EXACT AND HEURISTIC ALGORITHMS FOR WEIGHTED CLUSTER EDITING

Sven Rahmann, Tobias Wittkop, Jan Baumbach, and Marcel Martin
Computational Methods for Emerging Technologies group,
Genome Informatics, Technische Fakultät,
Bielefeld University, D-33594 Bielefeld, Germany
Address correspondence to: Sven.Rahmann@cebitec.uni-bielefeld.de

Anke Truß and Sebastian Böcker
Lehrstuhl für Bioinformatik, Friedrich-Schiller-Universität Jena,
Ernst-Abbe-Platz 2, D-07743 Jena, Germany

Clustering objects according to given similarity or distance values is a ubiquitous problem in computational biology with diverse applications, e.g., in defining families of orthologous genes, or in the analysis of microarray experiments. While there exists a plenitude of methods, many of them produce clusterings that can be further improved. "Cleaning up" initial clusterings can be formalized as *projecting a graph on the space of transitive graphs*; it is also known as the *cluster editing* or *cluster partitioning* problem in the literature. In contrast to previous work on cluster editing, we allow arbitrary weights on the similarity graph. To solve the so-defined weighted transitive graph projection problem, we present (1) the first exact fixed-parameter algorithm, (2) a polynomial-time greedy algorithm that returns the optimal result on a well-defined subset of "close-to-transitive" graphs and works heuristically on other graphs, and (3) a fast heuristic that uses ideas similar to those from the Fruchterman-Reingold graph layout algorithm. We compare quality and running times of these algorithms on both artificial graphs and protein similarity graphs derived from the 66 organisms of the COG dataset.

1. INTRODUCTION

The following problem arises frequently in clustering applications: Given a set of objects V and a similarity or distance measure for each unordered pair $\{u, v\}$ of objects, we want to partition V into disjoint clusters. A common strategy is to choose a similarity threshold and construct the corresponding threshold graph: The objects constitute the nodes of the graph, and an edge is drawn between u and v if their similarity exceeds (distance falls below) the given threshold. In this case, u and v are called "similar", which we write as $u \sim v$. However, the resulting graph need not be transitive, meaning that $u \sim v$ and $v \sim w$ do not necessarily imply $u \sim w$. We wish to "clean up" such a preliminary clustering with as few edge changes as possible. Formal definitions are given below.

The similarity graph. We write V for the set of objects to be clustered; these are the vertices or nodes of the graph. We write $\binom{V}{k}$ for the set of k-element subsets of V. We use uv shorthand for an unordered pair $\{u, v\} \in \binom{V}{2}$.

We assume the availability of a symmetric *similarity function* $s : \binom{V}{2} \to \mathbb{R}$ such that u and v are *similar*, $u \sim v$, if and only if $s(u, v) :\equiv s(uv) > 0$. The edge set of the similarity graph is $E := \{uv : u \sim v\}$. Note that the similarity of an object to itself is not and need not be defined here. For any set $F \subseteq \binom{V}{2}$, we define $s(F) := \sum_{uv \in F} s(u, v)$.

A perfect clustering is characterized by the condition that the graph $G = (V, E)$ is *transitive*, defined by any of the following equivalent conditions:

(1) For each triple $uvw \in \binom{V}{3}$, the implication $uv \in E$ and $vw \in E \Rightarrow uw \in E$ holds.
(2) G contains no induced paths of length 2, i.e., for each triple $uvw \in \binom{V}{3}$, we have $|E \cap \{uv, vw, uw\}| \neq 2$.
(3) G is a disjoint union of cliques (i.e., of complete graphs).

Our goal is to edit a given graph $G = (V, E)$ by removing and adding edges in such a way that it becomes transitive. Each operation incurs a nonnegative cost: If $uv \in E$, the *edge removal cost* of uv is $s(u, v)$. If $uv \notin E$, the *edge addition cost* of uv is $-s(u, v)$. Note the following subtlety: If $s(u, v) = 0$, then initially $uv \notin E$, but it costs nothing to add this edge.

The *cost* to transform the initial graph $G = (V, E)$ into a graph $G' = (V, E')$ with different edge set E' is consequently defined as $cost(G \to G') :=$

$s(E \setminus E') - s(E' \setminus E)$.

Problem statement. The *weighted transitive graph projection problem* (WTGPP) is defined as follows. Given a similarity function $s : \binom{V}{2} \to \mathbb{R}$ and the weighted undirected graph $G = (V, E, s)$ with $E := \{uv : s(uv) > 0\}$, compute $\delta(G) := \min\{cost(G \to G') : G' \text{ transitive}\}$ and find one or all transitive G^* with $cost(G \to G^*) = \delta(G)$. Such G^* are called *best transitive approximations* to G or *transitive projections* or *least-cost cluster edits* of G. We also call this problem the *weighted cluster editing problem*.

Previous work and results. The *unweighted* version of this problem, where $s(u, v) \in \{+1, -1\}$ and $cost(G \to G') = |E \setminus E'| + |E' \setminus E| = |E \triangle E'|$, has been extensively studied and is also known as *cluster editing* in the literature. The first study that we are aware of goes back to Zahn[18] in 1964 and solves the problem on specially structured graphs (2-level hierarchies). On the negative side, the problem has been proven NP-hard in general at least twice independently[4, 16].

On the positive side, fixed-parameter tractability (FPT) of the unweighted cluster editing problem using the *minimum number of edge changes* as parameter k is well-studied. Gramm et al.[9] give a simple algorithm with running time $O(3^k+|V|^3)$ and, by applying a refined branching strategy, improve the time complexity to $O(2.27^k+|V|^3)$. recent experiments by Dehne et al.[3] suggest that the $O(2.27^k + |V|^3)$ algorithm is indeed faster than the $O(3^k+|V|^3)$ algorithm in practice. In theory, the best known algorithm[8] for the problem has running time $O(1.92^k + |V|^3)$, but this algorithm uses very complicated branching rules (137 initial cases) and has never been implemented. Damaschke[2] shows how to enumerate all optimal solutions.

Unfortunately, it is also known that almost all graphs are almost maximally far away from transitivity in the following sense, as shown by Moon[12]. Let $\mathcal{G}_n$ be the set of all $2^{\binom{n}{2}}$ graphs on n vertices. Note that each $(V, E) = G \in \mathcal{G}_n$ satisfies $\delta(G) \leq \binom{n}{2}/2$ because if $|E| \leq \binom{n}{2}/2$, we can remove all edges and obtain the transitive empty graph, and if $|E| \geq \binom{n}{2}/2$, we can add all missing edges and obtain the transitive complete graph. Now define the class $\mathcal{G}_{n,\varepsilon}$ of graphs that are "far away" from transitivity in the sense that $\delta(G) \geq (1-\varepsilon)\cdot\binom{n}{2}/2$. Then for any $\varepsilon > 0$, this class contains asymptotically almost all graphs, i.e., $|\mathcal{G}_{n,\varepsilon}|/|\mathcal{G}_n| \to 1$ as $n \to \infty$.

Nevertheless, the FPT results are important in practice because we can expect the preliminary clusterings that we obtain from real-world datasets to be "almost transitive" already.

To our knowledge, the WTGPP has not been subject to fixed-parameter approaches until now. Grötschel and Wakabayashi[10] formulated it as an integer linear program and gave a cutting plane algorithm, but apparently it has not been tried on large instances.

Our contributions. We present the first fixed-parameter algorithm for the WTGPP in Section 2, which is an extension of the FP algorithm from Ref. 9 for the unweighted case. Assuming $|s(uv)| \geq 1$ for all $uv \in \binom{V}{2}$, our algorithm checks in time $O(3^k + |V|^3)$ if there exists a transitive projection of cost at most k. In fact, the running time of our algorithm is much better in practice. We also approach the problem from another end and present a new $O(|V|^3 + |V||E| + |E|^2)$ time greedy algorithm that provably returns the correct transitive projection on a well-defined class of graphs that are "not too far away" from transitivity, but may return suboptimal solutions for other graphs (Section 3). In Section 4, we present a fast heuristic based on ideas from graph layouting. In practice, its running time is $O(|V|^2)$, while a worst-case analysis gives $O(|V|^4)$ for cases that do not seem to occur in practice.

Although this paper focuses on the new algorithms, we also present applications to simulated graphs and to protein similarity graphs derived from BLAST scores on the 66 organisms of the COG dataset[17]; these appear in Section 5. A concluding discussion is found in Section 6.

Preliminaries. Without loss of generality, the vertex set will be denoted by $V := \{1, \ldots, n\}$. The input to all algorithms in subsequent sections is the (symmetric) similarity function $s : \binom{V}{2} \to \mathbb{R}$, and the initial edge set $E := \{ij : s(ij) > 0\}$. We set $m := |E|$.

Without loss of generality, we may assume that the input graph consists of a single connected component. If not, we can treat each connected component separately, because an optimal solution will never join separate components; this is easily proved by contradiction.

The output of each algorithm is an edge set E^* and a cost $c^* := cost(G \to (V, E^*))$. We say that an algorithm *correctly solves* instance $G = (V, E, s)$ if $c^* = \delta(G)$.

Let $N(v) := \{u : s(uv) > 0\} \subset V$ denote the set of neighbors of v. We call $N_\cap(u, v) := N(u) \cap N(v)$ the *common neighbors* of u and v and $N_\triangle(u, v) := (N(u) \triangle N(v)) \setminus \{u, v\}$ their *non-common neighbors*; here $A \triangle B$ is the symmetric set difference of sets A and B.

Let $\mathcal{C}(G)$ be the set of all *conflict triples*, i.e., $uvw \in \binom{V}{3}$ that induce a path of length two: $\mathcal{C}(G) := \{uvw \in \binom{V}{3} : |E \cap \{uv, vw, uw\}| = 2\}$. As noted, G is transitive if and only if $\mathcal{C}(G) = \{\}$.

2. FIXED-PARAMETER ALGORITHM

Fixed-parameter algorithmics were introduced by Downey and Fellows in the late nineties[5]. They enable us to find exact solutions for several NP-hard problems. The basic idea is to choose a parameter for a given problem such that the problem is solvable in polynomial time when the parameter is fixed. A problem is *fixed-parameter tractable* with respect to the given parameter if there exists an algorithm which solves the problem in a running time of $O(f(k) \cdot |I|^c)$, where f is a function only dependent on the parameter k, $|I|$ is the size of the input, and c is a constant. See Ref. 13 for a recent overview on fixed-parameter algorithms.

In the following, we propose a fixed-parameter algorithm for the WTGPP parameterized with the (real-valued) cost k of an optimal solution. Given an instance of the problem and fixed k, the algorithm is guaranteed to find an optimal solution with cost at most k or to return that no such solution exists. The algorithm roughly adopts the branching strategy and data reduction rules of the $O(3^k + |V|^3)$ algorithm from Ref. 9 and runs in time $O(3^k + |V|^3)$ if *every* edge deletion or insertion has a cost of at least 1 (if not, costs may be scaled up to fulfill this requirement). While our algorithm accepts any positive real numbers as input, minimum edit costs are required to achieve a provable running time because there can be no fixed-parameter algorithm solving the problem with arbitrarily small weights unless P=NP.

Our algorithm requires a cost parameter k. So in order to find an optimal solution, i.e., the smallest k for which a G^* with $cost(G \to G^*) \leq k$ exists, we call the algorithm repeatedly, starting with $k = 1$. If we do not find a solution with this value, we increase k by 1, call the algorithm again and so forth. Note that for every k, we have to traverse the complete search tree and find the *best* solution with cost $\leq k$, if any. The overall structure of the algorithm is recursive. In the beginning, we start with the full input graph and the given parameter k.

Given G and $k \geq 0$, we first call the data reduction procedure described below. Then we pick a conflict triple $uvw \in \mathcal{C}(G)$ and repair it in each possible way by recursively branching into three subproblems. In order to ensure that the sub-problems do not overlap, we will in the process set some nonexistent edges to "forbidden" (so we can never add them) and some existent edges to "permanent" (so they cannot be removed). Initially all edges have no such label.

Data reduction. The following operations reduce the problem size. They are performed initially and for every sub-problem.

- *Remove cliques*: Identify connected components and remove all components that are cliques from the input graph. The algorithm can be called separately for each component.
- *Check for unaffordable edge modifications*: For each $uv \in \binom{V}{2}$, we calculate a lower bound $icf(uv)$ and $icp(uv)$ for setting uv to forbidden or permanent, respectively. When setting uv to forbidden, we state that u and v should be in different components and therefore should have no common neighbors. Conversely, setting uv to permanent means getting rid of all non-common neighbors. Lower bounds on the induced costs are obtained as

$$icf(uv) = \sum_{w \in N_\cap(u,v)} \min\{s(uw), s(vw)\};$$

$$icp(uv) = \sum_{w \in N_\triangle(u,v)} \min\{|s(uw)|, |s(vw)|\}.$$

  We maintain lists in which these costs are sorted by size and update these lists every time an edit operation is carried out. Data reduction now works as follows:

  (a) For all $uv \in V$ where $icf(uv) > k$ (i.e., which cannot be forbidden): Insert uv if necessary, and set uv to "permanent".
  (b) For all $uv \in V$ where $icp(uv) > k$ (i.e., which cannot be made permanent): Delete uv if necessary, and set uv to "forbidden".

If there is a pair uv such that both $icp(uv) > k$ and $icf(uv) > k$, the (sub-)problem instance is not solvable with parameter k.

- *Merge vertices incident to permanent edges*: As soon as we set an edge uv to permanent, it is obvious that u and v must be in the same clique in each solution found in this branch of the algorithm. In this case we *merge* u and v, creating a new vertex x.

  Note that if w is a neighbor both of u and of v, we create a new edge xw whose deletion costs as much as the deletion of both uw and vw. If w is neither a neighbor of u nor of v, we calculate the insertion cost of the nonexistent edge xw analogously. In case w is a neighbor either of u or of v but not both, uvw is a conflict triple, and we have to decide whether we delete the edge connecting w with u or v or we insert the nonexistent edge. By summing the similarities (one of which is negative) to calculate the respective value for xw we carry out the cheaper operation and maintain the possibility to edit xw later.

  Thus, we merge u and v into a new vertex x as follows: For each vertex $w \in V \setminus \{u, v\}$, set $s(xw) \leftarrow s(uw) + s(vw)$. Let $k \leftarrow k - icp(uv)$, and delete u and v from the graph.

Branching Strategy. After data reduction, let $uvw \in \mathcal{C}(G)$ be a conflict triple, and let u be the vertex of degree two and v, w be the leaves. We recursively branch into three cases.

(1) Insert the missing edge vw, and set all edges uv, uw, vw to "permanent".
(2) Delete edge uv, and set the remaining edge uw to "permanent" and the absent edges uv and vw to "forbidden".
(3) Delete edge uw, set it to "forbidden" (do not set the other edge labels).

In each branch, we lower k by the insertion or deletion cost required for the executed operation. If this would lead to $k < 0$, we skip this branch. This branching strategy gives us a search tree of size $O(3^k)$, but usually much smaller in practice.

Time complexity analysis. If we set an edge to forbidden or permanent, this can reduce the parameter k because we have to delete or insert an edge. This, in turn, may trigger other edges to be forbidden or permanent. We can show that the running time for merging two vertices is $O(|V|^2)$, and the total running time for data reduction of an arbitrary input graph is $O(|V|^3)$. A detailed proof is deferred to a full journal version of this paper.

If *every* edge deletion or insertion has a cost of at least 1, then we can show that our data reduction results in a problem kernel with at most $2k^2 + k$ vertices. For the weighted cluster editing algorithm, this would result in a total running time of $O(3^k \cdot k^4 + |V|^3)$. We use interleaving[14] by performing data reduction repeatedly during the course of the search tree algorithm whenever possible. This reduces the total running time to $O(3^k + |V|^3)$.

We stress that the faster $O(2.27^k + |V|^3)$ algorithm of Gramm et al.[9] for the unweighted case cannot be used to solve the WTGPP, because the branching strategy is based on an observation that does not hold for weighted graphs (Lemma 5 in Ref. 9). We are currently working on adapting this branching strategy to the weighted case.

3. GREEDY HEURISTIC

As in the fixed-parameter algorithm, all conflict triples $uvw \in \mathcal{C}(G)$ must be repaired to make G transitive. A repair consists of either removing one of the two existing edges or adding the missing edge. Observe that the hard part is to correctly "guess" the set of edges to remove. Thereafter, the edge insertions can easily be found by *transitive closure*, that is, adding those edges required to make each connected component a clique.

Our idea is to define a function that scores edge removals and then let the algorithm greedily delete the highest-scoring edge in each step until further deletions do not improve the solution.

Scoring edges. We define G's *deviation from transitivity* $D(G)$ as

$$D(G) := \sum_{uvw \in \mathcal{C}(G)} \min\{|s(uv)|, |s(vw)|, |s(uw)|\}. \quad (1)$$

We can now score edge removals: Let uv be an edge in $G = (V, E, s)$. Removing it yields $G'_{uv} := (V, E \setminus \{uv\}, s')$, where $s'(xy) = s(xy)$, except $s'(uv) = -\infty$ ("forbidden"). We call

$$\Delta_{uv}(G) := D(G) - D(G'_{uv}) - s(uv) \quad (2)$$

the *transitivity improvement* of edge uv. The term $s(uv)$ penalizes the edge removal.

Algorithm. In addition to the main algorithm, the greedy heuristic consists of two auxiliary functions, which we describe first.

Algorithm Remove–Culprit(G) returns the highest-scoring edge $\operatorname{argmax}_{uv\in E}\{\Delta_{uv}(G)\}$ and removes it from G. There are m edges; computing each $\Delta_{uv}(G)$ can be done in $O(n)$ since only triples containing uv need to be considered. Thus, the runtime of the first invocation is $O(mn)$. Subsequent invocations need only $O(m+n)$ time, $O(n)$ to update scores for edges around the deleted edge, and $O(m)$ to find the maximum score.

Algorithm Transitive–Closure–Cost(G) assumes G is connected; it returns the total cost of all edge additions required for a transitive closure of G, $\sum_{uv\in\binom{V}{2}} \max\{-s(uv), 0\}$, in time $O(n^2)$.

Algorithm Greedy–Heuristic(G) is the main algorithm. It returns a pair (*deletions*, *cost*), where *deletions* is the list of edges to be removed from G and *cost* is the total cost of all edit operations (both removals and additions). Remember that G is connected.

(1) *cost* ← Transitive–Closure–Cost(G).
(2) If *cost* = 0, return an empty list and cost 0.
(3) Set *deletions* ← *empty list*; *delcost* ← 0.
(4) Repeat the following steps until G consists of two connected components G_1 and G_2.
  (a) uv ← Remove–Culprit(G)
  (b) append uv to *deletions*
  (c) increase *delcost* by $s(uv)$
(5) Adjust *deletions* such that it only includes edges that contribute to the cut between G_1 and G_2. Adjust *delcost* accordingly, and re-add incorrect edges to G_1 and G_2.
(6) Solve the problem recursively for G_1 and G_2, as long as there is a chance for a better solution:
  If *delcost* ≥ *cost*, return (*empty list*, *cost*).
  ($list_1$, $cost_1$) ← Greedy–Heuristic(G_1).
  If *delcost* + $cost_1$ ≥ *cost*,
    return (*empty list*, *cost*).
  ($list_2$, $cost_2$) ← Greedy–Heuristic(G_2)
  If *delcost* + $cost_1$ + $cost_2$ ≥ *cost*,
    return (*empty list*, *cost*).
(7) Append $list_1$ and $list_2$ to *deletions*.
  Return (*deletions*, *delcost*+$cost_1$+$cost_2$).

If the "safety net" in step 5 is never invoked, Greedy–Heuristic deletes each of the m edges at most once across all recursions. After each deletion, both determining connected components and Remove–Culprit require $O(m+n)$ time. Also, Transitive–Closure–Cost takes $O(n^2)$ time for each cut, of which there are at most $n-1$. Thus, the runtime is $O\big(m(m+n)+n^3\big)$.

Correctness of the greedy heuristic for special graphs. We show that the greedy heuristic correctly computes the transitive projection of certain classes of graphs in the unweighed case, where $s(i,j) \in \{\pm 1\}$. Here Eq. (1) becomes $D(G) = |\mathcal{C}(G)|$ and Eq. (2) becomes $\Delta_{uv}(G) = |\mathcal{C}(G'_{uv})| - |\mathcal{C}(G)| - 1 = |N_\triangle(u,v)| - |N_\cap(u,v)| - 1$, since triples not containing edge uv cancel out.

Let T be an unweighted transitive graph consisting of r cliques $C_1, \ldots, C_r$ with $n_i := |C_i|$. Graph G is obtained from T by edge modifications. Let δ_u be the number of u-incident edges deleted from T, and ι_u the number u-incident edges added to T to obtain G.

Lemma. (1) Let $uv \in E(G) \cap E(T)$ be an intra-cluster edge of C_i.
Then $\Delta_{uv}(G) \le 2\delta_u + 2\delta_v + \iota_u + \iota_v - n_i + 1$.
(2) Let $xy \in E(G) \setminus E(T)$ be an inter-cluster edge between $C_i \ni x$ and $C_j \ni y$.
Then $\Delta_{xy}(G) \ge n_i + n_j - (\delta_x + \delta_y + 2\iota_x + 2\iota_y) + 1$.

Proof. We count the common and non-common neighbors of u.

(1) There are no non-common neighbors of uv in T, and each edge deletion or insertion incident to u or v creates at most one. Therefore $|N_\triangle(u,v)| \le \delta_u + \delta_v + \iota_u + \iota_v$. There are $n_i - 2$ common neighbors of uv in T, and each edge deletion incident to u or v removes at most one. Thus $|N_\cap(u,v)| \ge n_i - 2 - (\delta_u + \delta_v)$.

(2) After inserting xy into T, this edge has $(n_i - 1) + (n_j - 1)$ non-common neighbors. Each deletion incident to x or y decreases this number, and each of the $\iota_x - 1$ plus $\iota_y - 1$ additional insertions incident to x or y might also decrease this number. Thus $|N_\triangle(x,y)| \ge n_i + n_j - (\delta_x + \delta_y + \iota_x + \iota_y)$. On the other hand, each insertion can also create a common neighbor; thus $|N_\cap(x,y)| \le \iota_x + \iota_y - 2$.

Theorem. GREEDY–HEURISTIC(G) recovers the original transitive graph T if the following assumption holds: For each vertex from any C_i in T, at most $2n_i/9$ edges to vertices in other clusters are added and at most $2n_i/9$ of the edges to vertices in the same cluster are removed to obtain G.

Proof. We show that $\Delta_e(G) > \Delta_f(G)$ for any inter-cluster edge e and intra-cluster edge f. Assume that $e = xy$ lies between C_i and C_j, and that $f = uv$ lies in C_i. Using the Lemma and the 2/9-assumption, $\Delta_e(G) - \Delta(f) \geq 2n_i - (\delta_x + 2\iota_x + 2\delta_u + 2\delta_v + \iota_u + \iota_v) + n_j - (\delta_y + 2\iota_y) \geq n_j/3 > 0$, as all n_i-terms cancel out. Therefore, GREEDY-HEURISTIC will always remove inter-cluster edges first. This also shows that the "safety net" (step 5) of the algorithm is unnecessary here.

4. LAYOUT-BASED HEURISTIC

Our final heuristic is based on physical intuition and motivated by graph layouting, initially introduced by Fruchterman and Reingold[6]. It has later been extended and used for the visualization of structural and functional relationships in biological networks, e.g., in BioLayout[7].

The main idea of these layout algorithms is to arrange all nodes on a 2-dimensional plane to fit esthetic criteria (such as even node distribution in a frame and inherent symmetry reflection). The graph's nodes are interpreted as magnets (or electrical charges of the same kind), and edges are replaced by rubber bands to form a physical system. The nodes are initially placed randomly or in a circle, for example, and then left to the forces of the system, so that the magnetic repulsion and the band's attraction forces on the nodes move the system to a minimal energy state. While a physical system provides the motivation for these algorithms, in the actual implementation the nodes need not move according to exact physical laws.

We have adapted and extended these ideas: The layout of the graph is used to partition it into disjoint connected components. Our algorithm proceeds in three phases: (1) layout, (2) partitioning, and (3) postprocessing.

Layout phase. The goal is to find a position $pos[i] = (pos[i]_1, pos[i]_2) \in \mathbb{R}^2$ for each node $1 \leq i \leq n$, starting with a circular layout of radius ρ_0 (a user-defined parameter) around the origin. We define the *distance* $d(i,j)$ of nodes i and j as their Euclidean distance in the layout: $d(i,j) := \left(\sum_{d=1}^{2} (pos[i]_d - pos[j]_d)^2\right)^{1/2}$.

For a user-defined number R of iterations, we compute the displacement of each node, and update the position $pos[i]$ of each node i accordingly. We have allowed ourselves some freedom in deriving a good displacement vector. In particular, we do not compute forces, accelerations, and velocities of points, but for simplicity's sake, directly apply a displacement vector to a node once it has been computed according to the rules below. In this sense, the physical system described above serves only as a motivation, but not as a model for the algorithm.

In round $r \in \{1, \ldots, R\}$, we compute the displacement of node i as follows. For each node $j \neq i$ with $s(ij) > 0$, we move i into the direction of j (the unit vector of this direction is $(pos[j] - pos[i])/d(i,j)$) by an amount of $f_{\text{att}} \cdot F_{\text{att}}(d(i,j)) \cdot s(ij)$. Here $F_{\text{att}}(d)$ is a strictly increasing function of the distance — we use $F_{\text{att}}(d) := \log(d+1)$ —, and $f_{\text{att}} > 0$ is a user-defined scaling factor for attraction. Conversely, for each node $j \neq i$ with $s(ij) < 0$, we move i away from j by an amount of $f_{\text{rep}} \cdot F_{\text{rep}}(d(i,j)) \cdot |s(ij)|$, where $F_{\text{rep}}(d) := 1/F_{\text{att}}(d)$ is strictly decreasing, and $f_{\text{rep}} > 0$ is another scaling factor.

Finally, the magnitude of the displacement vector is cut off at a maximal value $M(r)$ that depends on the iteration r: We use $M(r) = n \cdot M_0 \cdot (1/(r+1))^2$ to obtain increasingly small displacements in later iterations. Again, $M_0 > 0$ is a user-defined parameter.

After the displacement of a node i has been computed, the node is immediately moved, before the displacement of node $i+1$ is computed. While this does not agree with physical model, we have found that it speeds up convergence of the layout and saves memory for the displacement vectors for each node. After all nodes have been moved, the next iteration starts. The layout phase obviously runs in $\Theta(R \cdot n^2)$ time. The actions of the algorithm are visualized in Figure 1.

For the cluster editing problem based on protein sequence similarities, we use the following parameters: number of iterations $R = 186$, initial circular layout radius $\rho_0 = 200$, repulsion scaling factor $f_{\text{rep}} = 1.687$, attraction scaling factor $f_{\text{att}} = 1.245$, $M_0 = 633$. The best parameter constellation is

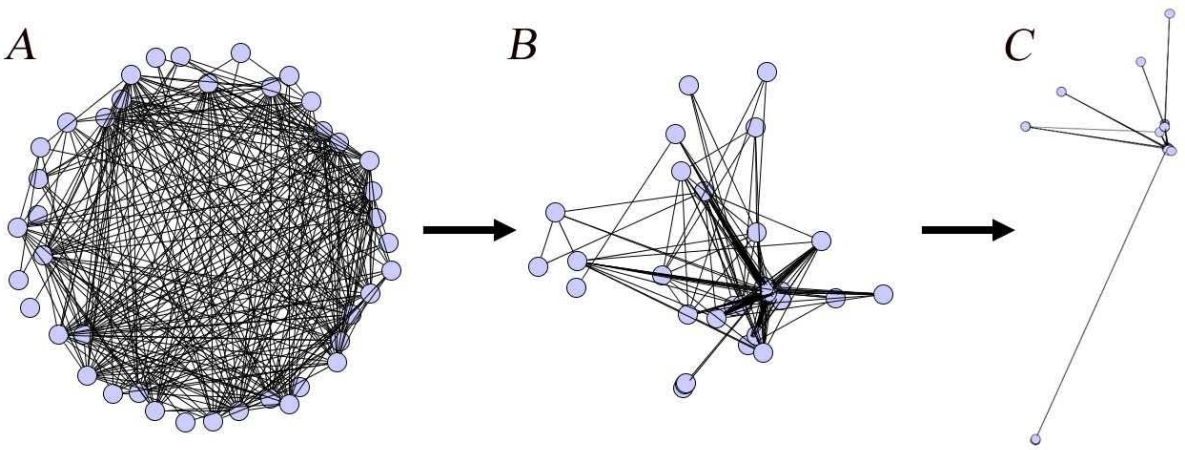

Fig. 1. Layout of a graph with 41 nodes after (A) 3, (B) 10 and (C) 90 iterations.

(more or less) specific to the concrete problem and has been obtained by an evolutionary training procedure by using the cost function as quality function. It is included in our implementation to enable the user to perform parameter calibration for arbitrary applications.

Partitioning phase. The nodes' positions after R rounds are used to partition the graph geographically. Given a distance parameter δ, we single-linkage cluster all nodes, meaning that nodes i and j belong to the same cluster if there exist nodes $i = i_0, i_1, \ldots i_K = j$ such that $d(i_{k-1}, i_k) \leq \delta$ for all $k = 1, \ldots, K$. We determine $cost(G \to G^*_\delta)$ for the so defined transitive graph G^*_δ.

To find a good G^*_δ, we start with a small distance parameter $\delta_{\text{init}} := 0$ and increase it ($\delta \leftarrow \delta + \sigma$) by a growing step size σ: Initially $\sigma \leftarrow \sigma_{\text{init}} := 0.01$; subsequently $\sigma \leftarrow \sigma \cdot f_\sigma$ with factor $f_\sigma := 1.1$. This continues until $\delta \geq \delta_{\max} := 300$.

The best value for δ, along with its cost, is remembered. Obviously, the time complexity of the partitioning phase is $O(D \cdot n^2)$, where D is the number of different values for δ.

Postprocessing. The geometric single-linkage clustering is further improved by the postprocessing, which takes $O(n^4)$ time in the worst case, but this almost never happens in practice. Effectively, the running overall time is $O(n^2)$. The two postprocessing steps are:

(1) For each pair of clusters, we check if joining them into a single cluster decreases overall cost and perform this operation if appropriate. During this step, we especially reduce the number of erroneous singleton nodes. This happens in arbitrary but deterministic order as long as merging a pair of clusters results in an improvement.

(2) For each node i and cluster C with $i \notin C$, we check if moving i to C decreases overall cost and perform this operation if appropriate. We also repeat this step as long as further improvements result.

5. RESULTS

We implemented the FP algorithm in C++, the greedy heuristic in Python, and the layout-based heuristic in Java. While with modern Java virtual machines, running times of Java programs are comparable to those of C++ programs, there is a higher start-up cost, which especially hurts performance for small problem instances. Python running times are about 10 times slower than those of the comparable C++ implementation. This should be kept in mind when comparing the running times of our implementations. All measurements were taken on a SunFire 880 with 900-MHz UltraSPARC III+ processors and 32 GB of RAM.

Artificial graphs. We generate random artificial graphs as follows. Given the number of nodes n, we randomly select an integer $k \in [1, n]$ and define the corresponding nodes to be a cluster. We proceed in the same way with the remaining $n - k$

Table 1. Results on artificial graphs with different numbers of nodes n, resulting in different ranges of edge numbers m. For each $n \leq 50$, ten random instances were generated. For each $n \geq 60$, where the FP algorithm did not finish in reasonable time, only five instances were generated. Costs and running times are averages over these 10 resp. 5 instances. Smallest costs and running times are marked in boldface. The (Diff.) columns show the relative cost difference against the optimal solution returned by FP, where possible. Abbreviations: FP: fixed parameter algorithm; Greedy: greedy heuristic; Layout: layout-based heuristic.

Parameters		Costs					Running Times [s]		
n	$m \in$	FP	Greedy	(Diff.)	Layout	(Diff.)	FP	Greedy	Layout
10	[11,30]	**95.75**	96.17	(+0%)	**95.75**	(+0%)	**0.035**	0.242	0.845
20	[65,165]	**301.89**	305.22	(+1%)	**301.89**	(+0%)	**0.152**	0.538	1.407
30	[138,296]	**671.25**	671.51	(+0%)	**671.25**	(+0%)	2.756	**1.157**	1.876
40	[251,533]	**1238.3**	1238.31	(+0%)	1238.31	(+0%)	72.109	3.167	**2.816**
50	[402,821]	**1859.99**	**1859.99**	(+0%)	**1859.99**	(+0%)	2204.862	8.315	**3.353**
60	[515,1252]	—	**2742.3**	(—)	**2742.3**	(—)	—	19.198	**3.972**
70	[694,1911]	—	**3608.54**	(—)	3609.48	(—)	—	58.124	**4.358**
80	[1141,2094]	—	4729.52	(—)	**4722.08**	(—)	—	69.056	**4.698**
90	[1248,2969]	—	**6106.56**	(—)	**6106.56**	(—)	—	128.986	**5.384**
100	[1711,3157]	—	**7494.36**	(—)	**7494.36**	(—)	—	207.958	**5.464**

nodes until no nodes are left. This gives us a random number of clusters of random sizes. Then the similarities of objects within a cluster are drawn from a Gaussian distribution $\mathcal{N}(\mu_{\text{in}}, \sigma_{\text{in}}^2)$; they are positive on average, but negative with some probability. Similarities of objects in different clusters are conversely drawn from a Gaussian distribution $\mathcal{N}(\mu_{\text{ex}}, \sigma_{\text{ex}}^2)$, which leads to negative values on average. If the parameters are chosen carefully, this construction leads to "almost transitive" graphs. For our experiments, we choose $\mu_{\text{in}} = 21$, $\mu_{\text{ex}} = -21$, $\sigma_{\text{in}} = \sigma_{\text{ex}} = 20$, so that the probability of seeing an undesired or missing edge is about 0.147 per node pair.

Table 1 shows the results. We see that the FP algorithm is the fastest one for small graphs, but reaches its limits above 50 nodes. On the other hand, the greedy and layout-based heuristics perform almost as well, while requiring significantly less time. The layout-based heuristic is much faster on large components, but first requires a good choice of parameters, as discussed in Section 4.

Protein similarity graph from the COG dataset. We test the algorithms on the 66 organisms of the COG dataset[17] from `http://www.ncbi.nlm.nih.gov/COG/`, i.e., on the protein sequences from `ftp://ftp.ncbi.nih.gov/pub/COG/COG/myva/`.

We define the similarity score of two proteins as follows: First let $s(u \to v) := \sum_{H \in \mathcal{H}(u \to v)} [-\log_{10} E(H)] - 2 \cdot (|\mathcal{H}(u \to v)| - 1)$. Here $\mathcal{H}(u \to v)$ denotes the set of high-scoring pairs (HSPs) with E-value better than 10^{-2} returned when BLASTing u against v. We subtract a penalty of 2 score points for each HSP beyond the highest-scoring one. We similarly define the score $s(v \to u)$ by BLASTing v against u. Finally we define the symmetric similarity score $s(u, v) := \min\{s(u \to v), s(v \to u)\} - T$, where we use a threshold of $T = 10$, corresponding to an E-value of 10^{-10}.

The resulting similarity matrix defines a graph of 42563 (trivially transitive) connected components of size 1 and 2, and 8037 larger components, 3964 of which are not transitive; these are the input to our algorithms. Figure 2 shows a histogram of initial component sizes $|V|$. There are 70 intransitive components with $|V| > 200$ that are not shown in the histogram, the largest of size 8836.

As all three algorithms perform well on very small components (which could be solved by exhaustive enumeration), we now restrict our attention to the 1243 components with $|V| \geq 20$. For each instance and each algorithm, we limit computation time to 48 hours; thus we could find the exact FP solution for 825 of the 1243 components in the alloted time.

Figure 3 (left) shows the relative cost of the solutions found by Greedy and Layout in comparison to the optimal one found by FP for the 825 components. Both heuristics work quite well: In 635 out of 825 cases, Greedy returns the optimal solution; and in 811 out of 825 cases, Layout returns the optimal solution. This behavior is relatively independent of the size or complexity of the graph (shown on the x-axis). The solution returned by Layout deviates in only two cases by more than 5% from the optimal so-

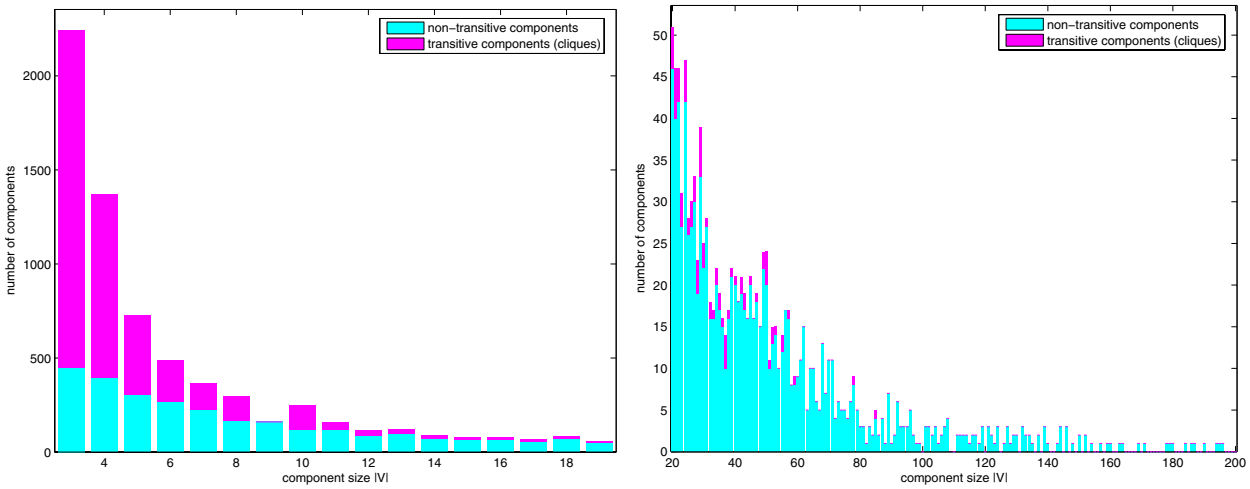

Fig. 2. Initial distribution of component sizes $|V|$ for the complete COG dataset in the range $3 \leq |V| < 20$ (left) and $20 \leq |V| \leq 200$ (right). Cyan (lower bars): number of non-transitive components. Magenta (upper bars): number of transitive components.

lution; this happens in 95 cases with Greedy, whose maximal deviation is about 50% in rare cases.

Figure 3 (right) visualizes the running times of the different algorithms against component complexity for all 1243 components. It is evident that the FP algorithm is fastest for small components, but quickly hits a wall for larger ones. Greedy is quickest for medium-sized components, but its running time grows faster with graph complexity than that of Layout, which is the only feasible algorithm for the largest components.

6. DISCUSSION AND CONCLUSION

We have put forward three algorithms for weighted transitive graph projection or weighted cluster editing that cover the whole spectrum from an exact fixed-parameter algorithm to pure heuristics. If graphs that arise from "real" data are *not* far away from transitivity (in contrast to random graphs, which are highly intransitive with high probability according to Moon's result[12]), we can find the optimal solution to the WTGPP with an FP algorithm in reasonable time for medium-sized components, and close-to-optimal solutions with well-engineered heuristics in guaranteed polynomial time. The FP and the Greedy algorithm complement each other well: The former guarantees the exact solution (and runs quickly for almost transitive graphs); the latter always runs in polynomial time and guarantees an optimal solution for close-to-transitive graphs. The Layout heuristic works very well in practice, but has no provable guarantees.

Our study shows that real protein similarity graphs are indeed close-to-transitive, and the three algorithms perform quite well on these WTGPP instances. Although not in the scope of this paper, the WTGPP has numerous potential applications to be investigated. Here we merely used the COG dataset as a comparative illustration of the respective capabilities of our three algorithms. Applications naturally arise in delineating gene and protein families[11, 17] (which in turn can be used as a preprocessing method for gene cluster discovery[15]) and in the discovery of structure in protein complexes or of communities in social or biological networks.

To further understand and improve the FP algorithm, it is of interest to systematically compare the branching strategy of our FP algorithm with that of a general ILP solver, using the cutting plane algorithm of Ref. 10, which so far has not been attempted on large components.

Acknowledgments and availability.

Tobias Wittkop is supported by the DFG GK Bioinformatik. Jan Baumbach is supported by the International NRW Graduate School in Bioinformatics and Genome Research. The fixed-parameter algorithm was implemented and engineered by Sebastian Briesemeister. We thank M. Madan Babu for many constructive comments, and Andreas Dress for point-

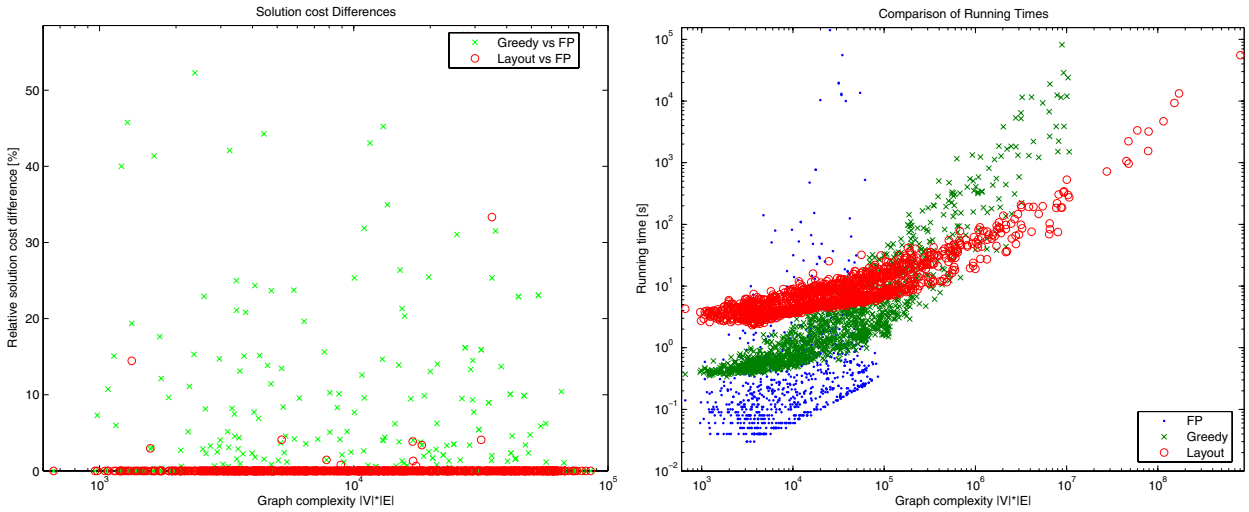

Fig. 3. *Left*: Relative cost differences of the solutions in percent (y-axis) found by Greedy and Layout in comparison to the exact Fixed-Parameter (FP) algorithm. Only those components whose exact solution could be computed in less than 48 hours are shown. For both Greedy and Layout, in the majority of cases, the optimal solution is found. Note that the x-axis, which shows the component complexity (we use $|V| \cdot |E|$), is logarithmic. *Right*: Running times of FP, Greedy, and Layout against component complexity. Both axes are logarithmic.

ing out the work of Grötschel and Wakabayashi.

Supplementary material and source code is available at `http://gi.cebitec.uni-bielefeld.de/transitivegraphprojection/`.

References

1. S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Res*, 25(17):3389–3402, 1997.
2. P. Damaschke. On the fixed-parameter enumerability of cluster editing. In D. Kratsch, editor, *Proc. of International Workshop on Graph Theoretic Concepts in Computer Science (WG 2005)*, volume 3787 of *LNCS*, pages 283–294. Springer, 2005.
3. F. Dehne, M. A. Langston, X. Luo, S. Pitre, P. Shaw, and Y. Zhang. The cluster editing problem: Implementations and experiments. In *Proc. of International Workshop on Parameterized and Exact Computation (IWPEC 2006)*, volume 4169 of *LNCS*, pages 13–24. Springer, 2006.
4. S. Delvaux and L. Horsten. On best transitive approximations to simple graphs. *Acta Informatica*, 40(9):637–655, 2004.
5. R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer, 1999.
6. T. M. J. Fruchterman and E. M. Reingold. Graph drawing by force-directed placement. *Software - Practice and Experience*, 21(11):1129–1164, 1991.
7. L. Goldovsky, I. Cases, A. J. Enright, and C. A. Ouzounis. BioLayout(Java): versatile network visualisation of structural and functional relationships. *Applied Bioinformatics*, 4(1):71–74, 2005.
8. J. Gramm, J. Guo, F. Hüffner, and R. Niedermeier. Automated generation of search tree algorithms for hard graph modification problems. *Algorithmica*, 39(4):321–347, 2004.
9. J. Gramm, J. Guo, F. Hüffner, and R. Niedermeier. Graph-modeled data clustering: Exact algorithm for clique generation. *Theor. Comput. Syst.*, 38(4):373–392, 2005.
10. M. Grötschel and Y. Wakabayashi. A cutting plane algorithm for a clustering problem. *Mathematical Programming, Series B*, 45:59–96, 1989.
11. A. Krause, J. Stoye, and M. Vingron. Large scale hierarchical clustering of protein sequences. *BMC Bioinformatics*, 6:15, 2005.
12. J. W. Moon. A note on approximating symmetric relations by equivalence classes. *SIAM Journal of Applied Mathematics*, 14(2):226–227, 1966.
13. R. Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Oxford University Press, 2006.
14. R. Niedermeier and P. Rossmanith. A general method to speed up fixed-parameter-tractable algorithms. *Inform. Process. Lett.*, 73:125–129, 2000.
15. S. Rahmann and G. W. Klau. Integer linear programs for discovering approximate gene clusters. In P. Bucher and B. Moret, editors, *Proceedings of the 6th Workshop on Algorithms in Bioinformatics (WABI)*, volume 4175 of *LNBI*, pages 298–309. Springer, 2006.
16. R. Shamir, R. Sharan, and D. Tsur. Cluster graph modification problems. *Discrete Applied Mathematics*, 144:173–182, 2004.

17. R. L. Tatusov, N. D. Fedorova, J. D. Jackson, A. R. Jacobs, B. Kiryutin, E. V. Koonin, D. M. Krylov, R. Mazumder, S. L. Mekhedov, A. N. Nikolskaya, B. S. Rao, S. Smirnov, A. V. Sverdlov, S. Vasudevan, Y. I. Wolf, J. J. Yin, and D. A. Natale. The COG database: an updated version includes eukaryotes. *BMC Bioinformatics*, 4:41, 2003.
18. C. T. Zahn Jr. Approximating symmetric relations by equivalence relations. *Journal of the Society of Industrial and Applied Mathematics*, 12(4):840–847, 1964.