PACE Solver Description: μSolver – Heuristic track[*†]

Valentin Bartier
G-SCOP, Université Grenoble Alpes, 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

1 Large neighborhood local search for Cluster Editing

Our solver, μSolver [1], uses various local moves to iteratively improve a solution (that is, a partition of the input into clusters) during the allotted time. The initial solution is a trivial solution with n clusters: all vertices belong to the same cluster and the remaining $n-1$ clusters are empty.

The first local move that we use is the `best_move`: it takes as parameter a vertex v and a solution S, and returns a solution S' in which v has been moved to the cluster c that minimizes the cost of S'. One can notice that c is either the cluster of a neighbor of v or an empty cluster. Therefore, the `best_move` function can be implemented to run in $O(d(v))$ time, where $d(v)$ is the degree of v. This local move is not sufficient to obtain good solutions, as it is very sensitive to local minima.

Our second local move, `bfs_greedy`, aims to avoid these local minima by making larger moves. Instead of moving a single vertex at a time, we select a random subset X of t vertices, isolate them in the graph (by moving them to a cluster of size 0), and then insert them back using `best_move`. In order to maximize the efficiency of this move, we fill X with vertices that are close to one another. Namely, we call the function `select_BFS` on a vertex v, which performs a BFS starting from v, and returns the first t vertices that it visits. We then use this move on every vertex of the graph: the `select_BFS` function only returns vertices that were not returned before. Running `bfs_greedy` on every vertex can be done in $O(n+m)$.

Our final algorithm works in passes: a pass runs the `bfs_greedy` move on every vertex of every connected component, and runs passes until it timeouts.

2 Parameter learning

Notice that our `bfs_greedy` heuristic requires a parameter t. Our experiments showed a trade-off: setting $t \simeq 15$ yields good results on sparse instances, while setting $t \simeq 50$ yields

[*] 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.

[†] This work was supported by ANR project GrR (ANR-18-CE40-0032).

good results on dense instances. Instead of settling on a single value for every instance, we opted for a learning algorithm: we specify a set of values $t_1, \dots t_\ell$ with associated weights $w_1, \dots, w_\ell$. At each step, we sample a value t_s according to the weights (w_i) and select a set X of size t_s. If the call to the heuristic with this choice of t yields an improved solution, we add 1 to the value w_s.

3 Overview

We combine our local search with some kernelization rules, that apply mostly on sparse instances (see next section). Moreover, we process each connected component of the graph separately, as it is easy to see that vertices from different connected components cannot be in the same cluster of an optimal cluster editing.

Algorithms 1 and 2 give a global overview of the pseudo code of our solver.

Algorithm 1 Main solver

```
Input: $G, t, w$
 1: $G \leftarrow$ kernelize$(G)$
 2: for each connected component $C_i$ do
 3:     $S_i \leftarrow$ trivial_solution$(C_i)$
 4: while timeout is not reached do
 5:     for each connected component $C_i$ do
 6:         $j \leftarrow$ sample_weights$(w_1, \dots, w_\ell)$
 7:         $S' \leftarrow$ bfs_greedy$(C_i, S_i, t[j])$
 8:         if cost$(S')$ < cost$(S_i)$ then
 9:             $S_i \leftarrow S'$
10:             $w[j] \leftarrow w[j] + 1$
11: return $\mathcal{S} = \bigcup_i S_i$
```

Algorithm 2 `bfs_greedy` heuristic

```
Input: $C, S, t$
 1: $seen \leftarrow [false] * n$
 2: for each $v \in C$ in a random order do
 3:     if $seen[v]$ then
 4:         go to the next vertex
 5:     $X \leftarrow$ select_BFS$(C, v, seen, t)$
 6:     $S' \leftarrow S$
 7:     for $u$ in $X$ do
 8:         $S' \leftarrow$ isolate$(u, S')$
 9:     for $u$ in $X$ do
10:         $S' \leftarrow$ best_move$(u, S')$
11: return $S'$
```

4 Kernelization rules

We present here our kernelization rules (see figure) without their safeness proofs due to space constraints. Our algorithm applies these rules until none of them can be applied.

▶ **Rule 4.1.** *Let u be a vertex with either a 1-neighbor or two adjacent 2-neighbors. If u has another neighbor v such that u and v have no common neighbor, then delete uv.*

This implies that we delete all edges but one between each vertex and its 1-neighbors.

▶ **Rule 4.2.** *Let $uvxw$ be an induced C_4 where v and w have degree 2. Delete uv and wx.*

▶ **Rule 4.3.** *Let u be a 3-vertex with neighbors a, b, c.*

- *If ab, bc and ac are edges, a has degree 3 and b, c both have degree at most 5, delete all edges bx and cx for $x \notin \{u, a, b, c\}$.*
- *If ac and bc are edges, ab is a non-edge, and a, b, c all have degree at most 3, delete all edges ax and bx for $x \notin \{u, c\}$.*
- *If ab is an edge but not ac and bc, and a, b both have degree at most 3, delete uc and all edges ax and bx for $x \notin \{u, a, b\}$.*

The last item is actually a special case of a more general rule that we present below (with $k = 3$). The two other items are actually specification designed to handle the case where the disjointness hypothesis is not met anymore. However, we did not implement it since it does not provide a better kernelization on the public instances.

▶ **Rule 4.4.** *Let K be a clique on k vertices such that each vertex outside of K has at most one neighbor in K. For every $u \in K$, denote by $f(u)$ the number of neighbors of u outside of k. Write $K = \{u_1, \dots, u_k\}$ where $f(u_1) \leqslant \dots \leqslant f(u_k)$. If, for every $i \in [1, k]$, $\sum_{j=i+1}^{k} f(u_j) \leqslant \binom{k}{2} - \binom{i}{2}$, delete all edges with exactly one endpoint in K.*

As a final remark, note that Rule 4.3 allows to solve the Cluster Editing problem in polynomial time on subcubic graphs. Indeed, applying Rule 4.3 removes all the triangles in subcubic graphs. It then remains to find a maximum matching problem in the kernel.

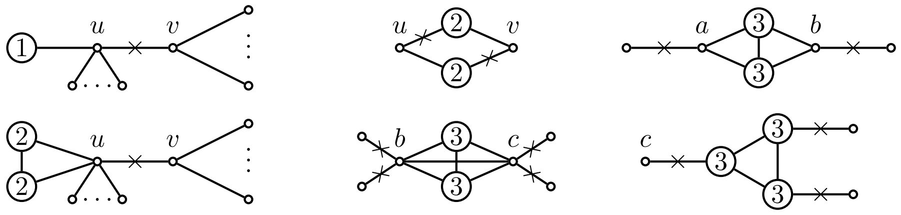

Figure 1 Kernelization rules. Vertices labeled by an integer i are of degree i.

References

1 Valentin Bartier, Gabriel Bathie, Nicolas Bousquet, Marc Heinrich, Théo Pierron, and Ulysse Prieto. Pace 2021 musolver, 2021. URL: `https://zenodo.org/record/4947325`, `doi:10.5281/ZENODO.4947325`.