# PACE Solver Description: PaSTEC - PAths, Stars and Twins to Edit Towards Clusters*

**Valentin Bartier**
G-SCOP, Grenoble INP, Univ. Grenoble-Alpes, Grenoble, France

**Gabriel Bathie**
École Normale Supérieure de Lyon, France

**Nicolas Bousquet**
LIRIS, CNRS, Université Claude Bernard Lyon 1, Université de Lyon, France

**Marc Heinrich**
University of Leeds, UK

**Théo Pierron**
LIRIS, CNRS, Université Claude Bernard Lyon 1, Université de Lyon, France

**Ulysse Prieto**
Independent Researcher, Paris, France

**Abstract**

This document describes our exact Cluster Editing solver, PaSTEC, which got the third place in the 2021 PACE Challenge.

**2012 ACM Subject Classification** Theory of computation → Parameterized complexity and exact algorithms

**Keywords and phrases** cluster editing, exact algorithm, star packing, twins

**Digital Object Identifier** 10.4230/LIPIcs.IPEC.2021.29

**Supplementary Material** *Software (Source Code)*: `https://github.com/valbart/pace-2021`
*Software (Source Code)*: `https://doi.org/10.5281/zenodo.4935569`

**Funding** This Work Was Supported by ANR Project GrR (ANR-18-CE40-0032).

## 1 General description of the solver

Our solver consists in a branch and bound (BB) algorithm. It first starts with a preprocessing phase that we describe below in this section. The algorithm then constructs solutions starting from the initial clustering where all the vertices are in their own clusters of size 1. At each branching step, it either merges two clusters or definitively separate them. We then apply rules to enforce pairs of vertices to be either on the same cluster, or to be in separated clusters. We denote these rules as *enforcing rules*. The branching phase and enforcing rules are described with more details in Section 3.

### Overview of the preprocessing phase

We first partition the graph into connected components that will be solved independently one after the other. Indeed, it is not hard to check that no two vertices that belong to separate connected components are in the same cluster of any optimal solution (see for instance [2]). We then compute an upper bound with an heuristic (the reader is referred to [1] for the description of the heuristic). We then label some pairs of vertices as "edges"

* 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.

if they necessarily belong to any optimal solution or “non edges” if they do not belong to an optimal solution. This labeling phase permits to avoid unnecessary branchings in the branching phase of the algorithm. We describe this labeling phase of the preprocessing in the next section.

## 2 Description of the labeling phase

Let us begin with some definitions. The *twinness* of two vertices $u, v$ is the size of the symmetric difference of their neighborhoods. Two adjacent vertices $u, v$ are $i$*-twins* if their twinness is at most $i$. Note that 0-twins are what is usually called in the literature *true twins*. One can easily prove that two 0-twins are always in the same cluster of an optimal solution.

On the other hand, two false twins are not necessarily in the same cluster of an optimal solution (consider e.g. a $P_3$). However, we can prove that if two false twins $u, v$ are not in the same cluster, we can move any of them to the cluster of the other without increasing the cost of an optimal solution. So we can assume that all the false twins are in the same cluster and then we can label as edges pairs of false twins. Similarly, we can label as edge any pair of 1-twins.

One can wonder what can happen when $i$ is increasing. If there is a triangle of 2-twins then we can prove that there exists an optimal solution where there are in the same cluster. Unfortunately, it is not enough to force all these edges. Indeed for the butterfly (two triangles sharing one vertex), there are two triangles of 2-twins but we cannot force all the edges of all the triangles of 2-twins to be in an optimal solution since the optimal solution contains two clusters. We can however prove that if there is a $K_4$ of 2-twins, these vertices are in the same cluster in any optimal solution.

Generalizing to any $i$ becomes harder and harder since the size of the clique is increasing. Instead we use the following fact, easier to prove: for every $i \leq 8$, if a vertex $u$ has at least $4i-1$ $i$-twins $X$ then $u \cup X$ are in the same cluster in any possible optimal solution. Note that we do not make any assumption on the twinness of pairs in $X$ nor on the existence of the edges in $X$. Our proof is computer assisted (the program runs in a few minutes for $i = 8$). We conjecture that this statements holds for any $i$. However, the lower bound on the number of $i$-twins might not be tight. For instance, it only gives 7 for $i = 2$ but it might be true that 5 is enough.

Let us summarize this labeling phase:

- For every pair of vertices $u, v$ that are either true twins, false twins, or 1-twins, label $(u, v)$ as an edge.
- For every subset of 4 vertices that forms a $K_4$ of 2-twins, label the pairs of the $K_4$ as edges.
- For every $i \leq 7$ and every vertex $u$ such that $u$ has a set $X$ of at least $4i - 1$ $i$-twins, label the pairs of $u \cup X$ as edges.

## 3 Description of the branching phase

Let $G$ be the input graph. A *cluster graph* is a partition of the vertices of $G$ into disjoint cliques. The *cost* of graph is the number of edge editions that has to be made to transform $G$ into this solution. We start with a solution where each vertex of the input graph $G$ is in its own cluster (and thus has a cost equal to $|E(G)|$). Furthermore, all along the algorithm we maintain a *current graph*, which is the graph obtained from $G$ by performing the edge editions corresponding to the branchings. The main routine on the branching step is the following: at each step we choose a pair of clusters $C_1$ and $C_2$ to branch on and either merge them (we add all the edges between vertices of $C_1$ and $C_2$, and these edges will never be removed in the current branch), or definitively separate them (we remove all the edges between vertices of $C_1$ and $C_2$, and no edges between vertices of $C_1$ and $C_2$ can be added back in the current branch). We denote edges (resp. non-edges) which cannot be removed (resp. added) in the current branch as *fixed*.

After each branching step we try to:

- cut the branch if possible, or
- merge or separate clusters according to enforcing rules.

Both the cutting of branches and the enforcing rules heavily rely on the computation of an upper bound (that we obtain through an heuristic) and the computation of a lower bound, that we describe below. The rule according to which we cut the branch is rather simple: we cut the branch if the lower bound on the current graph plus its cost (number of edges edited so far) is at least equal to the cost of the upped bound.

## Computation of the lower bound

To obtain the lower bound, we consider the current graph and compute how many edges (at least) has to be edited to find a solution. One can note that, for each $P_3$ on the graph, we have to edit at least one edge. So if we can find a collection of $P_3$ that pairwise intersect on at most one vertex, the cost of any solution is at least the cost of the current solution plus the number of $P_3$ in the collection. Unfortunately this rule is too weak since we often need to edit more than $|E|/2$ edges in the instances and the lower bound obtained this way is at most $|E|/2$. We can improve this rule by noting that, for every star $K_{1,\ell}$ the number of edits is at least $\ell - 1$. Our algorithm computes greedily a collection of $P_3$ that it extends to stars, from which we obtain the lower bound.
The collection of stars is then updated after each branching step. Note that we do not recompute the collection of stars from scratch, but rather only modify the stars that contain at least one endpoint of a pair edited during the current branching step. This allows us to always maintain a lower bound that corresponds to the current graph (and not only a lower bound on the initial graph), while only paying for a reasonable computational cost (since we only perform local modifications).

**Linear Programming.** For small instances (up to 120 vertices) we run a linear programming algorithm to compute a lower bound using fractional stars. We then round the solution to an integral solution whose size often matches the lower bound (which permits to conclude without the branching algorithm).

## Enforcing rules

There are a few cases where we can ensure, during the branching phase, that an edge will or will not be in the final solution, without computing twinness of vertices. Suppose that the current graph contains two clusters $C_1, C_2$ such that the number of (non yet fixed) non-edges between $C_1$ and $C_2$ plus the lower bound on the current graph is greater or equal to the upper bound. Then merging $C_1$ and $C_2$ cannot lead to a better solution, and thus we can delete all the $C_1, C_2$ edges and mark all the $C_1, C_2$ pairs as fixed non-edges. Similarly if the number of (non yet fixed) edges between the two clusters plus the lower bound is greater or equal to the upper bound, then we add all the missing $C_1, C_2$ edges and mark alll the $C_1, C_2$ pairs as fixed edges. We have a total of 5 such rules that we apply after each two branchings.

## References

1 Valentin Bartier, Gabriel Bathie, Nicolas Bousquet, Marc Heinrich, Théo Pierron, and Ulysse Prieto. Pace solver description: $\mu$solver – heuristic track. In *IPEC'21*, 2021.

2 Yixin Cao and Jianer Chen. Cluster editing: Kernelization based on edge cuts. *Algorithmica*, 64, August 2010. doi:10.1007/s00453-011-9595-1.