Noname manuscript No.
(will be inserted by the editor)

Tiers for peers

a practical algorithm for discovering hierarchy in weighted networks

Nikolaj Tatti

the date of receipt and acceptance should be inserted later

Abstract Interactions in many real-world phenomena can be explained by a strong hierarchical structure. Typically, this structure or ranking is not known; instead we only have observed outcomes of the interactions, and the goal is to infer the hierarchy from these observations. Discovering a hierarchy in the context of directed networks can be formulated as follows: given a graph, partition vertices into levels such that, ideally, there are only edges from upper levels to lower levels. The ideal case can only happen if the graph is acyclic. Consequently, in practice we have to introduce a penalty function that penalizes edges violating the hierarchy. A practical variant for such penalty is agony, where each violating edge is penalized based on the severity of the violation. Hierarchy minimizing agony can be discovered in $\mathcal{O}(m^2)$ time, and much faster in practice. In this paper we introduce several extensions to agony. We extend the definition for weighted graphs and allow a cardinality constraint that limits the number of levels. While, these are conceptually trivial extensions, current algorithms cannot handle them, nor they can be easily extended. We solve the problem by showing the connection to the capacitated circulation problem, and we demonstrate that we can compute the exact solution fast in practice for large datasets. We also introduce a provably fast heuristic algorithm that produces rankings with competitive scores. In addition, we show that we can compute agony in polynomial time for any convex penalty, and, to complete the picture, we show that minimizing hierarchy with any concave penalty is an **NP**-hard problem.

Keywords Hierarchy discovery; agony; capacitated circulation; weighed graphs

The research described in this paper builds upon and extends the work appearing in ICDM15 as [20].

Nikolaj Tatti
Helsinki Institute for Information Technology (HIIT) and
Department of Information and Computer Science, Aalto University, Finland
E-mail: nikolaj.tatti@aalto.fi

1 Introduction

Interactions in many real-world phenomena can be explained by a strong hierarchical structure. As an example, it is more likely that a line manager in a large, conservative company will write emails to her employees than the other way around. Typically, this structure or ranking is not known; instead we only have observed outcomes of the interactions, and the goal is to infer the hierarchy from these observations. Discovering hierarchies or ranking has applications in various domains: (*i*) ranking individual players or teams based on how well they play against each other [4], (*ii*) discovering dominant animals within a single herd, or ranking species based on who-eats-who networks [9], (*iii*) inferring hierarchy in work-places, such as, U.S. administration [12], (*iv*) summarizing browsing behaviour [11], (*v*) discovering hierarchy in social networks [7], for example, if we were to rank twitter users, the top-tier users would be the content-providers, middle-tiers would spread the content, while the bottom-tier are the consumers.

We consider the following problem of discovering hierarchy in the context of directed networks: given a directed graph, partition vertices into ranked groups such that there are only edges from upper groups to lower groups.

Unfortunately, such a partitioning is only possible when the input graph has no cycles. Consequently, a more useful problem definition is to define a penalty function p on the edges. This function should penalize edges that are violating a hierarchy. Given a penalty function, we are then asked to find the hierarchy that minimizes the total penalty.

The feasibility of the optimization problem depends drastically on the choice of the penalty function. If we attach a constant penalty to any edge that violates the hierarchy, that is, the target vertex is ranked higher or equal than the source vertex, then this problem corresponds to a feedback arc set problem, a well-known **NP**-hard problem [2], even without a known constant-time approximation algorithm [5].

A more practical variant is to penalize the violating edges by the severity of their violation. That is, given an edge (u, v) we compare the ranks of the vertices $r(u)$ and $r(v)$ and assign a penalty of $\max(r(u) - r(v) + 1, 0)$. Here, the edges that respect the hierarchy receive a penalty of 0, edges that are in the same group receive a penalty of 1, and penalty increases linearly as the violation becomes more severe, see Figure 1. This particular score is referred as *agony*. Minimizing agony was introduced by Gupte et al. [7] where the authors provide an exact $\mathcal{O}\left(nm^2\right)$ algorithm, where n is the number of vertices and m is the number of edges. A faster discovery algorithm with the computational complexity of $\mathcal{O}\left(m^2\right)$ was introduced by Tatti [19]. In practice, the bound $\mathcal{O}\left(m^2\right)$ is very pessimistic and we can compute agony for large graphs in reasonable time.

In this paper we specifically focus on agony, and provide the following main extensions for discovering hierarchies in graphs.

weighted graphs: We extend the notion of the agony to graphs with weighted edges. Despite being a conceptually trivial extension, current algorithms [7, 19] for computing agony are specifically design to work with unit weights, and cannot be used directly or extended trivially. Consequently, we need a new approach to minimize the agony, and in order to do so, we demonstrate that we can transform the problem into a capacitated circulation, a classic graph task known to have a polynomial-time algorithm.

cardinality constraint: The original definition of agony does not restrict the number of groups in the resulting partition. Here, we introduce a cardinality constraint k and we are asking to find the optimal hierarchy with at most k groups. This constraint works both with weighted and non-weighted graphs. Current algorithms for solving agony cannot handle cardinality constraints. Luckily, we can enforce the constraint when we transform the problem into a capacitated circulation problem.

fast heuristic: We introduce a fast divide-and-conquer heuristic. This heuristic is provably fast, see Table 1, and—in our experiments—produces competitive scores when compared to the optimal agony.

convex edge penalties: Minimizing agony uses linear penalty for edges. We show that if we replace the linear penalty with a convex penalty, see Figure 1, we can still solve the problem in polynomial time by the capacitated circulation solver. However, this extension increases the computational complexity.

concave edge penalties: To complete the picture, we also study concave edge penalties, see Figure 1. We show that in this case discovering the optimal hierarchy is an **NP**-hard problem. This provides a stark difference between concave and convex edge penalties.

canonical solution: A hierarchy minimizing agony may not be unique. For example, given a DAG any topological sorting of vertices will give you an optimal agony of 0. To address this issue we propose to compute a *canonical* solution, where, roughly speaking, the vertices are ranked as high as possible without compromising the optimality of the solution. We demonstrate that this solution is unique, it creates a hierarchy with the least amount of groups, and that we can compute it in $\mathcal{O}(n \log n + m)$ time, if we are provided with the optimal solution and the flow resulted from solving the capacitated circulation.

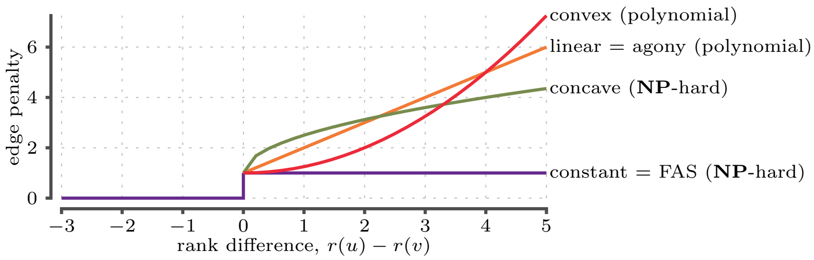

Fig. 1 A toy example of edge penalties as a function of the rank difference between the vertices.

Table 1 Summary of running times of different algorithms for computing agony: n is the number of vertices, m is the number of edges, k is the number of allowed ranks.

Algorithm	variant	input type	running time
Exact	plain		$\mathcal{O}(m \log n(m + n \log n))$
Exact	speed-up	unweighted	$\mathcal{O}(m(\min(kn, m) + n \log n))$
Exact	speed-up	weighted	$\mathcal{O}(m \log n(m + n \log n))$
Canonical	–	optimal rank and the flow	$\mathcal{O}(m + n \log n)$
Heuristic	plain	no cardinality constraint	$\mathcal{O}(m \log n)$
Heuristic	plain	cardinality constraint	$\mathcal{O}\big(m \log n + k^2 n\big)$
Heuristic	SCC	no cardinality constraint	$\mathcal{O}(m \log n)$
Heuristic	SCC	cardinality constraint	$\mathcal{O}\big(m \log n + k^2 n + km \log n\big)$

This paper is an extension of a conference paper [20]. In this extension we significantly speed-up the exact algorithm, propose a provably fast heuristic, and provide a technique for selecting unique canonical solutions among the optimal rankings.

The rest of the paper is organized as follows. We introduce the notation and formally state the optimization problem in Section 2. In Section 3 we transform the optimization problem into a capacitated circulation problem, allowing us a polynomial-time algorithm, and provide a speed-up in Section 4. In Section 5 we discuss alternative edge penalties. We demonstrate how to extract a canonical optimal solution in Section 6. We discuss the related work in Section 8 and present experimental evaluation in Section 9. Finally, we conclude the paper with remarks in Section 10.

2 Preliminaries and problem definition

We begin with establishing preliminary notation and then defining the main problem.

The main input to our problem is a *weighted directed graph* which we will denote by $G = (V, E, w)$, where w is a function mapping an edge to a real positive number. If w is not provided, we assume that each edge has a weight of 1. We will often denote $n = |V|$ and $m = |E|$.

As mentioned in the introduction, our goal is to partition vertices V. We express this partition with a *rank assignment* r, a function mapping a vertex to an integer. To obtain the groups from the rank assignment we simply group the vertices having the same rank.

Given a graph $G = (V, E)$ and a rank assignment r, we will say that an edge (u, v) is *forward* if $r(u) < r(v)$, otherwise edge is *backward*, even if $r(u) = r(v)$. Ideally, rank assignment r should not have backward edges, that is, for any $(u, v) \in E$ we should have $r(u) < r(v)$. However, this is only possible when G is a DAG. For a more general case, we assume that we are given a penalty function p, mapping an integer to a real number. The penalty for a single edge (u, v) is then equal to $p(d)$, where $d = r(u) - r(v)$. If $p(d) = 0$, whenever $d < 0$, then the forward edges will receive 0 penalty.

We highlight two penalty functions. The first one assigns a constant penalty to each backward edge,

$$p_c(d) = \begin{cases} 1 & \text{if } d \geq 0 \\ 0 & \text{otherwise} \end{cases} \quad .$$

The second penalty function assigns a linear penalty to each backward edge,

$$p_l(d) = \max(0, d+1) \quad .$$

For example, an edge (u, v) with $r(u) = r(v)$ is penalized by $p_l(r(u) - r(v)) = 1$, the penalty is equal to 2 if $r(u) = r(v) + 1$, and so on.

Given a penalty function and a rank assignment we can now define the the score for the ranking to be the sum of the weighted penalties.

Definition 1 Assume a weighted directed graph $G = (V, E, w)$ and a rank assignment r. Assume also a cost function p mapping an integer to a real number. We define a score for a rank assignment to be

$$q(G, r, p) = \sum_{e=(u,v)\in E} w(e) p(r(u) - r(v)) \quad .$$

We will refer the score $q(G, r, p_l)$ as *agony*.

Example 1 Consider the left ranking r_1 of a graph G given in Figure 2. This ranking has 5 backward edges, consequently, the penalty is $q(G, r_1, p_c) = 5$. On the other hand, there are 2 edges, (i, a) and (e, g), having the agony of 1. Moreover, 2 edges has agony of 2 and (d, b) has agony of 3. Hence, agony is equal to

$$q(G, r_1, p_l) = 2 \times 1 + 2 \times 2 + 1 \times 3 = 10 \quad .$$

The agony for the right ranking r_2 is $q(G, r_2, p_l) = 7$. Consequently, r_2 yields a better ranking in terms of agony.

Fig. 2 Toy graphs. Backward edges are represented by dotted lines, while the forward edges are represented by solid lines. Ranks are represented by dashed grey horizontal lines.

We can now state our main optimization problem.

Problem 1 Given a graph $G = (V, E, w)$, a cost function p, and an integer k, find a rank assignment r minimizing $q(r, G)$ such that $0 \leq r(v) \leq k - 1$ for every $v \in V$. We will denote the optimal score by $q(G, k, p)$.

We should point out that we have an additional constraint by demanding that the rank assignment may have only k distinct values, that is, we want to find at most k groups. Note that if we assume that the penalty function is non-decreasing and does not penalize the forward edges, then setting $k = |V|$ is equivalent of ignoring the constraint. This is the case since there are at most $|V|$ groups and we can safely assume that these groups obtain consecutive ranks. However, an optimal solution may have less than k groups, for example, if G has no edges and we use p_l (or p_c), then a rank assigning each vertex to 0 yields the optimal score of 0. We should also point out that if using p_c, there is always an optimal solution where each vertex has its own rank. This is not the case for agony.

It is easy to see that minimizing $q(G, p_c)$ is equivalent to finding a directed acyclic subgraph with as many edges as possible. This is known as FEEDBACK ARC SET (FAS) problem, which is **NP**-complete [2].

On the other hand, if we assume that G has unit weights, and set $k = |V|$, then minimizing agony has a polynomial-time $\mathcal{O}(m^2)$ algorithm [7, 19].

3 Computing agony

In this section we present a technique for minimizing agony, that is, solving Problem 1 using p_l as a penalty. In order to do this we show that this problem is in fact a dual problem of the known graph problem, closely related to the minimum cost max-flow problem.

3.1 Agony with shifts

We begin with an extension to our optimization problem.

Problem 2 (Agony-with-shifts) Given a graph $G = (V, E, w, s)$, where w maps an edge to a, possibly infinite, non-negative value, and s maps an edge to a possibly negative integer, find a rank assignment r minimizing

$$\sum_{e=(u,v)\in E} w(e) \times \max(r(u) - r(v) + s(e), 0) \quad .$$

We denote the optimal sum with $q(G)$.

In order to transform the problem of minimizing agony to AGONY-WITH-SHIFTS, assume a graph $G = (V, E, w)$ and an integer k. We define a graph $H = (W, F, w, s)$ as follows. The vertex set W consists of 2 groups: *(i)* $|V|$ vertices, each vertex corresponding to a vertex in G *(ii)* 2 additional vertices α and ω. For each edge $e = (u, v) \in E$, we add an edge $f = (u, v)$ to F. We set $w(f) = w(e)$ and $s(f) = 1$. We add edges (v, ω) and (α, v) for every $v \in V$ with $s(v, \omega) = s(\alpha, v) = 0$ and $w(v, \omega) = w(\alpha, v) = \infty$. Finally we add (ω, α) with $s(\omega, \alpha) = 1 - k$ and $w(\omega, \alpha) = \infty$. We will denote this graph by $H(G, k) = H$.

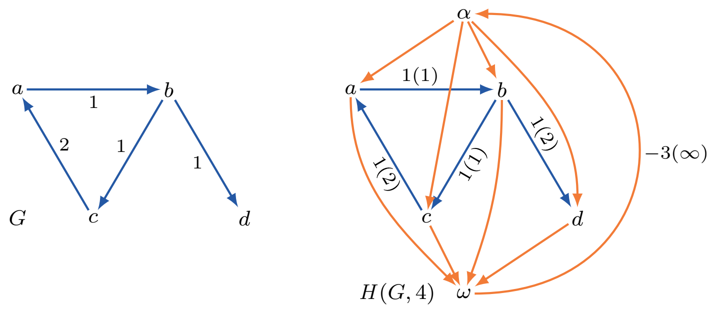

Fig. 3 Toy graph G and the related circulation graph $H(G, 4)$. Edge costs and shifts for (α, v) and (v, ω) are omitted to avoid clutter.

Example 2 Consider $G = (V, E)$, a graph with 4 vertices and 4 edges, given in Figure 3. Set cardinality constraint $k = 4$. In order to construct $H(G, k)$ we add two additional vertices α and ω to enforce the cardinality constraint k. We set edge costs to -1 and edges capacities to be the weights of the input graph. We connect α and ω with a, b, c, and d, and finally we connect ω to α. The resulting graph is given in Figure 3.

3.2 Agony is a dual problem of Circulation

Minimizing agony is closely related to a circulation problem, where the goal is to find a circulation with a minimal cost satisfying certain balance equations.

Problem 3 (Capacitated circulation) Given a graph $G = (V, E, c, s)$, where c maps an edge to a, possibly infinite, non-negative value, and s maps an edge to a possibly negative integer, find a flow f such that $0 \leq f(e) \leq c(e)$ for every $e \in E$ and

$$\sum_{e=(v,u)\in E} f(e) = \sum_{e=(u,v)\in E} f(e), \quad \text{for every } v \in V$$

maximizing

$$\sum_{e \in E} s(e) f(e) \quad .$$

We denote the above sum as $circ(G)$.

This problem is known as capacitated circulation problem, and can be solved in $\mathcal{O}(m \log n(m + n \log n))$ time with an algorithm presented by Orlin [14]. We should stress that we allow s to be negative. We also allow capacities for certain edges to be infinite, which simply means that $f(e) \leq c(e)$ is not enforced, if $c(e) = \infty$.

The following proposition shows the connection between the agony and the capacitated circulation problem.

Proposition 1 *Assume a weighted directed graph with shifts, $G = (V, E, w, s)$. Then $q(G) = circ(G)$.*

Proof Let $G = (V, E, w, s)$. To prove this result we will show that computing $circ(G)$ is a linear program, whose dual corresponds to optimizing Agony-with-shifts. In order to do this, we first express a general Capacitated circulation problem as a linear program,

$$\begin{aligned} &\text{maximize} \sum_{(u,v) \in E} s(u,v) f(u,v) && \text{such that} \\ &\sum_{(v,u) \in V} f(v,u) = \sum_{(u,v) \in V} f(u,v), && \text{for every } v \in V, \\ &w(u,v) \geq f(u,v) \geq 0, && \text{for every } (u,v) \in E \quad . \end{aligned}$$

This program has the following dual program,

$$\begin{aligned} &\text{minimize} \sum_{(u,v) \in E} \eta(u,v) w(u,v) \\ &\text{such that for every } (u,v) \in E \\ &\pi(v) - \pi(u) + \eta(u,v) \geq s(u,v), && \text{if } w(u,v) < \infty, \\ &\pi(v) - \pi(u) \geq s(u,v), && \text{if } w(u,v) = \infty, \\ &\eta(u,v) \geq 0, && (1) \end{aligned}$$

which is optimized over the variables π and η.

If π are integers, then they correspond to the ranking r. Moreover, $\eta(u, v) = \max(\pi(u) - \pi(v) + s(u, v), 0)$. So that, $w(u, v)\eta(u, v)$ corresponds to the penalty term in the sum of Agony-with-shifts, and the objective function of the dual program corresponds exactly to the objective of Agony-with-shifts.

To complete the proof we need to show that there is an optimal integer-valued dual solution π and η. This result follows from the fact that the constraints of the dual form an arc-vertex incidence matrix, which is known to be totally unimodular [15, Corollary of Theorem 13.3], Since $s(u, v)$ are integers, Theorem 13.2 in [15] implies that there is an optimal solution with integer-valued π, completing the proof. □

3.3 Algorithm for minimizing agony

Proposition 1 states that we can compute agony but it does not provide direct means to discover an optimal rank assignment. However, a closer look at the proof reveals that minimizing agony is a dual problem of Capacitated circulation. That is, if we were to solve the dual optimization problem given in Equation 1, then we can extract the optimal ranking from the dual parameters π by setting $r(v) = \pi(v) - \pi(\alpha)$ for $v \in V$, where α is the special vertex added during the construction of H.

Luckily, the algorithms for solving Capacitated circulation by Edmonds and Karp [3] or by Orlin [14] in fact solve Equation 1 and are guaranteed to have integer-valued solution as long as the capacities $s(u, v)$ are integers, which is the case for us.

If we are not enforcing the cardinality constraint, that is, we are solving $q(G, k)$ with $k = |V|$, we can obtain a significant speed-up by decomposing G to strongly connected components, and solve ranking for individual components.

Proposition 2 *Assume a graph G, and set $k = |V|$. Let $\{C_i\}$ be the strongly connected components of G, ordered in a topological order. Let r_i be the ranking minimizing $q(G(C_i), |C_i|)$. Let $b_i = \sum_{j=1}^{i-1} |C_j|$. Then the ranking $r(v) = r_i(v) + b_i$, where C_i is the component containing v, yields the optimal score $q(G, k)$.*

Proof Note that $\max r(v) \leq k$, hence r is a valid ranking. Let r' be the ranking minimizing $q(G, k)$. Let r'_i be the projection of the ranking to C_i. Then

$$q(G, r') \geq \sum_{i=1} q(G(C_i), r'_i) \geq \sum_{i=1} q(G(C_i), r_i) = q(G, r)\,,$$

where the last equality holds because any cross-edge between the components is a forward edge. □

4 Speeding up the circulation solver

In this section we propose a modification to the circulation solver. This modification provides us with a modest improvement in computational complexity, and—according to our experimental evaluation—significant improvement in running time in practice.

Before explaining the modification, we first need to revisit the original Orlin's algorithm. We refer the reader to [14] for a complete expose.

The solver actually solves a slightly different problem, namely, an uncapacitated circulation.

Problem 4 (Circulation) Given a directed graph $F = (W, A, t, b)$ with weights on edges and biases on vertices, find a flow f such that $0 \leq f(e)$ for every $e \in A$ and

$$\sum_{(v,u) \in A} f(v, u) - \sum_{(u,v) \in A} f(u, v) = b(v), \quad \text{for every } v \in W \qquad (2)$$

minimizing

$$\sum_{(u,v) \in A} t(u, v) f(u, v) \quad .$$

To map our problem to CIRCULATION, we us the trick described by Orlin [14]: we replace each capacitated edge $e = (v, w)$ with a vertex u and two edges (v, u) and (w, u). We set $b(u) = -c(e)$, and add $c(v, w)$ to $b(w)$. The costs are set to $t(v, u) = \max(-s(e), 0)$ and $t(w, u) = \max(s(e), 0)$. For each uncapacitated edge (v, w), we connect v to w with $t(v, w) = -s(v, w)$.[1]

From now on, we will write $H = H(G, k)$, and $F = (W, A, s, b)$ to be the graph modified as above. We split W to W_1 and W_2: W_1 are the original vertices in H, while W_2 are the vertices rising from the capacitated edges.

We also write n and m to be the number of vertices and edges in H, respectively, and n' and m' to be the number of vertices and edges in F, respectively. Note that $n', m' \in \mathcal{O}(m)$.

Consider the dual of uncapacitated circulation.

Problem 5 (dual to Circulation) Given a directed graph $F = (W, A, s, b)$ with weights on edges and biases on vertices, find dual variables π on vertices maximizing

$$\sum_{(u,v) \in A} b(v)\pi(v)$$

such that

$$t(e) + \pi(w) - \pi(v) \geq 0, \quad \text{for every } e = (v, w) \in A \quad . \tag{3}$$

The standard linear programming theory states that f and π satisfying Eq. 2–3 are optimal solutions to their respective problems if and only if the slackness conditions hold,

$$(t(e) + \pi(w) - \pi(v))f(e) = 0, \quad \text{for every} \quad e = (v, w) \in A \quad . \tag{4}$$

The main idea behind Orlin's algorithm is to maintain a flow f and a dual π satisfying Eqs. 3–4, and then iteratively enforce Eq. 2. More specifically, we first define an *excess* of a vertex to be

$$e(v) = b(v) + \sum_{(w,v) \in A} f(w, v) - \sum_{(v,w) \in A} f(v, w) \quad .$$

Our goal is to force $e(v) = 0$ for every v. This is done in gradually in multiple iterations. Assume that we are given Δ, a granularity which we will use to modify the flow. The following steps are taken: *(i)* We first construct a residual graph R which consists of all the original edges, and reversed edges for all edges with positive flow. *(ii)* We then select a source s with $e(s) \geq \alpha\Delta^2$, and construct a shortest path tree T in R, weighted by $t(e) + \pi(w) - \pi(v)$, for $e = (v, w)$. *(iii)* The dual variables are updated to $\pi(v) - d(v)$, where d is the shortest distance from s to v. *(iv)* We select a sink r with $e(r) \leq -\alpha\Delta$, and augment the flow along the path in T from r to s. This is repeated until there are no longer viable options for s or r. After that we half Δ, and repeat.

[1] The reason for the minus sign is that we expressed CIRCULATION as a minimization problem and CAPACITATED CIRCULATION as a maximization problem.

[2] Here α is a fixed parameter $1/2 < \alpha < 1$, we use $\alpha = 3/4$.

To guarantee polynomial convergence, we also must contract edges for which $f(e) \geq 3n'\Delta$, where n' is the number of vertices in the (original) input graph. Assume that we contract (v, w) into a new vertex u. We set $\pi(u) = \pi(v)$, $b(u) = b(v)+b(w)$. We delete the edge (v, w), and edges adjacent to v and w are migrated to u; the cost of an edge $t(w, x)$ must be changed to $t(w, x) + t(v, w)$, and similarly the cost of an edge $t(x, w)$ must be changed to $t(x, w) - t(v, w)$. A high-level pseudo-code is given in Algorithm 1.

Algorithm 1: Orlin's algorithm for solving CIRCULATION.

$\Delta \leftarrow \min(\max e(v), \max -e(v))$;
while there is excess **do**
  contract any edges with $f(e) \geq 3n\Delta$;
  while $\max e(v) \geq \alpha\Delta$ **and** $\min e(v) \leq -\alpha\Delta$ **do**
    $s \leftarrow$ a vertex with $e(s) \geq \alpha\Delta$;
    $r \leftarrow$ a vertex with $e(r) \leq -\alpha\Delta$;
    $T \leftarrow$ shortest path tree from s in residual graph, weighted by $t(e) + \pi(w) - \pi(v)$;
    update π using T;
    $P \leftarrow$ path in T from r to s;
    augment flow by Δ along P;
  $\Delta \leftarrow \Delta/2$;

The bottleneck of this algorithm is computing the shortest path tree. This is the step that we will modify. In order to do this we first point out that Orlin's algorithm relies on two things that inner loop should do: *(i)* Eqs. 3–4 must be maintained, and *(ii)* path augmentations are of granularity Δ, after the augmentations there should not be a viable vertex for a source or a viable vertex for a sink. As long as these two conditions are met, the correctness proof given by Orlin [14] holds.

Our first modification is instead of selecting one source s, we select *all* possible sources, $S \leftarrow \{v \in V \mid e(v) \geq \alpha\Delta\}$, and compute the shortest path tree using S as roots. Once this tree is computed, we subtract the shortest distance from π, select a sink t, and augment flow along the path from t to some root $s \in S$.

The following lemma guarantees that Eqs. 3–4 are maintained when f and π are modified.

Lemma 1 *Let f and π be flow and dual variables satisfying the slackness conditions given in Eq. 4. Let S be a set of vertices. Define $d(v)$ be the shortest distance from S to v in the residual graph with weighted edges $t(e)+\pi(w)-\pi(v)$. Let $\pi' = \pi - d$. Then π' satisfy Eq. 3, and f and π' respect the slackness conditions in Eq. 4. Moreover, $t(e) + \pi'(w) - \pi'(v) = 0$ for every edge in the shortest path tree.*

Note that since we modify f only along the edges of the shortest path tree, this lemma guarantees that Eq. 4 is also maintained when we augment f. The proof of this lemma is essentially the same as the single-source version given by Orlin [14].

Proof Let $e = (v, w) \in E$. Then e is also in residual graph, and $d(w) \leq d(v) + t(e) + \pi(w) - \pi(v)$. This implies

$$t(e) + \pi'(w) - \pi'(v) = t(e) + \pi(w) - d(w) - \pi(v) + d(v) \geq 0,$$

proving the first claim. If e is in the shortest path tree, then $d(w) = d(v) + t(e) + \pi(w) - \pi(v)$, which implies the third claim, $t(e) + \pi(w) - \pi(v) = 0$.

To prove the second claim, if $f(e) > 0$, then $t(e) + \pi(w) - \pi(v) = 0$. Since $(w, v) = e'$ is also in residual graph, we must have $d(v) = d(w)$. Thus,

$$t(e) + \pi'(w) - \pi'(v) = t(e) + \pi(w) - d(w) - \pi(v) + d(v) = 0 \quad .$$

This completes the proof. □

Once we augment f, we need to update the shortest path tree. There are three possible updates: *(i)* adding a flow may result in a new backward edge in the residual graph, *(ii)* reducing a flow may result in a removing a backward edge in the residual graph, and *(iii)* deleting a source from S requires that the tree is updated.

In order to update the tree we will use an algorithm by Ramalingam and Reps [16] to which we will refer as RR. The pseudo-code for the modified solver is given in Algorithm 2.

Algorithm 2: A modified algorithm for CIRCULATION.

$\Delta \leftarrow \min(\max e(v), \max -e(v))$;
while there is excess **do**
  contract any edges with $f(e) \geq 3n'\Delta$;
  $S \leftarrow \{v \in V \mid e(v) \geq \alpha\Delta\}$;
  $Q \leftarrow \{v \in V \mid e(v) \leq -\alpha\Delta\}$;
  $T \leftarrow$ shortest path tree from S in residual graph, weighted by $t(e) + \pi(w) - \pi(v)$;
  update π using T, see Lemma 1;
  while $S \neq \emptyset$ **and** $Q \neq \emptyset$ **do**
    select $r \in Q$;
    $P \leftarrow$ path in T from r to some $s \in S$;
    augment flow by Δ along P;
    update residual graph;
    if $e(s) < \alpha\Delta$ **then** delete s from S;
    if $e(r) > -\alpha\Delta$ **then** delete r from Q;
    update T using [16];
    update π using T, see Lemma 1;
  $\Delta \leftarrow \Delta/2$;

Before going further, we need to address one technical issue. RR requires that edge weights are positive, whereas we can have weights equal to 0. We solve this issue by adding $\epsilon = 1/n'$ to each edge. Since the original weights are integers and a single path may have $n' - 1$ edges, at most, the obtained shortest path tree is a valid shortest path tree for the original weights. We use ϵ only for computing and updating the tree; we will not use it when we update the dual variables.

In order to update the tree, first note that the deleting the source s from S is essentially the same as deleting an edge: computing the tree using S as roots is equivalent to having one auxiliary root, say σ, with only edges connecting σ to S. Removing s from S is then equivalent to deleting an edge (σ, s).

The update is done by first adding the new edges, and then deleting the necessary edges. We first note that the edge additions do not require any updates by RR. This is because the internal structure of RR is a subgraph of *all* edges that can be used to form the shortest path. Any edge that is added will be from a child to a parent, implying that it cannot participate in a shortest path.[3]

Proposition 3 *Algorithm 2 runs in* $\mathcal{O}(m(\min(kn, m) + n \log n))$ *time, assuming* G *is not weighted.*

To prove the result we need the following lemma.

Lemma 2 *At any point of the algorithm, the dual variables* π *satisfy* $\pi(v) - \pi(u) \leq k$ *for any* u, v.

Proof Let us first prove that this result holds if we have done no edge contractions. Let α and ω be the vertices in H, enforcing the cardinality constraint. Assume that v and w are both in W_1. Then Eq. 3 guarantees that $\pi(u) \geq \pi(\alpha)$ and $\pi(\omega) \geq \pi(v)$ implying

$$\pi(v) - \pi(u) \leq \pi(\omega) - \pi(\alpha) \leq t(\omega, \alpha) = k - 1 \quad .$$

Now assume that v (and/or u) is in W_2. Then the shortest path tree connects it to a vertex $x \in W_1$, and either $\pi(u) = \pi(x)$ or $\pi(u) = \pi(x) - 1$. This leads to that the difference $\pi(v) - \pi(u)$ can be at most k.

To see why the lemma holds despite edge contractions, note that we can always unroll the contractions to the original graph, and obtain π' that satisfies Eq. 3. Moreover, if x is a new vertex resulted from a contraction, after unrolling, there is a vertex $u \in W$ such that $\pi(x) = \pi'(u)$. This is because when we create x, we initialize $\pi(x)$ to be dual of one contracted vertices. Consequently, the general case reduces to the first case. □

Proof (of Proposition 3) Since $b(v) = -1$, for $v \in W_2$ and $b(v) \geq 0$ for $v \in W_1$, we have $\Delta = 1$, and after a single iteration $e(v) = 0$, for $v \in W$. So, we need only one outer iteration. Consequently, we only need to show that the inner loop needs $\mathcal{O}(m(\min(kn, m) + n \log n))$ time.

Let us write O to be the vertices who are either in W_1, or, due to a contraction, contain a vertex in W_1. Let P_i be the path selected during the ith iteration. Let us write n'_i to be the number of vertices whose distance is changed[4]

[3] The edge can participate later when we delete edges.

[4] taking into account the ϵ trick

during the ith iteration of the inner loop; let m_i be the number of edges adjacent to these vertices. Finally, let us write n_i to be the number of vertices in O whose distance is changed.

Ramalingam and Reps [16] showed that updating a tree during the ith iteration requires $\mathcal{O}(m_i + n'_i \log n'_i)$ time. More specifically, the update algorithm first detects the affected vertices in $\mathcal{O}(m_i)$ time, and then computes the new distances using a standard Dijkstra algorithm with a binomial heap in $\mathcal{O}(m_i + n'_i \log n'_i)$ time.

We can optimize this to $\mathcal{O}(m_i + n'_i + n_i \log n_i)$ by performing a trick suggested by Orlin: Let X be the vertices counted towards n_i and let Y be the remaining vertices counted towards n'_i. A vertex in Y is either in a path between two vertices in X, or is a leaf. In the latter case it may be only connected to only two (known) vertices. We can first compute the distances for X by frogleaping the vertices in Y in $\mathcal{O}(m_i + n_i \log n_i)$ time. This gives us the updated distances for X and for vertices in Y that are part of some path. Then we can proceed to update the leaf vertices in Y in $\mathcal{O}(m_i + n_i)$ time.

The total running time of an inner loop is then

$$\mathcal{O}\left(\sum_i |P_i| + m_i + n'_i + n_i \log n_i\right) \subseteq \mathcal{O}\left(\sum_i |P_i| + m_i + n'_i + n_i \log n\right) \quad .$$

First note that we can have at most $\mathcal{O}(m)$ terms in the sum. This is because we either have $\sum \max(e(i), 0) \leq 2\Delta\alpha n'$ or $\sum \max(-e(i), 0) \leq 2\Delta\alpha n'$ due to to the previous outer loop iteration, and since the contractions can only reduce these terms.

A path from a leaf to a root in T cannot contain two consecutive vertices that are outside O. Hence, the length of a path is at most $\mathcal{O}(n)$.

Let us now bound the number of times a single vertex, say v, needs to be updated. Assume that we have changed the distance but the dual $\pi(v)$ has not changed. In other words, we have increased the ϵ part of the distance. This effectively means that $\pi(v)$ remained constant but we have increased the number of edges from the vertex to the root. Since we can have at most $\mathcal{O}(n)$ long path, we can have at most $\mathcal{O}(n)$ updates without before updating $\pi(v)$. Note that at least one root, say $s \in S$, will not have its dual updated until the very last iteration. Lemma 2 now implies that we can update, that is, decrease, $\pi(v)$ only $\mathcal{O}(k)$ times. Consequently, we can only update v $\mathcal{O}(nk)$ times.

This immediately implies that $\sum_i n'_i \in \mathcal{O}(mnk)$, $\sum_i n_i \in \mathcal{O}\big(n^2 k\big)$, and $\sum_i m_i \in \mathcal{O}(mnk)$. Since path lengths are $\mathcal{O}(n)$, we also have $\sum_i |P_i| \in \mathcal{O}(mn)$. This gives us a total running time of

$$\mathcal{O}\big(mn + mnk + n^2 k \log n\big) = \mathcal{O}(nk(m + n \log n)) \quad .$$

We obtain the final bound by alternatively bounding RR with $\mathcal{O}(m + n \log n)$, and observing that you need only $\mathcal{O}(m)$ updates. □

The theoretical improvement is modest: we essentially replaced m with $\min(nk, m)$. However, in practice this is a very pessimistic bound, and we will see that this approach provides a significant speed-up. Moreover, this result suggests—backed up by our experiments—that the problem is easier for smaller values of k. This is opposite to the behavior of the original solver presented in [20]. Note also, that we assumed that G has no weights. If we have integral weights of at most ℓ, then the running time increases by $\mathcal{O}(\log \ell)$ time.

5 Alternative penalty functions

We have shown that we can find ranking minimizing edge penalties p_l in polynomial time. In this section we consider alternative penalties. More specifically, we consider convex penalties which are solvable in polynomial time, and show that concave penalties are **NP**-hard.

5.1 Convex penalty function

We say that the penalty function is *convex* if $p(x) \leq (p(x-1) + p(x+1))/2$ for every $x \in \mathbb{Z}$.

Let us consider a penalty function that can be written as

$$p_s(x) = \sum_{i=1}^{\ell} \max(0,\, \alpha_i(x - \beta_i)),$$

where $\alpha_i > 0$ and $\beta_i \in \mathbb{Z}$ for $1 \leq i \leq \ell$. This penalty function is convex. On the other hand, if we are given a convex penalty function p such that $p(x) = 0$ for $x < 0$, then we can safely assume that an optimal rank assignment will have values between 0 and $|V| - 1$. We can define a penalty function p_s with $\ell \leq |V|$ terms such that $p_s(x) = p(x)$ for $x < |V|$. Consequently, finding an optimal rank assignment using p_s will also yield an optimal rank assignment with respect to p.

Note that p_l is a special case of p_s. This hints that we can solve $q(G, k, p_s)$ with a technique similar to the one given in Section 3. In fact, we can map this problem to Agony-with-shifts. In order to do this, assume a graph $G = (V, E, w)$ and an integer k. Set $n = |V|$ and $m = |E|$. We define a graph $H = (W, F, w, s)$ as follows. The vertex set W consists of 2 groups: *(i)* n vertices, each vertex corresponding to a vertex in G *(ii)* 2 additional vertices α and ω. For each edge $e = (v, w) \in E$, we add ℓ edges $f_i = (u, v)$ to F. We set $s(f_i) = -\beta_i$ and $w(f_i) = \alpha_i w(e)$. We add edges to α and ω to enforce the cardinality constraint, as we did in Section 3.1. We denote this graph by $H(G, k, p_s) = H$.

Example 3 Consider a graph G given in Figure 4 and a penalty function $p_s(d) = \max(0, d + 1) + 2\max(0, d - 3)$. The graph $H = H(G, 3, p_s)$ has 5 vertices, the original vertices and the two additional vertices. Each edge in G results in two edges in H. This gives us 6 edges plus the 7 edges adjacent to α or ω. The graph H without α and ω is given in Figure 4.

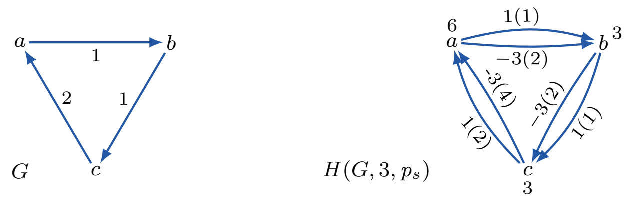

Fig. 4 Toy graph G and the related circulation graph $H(G, 3, p_s)$. To avoid clutter the vertices α and ω and the adjacent edges are omitted.

Finally, let us address the computational complexity of the problem. The circulation graph $H(G, k, p_s)$ will have $n + 2$ vertices and $\ell m + n$ edges. If the penalty function p is convex, then we need at most $\ell = n$ functions to represent p between the range of $[0, n-1]$. Moreover, if we enforce the cardinality constraint k, we need only $\ell = k$ components. Consequently, we will have at most $dm + n$, edges where $d = \min(k, \ell, n)$ for p_s, and $d = \min(k, n)$ for a convex penalty p. This gives us computational time of $\mathcal{O}(dm \log n(dm + n \log n))$.

5.2 Concave penalty function

We have shown that we can solve Problem 1 for any convex penalty. Let us consider concave penalties, that is penalties for which $p(x) \geq (p(x-1) + p(x+1))/2$. There is a stark difference compared to the convex penalties as the minimization problem becomes computationally intractable.

Proposition 4 *Assume a monotonic penalty function $p : \mathbb{Z} \to \mathbb{R}$ such that $p(x) = 0$ for $x < 0$, $p(2) > p(1)$, and there is an integer t such that*

$$p(t) > \frac{p(t-1) + p(t+1)}{2} \tag{5}$$

and

$$\frac{p(s)}{s+1} \geq \frac{p(y)}{y+1},$$

*for every $0 \leq s \leq y$ and $y \in [t-1, t, t+1]$. Then, determining whether $q(G, k, p) \leq \sigma$ for a given graph G, integer k, and threshold σ is an **NP**-hard problem.*

We provide the proof in Appendix.

While the conditions in Proposition 4 seem overly complicated, they are quite easy to satisfy. Assume that we are given a penalty function that is concave in $[-1, \infty]$, and $p(-1) = 0$. Then due to concavity we have

$$\frac{p(x)}{x+1} \geq \frac{p(x+1)}{x+2}, \quad \text{for} \quad x \geq 0 \quad .$$

This leads to the following corollary.

Corollary 1 *Assume a monotonic penalty function* $p : \mathbb{Z} \rightarrow \mathbb{R}$ *such that* $p(x) = 0$ *for* $x < 0$, $p(2) > p(1)$, *and* p *is concave and non-linear in* $[-1, \ell]$ *for some* $\ell \geq 1$. *Then, determining whether* $q(G, k, p) \leq \sigma$ *for a given graph* G, *integer* k, *and threshold* σ *is* ***NP****-hard problem.*

Note that we require p to be non-linear. This is needed so that the proper inequality in Equation 5 is satisfied. This condition is needed since p_l satisfies every other requirement. Corollary 1 covers many penalty functions such as $p(x) = \sqrt{x+1}$ or $p(x) = \log(x+2)$, for $x \geq 0$. Note that the function needs to be convex only in $[-1, \ell]$ for some $\ell \geq 1$. At extreme, $\ell = 1$ in which case $t = 0$ satisfies the conditions in Proposition 4.

6 Selecting canonical solution

A rank assignment minimizing agony may not be unique. In fact, consider a graph G with no edges, then any ranking will have the optimal score of 0. Moreover, if the input graph G is a DAG, then any topological sorting of vertices will yield the optimal score of 0.

In this section we introduce a technique to select a unique optimal solution. The idea here is to make the ranks as small as possible without compromising the optimality of the solution. More specifically, let us define the following relationship between to rankings.

Definition 2 Given two rank assignments r and r', we write $r \preceq r'$ if $r(x) \leq r'(x)$ for every x.

The following proposition states that there exists exactly one ranking with the optimal score that is minimal with respect to the $\preceq$ relation. We will refer to this ranking as *canonical ranking*.

Proposition 5 *Given a graph* G *and an integer* k, *there exists a unique optimal rank assignment* r *such that* $r \preceq r'$ *for every optimal rank assignment* r'.

The proof of this proposition is given in Appendix.

Canonical ranking has many nice properties. The canonical solution for a graph without edges assigns rank 0 to all vertices. More generally, if $G = (V, E)$ is a DAG, then the source vertices S of G will receive a rank of 0, the source vertices of $G(V \setminus S)$ will receive a rank of 1, and so on. For general graphs we have the following proposition.

Proposition 6 *Let* r *be the canonical ranking. Then* r *has the least distinct rank values among all optimal solutions.*

In other words, the partition of V corresponding to the canonical ranking has the smallest number of groups.

Our next step is to provide an algorithm for discovering canonical ranking. In order to do so we assume that we use Orlin's algorithm and obtain the flow

f and the dual π, described in Problem 4 and 5. We construct the residual graph R, as described in Section 4, edges weighted by $t(e) + \pi(w) - \pi(v)$. We then compute, $d(v)$ which is the shortest path in R from α to v. Finally, we set $r^*(v) = r(v) - d(v)$.

Once, we have computed the residual graph, we simply compute the shortest path distance from q and subtract the distance from the optimal ranking, see Algorithm 3.

Algorithm 3: CANON(G), computes canonical optimal solution

```
f, π ← optimal flow and dual of CIRCULATION;
R ← residual graph;
d(v) ← shortest weighted distance from α to v in R;
foreach v ∈ V do r*(v) ← r(v) − d(v);
return r*;
```

Proposition 7 *Algorithm* CANON *returns canonical solution with optimal score.*

We give the proof of this proposition in Appendix.

Proposition states that to compute the canonical ranking it is enough to form the residual graph, compute the shortest edge distances $d(v)$ from the vertex q, and subtract them from the input ranking. The computational complexity of these steps is $\mathcal{O}(m + n \log n)$. Moveover, this proposition holds for a more general convex penalty function, described in Section 5.1.

7 A fast divide-and-conquer heuristic

In this section we propose a simple and fast divide-and-conquer approach. The main idea is as follows: We begin with the full set of vertices and we split them into two halves: the left half will have smaller ranks than the right half. We then continue splitting the smaller sets recursively, and obtain a tree. We show that this can be done in $\mathcal{O}(m \log n)$ time. If we are given a cardinality constraint k, then we prune the tree using dynamic program that runs in $\mathcal{O}(k^2 n)$ time. We also propose a variant, where we perform SCC decomposition, and perform then divide-and-conquer on individual components. To enforce the cardinality constraint in this case, we need additional $\mathcal{O}(km \log n + k^2 n)$ time.

7.1 Constructing a tree by splitting vertices

As mentioned above, our goal is to construct a tree T. This tree is binary and ordered, that is, each non-leaf vertex has a left child and a right child.

Each leaf α[5] in this tree T is associated with a set of vertices that we denote by V_α. Every vertex of the input graph should belong to some leaf, and

[5] we will systematically denote the vertices in T with Greek letters

no two leaves share a vertex. If α is a non-leaf, then we define V_α to be the union of vertices related to each descendant leaf of α. We also define E_α to be the edges in E that have both endpoints in V_α.

Since the tree is ordered, we can sort the leaves, left first. Using this order, we define a rank $r(v)$ to be the rank of the leaf in which v is included. We define $q(T) = q(r)$.

Our goal is to construct T with good $q(T)$. We do this by splitting V_α of a leaf α to two leaves such that the agony is minimized.

Luckily, we can find the optimal split efficiently. Let us first express the gain in agony due to a split. In order to do so, assume a tree T, and let α be a leaf. Let X be the vertices in leaves that are left to α, and let Z be the vertices in leaves that are right to α.

We define $b(\alpha)$ to be the total weight of the edges from Z to X,

$$b(\alpha) = \sum_{\substack{(z,x) \in E \\ x \in X, z \in Z}} w(z, x) \quad .$$

Let y be a vertex in V_α. We define

$$ib(y; \alpha) = \sum_{\substack{(z,y) \in E \\ z \in Z}} w(z, y) \quad \text{and} \quad ob(y; \alpha) = \sum_{\substack{(y,x) \in E \\ x \in X}} w(y, x)$$

to be the total weight of the backward edges adjacent to y and Z or X. We also define the total weights

$$ib(\alpha) = \sum_{y \in V_\alpha} ib(y; \alpha) \quad \text{and} \quad ob(\alpha) = \sum_{y \in V_\alpha} ob(y; \alpha) \quad .$$

Let

$$\mathit{flux}(y; \alpha) = \sum_{(x,y) \in E_\alpha} w(x, y) - \sum_{(y,x) \in E_\alpha} w(y, x)$$

to be the total weight of incoming edges minus the total weight of the outgoing edges.

Finally, let us define

$$d(y; \alpha) = \mathit{flux}(y; \alpha) + ib(y; \alpha) - ob(y; \alpha) \quad .$$

We can now use these quantities to express how a split changes the score.

Proposition 8 *Let α be a leaf of a tree T. Assume a new tree T', where we have split α to two leaves. Let Y_1 be the vertex set of the new left leaf, and Y_2 the vertex set of the new right leaf. Then the score difference is*

$$q(T') - q(T) = b(\alpha) + ib(\alpha) - \sum_{y \in Y_2} d(y; \alpha)$$

that can be rewritten as

$$q(T') - q(T) = b(\alpha) + ob(\alpha) + \sum_{y \in Y_1} d(y; \alpha) \quad .$$

Proof We will show that

$$q(T') - q(T) = b(\alpha) + \sum_{y \in Y_1} ib(y;\alpha) + \sum_{y \in Y_2} ob(y;\alpha) + \sum_{y \in Y_1} \mathit{flux}(y;\alpha) \quad . \tag{6}$$

Equation 6 can be then rewritten to the forms given in the proposition.

Let Y_0 be the set of all vertices to the left of α. Let Y_3 be the set of all vertices to the right of α. Note that $Y_0 \cup Y_1 \cup Y_2 \cup Y_3 = V$. Write $t(i, j)$ to be the total weight of edges from Y_i to Y_j. Also, write $c(i, j)$ to be the total change in the penalty of edges from Y_i to Y_j due to a split.

Note that $c(0, 1) = c(0, 2) = c(0, 3) = c(1, 3) = c(2, 3) = 0$ since these are forward edges that remain forward. Also, $c(0, 0) = c(1, 1) = c(2, 2) = c(3, 3) = 0$ since the rank difference of these edges has not changed. For the same reason, $c(3, 2) = c(1, 0) = 0$.

Case (*i*): Since a split shifts Y_3 by one rank, $c(3, 0) = t(3, 0)$ and $c(3, 1) = t(3, 1)$. Case (*ii*): Since a split shifts Y_2 by one rank, $c(2, 0) = t(2, 0)$. Case (*iii*): The penalty of an edge from Y_2 to Y_1 increases by $w(e)$. Summing over these edges leads to $c(2, 1) = t(2, 1)$. Case (*iv*): The penalty of an edge from Y_1 to Y_2 decreases by $w(e)$. Summing over these edges leads to $c(1, 2) = -t(1, 2)$.

This leads to

$$q(T') - q(T) = \sum_{i,j} c(i, j) = t(3, 0) + t(3, 1) + t(2, 0) + t(2, 1) - t(1, 2) \quad .$$

First, note that

$$t(3, 0) = b(\alpha)\,, \quad t(3, 1) = \sum_{y \in Y_1} ib(y;\alpha)\,, \quad t(2, 0) = \sum_{y \in Y_2} ob(y;\alpha) \quad .$$

To express $t(2, 1) - t(1, 2)$, we can write

$$\begin{aligned} \sum_{y \in Y_1} \mathit{flux}(y;\alpha) &= \sum_{\substack{(x,y) \in E_\alpha \\ y \in Y_1}} w(x, y) - \sum_{\substack{(y,x) \in E_\alpha \\ y \in Y_1}} w(y, x) \\ &= t(1, 1) + t(2, 1) - t(1, 1) - t(1, 2) = t(2, 1) - t(1, 2) \quad . \end{aligned}$$

This proves Eq. 6, and the proposition. □

Proposition 8 gives us a very simple algorithm for finding an optimal split: A vertex y for which $d(y) \geq 0$ should be in the right child, while the rest vertices should be in the left child. If the gain is negative, then we have improved the score by splitting. However, it is possible to have positive gain, in which case we should not do a split at all. Note that the gain does not change if we do a split in a different leaf. This allows to treat each leaf independently, and not care about the order in which leaves are tested.

The difficulty with this approach is that if we simply recompute the quantities every time from the scratch, we cannot guarantee a fast computation time. This is because if there are many uneven splits, we will enumerate over some edges too many times. In order to make the algorithm provably fast, we argue that we can detect which of the new leaves has *fewer* adjacent edges, and we only enumerate over these edges.

Let us describe the algorithm in more details. We start with the full graph, but as we split the vertices among leaves, we only keep the edges that are intra-leaf; we delete any cross-edges between different leaves. As we delete edges, we also maintain 4 counters for each vertex, $\mathit{flux}(y; \alpha)$, $\mathit{ib}(y; \alpha)$, $\mathit{ob}(y; \alpha)$, and the unweighted degree, $\deg(y)$, where α is the leaf containing y.

For each leaf α, we maintain four sets of vertices,

$$\begin{aligned}
N_\alpha &= \{y \in V_\alpha \mid \deg(y; \alpha) > 0, d(y; \alpha) < 0\}, \\
P_\alpha &= \{y \in V_\alpha \mid \deg(y; \alpha) > 0, d(y; \alpha) \geq 0\}, \\
N^*_\alpha &= \{y \in V_\alpha \mid \deg(y; \alpha) = 0, d(y; \alpha) < 0\}, \\
P^*_\alpha &= \{y \in V_\alpha \mid \deg(y; \alpha) = 0, d(y; \alpha) \geq 0\} \quad .
\end{aligned}$$

The reason why we treat vertices with zero degree differently is so that we can bound $|N_\alpha|$ or $|P_\alpha|$ by the number of adjacent edges.

Note that we maintain these sets only for leaves. To save computational time, when a leaf is split, its sets are reused by the new leaves, and in the process are modified.

In addition, we maintain the following counters

1. the total weights $b(\alpha)$, $\mathit{ib}(\alpha)$, $\mathit{ob}(\alpha)$, and
2. in order to avoid enumerating over N^*_α and P^*_α when computing the gain, we also maintain the counters

$$\mathit{db}_N(\alpha) = \sum_{y \in N_\alpha} \mathit{ib}(y) - \mathit{ob}(y), \quad \mathit{db}_P(\alpha) = \sum_{y \in P_\alpha} \mathit{ib}(y) - \mathit{ob}(y) \quad .$$

We also maintain $\mathit{gain}(\alpha)$ for non-leaves, which is the agony gain of splitting α. We will use this quantity when we prune the tree to enforce the cardinality constraint.

If we decide to split, then we can do this trivially: according to Proposition 8 N_α and N^*_α should be in the left child while P_α and P^*_α should be in the right child. Our task is to compute the gain, and see whether we should split the leaf, and compute the structures for the new leaves.

Given a leaf α, our first step is to determine whether N_α or P_α has fewer edges. More formally, we define $\mathit{adj}(X)$ to be the edges that have *at least one* end point in X. We then need to compute whether $|\mathit{adj}(N_\alpha)| \leq |\mathit{adj}(P_\alpha)|$. This is done by cleverly enumerating over elements of N_α and P_α simultaneously. The pseudo-code is given in Algorithm 4.

Proposition 9 *Let* $m_1 = |\mathit{adj}(N_\alpha)|$ *and* $m_2 = |\mathit{adj}(P_\alpha)|$. *Then* LeftSmaller *returns true if and only if* $m_1 \leq m_2$ *in* $\mathcal{O}(\min(m_1, m_2))$ *time.*

Proof Assume that the algorithm returns true, so $Y_1 = \emptyset$ and $c_1 \leq c_2$. Since $Y_1 = \emptyset$, then $c_1 = m_1$, which leads to $m_1 = c_1 \leq c_2 \leq m_2$. Assume that the algorithm returns false. Then the while loop condition guarantees that $Y_2 = \emptyset$ and $c_1 \geq c_2$. Since $Y_2 = \emptyset$, then $c_2 = m_2$. Either $Y_1 \neq \emptyset$ or $c_1 > c_2$. If latter,

Algorithm 4: LeftSmaller(α), tests whether $|adj(N_\alpha)| \le |adj(P_\alpha)|$.

$Y_1 \leftarrow N_\alpha, Y_2 \leftarrow P_\alpha$;
$c_1 \leftarrow 0$; $c_2 \leftarrow 0$;
until ($Y_1 = \emptyset$ **and** $c_1 \le c_2$) **or** ($Y_2 = \emptyset$ **and** $c_1 \ge c_2$) **do**
  if $c_1 \le c_2$ **then**
    $y \leftarrow$ vertex in Y_1; delete y from Y_1;
    $c_1 \leftarrow c_1 + \deg(y)$;
  else
    $y \leftarrow$ vertex in Y_2; delete y from Y_2;
    $c_2 \leftarrow c_2 + \deg(y)$;
return $Y_1 = \emptyset$ **and** $c_1 \le c_2$;

then $m_2 = c_2 < c_1 \le m_1$. If former, then $m_2 = c_2 \le c_1 < m_1$. This proves the correctness.

To prove the running time, first note, since there are no singletons, each iterations will increase either c_1 or c_2. Assume that $m_1 \le m_2$. If we have not terminated after $2m_1$ iterations, then we must have $m_1 < c_2$. Since $c_1 \le m_1 < c_2$, we will then only increase c_1. This requires at most m_1 iterations (actually, we can show that we only need 1 more iteration). In conclusion, the algorithm runs in $\mathcal{O}(m_1)$ time. The case for $m_1 \ge m_2$ is similar. □

We can now describe our main algorithm, given in Algorithms 5, 6, and 7. Split is given a leaf α. As a first step, Split determines which side has fewer edges using LeftSmaller. After that it computes the gain, and checks whether a split is profitable. If it is, then it calls either ConstructLeft or ConstructRight, depending which one is faster. These two algorithms perform the actual split and updating the structures, and then recurse on the new leaves.

Algorithm 5: Split(α), checks if we can improve by splitting α, and decides which side is more economical to split. Calls either ConstructLeft or ConstructRight to update the structures.

if LeftSmaller(α) **then**
  $g \leftarrow b(\alpha) + ob(\alpha) + db_N(\alpha) + \sum_{y \in N_\alpha} d(y; \alpha)$;
  if $g < 0$ **then** ConstructLeft(α); $gain(\alpha) \leftarrow g$;
else
  $g \leftarrow b(\alpha) + ib(\alpha) - db_P(\alpha) - \sum_{y \in P_\alpha} d(y; \alpha)$;
  if $g < 0$ **then** ConstructRight(α); $gain(\alpha) \leftarrow g$;

Let us next establish the correctness of the algorithm. We only need to show that during the split the necessary structures are maintained properly. We only show it for ConstructLeft, as the argument is exactly the same for ConstructRight.

Proposition 10 ConstructLeft *maintains the counters and the vertex sets.*

Algorithm 6: ConstructLeft(α), performs a single split using N_α. Recurses to Split for further splits.

create a new leaf β with sets $N_\beta = N_\alpha$, $P_\beta = \emptyset$, $N^*_\alpha = P^*_\alpha$, and $P^*_\alpha = \emptyset$;
$b(\beta) \leftarrow b(\alpha) + ob(\alpha)$;
create a new leaf γ with sets $N_\gamma = \emptyset$, $P_\gamma = P_\gamma$, $N^*_\gamma = \emptyset$, and $P^*_\gamma = P^*_\gamma$;
$b(\gamma) \leftarrow b(\alpha)$;
foreach $x \in N_\alpha$ **do**
    $b(\gamma) \leftarrow b(\gamma) + ib(x)$;
    $b(\beta) \leftarrow b(\beta) - ob(x)$;
    delete edges (x, z) or (z, x) for any $z \in P_\alpha$, and update *flux*, deg, *ib*, *ob* ;
check the affected vertices and update P_β, N_β, P^*_β, N^*_β, $db_N(\beta)$ and $db_P(\beta)$;
check the affected vertices and update P_γ, N_γ, P^*_γ, N^*_γ, $db_N(\gamma)$ and $db_P(\gamma)$;
Split(β); Split(γ);

Algorithm 7: ConstructRight(α), performs a single split using P_α. Recurses to Split for further splits.

create a new leaf β with sets $N_\beta = N_\alpha$, $P_\beta = \emptyset$, $N^*_\alpha = P^*_\alpha$, and $P^*_\alpha = \emptyset$;
$b(\beta) \leftarrow b(\alpha)$;
create a new leaf γ with sets $N_\gamma = \emptyset$, $P_\gamma = P_\gamma$, $N^*_\gamma = \emptyset$, and $P^*_\gamma = P^*_\gamma$;
$b(\gamma) \leftarrow b(\alpha) + ib(\alpha)$;
foreach $x \in P_\alpha$ **do**
    $b(\gamma) \leftarrow b(\gamma) - ib(x)$;
    $b(\beta) \leftarrow b(\beta) + ob(x)$;
    delete edges (x, z) or (z, x) for any $z \in P_\alpha$, and update *flux*, deg, *ib*, *ob* ;
check the affected vertices and update P_β, N_β, P^*_β, N^*_β, $db_N(\beta)$ and $db_P(\beta)$;
check the affected vertices and update P_γ, N_γ, P^*_γ, N^*_γ, $db_N(\gamma)$ and $db_P(\gamma)$;
Split(β); Split(γ);

Proof During a split, our main task is to remove the cross edges between N_α and P_α and make sure that all the counters and the vertex sets in the new leaves are correct.

Let $y \in V_\beta$. If there is no cross edge attached to a vertex y in E_α, then $d(y; \beta) = d(y; \alpha)$ and $\deg(y; \beta) = \deg(y; \alpha)$. This means that we only need to check vertices that are adjacent to a cross edge, and possibly move them to a different set, depending on $\deg(y; \beta)$ and $d(y; \beta)$. This is exactly what the algorithm does. The case for $y \in V_\gamma$ is similar.

The only non-trivial counters are $b(\gamma)$ and $b(\beta)$. Note that $b(\gamma)$ consists of $b(\alpha)$ as well as additional edges to N_α, namely $b(\gamma) = b(\alpha) + \sum_{y \in N_\alpha} ib(y)$, which is exactly what algorithm computes. Also, $b(\beta) = b(\alpha) + \sum_{y \in P_\alpha} ob(y) = b(\alpha) + ob(\alpha) - \sum_{y \in N_\alpha} ob(y)$. The remaining counters are trivial to maintain as we delete edges or move vertices from one set to another. □

We conclude this section with the computational complexity analysis.

Proposition 11 *Constructing the tree can be done in $\mathcal{O}(m \log n)$ time, where m is the number of edges and n is the number of vertices in the input graph.*

To prove the proposition, we need the following lemmas.

Lemma 3 *Let $m = |adj(N_\alpha)|$. Updating the new leaves in* ConstructLeft *can be done in $\mathcal{O}(m)$ time.*

Proof The assignments $N_\beta = N_\alpha$, $N^*_\beta = N^*_\alpha$, $P_\gamma = P_\alpha$, $P^*_\gamma = P^*_\alpha$ are done by reference, so they can be done in constant time. Since there are no singletons in N_α, there are most $2m$ vertices in N_α. Deleting an edge is done in constant time, so the for-loop requires $\mathcal{O}(m)$ time. There are at most $2m$ affected vertices, thus updating the sets also can be done in $\mathcal{O}(m)$ time. □

Lemma 4 *Let $m = |adj(P_\alpha)|$. Updating the new leaves in* ConstructRight *can be done in $\mathcal{O}(m)$ time.*

The proof for the lemma is the same as the proof for Lemma 3.

Proof Let us write $m_\alpha = \min(|adj(N_\alpha)|, |adj(P_\alpha)|)$ to be the smaller of the two adjacent edges.

Lemmas 3–4 implies that the running time is $\mathcal{O}(\sum_\alpha m_\alpha)$, where α runs over every vertex in the final tree.

We can express the sum differently: given an edge e, write

$$i_{e\alpha} = \begin{cases} 1 & e \in adj(N_\alpha)\,, \quad \text{and} \quad |adj(N_\alpha)| \leq |adj(P_\alpha)|, \\ 1 & e \in adj(P_\alpha)\,, \quad \text{and} \quad |adj(N_\alpha)| > |adj(P_\alpha)|, \\ 0 & \text{otherwise} \quad . \end{cases}$$

That is, $m_\alpha = \sum_e i_{e\alpha}$. Write $i_e = \sum_\alpha i_{e\alpha}$. To prove the proposition, we show that $i_e \in \mathcal{O}(\log n)$.

Fix e, and let α and β be two vertices in a tree for which $i_{e\alpha} = i_{e\beta} = 1$. Either α is a descendant of β, or β is a descendant of α. Assume the latter, without the loss of generality. We will show that $2|E_\beta| \leq |E_\alpha|$, and this immediately proves that $i_e \in \mathcal{O}(\log n)$.

To prove this, let us define c_α to be the number of cross edges between N_α and P_α, when splitting α. Assume, for simplicity, that β is the left descendant of α. Then $|E_\beta| \leq |adj(N_\alpha)| - c_\alpha$. Also, $e \in adj(N_\alpha)$, and by definition of $i_{e\alpha}$, $m_\alpha = |adj(N_\alpha)|$. This gives us,

$$2|E_\beta| \leq 2m_\alpha - 2c_\alpha \leq (|adj(N_\alpha)| - c_\alpha) + (|adj(P_\alpha)| - c_\alpha) = |E_\alpha| - c_\alpha \leq |E_\alpha| \quad .$$

The case when β is the right descendant is similar, proving the result. □

7.2 Enforcing the cardinality constraint by pruning the tree

If we did not specify the cardinality constraint, then once we have obtained the tree, we can now assign individual ranks to the leaves, and consequently to the vertices. If k is specified, then we may violate the cardinality constraint by having too many leaves.

In such case, we need to reduce the number of leaves, which we do by pruning some branches. Luckily, we can do this optimally by using dynamic programming. To see this, let T' be a subtree of T obtained by merging some of the branches, making them into leaves. Then Proposition 8 implies that $q(T')$ is equal to

$$q(T') = W + \sum_{\alpha \text{ is a non-leaf in } T'} gain(\alpha)\,,$$

where W is the total weight of edges.

This allows us to define the following dynamic program. Let $opt(\alpha; h)$ be the optimal gain achieved in branch starting from α using only h ranks. If α is the root of T, then $opt(\alpha; k)$ is the optimal agony that can be obtained by pruning T to have only k leaves.

To compute $opt(\alpha; h)$, we first set $opt(\alpha; 1) = 0$ for any α, and $opt(\alpha; h) = 0$ if α is a leaf in T. If α is a non-leaf and $k > 1$, then we need to distribute the budget among the two children, that is, we compute

$$opt(\alpha; h) = gain(\alpha) + \min_{1 \leq \ell \leq h-1} opt(\beta; \ell) + opt(\gamma; h - \ell) \quad .$$

We also record the optimal index ℓ, that allows us to recover the optimal tree. Computing a single $opt(\alpha; h)$ requires $\mathcal{O}(k)$ time, and we need to compute at most $\mathcal{O}(nk)$ entries, leading to $\mathcal{O}\big(nk^2\big)$ running time.

7.3 Strongly connected component decomposition

If the input graph has no cycles and there is no cardinality constraint, then the optimal agony is 0. However, the heuristic is not guaranteed to produce such a ranking. To guarantee this, we add an additional—and optional—step. First, we perform the SCC decomposition. Secondly, we pack strongly connected components in the minimal number of layers: source components are in the first layer, second layer consists of components having edges edge only from the first layer, and so on. We then run the heuristic on each individual layer.

If k is not set, we can now create a global ranking, where the ranks of the ith layer are larger than the ranks of the $(i-1)$th layer. In other words, edges between the SCCs are all forward.

If k is set, then we need to decide how many individual ranks each component should receive. Moreover, we may have more than k layers, so some of the layers must be merged. In such a case, we will demand that the merged layers must use exactly 1 rank, together. The reason for this restriction is that it allows us to compute the optimal distribution quickly using dynamic programming.

The gain in agony comes from two different sources. The first source is the improvement of edges within a single layer. Let us adopt the notation from the previous section, and write $opt(i; h)$ to be the optimal gain for ith layer using h ranks. We can compute this using the dynamic program in the previous section. The second source of gain is making the inter-layer edges forward. Instead of computing the total weight of such edges, we compute how many edges are not made forward. These are exactly the edges that are between the layers that have been merged together. In order to express this we write $w(j, i)$ to be the total weight of inter-layer edges having both end points in layers $j, \ldots, i$.

To express, the agony of the tree, let k_i be the budget of individual layers. We also, write $[a_j, b_j]$ to mean that layers $a_j, \ldots, b_j$ have been merged, and must share a single rank. We can show that the score of the tree T' that uses this budget distribution of is then equal to

$$q(T') = W + \sum_i opt(i, k_i) + \sum_j w(a_j, b_j),$$

where W is the total weight of the *intra-layer* edges. Note that W is a constant and so we can ignore it.

To find the optimal k_i and $[a_j, b_j]$, we use the following dynamic program. Let us write $o(i; h)$ to be the gain of $1, \ldots, i$ layers using h ranks. We can express $o(i; h)$ as

$$o(i; h) = \min(\min_j w(j, i) + o(j-1; h-1)\,,\ \min_\ell opt(i, \ell) + o(i-1, h-\ell)) \quad .$$

The first part represents merging $j, \ldots, i$ layers, while the second part represents spending ℓ ranks on the ith layer. By recording the optimal j and ℓ we can recover the optimal budget distribution for each i and h.

Computing the second part can be done in $\mathcal{O}(k)$ time, and computing the first part can be done in $\mathcal{O}(n)$ time, naively. This leads to $\mathcal{O}\big(n^2k + nk^2\big)$ running time, which is too expensive.

Luckily we can speed-up the computation of the first term. To simplify notation, fix h, and let us write $f(j, i) = w(j, i) + o(j-1; h-1)$. We wish to find $j(i)$ such that $f(j(i), i)$ is minimal for each i. Luckily, f satisfies the condition,

$$f(j_1, i_1) - f(j_2, i_1) \leq f(j_1, i_2) - f(j_2, i_2),$$

where $j_1 \leq j_2 \leq i_1 \leq i_2$. Aggarwal et al. [1] now guarantees that $j(i_1) \leq j(i_2)$, for $i_1 \leq i_2$. Moreover, Aggarwal et al. [1] provides an algorithm that computes $j(i)$ in $\mathcal{O}(n)$ time. Unfortunately, we cannot use it since it assumes that $f(j, i)$ can be computed in constant time, which is not the case due to $w(j, i)$.

Fortunately, we can still use the monotonicity of $j(\cdot)$ to speed-up the algorithm. We do this by computing $j(i)$ in an interleaved manner. In order to do so, let ℓ be the number of layers, and let t be the largest integer such that $s = 2^t \leq \ell$. We first compute $j(s)$. We then proceed to compute $j(s/2)$ and $j(3s/2)$, and so on. We use the previously computed values of $j(\cdot)$ as sentinels: when computing $j(s/2)$ we do not test $j > j(s)$ or when computing $j(3s/2)$ we do not test $j < j(s)$. The pseudo-code is given in Algorithm 8.

To analyze the complexity, note that for a fixed s, the variables i, a and b are only moving to the right. This allows us to compute $w(j, i)$ incrementally: whenever we increase i, we add the weights of new edges to the total weight, whenever we increase j, we delete the weights of expiring edges from the

Algorithm 8: Fast algorithm for computing $j(i)$ minimizing $f(j(i), i)$

$\ell \leftarrow$ largest possible i;
$s \leftarrow \max\left\{2^t \leq \ell, t \in \mathbb{N}\right\}$;
while $s \geq 1$ **do**
    foreach $i = s, 3s, 5s, \ldots, i \leq \ell$ **do**
        $a \leftarrow 1$; $b \leftarrow i$;
        if $i - s \geq 1$ **then** $a \leftarrow j(i - s)$;
        if $i + s \leq \ell$ **then** $b \leftarrow \min(b, j(i + s))$;
        $j(i) \leftarrow \min_{a \leq j \leq b} f(j, i)$;
    $s \leftarrow s/2$;

total weight. Each edge is visited twice, and this gives us $\mathcal{O}(m)$ time for a fixed s. Since s is halved during each outer iteration, there can be at most $\mathcal{O}(\log n)$ iterations. We need to do this for each h, so the total running time is $\mathcal{O}\big(km \log n + nk^2\big)$.

As our final remark, we should point out that using this decomposition may not necessarily result in a better ranking. If k is not specified, then the optimal solution will have inter-layer edges as forward, so we expect this decomposition to improve the quality. However, if k is small, we may have a better solution if we allow to inter-layer edges go backward. At extreme, $k = 2$, we are guaranteed that the heuristic *without* the SCC decomposition will give an optimal solution, so the SCC decomposition can only harm the solution. We will see this behaviour in the experimental section. Luckily, since both algorithms are fast, we can simply run both approaches and select the better one.

8 Related work

The problem of discovering the rank of an object based on its dominating relationships to other objects is a classic problem. Perhaps the most known ranking method is Elo rating devised by Elo [4], used to rank chess players. In similar fashion, Jameson et al. [9] introduced a statistical model, where the likelihood of the the vertex dominating other is based on the difference of their ranks, to animal dominance data.

Maiya and Berger-Wolf [12] suggested an approach for discovering hierarchies, directed trees from weighted graphs such that parent vertices tend to dominate the children. To score such a hierarchy the authors propose a statistical model where the probability of an edge is high between a parent and a child. To find a good hierarchy the authors employ a greedy heuristic.

The technical relationship between our approach and the previous studies on agony by Gupte et al. [7] and Tatti [19] is a very natural one. The authors of both papers demonstrate that minimizing agony in a unweighted graph is a dual problem to finding a maximal eulerian subgraph, a subgraph in which, for each vertex, the same number of outdoing edges and the number of incoming edges is the same. Discovering the maximum eulerian subgraph is a special case of the capacitated circulation problem, where the capacities are set to 1. However, the algorithms in [7, 19] are specifically designed to work with unweighted edges. Consequently, if our input graph edges or we wish to enforce the cardinality constraint, we need to solve the problem using the capacitated circulation solver.

The stark difference of computational complexities for different edge penalties is intriguing: while we can compute agony and any other convex score in polynomial-time, minimizing the concave penalties is **NP**-hard. Minimizing the score $q(G, k, p_c)$ is equivalent to FEEDBACK ARC SET (FAS), which is known to be **APX**-hard with a coefficient of $c = 1.3606$ [2]. Moreover, there is no known constant-ratio approximation algorithm for FAS, and the best known approximation algorithm has ratio $\mathcal{O}(\log n \log\log n)$ [5]. In this paper we have shown that minimizing concave penalty is **NP**-hard. An interesting theoretical question is whether this optimization problem is also **APX**-hard, and is it possible to develop an approximation algorithm.

Role mining, where vertices are assigned different roles based on their adjacent edges, and other features, has received some attention. Henderson et al. [8] studied assigning roles to vertices based on its features while McCallum et al. [13] assigned topic distributions to individual vertices. A potential direction for a future work is to study whether the rank obtained from minimizing agony can be applied as a feature in role discovery.

9 Experiments

In this section we present our experimental evaluation. Our main focus of the experiments is practical computability of the weighted agony.

9.1 Datasets and setup

For our experiments we took 10 large networks from SNAP repository [10]. In addition, for illustrative purposes, we used two small datasets: *Nfl*, consisting of National Football League teams. We created an edge (x, y) if team x has scored more points against team y during 2014 regular season, we assign the weight to be the difference between the points. Since not every team plays against every team, the graph is not a tournament. *Reef*, a food web of guilds of species [17], available at [18]. The dataset consisted of 3 food webs of coral reef systems: The Cayman Islands, Jamaica, and Cuba. An edge (x, y) appears if a guild x is known to prey on a guild y. Since the guilds are common among all 3 graphs, we combined the food webs into one graph, and weighted the edges accordingly, that is, each edge received a weight between 1 and 3.

The sizes of the graphs, along with the sizes of the largest strongly connected component, are given in the first 4 columns of Table 2.

The 3 *Higgs* and *Nfl* graphs had weighted edges, and for the remaining graphs we assigned a weight of 1 for each edge. We removed any self-loops as they have no effect on the ranking, as well as any singleton vertices.

For each dataset we computed the agony using Algorithm 2. We compared the algorithm to the baseline given by Tatti [20]. For the unweighted graphs we also computed the agony using Relief, an algorithm suggested by Tatti [19]. Note that this algorithm, nor the algorithm by Gupte et al. [7], does not work for weighted graphs nor when the cardinality constraint k given in Problem 1 is enforced. We implemented algorithms in C++ and performed experiments using a Linux-desktop equipped with a Opteron 2220 SE processor.[6]

Table 2 Basic characteristics of the datasets and the experiments. The 6th is the number of groups in the optimal ranking.

			largest SCC			time		baseline	
Name	$\|V\|$	$\|E\|$	$\|V'\|$	$\|E'\|$	k	SCC	plain	[20]	[19]
Amazon	403 394	3 387 388	395 234	3 301 092	17	24m7s	25m6s	6h24m	4h27m
Gnutella	62 586	147 892	14 149	50 916	24	4s	20s	8s	45s
EmailEU	265 214	418 956	34 203	151 132	9	10s	29s	10m	2m
Epinions	75 879	508 837	32 223	443 506	10	33s	44s	49m	20m
Slashdot	82 168	870 161	71 307	841 201	9	38s	61s	1h38m	1h5m
WebGoogle	875 713	5 105 039	434 818	3 419 124	31	10m31s	25m22s	8h50m	2h32m
WikiVote	7 115	103 689	1 300	39 456	12	2s	6s	43s	7s
Nfl	32	205	32	205	6	4ms	5ms	22ms	–
Reef	258	4232	1	0	19	8ms	100ms	10ms	–
HiggsReply	37 145	30 517	263	569	11	0.3s	5s	0.2s	–
HiggsRetweet	425 008	733 610	13 086	63 505	22	12s	2m10s	10m	–
HiggsMention	302 523	445 147	4 786	19 848	21	6s	1m34s	2m	–

9.2 Results

Let us begin by studying running times given in Table 2. We report the running times of our approach with and without the strongly connected component decomposition as suggested by Proposition 2, and compare it against the baselines, whenever possible. Note that we can use the decomposition only if we do not enforce the cardinality constraint.

Our first observation is that the decomposition always helps to speed up the algorithm. In fact, this speed-up may be dramatic, if the size of the strongly connected component is significantly smaller than the size of the input graph, for example, with *HiggsRetweet*. The running times are practical despite the unappealing theoretical bound. This is due to several factors. First, note that the theoretical bound of $\mathcal{O}(\min(nk, m) m \log n)$ given in Section 4 only holds for unweighted graphs, and it is needed to bound the number of outer-loop iterations. In practice, however, the number of these iterations is small, even for weighted graphs. The other, and the main, reason is the pessimistic n in the $\min(nk, m)$ factor; we spend nk inner-loop iterations only if the dual $\pi(v)$

[6] The source code is available at `http://users.ics.aalto.fi/ntatti/agony.zip`

of each vertex v increases by $\mathcal{O}(k)$, and *between* the increases the shortest path from sources to v changes $\mathcal{O}(n)$ times. The latter change seems highly unlikely in practice, leading to a faster computational time.

We see that our algorithm beats consistently both baselines. What is more important: the running times remain practical, even if we do not use strongly connected components. This allows us to limit the number of groups for large graphs. This is a significant improvement over [20], where solving *HiggsRetweet* *without* the SCC decomposition required 31 hours.

Our next step is to study the effect of the constraint k, the maximum number of different rank values. We see in the 6th column in Table 2 that despite having large number of vertices, that the optimal rank assignment has low number of groups, typically around 10–20 groups, even if the cardinality constraint is not enforced.

Let us now consider agony as a function of k, which we have plotted in Figure 5 for *Gnutella* and *WikiVote* graphs. We see that for these datasets that agony remains relatively constant as we decrease k, and starts to increase more prominently once we consider assignments with $k \leq 5$.

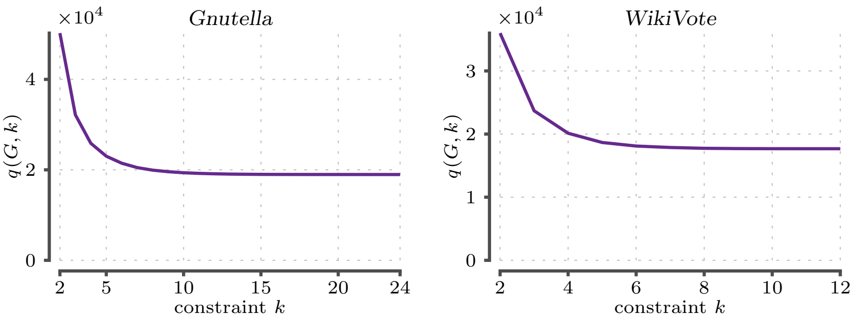

Fig. 5 Agony as a function of the constraint k for *Gnutella* and *WikiVote* datasets.

Enforcing the constraint k has an impact on running time. As implied by Proposition 3, low values of k should speed-up the computation. In Figure 6 we plotted the running time as a function of k, compared to the plain version without the speed-up.

As we can see lower values of k are computationally easier to solve. This is an opposite behavior of [20], where lowering k increased the computational time. To explain this behaviour, note that when we decrease k we increase the agony score, which is equivalent to the capacitated circulation. Both solvers increase incrementally the flow until we have reached the solution. As we lower k, we increase the amount of optimal circulation, and we need more iterations to reach the optimal solution. The difference between the algorithm is that for lower k updating the residual graph becomes significantly faster than computing the tree from scratch. This largely overpowers the effect of

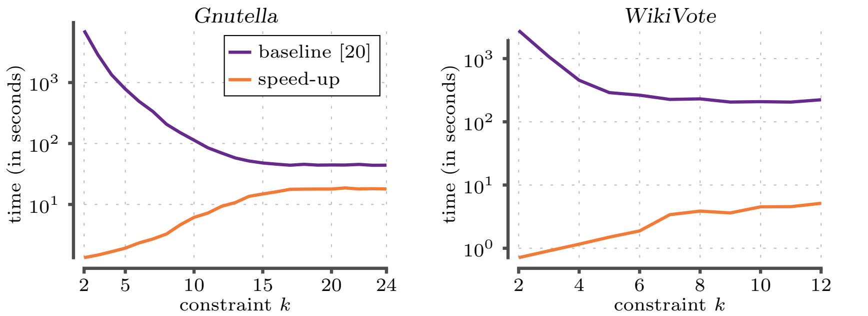

Fig. 6 Execution time as a function of the constraint k for *Gnutella* and *WikiVote* datasets. Note that the y-axis is logarithmic.

Table 3 Scores, compared to the optimal, and running times of the heuristic. Here, *SCC* is the heuristic with SCC decomposition, while *plain* is the plain version, *opt* is the optimal agony.

						Time (sec.)	
Name	$\frac{q(\text{SCC})}{q(\text{opt})}$	$\frac{q(\text{plain})}{q(\text{opt})}$	q(SCC)	q(plain)	q(opt)	SCC	plain
Amazon	1.036	1.037	2 044 609	2 046 344	1 973 965	9.24	8.49
Gnutella	1.256	1.350	23 820	25 603	18 964	0.35	0.34
EmailEU	1.008	1.012	121 820	122 362	120 874	0.47	0.45
Epinions	1.024	1.030	271 419	273 016	264 995	0.40	0.37
Slashdot	1.001	1.003	749 448	750 760	748 582	0.68	0.64
WebGoogle	1.051	1.079	1 935 476	1 985 831	1 841 215	6.80	6.64
WikiVote	1.043	1.091	18 430	19 276	17 676	0.05	0.05
Nfl	1.047	1.047	1172	1172	1119	0.002	0.002
Reef	–	–	0	452	0	0.008	0.006
HiggsReply	1.007	1.103	5 499	6 022	5459	2.29	1.03
HiggsRetweet	1.259	1.606	19 264	24 579	15 302	0.12	0.05
HiggsMention	1.078	1.322	24 165	29 632	22 418	1.82	1.65

needing many more iterations to converge. However, there are exceptions: for example, computing agony for *WikiVote* with $k = 8$ is slower than $k = 9$.

Let us now consider the performance of the heuristic algorithm. We report the obtained scores and the running times in Table 3. We tested both variants: with and without SCC decomposition, and we do not enforce k. We first observe that both variants are expectedly fast: processing the largest graphs, *Amazon* and *WebGoogle*, required less than 10 seconds, while the exact version needed 10–25 minutes. The plain version is cosmetically faster. Heuristic also produces competitive scores but the performance depends on the dataset: for *Gnutella* and *HiggsRetweet* the SCC variant produced 25% increase to agony, while for the remaining datasets the increase was lower than 8%. Note that, *Reef* has agony of 0, that is, the network is a DAG but the plain variant was not able to detect this. This highlights the benefit of doing the SCC decomposition. In general, the SCC variant outperforms the plain variant when we do not enforce the cardinality constraint.

As we lower the cardinality constraint k, the plain variant starts to outperform the SCC variant, as shown in Figure 7. The reason for this is that the SCC variant requires that a edge (u, v) between two SCCs is either forward or $r(u) = r(v)$. This restriction becomes too severe as we lower k and it becomes more profitable to allow $r(u) < r(v)$. At extreme $k = 2$, the plain version is guaranteed to find the optimal solution, so the SCC variant can only harm the solution.

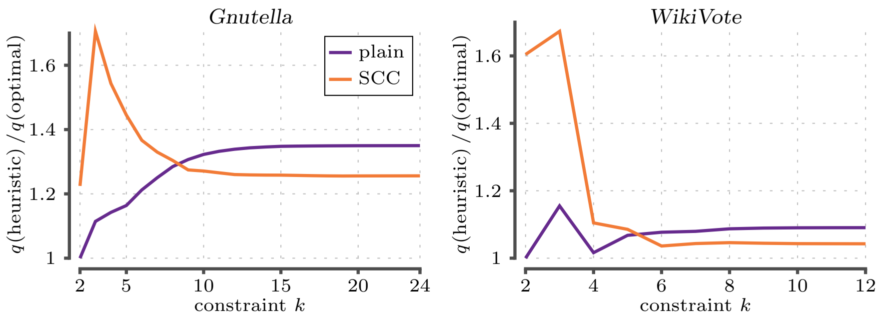

Fig. 7 Ratio of the agony given by the heuristic and the optimal agony as a function of the constraint k for *Gnutella* and *WikiVote* datasets.

Table 4 Rank assignment discovered for *Nfl* dataset with $k = 3$ groups

Rank	Teams
1.	DEN BAL NE DAL SEA PHI KC GB PIT
2.	STL NYG MIA CAR NO SD MIN CIN BUF DET IND HOU SF ARI
3.	WSH OAK TB JAX TEN CLE ATL NYJ CHI

Let us look at the ranking that we obtained from *Nfl* dataset using $k = 3$ groups, given in Table 4. We see from the results that the obtained ranking is very sensible. 7 of 8 teams in the top group consists of playoff teams of 2014 season, while the bottom group consists of teams that have a significant losing record.

Finally, let us look at the rankings obtained *Reef* dataset. The graph is in fact a DAG with 19 groups. To reduce the number of groups we rank the guilds into $k = 4$ groups. The condensed results are given in Table 5. We see that the top group consists of large fishes and sharks, the second group contains mostly smaller fishes, a large portion of the third group are crustacea, while the last group contains the bottom of the food chain, planktons and algae. We should point out that this ranking is done purely on food web, and not on type of species. For example, *cleaner crustacea* is obviously very different than plankton. Yet *cleaner crustacea* only eats *planktonic bacteria* and *micro-*

Table 5 Ranked guilds of *Reef* dataset, with $k = 4$. For simplicity, we removed the duplicate guilds in the same group, and grouped similar guilds (marked as italic, the number in parentheses indicating the number of guilds).

Sharks (6), Amberjack, Barracuda, Bigeye, Coney grouper, Flounder, Frogfish, Grouper, Grunt, Hind, Lizardfish, Mackerel, Margate, Palometa, Red hind, Red snapper, Remora, Scorpionfish, Sheepshead, Snapper, Spotted eagle ray
Angelfish, Atlantic spadefish, Ballyhoo, Barracuda, Bass Batfish, Blenny Butterflyfish, Caribbean Reef Octopus, Caribbean Reef Squid, Carnivorous fish II-V, Cornetfish, Cowfish, Damselfish, Filefish, Flamefish, Flounder, Goatfish, Grunt, Halfbeak, Hamlet, Hawkfish, Hawksbill turtle, Herring, Hogfish, Jack, Jacknife fish, Jawfish, Loggerhead sea turtle, Margate, Moray, Needlefish Porcupinefish I-II, Porkfish, Pufferfish, Scorpionfish, Seabream, Sergeant major, Sharptail eel, Slender Inshore Squid, Slippery dick, Snapper, Soldierfish, Spotted drum, Squirrelfish, Stomatopods II, Triggerfish, Trumpetfish, Trunkfish, Wrasse, Yellowfin mojarra
Crustacea (31), Ahermatypic benthic corals, Ahermatypic gorgonians, Anchovy, Angelfish, Benthic carnivores II, Blenny, Carnivorous fish I, Common Octopus, Corallivorous gastropods IV, Deep infaunal soft substrate suspension feeders, Diadema, Echinometra, Goby, Green sea turtle, Herbivorous fish I-IV, Herbivorous gastropods I, Hermatypic benthic carnivores I, Hermatypic corals, Hermatypic gorgonians, Herring, Infaunal hard substrate suspension feeders, Lytechinus, Macroplanktonic carnivores II-IV, Macroplanktonic herbivores I, Molluscivores I, Omnivorous gastropod, Parrotfish, Pilotfish, Silverside, Stomatopods I, Tripneustes, Zooplanktivorous fish I-II,
Planktons (7), *Algae (6)*, *Sponges (2)*, *Feeders (11)*, Benthic carnivores I, Carnivorous ophiuroids, Cleaner crustacea I, Corallivorous polychaetes, Detritivorous gastropods I, Echinoid carnivores I, Endolithic polychaetes, Epiphyte grazer I, Epiphytic autotrophs, Eucidaris, Gorgonian carnivores I, Herbivorous gastropod carnivores I, Herbivorous gastropods II-IV, Holothurian detritivores, Macroplanktonic carnivores I, Micro-detritivores, Molluscivores II-III, Planktonic bacteria, Polychaete predators (gastropods), Seagrasses, Sponge-anemone carnivores I, Spongivorous nudibranchs

detritivores while being eaten by many other guilds. Consequently, it is ranked in the bottom group.

10 Concluding remarks

In this paper we studied the problem of discovering a hierarchy in a directed graph that minimizes agony. We introduced several natural extensions: *(i)* we demonstrated how to compute the agony for weighted edges, and *(ii)* how to limit the number of groups in a hierarchy. Both extensions cannot be handled with current algorithms, hence we provide a new technique by demonstrating that minimizing agony can be solved by solving a capacitated circulation problem, a well-known graph problem with a polynomial solution.

We also introduced a fast divide-and-conquer heuristic that produces the rankings with competitive scores.

We should point out that we can further generalize the setup by allowing each edge to have its own individual penalty function. As long as the penalty functions are convex, the construction done in Section 5.1 can still be used to solve the optimization problem. Moreover, we can further generalize cardinality constraint by requiring that only a subset of vertices must have ranks within some range. We can have multiple such constraints.

There are several interesting directions for future work. As pointed out in Section 5.1 minimizing convex penalty increases the number of edges when solving the corresponding circulation problem. However, these edges have very specific structure, and we conjecture that it is possible to solve the convex case without the additional computational burden.

References

1. A. Aggarwal, M. Klawe, S. Moran, P. Shor, and R. Wilber. Geometric applications of a matrix-searching algorithm. *Algorithmica*, 2(1–4):195–208, 1987.
2. I. Dinur and S. Safra. On the hardness of approximating vertex cover. *Annals of Mathematics*, 162(1):439–485, 2005.
3. J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of ACM*, 19(2):248–264, 1972.
4. A. E. Elo. *The rating of chessplayers, past and present.* Arco Pub., 1978.
5. G. Even, J. (Seffi) Naor, B. Schieber, and M. Sudan. Approximating minimum feedback sets and multicuts in directed graphs. *Algorithmica*, 20(2):151–174, 1998.
6. M. Garey and D. Johnson. *Computers and intractability: a guide to the theory of NP-completeness.* WH Freeman & Co., 1979.
7. M. Gupte, P. Shankar, J. Li, S. Muthukrishnan, and L. Iftode. Finding hierarchy in directed online social networks. In *Proceedings of the 20th International Conference on World Wide Web*, pages 557–566, 2011.
8. K. Henderson, B. Gallagher, T. Eliassi-Rad, H. Tong, S. Basu, L. Akoglu, D. Koutra, C. Faloutsos, and L. Li. Rolx: Structural role extraction & mining in large graphs. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1231–1239, 2012.
9. K. A. Jameson, M. C. Appleby, and L. C. Freeman. Finding an appropriate order for a hierarchy based on probabilistic dominance. *Animal Behaviour*, 57:991–998, 1999.
10. J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, Jan. 2015.
11. L. Macchia, F. Bonchi, F. Gullo, and L. Chiarandini. Mining summaries of propagations. In *IEEE 13th International Conference on Data Mining*, pages 498–507, 2013.
12. A. S. Maiya and T. Y. Berger-Wolf. Inferring the maximum likelihood hierarchy in social networks. In *Proceedings IEEE CSE'09, 12th IEEE International Conference on Computational Science and Engineering*, pages 245–250, 2009.

13. A. McCallum, X. Wang, and A. Corrada-Emmanuel. Topic and role discovery in social networks with experiments on enron and academic email. *J. Artif. Int. Res.*, 30(1):249–272, 2007.
14. J. B. Orlin. A faster strongly polynomial minimum cost flow algorithm. *Operations Research*, 41(2), 1993.
15. C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Inc., 1982.
16. G. Ramalingam and T. Reps. On the computational complexity of dynamic graph problems. *Theoretical Computer Science*, 158:233–277, 1996.
17. P. D. Roopnarine and R. Hertog. Detailed food web networks of three Greater Antillean Coral Reef systems: The Cayman Islands, Cuba, and Jamaica. *Dataset Papers in Ecology*, 2013, 2013.
18. H. R. Roopnarine PD. Data from: Detailed food web networks of three Greater Antillean Coral Reef systems: The Cayman Islands, Cuba, and Jamaica, 2012. Dryad Digital Repository, `http://dx.doi.org/10.5061/dryad.c213h`.
19. N. Tatti. Faster way to agony — discovering hierarchies in directed graphs. In *Prooceding of European Conference of Machine Learning and Knowledge Discovery in Databases, ECML PKDD 2014*, pages 163–178, 2014.
20. N. Tatti. Hierarchies in directed networks. In *Proceedings of the 15th IEEE International Conference on Data Mining (ICDM 2015)*, 2015.

A Proof of Proposition 4

Proof To prove the completeness we will provide reduction from MAXIMUM CUT [6]. An instance of MAXIMUM CUT consists of an undirected graph, and we are asked to partition vertices into two sets such that the number of cross edges is larger or equal than the given threshold σ.

Note that the conditions of the proposition guarantee that $p(0) > 0$.

Assume that we are given an instance of MAXIMUM CUT, that is, an undirected graph $G = (V, E)$ and a threshold σ. Let $m = |E|$. Define a weighted directed graph $H = (W, F, w)$ as follows. Add V to W. For each edge $(u, v) \in E$, add a path with t intermediate vertices from u to v, the length of the path is $t + 2$. Add also a path in reversed direction, from v to u. Set edge weights to be 1. Add 4 special vertices $\alpha_1, \ldots, \alpha_4$. Add edges (α_{i+1}, α_i), for $i = 1, \ldots, 3$ with a weight of

$$C = 2B\frac{p(0)}{p(2) - p(1)}, \quad \text{where} \quad B = 2(t + 1)m \quad .$$

Add edges (α_i, α_{i+1}), for $i = 1, \ldots, 3$ with a weight of

$$D = 4Cp(1)\,/p(0) + B \quad .$$

Add edges (α_0, v) and (v, α_4), for each $v \in V$, with a weight of D.

Let r be the optimal ranking for H. We can safely assume that $r(\alpha_1) = 0$. We claim that $r(\alpha_i) = i - 1$, and $r(v) = 1, 2$ for each $v \in V$. To see this, consider a ranking r' such that $r'(\alpha_i) = i - 1$ and the rank for the remaining vertices is 2. The score of this rank is

$$q\big(H, r'\big) = 3Cp(1) + 2(t + 1)mp(0) = 3Cp(1) + Bp(0) \quad .$$

Let $(u, v) \in F$ with the weight of D. If $r(u) \geq r(v)$, then the score of r is at least $Dp(0) = 4Cp(1) + Bp(0)$ which is more than $q(H, r')$. Hence, $r(u) < r(v)$. Let $(u, v) \in F$ with the weight of C. Note that $r(u) \geq r(v) + 1$. Assume that $r(u) \geq r(v) + 2$. Then the score is at least

$$3Cp(1) + C(p(2) - p(1)) = 3Cp(1) + 2Bp(0)\,,$$

which is a contradiction. This guarantees that $r(\alpha_i) = i - 1$, and $r(v) = 1, 2$ for each $v \in V$.

Consider $(u, v) \in E$ and let $u = x_0, \ldots, x_{t+1} = v$ be the corresponding path in H. Let $d_i = r(x_i) - r(x_{i+1})$ and set $\ell = t + r(u) - r(v)$. Let $P = \sum p(d_i)$ be the penalty contributed by this path. Note that $P \leq p(\ell)$, a penalty that we achieve by setting $r(x_i) = r(x_{i-1}) + 1$ for $i = 1, \ldots, t$. This implies that $d_i \leq \ell$. The condition of the proposition now implies

$$\begin{aligned} P = \sum_{i=0}^{t} p(d_i) &= \sum_{i=0, d_i \geq 0}^{t} p(d_i) \\ &\geq \sum_{i=0}^{t} \frac{\max(d_i + 1, 0)}{\ell + 1} p(\ell) \\ &\geq \sum_{i=0}^{t} \frac{d_i + 1}{\ell + 1} p(\ell) \\ &= \frac{p(\ell)}{\ell + 1} \sum_{i=0}^{t} 1 + r(x_i) - r(x_{i+1}) \\ &= \frac{p(\ell)}{\ell + 1} (t + 1 + r(u) - r(v)) = p(\ell) \quad . \end{aligned}$$

This guarantees that $P = p(\ell)$.

Partition edges E into two groups,

$$X = \{(u, v) \in E \in r(u) = r(v)\}$$

and

$$Y = \{(u, v) \in E \in r(u) \neq r(v)\} \quad .$$

Let $\Delta = p(t-1) + p(t+1) - 2p(t)$. Note that concavity implies that $\Delta < 0$. Then

$$\begin{aligned} q(H, r) &= 3C + |X| 2p(t) + |Y|(p(t-1) + p(t+1)) \\ &= 3C + m2p(t) + |Y|(p(t-1) + p(t+1) - 2p(t)) \\ &= 3C + m2p(t) + |Y|\Delta \quad . \end{aligned}$$

The first two terms are constant. Consequently, $q(H, r)$ is optimal if and only if $|Y|$, the number of cross-edges is maximal.

Given a threshold σ, define $\sigma' = 3C + m2p(t) + \Delta\sigma$. Then $q(H, r) \leq \sigma'$ if and only if there is a cut of G with at least σ cross-edges, which completes the reduction. □

B Proof of Proposition 5 and 7

We will prove both Propositions 5 and 7 with the same proof.

Proof Let r^* be the ranking returned by CANON, and let $\pi^* = \pi - d$ be the corresponding dual. Lemma 1 states that π^* satisfies the slackness conditions, so it remains a solution to Problem 5. This implies also that r^* is an optimal ranking.

To complete the proof we need to show that for any r', we have $r^* \preceq r'$. Note that this also proves that r^* is a unique ranking having such property.

Let r' be any optimal ranking, and let π' be the corresponding dual. We can assume that $\pi'(\alpha) = \pi^*(\alpha) = 0$. To prove the result we need to show that $\pi'(v) \geq \pi^*(v)$. We will prove this by induction over the shortest path tree T from α. This certainly holds for α.

Let u be a vertex and let v be its parent in T, and let $e \in E(T)$ be the connecting edge. Note that, by definition, $-t(e) = \pi^*(u) - \pi^*(v)$. By the induction assumption, $\pi^*(v) \leq \pi'(v)$.

If e is forward, then due to Eq. 3

$$\pi'(u) - \pi'(v) \geq -t(e) = \pi^*(u) - \pi^*(v) \geq \pi^*(u) - \pi'(v) \quad .$$

If e is backward, then $f(v, u) > 0$. and Eq. 4 implies

$$\pi'(u) - \pi'(v) = -t(e) = \pi^*(u) - \pi^*(v) \geq \pi^*(u) - \pi'(v) \quad .$$

This completes the induction step, and the proves the proposition. □