some of the other algorithms manage to find solutions with low CEVS-score for some of the graphs, suggesting that the characteristics the algorithms search for in some cases agree with the characteristics of solutions with low CEVS-score. On the other hand, frequently the algorithms find solutions with both high CEVS-score and high EQ. Therefore it seems unlikely that either of CEVS-score and EQ can serve as a general measure for quality of a solution to the overlapping community task.

It is worth noting that EQ and CEVS-score seems to evaluate different aspects of solutions. In the literature, most papers on overlapping community detection use EQ as the only ground truth independent quality measure. With Figure 6.8, the possibility of also using CEVS-score for this purpose is introduced. As remarked by Abu-Khzam et al. [3], CEVS-score can be calculated in running time that is practically linear in the number of vertices and edges, this making CEVS-score applicable to evaluate solutions found by any algorithm for the overlapping community detection task.

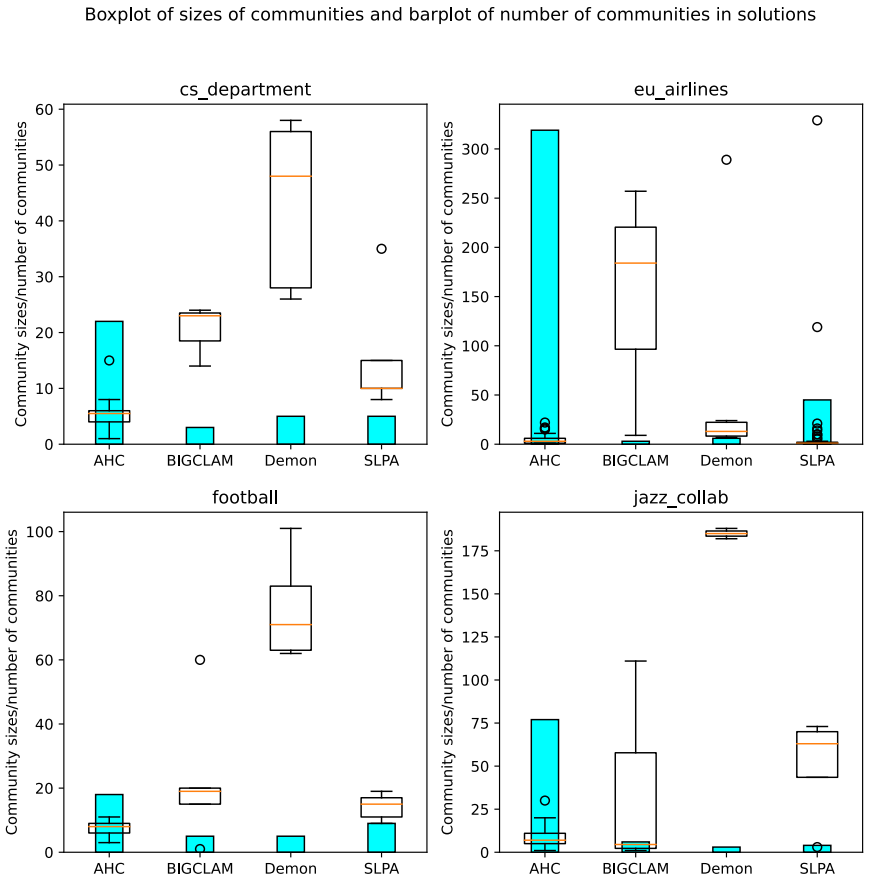


Figure 6.7: Boxplot of community distribution of solutions found by different algorithms on the graphs cs-department, eu_airlines, football, and jazz_collab. The bars show the number of communities, the boxplots show the distribution of community sizes.

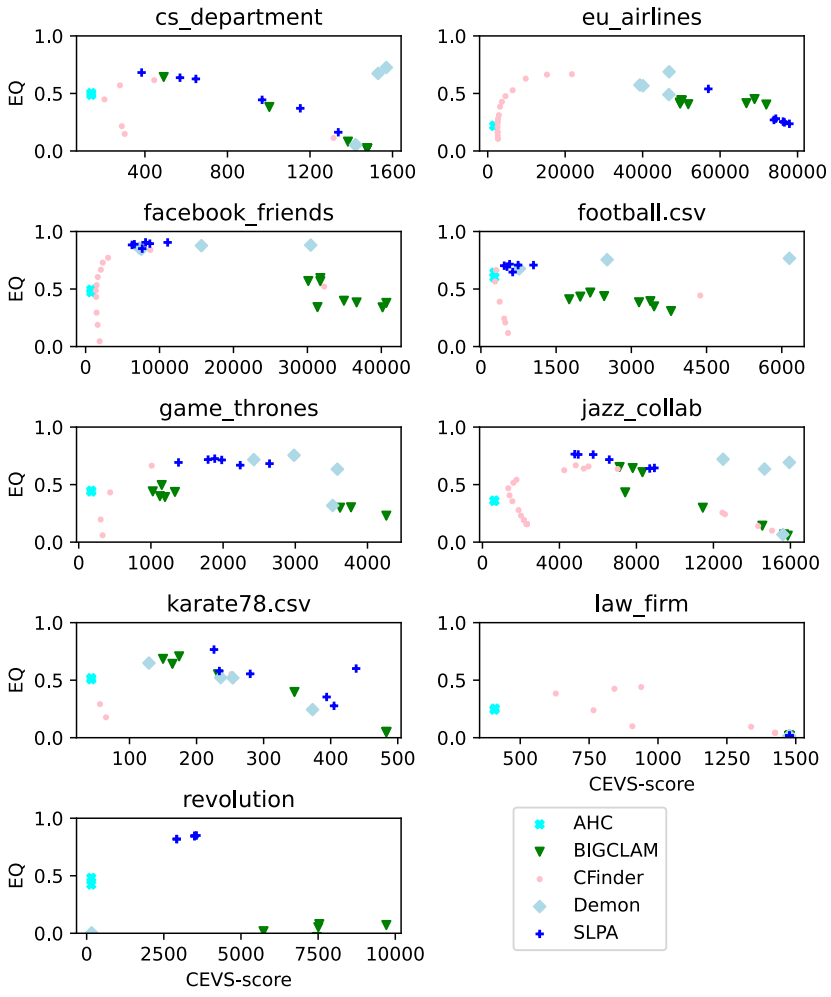


Figure 6.8: CEVS-score and EQ of solutions plotted for different graphs and algorithms. Lower CEVS-score and higher EQ indicate better solutions.

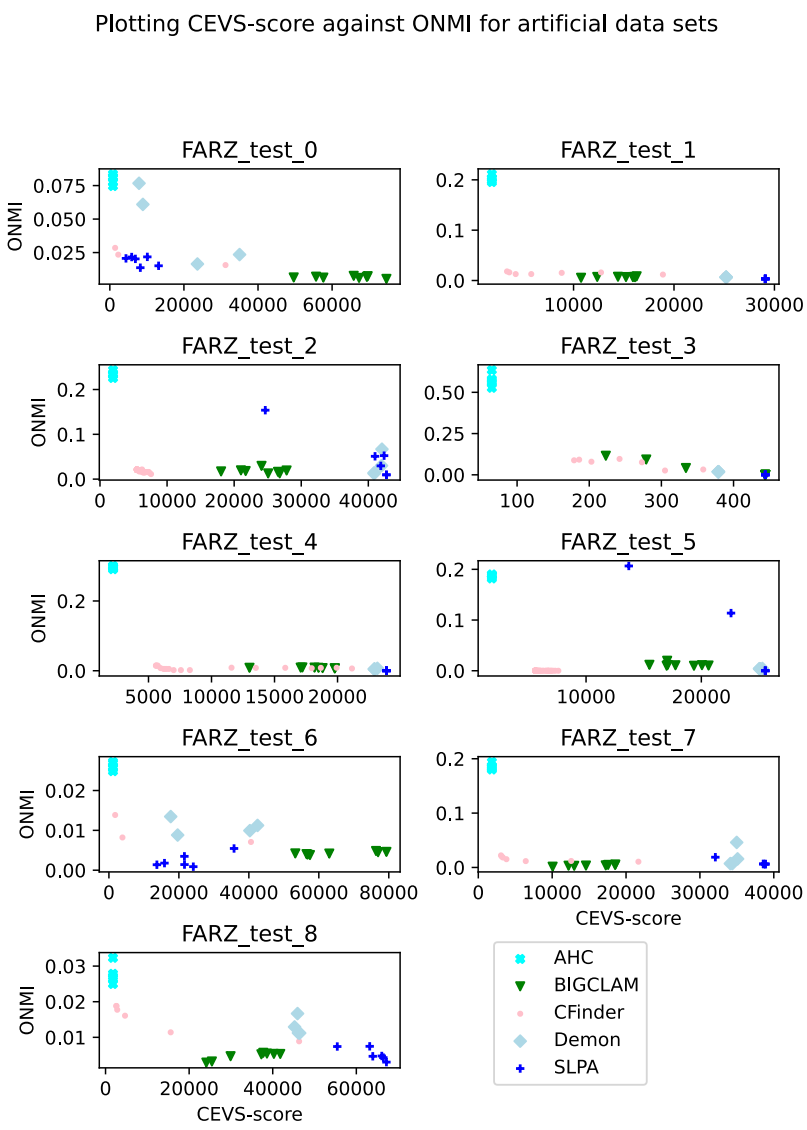


Figure 6.9: CEVS-score plotted against ONMI for solutions for different artificial data sets found by different algorithms. Lower CEVS-score and higher ONMI indicate better solutions.

We also plot CEVS-score against ONMI (see Figure 6.9). Artificial graphs are used for these plots as there are few real-world graphs with ground truth with overlapping communities available. Here AHC in most cases gives the solutions with best ONMI and best CEVS-score. However, the poor performance of the other algorithms than AHC on ONMI suggests that the characteristics of the ground truth communities decided by FARZ do not conform to the characteristics of the community structures the other algorithms search for. Nevertheless, the plots give further evidence that CEVS-score is a reasonable quality measure for solutions to the overlapping community detection task.

Also of interest is the number of splits in solutions, telling us about the degree of overlap in solutions. With a split we mean the number of times the split operator has been applied to an original ancestor vertex in the input graph or one of its descendants, indicating how many communities contain the vertex. The number of splits of representative solutions are visualized for the real-world graphs in the test data set in Figure 6.10. AHC seems to find solutions with more splits than the other algorithms, with some exceptions. In the experiments, BIGCLAM yielded solutions with no overlapping sets exclusively. Demon, SLPA and CFinder frequently found some solutions with and some solutions without splits for the same data set.

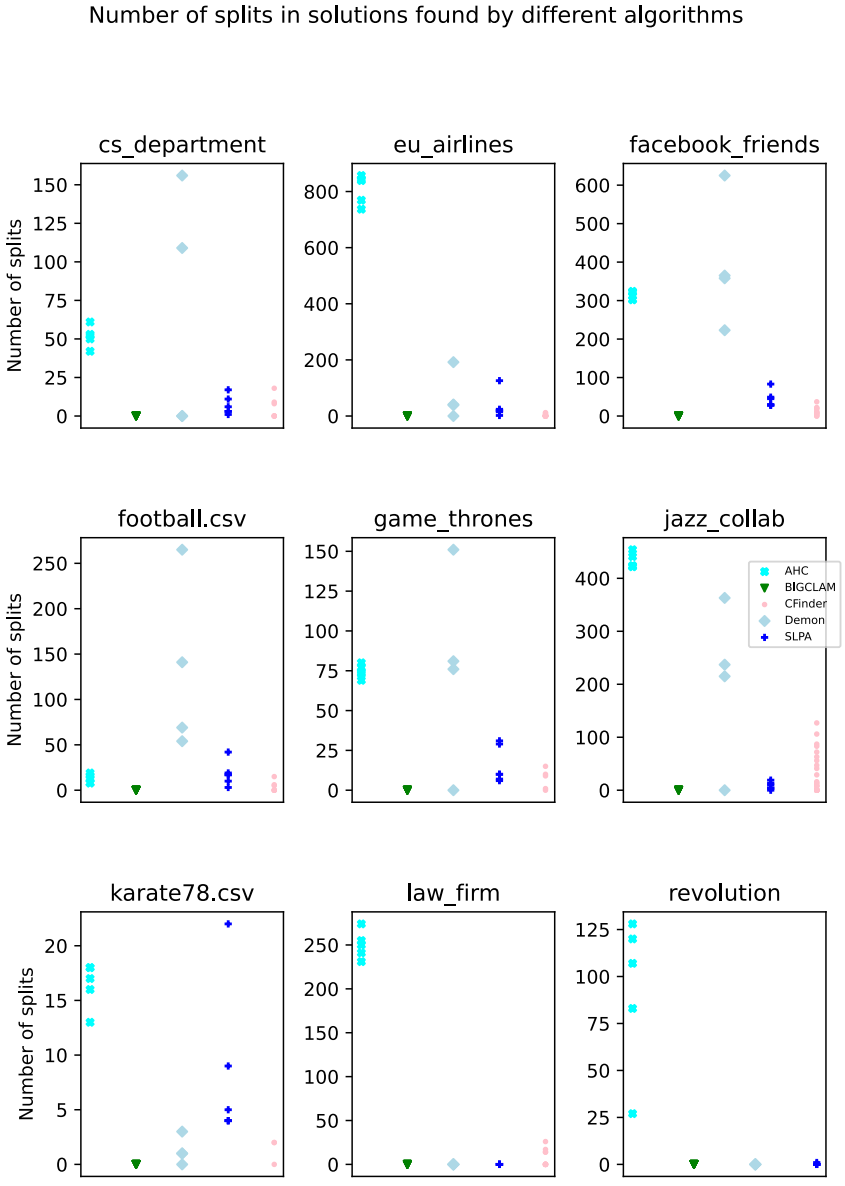


Figure 6.10: Number of splits in solutions to the overlapping communities task found by different algorithms for different graphs. The solutions are interpreted as solutions to CEVS and the number of splits in the solutions is defined accordingly. Solutions found by different algorithms are plotted in seperate columns for visibility. A vertex being split several times indicate that several communities overlap by containing that vertex.

Chapter 7

Conclusion

In this thesis we presented the algorithm AHC that uses metaheuristics to solve CEVS. The running time of the algorithm scales linearly with a factor of $2.1 \cdot 10^{-3}$ with the amount of edges in the input graph when controlling other graph parameters, and the algorithm is able to find low-cost solutions of CEVS on much larger graphs than what is viable with currently known exact algorithms. Parallelization of parts of the algorithm is investigated and yields modest but worthwhile improvements in running time. As AHC does not search for a solution with a specific number of clusters, it uses datastructures that are easy to change the size of but may be slower than necessary to access and alter, in particular red-black search trees. Further implementations of heuristics for CEVS could attempt to solve the problem for a fixed $|F|$ while using arrays for the datastructures, exploring which gains in running time this may yield and which limitations it imposes. This approach could also be used to investigate whether local minima for CEVS occur around certain values of $|F|$.

The selection heuristic of intermittent scan is likely applicable when solving other problems heuristically, in particular for problems where there is some distance between objects of interest (e. g. vertices) in the data so that several worthwhile actions may be discovered when considering a subset of all possible actions. The idea of using segment trees to store which vertices have been moved recently should also be generally applicable.

Experiments using DRLH in place of ALNS yielded comparatively poor results. This leaves an open problem of whether DRLH for CEVS can yield results matching or better than those of ALNS when a different pool of operators and possibly different state representation and parameters are used. The difficulty of using DRLH to solve CEVS suggests that CEVS presents hurdles for DRLH, and that the identification of these hurdles would lead to a better understanding of DRLH.

It remains unknown whether CEVS is NP-complete. We put forth the conjecture of a polynomial-time algorithm for solving CEVS on complete bipartite graphs as an open problem, which if expanded upon could inform results helpful for applying a crown decomposition [15] to CEVS, giving a linear kernel and

thus a preprocessing step to be used in conjunction with both exact algorithms and heuristics. It is worth noting that the notion of using a crown decomposition for CEVS with exclusive vertex split is put forth in Abu-Khzam et al. [2]. As seen in the results regarding the reduction utilizing critical cliques, the current combinatorial results on CEVS are not helpful in practice, urging further endeavours in this direction.

Our comparison of solutions found by AHC to solutions found by other algorithms for solving the overlapping community detection task indicates that solutions with low CEVS-score found by AHC frequently have better ONMI than solutions found by the other algorithms, as well as a decent EQ. However, AHC requires significantly larger computational resources than the other algorithms. The observation that high EQ does not necessarily imply low CEVS-score justifies the idea of using a two-dimensional scatter plot of EQ and CEVS-score of solutions to evaluate the quality of overlapping communities. This tool may capture more of the features of overlapping communities than what the earlier use of only EQ for this purpose has done. We also considered the community distribution and number of splits of solutions, with our results indicating that the solutions of lowest CEVS-score have more and smaller communities than solutions found by the other algorithms optimizing other criteria, and also that solutions with low CEVS-score have relatively more splits. This suggests AHC could be applied to real-life community detection problems in which communities exhibit similar structure to low-CEVS solutions and where other algorithms do not yield useful results. We believe that collection and wider availability of real-world graphs with ground truths that are overlapping communities would be useful for further study of the overlapping community detection task.

Bibliography

[1] kbalasu (github-username). *SLPA*. Accessed: 29.9.2022. 2013. URL: https://github.com/kbalasu/SLPA.

[2] Faisal N. Abu-Khzam, Joseph R. Barr, Amin Fakhereldine, and Peter Shaw. "A Greedy Heuristic for Cluster Editing with Vertex Splitting". In: *4th International Conference on Artificial Intelligence for Industries, AI4I 2021, Laguna Hills, CA, USA, September 20-22, 2021*. IEEE, 2021, pp. 38–41. DOI: 10.1109/AI4I51902.2021.00017. URL: https://doi.org/10.1109/AI4I51902.2021.00017.

[3] Faisal N. Abu-Khzam, Judith Egan, Serge Gaspers, Alexis Shaw, and Peter Shaw. "Cluster Editing with Vertex Splitting". In: *Combinatorial Optimization*. Ed. by Jon Lee, Giovanni Rinaldi, and A. Ridha Mahjoub. Cham: Springer International Publishing, 2018, pp. 1–13. ISBN: 978-3-319-96151-4.

[4] Enrique Amigó, Julio Gonzalo, Javier Artiles, and Felisa Verdejo. "A comparison of extrinsic clustering evaluation metrics based on formal constraints". In: *Inf. Retr.* 12.4 (2009), pp. 461–486. DOI: 10.1007/s10791-008-9066-8. URL: https://doi.org/10.1007/s10791-008-9066-8.

[5] Ian Anderson. *A First Course in Discrete Mathematics*. Springer, 2001.

[6] Gard Askeland. Accessed: 11.11.2022. 2022. URL: https://github.com/gardaskeland/CEVS_heur.

[7] Nikhil Bansal, Avrim Blum, and Shuchi Chawla. "Correlation Clustering". In: *Mach. Learn.* 56.1-3 (2004), pp. 89–113. DOI: 10.1023/B:MACH.0000033116.57574.95. URL: https://doi.org/10.1023/B:MACH.0000033116.57574.95.

[8] Andrew Beveridge and Jie Shan. "Network of Thrones". In: *Math Horizons* 23.4 (2016), pp. 18–22. DOI: 10.4169/mathhorizons.23.4.18. eprint: https://doi.org/10.4169/mathhorizons.23.4.18. URL: https://doi.org/10.4169/mathhorizons.23.4.18.

[9] Sebastian Böcker and Jan Baumbach. "Cluster Editing". In: *The Nature of Computation. Logic, Algorithms, Applications*. Ed. by Paola Bonizzoni, Vasco Brattka, and Benedikt Löwe. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 33–44. ISBN: 978-3-642-39053-1.

[10] Sebastian Böcker, Sebastian Briesemeister, Quang Bao Anh Bui, and Anke Truß. "A Fixed-Parameter Approach for Weighted Cluster Editing". In: *Proceedings of the 6th Asia-Pacific Bioinformatics Conference, APBC 2008, 14-17 January 2008, Kyoto, Japan*. Ed. by Alvis Brazma, Satoru Miyano, and Tatsuya Akutsu. Vol. 6. Advances in Bioinformatics and Computational Biology. Imperial College Press, 2008, pp. 211–220. URL: http://www.comp.nus.edu.sg/%5C%7Ewongls/psZ/apbc2008/apbc050a.pdf.

[11] Alessio Cardillo, Jesús Gómez-Gardeñes, Massimiliano Zanin, Miguel Romance, David Papo, Francisco del Pozo, and Stefano Boccaletti. "Emergence of network features from multiplexity". In: *Scientific Reports* 3 (2013). Article number: 1334. DOI: 10.1038/srep01344.

[12] Linda M. Collins and Clyde W. Dent. "Omega: A General Formulation of the Rand Index of Cluster Recovery Suitable for Non-disjoint Solutions". In: *Multivariate Behavioral Research* 23.2 (1988). PMID: 26764947, pp. 231–242. DOI: 10.1207/s15327906mbr2302_6. eprint: https://doi.org/10.1207/s15327906mbr2302_6. URL: https://doi.org/10.1207/s15327906mbr2302_6.

[13] Michele Coscia, Giulio Rossetti, Fosca Giannotti, and Dino Pedreschi. "DEMON: a local-first discovery method for overlapping communities". In: *The 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '12, Beijing, China, August 12-16, 2012*. Ed. by Qiang Yang, Deepak Agarwal, and Jian Pei. ACM, 2012, pp. 615–623. DOI: 10.1145/2339530.2339630. URL: https://doi.org/10.1145/2339530.2339630.

[14] Michele Coscia, Giulio Rossetti, Fosca Giannotti, and Dino Pedreschi. "Uncovering Hierarchical and Overlapping Communities with a Local-First Approach". In: *ACM Trans. Knowl. Discov. Data* 9.1 (2014), 6:1–6:27. DOI: 10.1145/2629511. URL: https://doi.org/10.1145/2629511.

[15] Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2016.

[16] Zhuanlin Ding, Xingyi Zhang, Dengdi Sun, and Bin Luo. "Overlapping Community Detection based on Network Decomposition". In: 6 (2014). DOI: https://doi.org/10.1038/srep24115.

[17] Tansel Dökeroglu, Ender Sevinç, Tayfun Kucukyilmaz, and Ahmet Cosar. "A survey on new generation metaheuristic algorithms". In: *Comput. Ind. Eng.* 137 (2019). DOI: 10.1016/j.cie.2019.106040. URL: https://doi.org/10.1016/j.cie.2019.106040.

[18] Peter Eades and Candido Ferreira Xavier de Mendonça Neto. "Vertex Splitting and Tension-Free Layout". In: *Graph Drawing, Symposium on Graph Drawing, GD '95, Passau, Germany, September 20-22, 1995, Proceedings*. Ed. by Franz-Josef Brandenburg. Vol. 1027. Lecture Notes in Computer Science. Springer, 1995, pp. 202–211. DOI: 10.1007/BFb0021804. URL: https://doi.org/10.1007/BFb0021804.

[19] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. *Graphviz and Dynagraph – Static and Dynamic Graph Drawing Tools*. Accessed: 21.10.2022. URL: https://graphviz.org/documentation/EGKNW03.pdf.

[20] L. Faria, C.M.H. de Figueiredo, and C.F.X. Mendonça. "splitting number is NP-complete". In: *Discrete Applied Mathematics* 108.1 (2001). Workshop on Graph Theoretic Concepts in Computer Science, pp. 65–83. ISSN: 0166-218X. DOI: https://doi.org/10.1016/S0166-218X(00)00220-1. URL: https://www.sciencedirect.com/science/article/pii/S0166218X00002201.

[21] Tom Fawcett. "An introduction to ROC analysis". In: *Pattern Recognit. Lett.* 27.8 (2006), pp. 861–874. DOI: 10.1016/j.patrec.2005.10.010. URL: https://doi.org/10.1016/j.patrec.2005.10.010.

[22] Fedor V. Fomin, Stefan Kratsch, Marcin Pilipczuk, Michal Pilipczuk, and Yngve Villanger. "Tight bounds for parameterized complexity of Cluster Editing with a small number of clusters". In: *J. Comput. Syst. Sci.* 80.7 (2014), pp. 1430–1447. DOI: 10.1016/j.jcss.2014.04.015. URL: https://doi.org/10.1016/j.jcss.2014.04.015.

[23] Fedor V. Fomin, Saket Saurabh, and Neeldhara Misra. "Graph Modification Problems: A Modern Perspective". In: *Frontiers in Algorithmics*. Ed. by Jianxin Wang and Chee Yap. Cham: Springer International Publishing, 2015, pp. 3–6. ISBN: 978-3-319-19647-3.

[24] M. Girvan and M. E. J. Newman. "Community structure in social and biological networks". In: *Proceedings of the National Academy of Sciences* 99.12 (2002), pp. 7821–7826. DOI: 10.1073/pnas.122653799. eprint: https://www.pnas.org/doi/pdf/10.1073/pnas.122653799. URL: https://www.pnas.org/doi/abs/10.1073/pnas.122653799.

[25] M. Girvan and M. E. J. Newman. "Community structure in social and biological networks". In: *Proceedings of the National Academy of Sciences* 99.12 (2002), pp. 7821–7826. DOI: 10.1073/pnas.122653799. eprint: https://www.pnas.org/doi/pdf/10.1073/pnas.122653799. URL: https://www.pnas.org/doi/abs/10.1073/pnas.122653799.

[26] Pablo M. Gleiser and Leon Danon. "Community Structure in Jazz". In: *Adv. Complex Syst.* 6.4 (2003), pp. 565–574. DOI: 10.1142/S0219525903001067.

[27] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[28] Lars Gottenbüren, Tobias Heuer, Thomas Bläsius, Philipp Fishbeck, Michael Hamann, Jonas Spinner, Crhistopher Weyand, and Marcus Wilhelm. *KaPoCE: A Heuristic Cluster Editing Algorithm*. 2021. URL: http://algo2.iti.kit.edu/heuer/kapoce/kapoce_heuristic.pdf.

[29] Jiong Guo. "A more effective linear kernelization for cluster editing". In: *Theoretical Computer Science* 410.8 (2009), pp. 718–726. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2008.10.021. URL: https://www.sciencedirect.com/science/article/pii/S0304397508007822.

[30] Ahmad Hemmati, Lars Magnus Hvattum, Kjetil Fagerholt, and Inge Norstad. "Benchmark Suite for Industrial and Tramp Ship Routing and Scheduling Problems". In: *INFOR Inf. Syst. Oper. Res.* 52.1 (2014), pp. 28–38. DOI: 10.3138/infor.52.1.28. URL: https://doi.org/10.3138/infor.52.1.28.

[31] Kashif Hussain, Mohd. Najib Mohd. Salleh, Shi Cheng, and Yuhui Shi. "Metaheuristic research: a comprehensive survey". In: *Artif. Intell. Rev.* 52.4 (2019), pp. 2191–2233. DOI: 10.1007/s10462-017-9605-z. URL: https://doi.org/10.1007/s10462-017-9605-z.

[32] Nabil Ibtehaz, M. Kaykobad, and M. Sohel Rahman. "Multidimensional segment trees can do range updates in poly-logarithmic time". In: *Theor. Comput. Sci.* 854 (2021), pp. 30–43. DOI: 10.1016/j.tcs.2020.11.034. URL: https://doi.org/10.1016/j.tcs.2020.11.034.

[33] Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. "Which Problems Have Strongly Exponential Complexity?" In: *J. Comput. Syst. Sci.* 63.4 (2001), pp. 512–530. DOI: 10.1006/jcss.2001.1774. URL: https://doi.org/10.1006/jcss.2001.1774.

[34] ISO. *Working Draft, Standard for Programming Language C++*. 2017. URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf.

[35] Jakob Kallestad. Private communication. 2022.

[36] Jakob Kallestad, Ahmad Hemmati, and Ramin Hasibi. *Developing an Intelligent Hyperheuristic for Combinatorial Optimization Problems using Deep Reinforcement Learning*. 2021. URL: https://bora.uib.no/bora-xmlui/bitstream/handle/11250/2827078/Master_Thesis___Jakob_Kallestad_3.pdf?sequence=1&isAllowed=y.

[37] Leon Kellerhals, Tomohiro Koana, André Nichterlein, and Philipp Zschoche. "The PACE 2021 Parameterized Algorithms and Computational Experiments Challenge: Cluster Editing". In: *16th International Symposium on Parameterized and Exact Computation, IPEC 2021, September 8-10, 2021, Lisbon, Portugal*. Ed. by Petr A. Golovach and Meirav Zehavi. Vol. 214. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 26:1–26:18. DOI: 10.4230/LIPIcs.IPEC.2021.26. URL: https://doi.org/10.4230/LIPIcs.IPEC.2021.26.

[38] John Kleinberg and Éva Tardos. *Pearson New International Edition: Algorithms Design*. First. Pearson, 2014.

[39] Jérôme Kunegis. "KONECT: The Koblenz Network Collection". In: *Proceedings of the 22nd International Conference on World Wide Web*. WWW '13 Companion. Rio de Janeiro, Brazil: Association for Computing Machinery, 2013, pp. 1343–1350. ISBN: 9781450320382. DOI: 10.1145/2487788.2488173. URL: https://doi.org/10.1145/2487788.2488173.

[40] Andrea Lancichinetti, Santo Fortunato, and János Kertész. "Detecting the overlapping and hierarchical community structure in complex networks". In: (Mar. 2009). URL: https://arxiv.org/pdf/0802.1218.pdf.

[41] Emmanuel Lazega. *The Collegial Phenomenon: The Social Mechanisms of Cooperation Among Peers in a Corporate Law Partnership*. Oxford University Press, Sept. 2001. ISBN: 9780199242726. DOI: 10.1093/acprof:oso/9780199242726.001.0001. URL: https://doi.org/10.1093/acprof:oso/9780199242726.001.0001.

[42] Guo-Hui Lin, Tao Jiang, and Paul E. Kearney. "Phylogenetic k-Root and Steiner k-Root". In: *Algorithms and Computation, 11th International Conference, ISAAC 2000, Taipei, Taiwan, December 18-20, 2000, Proceedings*. Ed. by D. T. Lee and Shang-Hua Teng. Vol. 1969. Lecture Notes in Computer Science. Springer, 2000, pp. 539–551. DOI: 10.1007/3-540-40996-3_46. URL: https://doi.org/10.1007/3-540-40996-3%5C_46.

[43] Yunlong Liu, Jianxin Wang, and Jiong Guo. "An overview of kernelization algorithms for graph modification problems". In: *Tsinghua Science and Technology* 19.4 (2014), pp. 346–357. DOI: 10.1109/TST.2014.6867517.

[44] Ulrike von Luxburg. "A tutorial on spectral clustering". In: *Stat. Comput.* 17.4 (2007), pp. 395–416. DOI: 10.1007/s11222-007-9033-z. URL: https://doi.org/10.1007/s11222-007-9033-z.

[45] Matteo Magani, Barbora Micenkova, and Luca Ross. *Combinatorial Analysis of Multiple Networks*. 2013. arXiv: 1303.4986. URL: http://arxiv.org/abs/1303.4986.

[46] Benjamin F. Maier and Dirk Brockmann. "Cover time for random walks on arbitrary complex networks". In: *CoRR* abs/1706.02356 (2017). arXiv: 1706.02356. URL: http://arxiv.org/abs/1706.02356.

[47] Julian J. McAuley and Jure Leskovec. "Learning to Discover Social Circles in Ego Networks". In: *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*. Ed. by Peter L. Bartlett, Fernando C. N. Pereira, Christopher J. C. Burges, Léon Bottou, and Kilian Q. Weinberger. 2012, pp. 548–556.

[48] Aaron F. McDaid, Derek Greene, and Neil Hurley. "Normalized Mutual Information to evaluate overlapping community finding algorithms". In: (Oct. 2011). arXiv: 1110.2515. URL: http://arxiv.org/abs/1110.2515.

[49] Mark Newman and Michelle Girvan. “Finding and Evaluating Community Structure in Networks”. In: *Physical review. E, Statistical, nonlinear, and soft matter physics* 69 (Mar. 2004), p. 026113. DOI: 10.1103/PhysRevE.69.026113.

[50] V. Nicosia, G. Mangioni, V. Carchiolo, and M. Malgeri. “Extending the defintion of modularity to directed graphs with overlapping communities”. In: *Journal of Statistical Mechanics: Thory and Experiment* 2009 (2009), P03024.

[51] OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 5.0*. 2018. URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf.

[52] Gergely Palla, Imre Derényi, Illés Farkas, and Tamás Vicsek. “Uncovering the overlapping community structure of complex networks in nature and society”. In: *Nature* 435 (July 2005), pp. 814–818.

[53] Gergely Palla, Imre Derényi, Illés Farkas, and Tamás Vicsek. *CFinder: Overlapping clusters/communities in networks*. Accessed: 29.9.2022. URL: https://www.cfinder.org/.

[54] Leto Peel, Daniel B. Larremore, and Aaron Clauset. “The ground truth about metadata and community detection in networks”. In: *Science Advances* 3.5 (2017), e1602548. DOI: 10.1126/sciadv.1602548. eprint: https://www.science.org/doi/pdf/10.1126/sciadv.1602548. URL: https://www.science.org/doi/abs/10.1126/sciadv.1602548.

[55] Tiago P. Peixoto. *The Netzschleuder network catalogue and repository*. Accessed: 14.9.2022. 2020. URL: https://networks.skewed.de/.

[56] Alexander Ponomarkenko, Leonidas Pitsoulis, and Marat Shamshetdinov. “Overlapping community detection in networks based on link partitioning and partitioning around medoids”. In: *PLOS ONE* 16 (2021). DOI: https://doi.org/10.1371/journal.pone.0255717.

[57] Reihaneh Rabbany. *FARZ - Benchmark for Community Detection Algorithms*. Accessed: 13.9.2022. URL: https://github.com/rabbanyk/FARZ.

[58] Nandini Raghavan, Réka Albert, and Soundar Kumara. “Near Linear Time Algorithm to Detect Community Structures in Large-Scale Networks”. In: *Physical review. E, Statistical, nonlinear, and soft matter physics* 76 (Oct. 2007), p. 036106. DOI: 10.1103/PhysRevE.76.036106.

[59] H. N. de Ridder et al. *Information System on Graph Classes and their Inclusions (ISGCI)*. https://www.graphclasses.org. Accessed: 3.10.2022. Feb. 2022.

[60] Stefan Ropke and David Pisinger. “An Adaptive Large Neighborhood Search Heuristic for the Pickup and Delivery Problem with Time Windows”. In: *Transp. Sci.* 40.4 (2006), pp. 455–472. DOI: 10.1287/trsc.1050.0135. URL: https://doi.org/10.1287/trsc.1050.0135.

[61] Guilio Rosetti. *DEMON*. Accessed: 29.9.2022. 2021. URL: https://github.com/GiulioRossetti/DEMON.

[62] Benedek Rozemberczki, Oliver Kiss, and Rik Sarkar. "Karate Club: An API Oriented Open-source Python Framework for Unsupervised Learning on Graphs". In: *Proceedings of the 29th ACM International Conference on Information and Knowledge Management (CIKM '20)*. ACM. 2020, pp. 3125–3132.

[63] Benedek Rozemberczki and Rik Sarkar. "Characteristic Functions on Graphs: Birds of a Feather, from Statistical Descriptors to Parametric Models". In: *Proceedings of the 29th ACM International Conference on Information and Knowledge Management (CIKM '20)*. ACM. 2020, pp. 1325–1334.

[64] Peter Sanders, Kurt Melhorn, Martin Dietzfelbinger, and Roman Dementiev. *Sequential and Parallel Algorithms and Data Structures*. Springer, 2019.

[65] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. *Proximal Policy Optimization Algorithms*. 2017. arXiv: 1707.06347. URL: http://arxiv.org/abs/1707.06347.

[66] Robert Sedgewick and Kevin Wayne. *Algorithms*. Fourth. Addison-Wesley, 2011.

[67] Ron Shamir, Roded Sharan, and Dekel Tsur. "Cluster graph modification problems". In: *Discret. Appl. Math.* 144.1-2 (2004), pp. 173–182. DOI: 10.1016/j.dam.2004.01.007. URL: https://doi.org/10.1016/j.dam.2004.01.007.

[68] Michael Sipser. *Introduction to the Theory of Computation*. Third International Edition. Cengage Learning, 2013.

[69] Vinícius da Fonseca Vieira, Carolina Ribeiro Xavier, and Alexandre Gonçalves Evsukoff. "A comparative study of overlapping community detection methods from the perspective of the structural properties". In: *Applied Network Science* 5 (2020). DOI: https://doi.org/10.1007/s41109-020-00289-9.

[70] Stefan Voß and Andreas Fink. "A hybridized tabu search approach for the minimum weight vertex cover problem". In: *J. Heuristics* 18.6 (2012), pp. 869–876. DOI: 10.1007/s10732-012-9211-9. URL: https://doi.org/10.1007/s10732-012-9211-9.

[71] Hsiang-Yun Wu, Martin Nöllenburg, and Ivan Viola. "Multi-Level Area Balancing of Clustered Graphs". In: *IEEE Trans. Vis. Comput. Graph.* 28.7 (2022), pp. 2682–2696. DOI: 10.1109/TVCG.2020.3038154. URL: https://doi.org/10.1109/TVCG.2020.3038154.

[72] Jierui Xie, Boleslaw K. Szymanski, and Xiaoming Liu. "SLPA: Uncovering Overlapping Communities in Social Networks via a Speaker-Listener Interaction Dynamic Process". In: *Data Mining Workshops (ICDMW), 2011 IEEE 11th International Conference on, Vancouver, BC, Canada, December 11, 2011*. Ed. by Myra Spiliopoulou, Haixun Wang, Diane J. Cook, Jian Pei, Wei Wang, Osmar R. Zaïane, and Xindong Wu. IEEE Computer Society, 2011, pp. 344–349. DOI: 10.1109/ICDMW.2011.154. URL: https://doi.org/10.1109/ICDMW.2011.154.

[73] Jaewon Yang and Jure Leskovec. "Overlapping community detection at scale: a nonnegative matrix factorization approach". In: *Sixth ACM International Conference on Web Search and Data Mining, WSDM 2013, Rome, Italy, February 4-8, 2013*. Ed. by Stefano Leonardi, Alessandro Panconesi, Paolo Ferragina, and Aristides Gionis. ACM, 2013, pp. 587–596. DOI: 10.1145/2433396.2433471. URL: https://doi.org/10.1145/2433396.2433471.

[74] W. W. Zachary. "An information flow model for conflict and fission in small groups." In: *Journal of Anthropological Research* 33 (1977), pp. 452–473. DOI: 10.1086/jar.33.4.3629752. URL: https://doi.org/10.1086/jar.33.4.3629752.

Index