PACE Solver Description: CluES – a Heuristic Solver for the Cluster Editing Problem*

Sylwester Swat
Institute of Computing Science, Poznań University of Technology, Poland

Abstract

This article briefly describes the most important algorithms and techniques used in the cluster editing heuristic solver called "CluES", submitted to the 6th Parameterized Algorithms and Computational Experiments Challenge (PACE 2021).

2012 ACM Subject Classification Mathematics of computing → Graph algorithms

Keywords and phrases Cluster editing, heuristic solver, graph algorithms, PACE 2021

Digital Object Identifier 10.4230/LIPIcs.IPEC.2021.32

Supplementary Material *Software (Source Code)*: `https://doi.org/10.5281/zenodo.4949728`

1 Problem description

A cluster editing set of a graph $G = (V, E)$ is a set of edges $E_{mod} \subseteq V \times V$ such that each connected component of a graph $G[E \Delta E_{mod}]$ is a clique, where $A \Delta B$ denotes the symmetric difference of sets A and B. The solver briefly described here is a heuristic approach to the cluster editing problem, where the goal is to find, for a given graph, a cluster editing set of the smallest size.

2 Solver description

In this paper we provide a short description of the most important algorithms implemented in solver CluES. Due to many parameters used in the implementation and a vast number of case distinctions that need to be taken into account, this description may not contain full information about the algorithms behavior in every possible situation.

All algorithms implemented in CluES operate on a data structure we called a *partition graph*. For a given graph $G = (V, E)$ and a partition $P = \{C_1, C_2, \ldots, C_p\}$ of V, a partition graph G_P is tuple (V_P, E_P, w_n, w_e), where V_P is the node set, E_P is the edge set and $w_n : V_P \mapsto \mathbb{N}$ and $w_e : E_P \mapsto \mathbb{N}$ are weight functions, defined as below:

$$\begin{aligned} V_P &= \{1, 2, \ldots, p\} \\ E_P &= \{(u, v) \in V_P \times V_P : \exists_{x \in V \cap C_u, y \in V \cap C_v} (x, y) \in E\} \\ w_n(u) &= |\{x \in V : x \in C_u\}| \\ w_e(u, v) &= |\{(x, y) \in E : x \in C_u, y \in C_v\}| \end{aligned}$$

It is worth noting here that by setting $V_P = V$, $E_P = E$, $w_n(u) = 1$, and $w_e(u, v) = 1$ we obtain the partition graph representing the original input graph.

* This is a brief description of one of the highest ranked solvers of PACE Challenge 2021. It has been made public for the benefit of the community and was selected based on the ranking. PACE encourages publication of work building on the ideas presented in this description in peer-reviewed venues.

The main algorithm works in independent iterations (main iterations). In each iteration a separate cluster editing set is found (independently from previous iterations) and the best result found in all main iterations is returned as a final cluster editing set. In main iterations, instead of looking for a set of edge modifications, we look for a partition $\{C_1, C_2, ..., C_p\}$ of set V corresponding to clusters in a graph after edge modifications. Each main iteration consists of the following phases:

1. Quickly create a set of feasible solutions using some fast heuristics.
2. Coarsen the graph. To do this, we create a partition $\{S_1, \ldots, S_t\}$ of V such that all nodes in each S_i occur always in the same cluster in all already found solutions. For each $1 \leq i \leq t$ find a maximal matching M_i of a graph $G[S_i]$. We take $M = \bigcup_{1 \leq i \leq t} M_i$ and "contract" (by creating a corresponding partition graph) all edges in M.
3. Repeat the process in steps $1 - 2$ for the coarsened graph. After the steps are repeated (the graph is coarsened) for the R-th time (R is some small integer, usually 3 or 4), proceed to step 4.
4. Consider the best solution found so far and improve the solution using local search.
5. If possible, uncoarsen the graph "by one level" (uncoarsen pairs of nodes from the most recently found matching in step 2) and go back to step 4.

3 Preprocessing

In main iterations we sometimes use preprocessing to determine certain subsets of nodes that are in the same cluster in some optimal solution. This information is used to create a proper partition graph before starting the first phase of a given main iteration. We use over 15 parameter-independent kernelization rules based on edge cuts and critical cliques. Nine of these rules are based on concepts related to critical cliques and can be found in [3] and [2]. One rule is a "similar neighborhoods" rule (see [1]) applied to an unweighted graph. The rule states that, if there exists an edge $(u, v) \in E$ with $N[u] = N[v] \cup \{w\}$ for some $w \notin N(v)$, then there exists an optimal solution with u and v belonging to the same cluster. Another three rules are specifications of the "almost clique rule" [1] applied for all triangles, K_4's and induced diamonds. The "almost clique rule" states that, if there exists a subset $C \subset V$ such that $G[C]$ is connected and the size of the minimum cut in $G[C]$ is not smaller than $|\{(a, b) \in C \times C : (a, b) \notin E\}| + |\{(a, b) \in E : a \in C, b \notin C\}|$, then there exists an optimal solution where all nodes in C belong to the same cluster. The other preprocessing rules are of our own invention and are based on properties of some intersections of neighborhoods of critical cliques and usually do not guarantee that their application will preserve the result optimality property, but are just a reasonable guess that often enables us to quickly and accurately identify nodes that most often belong to the same cluster in some good solution.

4 Local search techniques

In order to improve a solution, we use a local search technique. In each iteration of the local search, we call several procedures. All of them are based on moving some subsets of nodes from a cluster to which they belong in the current solution to some other cluster (possibly creating a new one). We allow moving subsets only if the total number of edges required to create the cluster graph obtained after the move is not larger than before the move. We consider the following possibilities of a move:

1. Move a single node v from its cluster to some other cluster.
2. Move two nodes belonging to the same edge to some cluster.

3. Move three nodes a, b, c with $b \in N(a)$ and $c \in N(b)$ to some cluster.
4. Interchange clusters of a pair of nodes a and b.
5. Move node a to a cluster containing node b and simultaneously move node b to some other cluster.
6. For a given cluster, create some permutation $(u_0, u_1, \ldots, u_x)$ of nodes belonging to that cluster. For each set $U_i = \{u_0, \ldots, u_i\}$, try to move U_i to some other (single) cluster.
7. For a given cluster, greedily move nodes from this cluster to some other cluster, always selecting the one that minimizes some objective function (e.g. the difference in the total number of edge modifications needed to obtain states before and after the move). This way it is possible to move several nodes from the given cluster to multiple clusters.
8. For a given cluster, greedily move nodes from other clusters to this cluster. This way it is possible to move several nodes from multiple clusters to the given cluster.

Let us note here that these steps work best when applied to a properly created partition graph. Moving a single node of a partition graph G_P corresponds to moving some subset of nodes of the original graph G. Hence it is possible to move large subsets of nodes from G by moving just a single node from G_P.

It is important to mention that the crucial point is the design and implementation of algorithms efficiently realizing described steps (working in a weighted scenario, since they are applied for the partition graph structure). Naively checking the moves described above, except perhaps the first one, could lead to a complexity of $\Omega(N^2)$ that would be prohibitive to perform for larger graphs, let alone calling the algorithms hundreds or thousands times during the local search. By efficiently we mean checking all possible structures (all nodes, edges, P_3's and pairs of nodes) that can be moved in steps 1-5, the guarantee of a feasible worst-case running time and the implementation maximally reducing the running time constant.

As an easy example of an algorithm efficiently realizing described steps, let us consider the following subproblem of step 3: "Does there exist a triangle (a, b, c) in G, such that making (a, b, c) a single, separate cluster would result in a better solution?". Naively checking all triangles in the graph could lead to an algorithm with $\Omega(N^2)$ complexity and that would be too slow, e.g., for a graph with $N = 10^5$ nodes and $M = 5 * 10^5$ edges. Enumeration of all triangles can, however, be done in time $O(E^{\frac{3}{2}})$ with a fairly small constant factor (especially in practice), what gives a more practical approach.

5 Availability

The source code of CluES is available at `https://doi.org/10.5281/zenodo.4949728` (please select the proper version of the solver to access required contest track: heuristic, exact or kernelization).

References

1 Sebastian Böcker, Sebastian Briesemeister, and Gunnar Klau. Exact algorithms for cluster editing: Evaluation and experiments. *Algorithmica*, 60:316–334, January 2008.

2 Jianer Chen and Jie Meng. A 2k kernel for the cluster editing problem. *Journal of Computer and System Sciences*, 78(1):211–220, 2012. JCSS Knowledge Representation and Reasoning.

3 Jiong Guo. A more effective linear kernelization for cluster editing. *Theor. Comput. Sci.*, 410:718–726, March 2009.