μSolver description – PACE 2021 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 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 $O(d(v))$. This local move is not sufficient to obtain good solutions, as it is very sensitive to local minima.

Our second local move, `multi-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 `multi-greedy` on every vertex can be done in $O(n+m)$.

Our final algorithm works in passes: a pass runs the `multi-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, but setting $t \simeq 50$ yields 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 X and t yields an improved solution, we add 1 to the value w_s.

* This work was supported by ANR project GrR (ANR-18-CE40-0032).

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

3 Overview

We combine our local search with the kernelization rules for sparse instances implemented in our kernelization solver (see our kernalization solver description, submission 35, for more details). 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

$G \leftarrow$ `kernelize`(G)
for each connected component C_i **do**
    $S_i \leftarrow$ `trivial_solution`(C_i)
while timeout is not reached **do**
    for each connected component C_i **do**
        $j \leftarrow$ `sample_weights`$(w_1, \ldots, w_\ell)$
        $S' \leftarrow$ `bfs-greedy`$(C_i, S_i, t[j])$
        if `cost`$(S') <$ `cost`(S_i) **then**
            $S_i \leftarrow S'$
            $w[j] \leftarrow w[j] + 1$
return $\mathcal{S} = \bigcup_i S_i$

Algorithm 2 `bfs-greedy` heuristic

Input: C, S, t

$seen \leftarrow [false] * n$
for each $v \in C$ in a random order **do**
    if $seen[v]$ **then**
        go to next vertex
    $X \leftarrow$ `select_BFS`$(C, v, seen, t)$
    $S' \leftarrow S$
    for u in X **do**
        $S' \leftarrow$ `isolate`(u, S')
    for u in X **do**
        $S' \leftarrow$ `best_move`(u, S')
return S'