Efficient Heuristic for Decomposing a Flow with Minimum Number of Paths

Mingfu Shao*[1] and Carl Kingsford†[1]

[1]Computational Biology Department, School of Computer Science, Carnegie Mellon University

November 14, 2016

Abstract

Motivated by transcript assembly and multiple genome assembly problems, in this paper, we study the following *minimum path flow decomposition* problem: given a directed acyclic graph $G = (V,E)$ with source s and sink t and a flow f, compute a set of s-t paths P and assign weight $w(p)$ for $p \in P$ such that $f(e) = \sum_{p \in P: e \in p} w(p)$, $\forall e \in E$, and $|P|$ is minimized. We propose an efficient pseudo-polynomial-time heuristic for this problem based on novel insights. Our heuristic gives a framework that consists of several components, providing a roadmap for continuing development of better heuristics. Through experimental studies on both simulated and transcript assembly instances, we show that our algorithm significantly improves the previous state-of-the-art algorithm. Implementation of our algorithm is available at `https://github.com/Kingsford-Group/catfish`.

1 Introduction

RNA sequencing (RNA-seq) is an established technology that enables identification of novel genes and transcripts as well as accurate measurement of expression abundances [1]. The RNA-seq protocol produces short sequencing reads sampled from the expressed transcripts. Thus, a fundamental computational problem is to recover the set of full-length expressed transcripts and their expression abundances in the sample from these reads. This is referred to the *transcript assembly* problem. There have been much research interests for this problem and many methods and software packages have been proposed. Depending on whether a reference genome is assumed available, these methods are divided into two categories, *reference-based* methods (e.g., Cufflinks [2], Scripture [3], IsoLasso [4], SLIDE [5], CLIIQ [6], CEM [7], MITIE [8], iReckon [9], Traph [10], and StringTie [11]), and *de novo* methods (e.g., TransABySS [12], Rnnotator [13], Trinity [14], SOAPdenovo-Trans [15], Velvet [16], and Oases [17]). However, according to the benchmarking studies [18, 19], the performance of these methods are far from satisfactory, especially when the expressed genes contain multiple splice isoforms. Hence, transcript assembly still remains an open and challenging problem, and thus requires further algorithm development.

Both reference-based and *de novo* methods share the similar pipeline. The first step of the pipeline is to build the so-called *splice graph* (or transcript graph, overlap graph, etc) from given reads. For reference-based methods, reads are first aligned to the reference genome using some spliced aligner (e.g., TopHat2 [20],

*mingfu.shao@cs.cmu.edu

†carlk@cs.cmu.edu

STAR [21], and HISAT [22]). Then the boundaries of exons and introns are inferred from the spliced aligned reads, and the splice graph is consequently constructed: each vertex represents an exon (or partial exon), each edge indicates there exist reads spanning the two corresponding exons, and the weight of this edge is usually computed as the number of such spanning reads. For *de novo* methods, reads are usually first organized by the *de bruijn* graph. Then the splice graph is constructed by collapsing simple paths of the *de bruijn* graph, and the weight of each edge is computed as the number of reads associated with the corresponding path. For both categories of methods, the constructed splice graph is the superposition of the expressed transcripts, each of which corresponds to an (unknown) path in the splice graph. If we assume the ideal case that the read coverage for each expressed transcript is a constant across the whole transcript, then the weights of edges shall form a flow of the spliced graph. In the realistic scenario where the read coverage varies across each transcript, there exist methods that can smooth the edge weights to make it a flow [10].

The second step of the pipeline is usually to decompose the splice graph into a set of paths and associate a weight with each path. These predict the expressed transcripts and their expression abundances. This step is the most algorithmically challenging part in the whole transcript assembly pipeline. Under the principle of parsimony, it is usually formulated as an optimization problem of decomposing a given flow into a minimum number of paths. Formally, let $G = (V, E)$ be a directed acyclic graph (DAG) with a source vertex s and a sink vertex t. We denote the set of all s-t paths of G by $\mathcal{P}$. Notice that $|\mathcal{P}|$ might be exponential compared with $|E|$. Let $f : E \to \mathbb{Z}$ be an s-t integral flow of G. We say (P, w) is a *decomposition* of (G, f) if we have $P \subseteq \mathcal{P}$, $w : P \to \mathbb{Z}$ assigns an integral weight $w(p)$ for each path $p \in P$, and $f(e) = \sum_{p \in P : e \in p} w(p)$ for every $e \in E$. Our paper focuses on the following problem.

Problem 1 (Minimum Path Flow Decomposition) *Given a DAG $G = (V, E)$ with source vertex s and sink vertex t and an integral s-t flow f, compute a decomposition (P, w) of (G, f) such that $|P|$ is minimized.*

While we formulate Problem 1 in the context of transcript assembly, we emphasize that Problem 1 can be naturally used to model a broader class of so-called *multiple genome assembly* problems [23]. These are problems in genomics where several unknown strings $S = \{s_1, s_2, \ldots, s_k\}$ and their (unknown) counts $C = \{c_1, c_2, \cdots, c_k\}$ must be reconstructed from many substrings (reads) that are sampled from these strings. This is a generalization of the standard genome assembly problem that seeks to reconstruct a single string. The above-mentioned transcript assembly problem is one such instance, where S is the set of alternatively spliced isoforms of a gene, and C is the expression levels of these isoforms. Other instances include metagenomic assembly (where S represents the genomes of microbes in a community and C represents their abundances), cancer genome assembly (where S is the set of genomes of subclones in a tumor sample and C is the frequency of each subclone), and virus quasi-species assembly (where S is the set of genomes in a population of viral genomes infecting a given individual and C is their frequency).

Problem 1 has been proven to be strongly NP-hard [24], meaning that even if all the flow values are in polynomial-size, there still does not exist a polynomial-time algorithm unless P = NP. It has been further proved that even if all flow values are chosen from $\{1, 2, 4\}$, this problem is still NP-hard [25].

Several algorithms and heuristics have been proposed for Problem 1. Vatinlen et al. [24] proposed two practical greedy algorithms, namely *greedy-length* and *greedy-width* algorithms, which are to iteratively choose the shortest path and the path with the largest flow, respectively, in the remaining flow, until the entire flow is decomposed. These two algorithms have been shown that they can guarantee obtaining a decomposition with at most $(|E| - |V| + 2)$ paths, a known upper bound for the number of paths in any optimal solution. Hartman et al. [25] designed two similar algorithms, called *bicriteria-width* and *bicriteria-length*, which can guarantee decomposing $(1 - \varepsilon)$ fraction of the entire flow into $O(|P^*|/\varepsilon^2)$ paths, for any $\varepsilon < 1$. Mumey et al. [26] proposed another two algorithms using parity balancing path flows, which have been proved to be $(L^{\log(F)} \cdot \log(F))$-approximation algorithms, where L is the maximum length of all s-t

paths in G, and F is the maximum flow values over all edges. We emphasize that over all existing algorithms, the greedy-width algorithm has been shown as the best heuristic through experimental comparisons [24, 25, 26], and because of that, this algorithm has been implemented in several widely used transcript assembly methods, for example, Traph [10] and StringTie [11].

Several problems related to Problem 1 have been also addressed. Hendel and Kubiak [27] studied the problem of decomposing a given flow into a set of paths such that the length of the longest path is minimized. They proposed a polynomial-time approximation scheme (PTAS) for this problem. Baier et al. [28, 29] studied the *k-splittable flow* problem, which is to compute a maximum flow such that it can be decomposed into at most k paths. For this problem they devised a $(2/3)$-approximation algorithm for $k = 2$ and a $(2/k)$-approximation algorithm for $k \geq 3$.

Our contribution to Problem 1 is a pseudo-polynomial-time heuristic based on the novel properties we prove, and we show that it considerably outperforms the best previous heuristic (greedy-width) through extensive experimental comparisons. The central idea of our heuristic is that equations involving flow values on edges are intuitively suggestive of the structure of paths. For example, if $f(e_1) = f(e_2) + f(e_3)$, then one might suspect that a path goes through e_1 and e_2 and another path goes through e_1 and e_3 (see examples in Figure 1). The first challenge of our heuristic is to identify these equations. Once identified, each equation can then be used to simplify the graph by merging edges that are involved in the same path. We define a operation, called *reverse*, on the graph that modifies the graph in such a way that the optimal solution does not change, but that attempts to make edges that are to be merged be incident on the same vertex. The entire approach iterates between these two steps: identifying a non-trivial relationship between the flow values on the edges and modifying the graph to facilitate edge merging (and merging the edges). We show that at each stage, either the algorithm fails, or the solution space gets reduced and the gap between the optimal and an upper bound on the optimal is reduced.

2 Motivation

In this Section, we prove some properties about Problem 1 upon which we design our algorithm. We first reformulate some definitions to facilitate using algebra tools. We arbitrarily assign indices to edges in E and paths in $\mathcal{P}$, i.e., $E = \{e_1, e_2, \cdots, e_{|E|}\}$ and $\mathcal{P} = \{p_1, p_2, \cdots, p_{|\mathcal{P}|}\}$. We can then write f as a row vector in $\mathbb{Z}^{1\times|E|}$, with $f[k] = f(e_k)$. We can also write $\mathcal{P}$ as a matrix in $\{0,1\}^{|\mathcal{P}|\times|E|}$, with $\mathcal{P}[i,j] = 1$ if and only if p_i contains edge e_j. Consequently, each subset $P \subseteq \mathcal{P}$ can also be represented as a binary matrix, which can be modified from matrix $\mathcal{P}$ by removing these rows whose corresponding paths are not in P. With these formulations, the definition that (P,w) is a decomposition of (G,f) can be equivalently written as $f = w \cdot P$, where $w \in \mathbb{Z}^{1\times|P|}$. In the following, we use $\mathcal{P}$ (and P) to represent both the set of paths as well as the corresponding matrix, but this will be obvious to distinguish. Let (P^*, w^*) be an optimal decomposition of (G,f). Let $\Delta = |E| - |V| + 2$. From [24], we know that Δ gives an upper bound for $|P^*|$, i.e., $|P^*| \leq \Delta$. We call $(\Delta - |P^*|)$ as the *optimality gap*. Let $\mathcal{N}(P^*)$ be the null space of P^*, i.e., $\mathcal{N}(P^*) = \{q \mid P^* \cdot q = 0\}$. We call vectors in $\mathcal{N}(P^*)$ as *null vectors* (*w.r.t.* P^*). The proofs of all Propositions are in the Appendix C.

Proposition 1 *We have* $f \cdot q = 0$ *for any* $q \in \mathcal{N}(P^*)$.

Proposition 1 is a necessary condition rather than a sufficient condition to characterize $\mathcal{N}(P^*)$. On the other side, if we assume that all weights in w^* are randomly assigned, then it is with small probability to observe that $f \cdot q = 0$ if $q \notin \mathcal{N}(P^*)$.

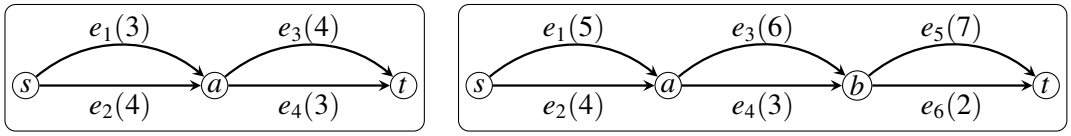
Proposition 2 *Suppose* w^*_j *is sampled independently and uniformly at random from* $\mathbb{Z}_k = \{1, 2, \cdots, k\}$ *for all* $j = 1, 2, \cdots, |P^*|$. *Then we have* $\Pr(f \cdot q = 0 \mid q \notin \mathcal{N}(P^*)) \leq 1/k$.

We further study the structure of $\mathcal{N}(P^*)$. Let $V_0 = V \setminus \{s,t\}$. For each $v \in V_0$, we define a column vector $q_v \in \{-1,0,+1\}^{|E|\times 1}$ where $q_v[k] = 0$ if e_k is not adjacent to v, $q_v[k] = +1$ if e_k points to v, and $q_v[k] = -1$ if e_k leaves v. Let $\mathcal{N}_0$ be the linear space spanned by $\{q_v \mid v \in V_0\}$. We call vectors in $\mathcal{N}_0$ as *trivial null vectors*, and call vectors in $\mathcal{N}(P^*) \setminus \mathcal{N}_0$ as *nontrivial null vectors*. We have the following results.

Proposition 3 $\mathcal{N}_0 \subseteq \mathcal{N}(P^*)$.

Proposition 4 $\dim(\mathcal{N}(P^*)) - \dim(\mathcal{N}_0) = \Delta - |P^*|$.

Proposition 4 says that the optimality gap is larger than 0 if and only if there exists nontrivial null vectors (see Figure 1 for examples). This inspires our algorithm: suppose that we have two oracles, the first one can return nontrivial null vectors, and the second one can use these vectors to reduce the optimality gap; then we can iteratively apply these two oracles, and when the first oracle returns the empty set, the optimality gap has been closed. Our algorithm then tries to approximate these two oracles.



(a) (b)

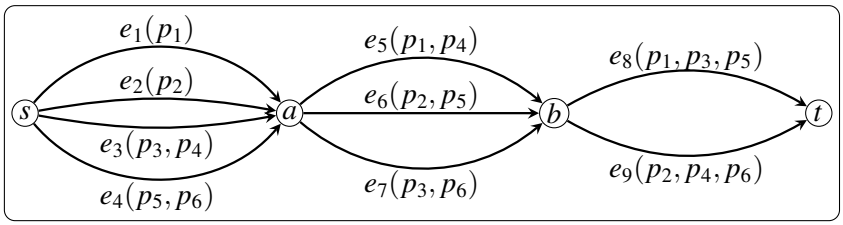
Figure 1: The flow value for each edge is put in parenthesis. **(a)** Observe that $f(e_1) = f(e_4)$, i.e, nontrivial null vector of $(1,0,0,-1)^T$. We have $\Delta = 3$, $|P^*| = 2$ ($p_1^* = \{e_1,e_4\}$, $w^*(p_1^*) = 3$, $p_2^* = \{e_2,e_3\}$, $w^*(p_2^*) = 4$). **(b)** Observe that $f(e_1) = f(e_4) + f(e_6)$, i.e., nontrivial null vector of $(1,0,0,-1,0,-1)^T$. We have $\Delta = 4$, $|P^*| = 3$ ($p_1^* = \{e_1,e_4,e_5\}$, $w^*(p_1^*) = 3$, $p_2^* = \{e_1,e_3,e_6\}$, $w^*(p_2^*) = 2$, $p_3^* = \{e_2,e_3,e_5\}$, $w^*(p_3^*) = 4$).

3 Algorithm

Our heuristic for Problem 1 consists of two phases. In its first phase, in each iteration, it tries to identify nontrivial null vectors (see Section 3.1), and then use them to partially decompose and update the graph and flow (see Section 3.2) in order to obtain a new graph with a smaller optimality gap. We expect that, the optimality gap can be reduced by at least 1 after each iteration, and the entire optimality gap can be closed after the first phase. Its second phase uses the greedy-width algorithm to decompose the remaining graph, which guarantees optimality if the optimality gap can be correctly closed in the first phase. The entire algorithm is formally described and analyzed in Section 3.4.

3.1 Identifying Nontrivial Null Vectors

The idea to identify nontrivial null vectors is based on Proposition 1 and Proposition 2, i.e., to compute vector $q \in Q$ satisfying that $f \cdot q = 0$ and that $q \notin \mathcal{N}_0$, where Q is a collection of candidate vectors that we guess in advance. We say a column vector $q \in \mathbb{R}^{|E|\times 1}$ is *simple* if all elements of q are in $\{-1,0,+1\}$. We choose Q as the set of all simple vectors. Notice that it could be the case that nontrivial null vectors exist, but none of them is simple (Figure 2). Let $Q_1 = \{q \in Q \mid f \cdot q = 0\}$. For any subset $E_1 \subseteq E$, we define $f(E_1) = \sum_{e\in E_1} f(e)$. We say two subsets $E_1, E_2 \subseteq E$ are *balanced* if $f(E_1) = f(E_2)$. For a vector $q \in Q$, we define $E_s(q) = \{e_j \in E \mid q[j] = +1\}$, and define $E_t(q) = \{e_j \in E \mid q[j] = -1\}$. It is clear that for any $q \in Q$, we have $q \in Q_1$ if and only if $f(E_s(q)) = f(E_t(q))$. We say a pair of balanced subsets E_1 and E_2 are *indivisible*, if there does not exist two strict subsets $E_1' \subset E_1$ and $E_2' \subset E_2$ such that $f(E_1') = f(E_2')$. Notice that two balanced subsets being indivisible implies that they are disjoint. We also say $q \in Q_1$ is *indivisible* if $E_s(q)$ and $E_t(q)$



(a)

Figure 2: In this example, P^* contains six paths, $\{p_1, p_2, \cdots, p_6\}$. The set of paths that go through each edge are labeled in the parenthesis. We also have $\Delta = 7$, thus there exists nontrivial null vectors, for example, $(2,-1,1,0,-1,1,0,-1,0)^T$. However, we can verify there does not exist simple ones for this example.

are indivisible. Let $Q_2 \subseteq Q_1$ be the subset that includes all indivisible vectors. We focus on computing indivisible and nontrivial null vectors, formally illustrated as Problem 2. We emphasize that $Q_2 \setminus \mathcal{N}_0$ is an approximation of $\mathcal{N}(P^*) \setminus \mathcal{N}_0$, i.e., vectors in $\mathcal{N}(P^*) \setminus \mathcal{N}_0$ are not necessarily in $Q_2 \setminus \mathcal{N}_0$ (because we restrict to simple vectors, see Figure 2), and vectors in $Q_2 \setminus \mathcal{N}_0$ are not necessarily in $\mathcal{N}(P^*) \setminus \mathcal{N}_0$ (because $f \cdot q = 0$ is not a sufficient condition).

Problem 2 *Given G and f, compute $Q_2 \setminus \mathcal{N}_0$.*

We give a heuristic for Problem 2. We first devise a dynamic programming heuristic to compute Q_2 (Algorithm 1), which extends the pseudo-polynomial-time algorithm for the subset-sum problem, then we devise an algorithm (Algorithm 2) based on the properties of Q_2 and $\mathcal{N}_0$ to filter out vectors in $\mathcal{N}_0$.

Algorithm 1: Heuristic to compute Q_2

Input: G and f (we define $|f| = \sum_{e \in E: e=(s,v)} f(e) = \sum_{e \in E: e=(v,t)} f(e)$, i.e., the value of f)

Output: $Q_2^\dagger$

1. Let $A \in \{0,1\}^{|E| \times |f|}$: $A[k,x] = 1$ if there exists $E_1 \subseteq \{e_1, \cdots, e_k\}$ with $f(E_1) = x$.
2. Let $B \in \{0,1\}^{|E| \times |f|}$: $B[k,x] = 1$ if there exists distinct $E_1, E_2 \subseteq \{e_1, \cdots, e_k\}$ with $f(E_1) = f(E_2) = x$.
3. Compute A with the recursion: $A[k,x] = 1$ if and only if $A[k-1,x] = 1$, or $A[k-1,x-f(e_k)] = 1$.
4. Compute B with the recursion: $B[k,x] = 1$ if and only if $B[k-1,x] = 1$, or $B[k-1,x-f(e_k)] = 1$, or $A[k-1,x] = 1$ and $A[k-1,x-f(e_k)] = 1$.
5. Compute $C = \{1 \le x \le |f| \mid B[|E|,x] = 1\}$.
6. Compute $D = \{x \in C \mid B[|E|,x-f(e)] = 0, \forall e \in E\}$.
7. For every $x \in D$, retrieve the two balanced subsets through tracing back matrices A and B, and then add the corresponding vector to $Q_2^\dagger$.

By the definition of matrix B, for each $x \in C$, there exists two subsets E_1 and E_2 such that $f(E_1) = f(E_2) = x$, but these two subsets might be not indivisible (or even not disjoint). We can prove that the two balanced subsets through tracing back from $x \in D$ are guaranteed indivisible (see proof of Proposition 5).

Proposition 5 *We have that $Q_2^\dagger$, i.e., the set returned by Algorithm 1, is a subset of $Q_2^\dagger$. Algorithm 1 runs in $O(|E| \cdot |f|)$ time.*

The remaining task is to compute $Q_2^\dagger \setminus \mathcal{N}_0 = Q_2^\dagger \setminus (Q_2^\dagger \cap \mathcal{N}_0)$. The following proposition gives an equivalent condition to decide whether a vector $q \in Q_2$ is in $\mathcal{N}_0$ or not. We say a subset $V_1 \subseteq V$ is *connected* if the subgraph of G induced by V_1 is weakly connected.

Proposition 6 *Let $q \in Q_2$. We have that $q \in \mathcal{N}_0$ if and only if there exists a connected subset $V_1 \subseteq V_0$ such that $q = \sum_{v \in V_1} q_v$ or $q = -\sum_{v \in V_1} q_v$.*

For a subset $V_1 \subseteq V_0$, we define $E_s(V_1)$ and $E_t(V_1)$ the sets of edges corresponding to cut $(V_1, V \setminus V_1)$, i.e., $E_s(V_1) = \{(u,v) \in E \mid u \in V \setminus V_1, v \in V_1\}$ and $E_t(V_1) = \{(u,v) \in E \mid u \in V_1, v \in V \setminus V_1\}$. Following Proposition 6, we have the following result.

Proposition 7 *Let $q \in \mathcal{Q}_2$. We have that $q \in \mathcal{N}_0$ if and only if there exists a connected subset $V_1 \subseteq V_0$ such that $E_s(q) = E_s(V_1)$ and $E_t(q) = E_t(V_1)$, or $E_s(q) = E_t(V_1)$ and $E_t(q) = E_s(V_1)$.*

Proposition 7 transforms the problem of deciding whether a vector $q \in \mathcal{Q}_2^\dagger$ is also in $\mathcal{N}_0$ into the problem of examining whether $E_s(q)$ and $E_t(q)$ form a set of cut edges, which can be accomplished through two rounds of breadth-first searching (Algorithm 2).

Algorithm 2: Exact algorithm to compute $\mathcal{Q}_2^\dagger \setminus \mathcal{N}_0$

Input: $\mathcal{Q}_2^\dagger \subseteq \mathcal{Q}_2$, G
Output: $\mathcal{Q}_2^\dagger \setminus \mathcal{N}_0$

1. **FOREACH** $q \in \mathcal{Q}_2^\dagger$, **DO** the following procedure for both $q' = q$ and $q' = -q$
2. Traverse G from edges in $E_s(q')$, stop searching when reaching edges in $E_t(q')$. **IF** all searching ends up with edges in $E_t(q')$, i.e., t is not reached, **THEN** add q to $\mathcal{Q}_2^\dagger \setminus \mathcal{N}_0$.
3. Traverse G reversely from edges in $E_t(q')$, stop searching when reaching edges in $E_s(q')$. **IF** all searching ends up with edges in $E_s(q')$, i.e., s is not reached, **THEN** add q to $\mathcal{Q}_2^\dagger \setminus \mathcal{N}_0$.

Algorithm 2 also runs in $O(|E| \cdot |f|)$ time, since $|\mathcal{Q}_2^\dagger| \leq |f|$ and for each $q \in \mathcal{Q}_2^\dagger$, Algorithm 2 takes $O(|E|)$ time to check its membership in $\mathcal{N}_0$. Thus, our heuristic for Problem 2, which consists of Algorithm 1 followed by Algorithm 2, also runs in $O(|E| \cdot |f|)$ time.

3.2 Reducing the Optimality Gap

In this section, we devise a heuristic (Algorithm 3) to use $q \in \mathcal{Q}_2^\dagger \setminus \mathcal{N}_0$ to reduce the optimality gap. The idea of is to iteratively choose one edge from $E_s(q)$ and one edge from $E_t(q)$, merge them through updating G, f and q, until finally q becomes 0.

Algorithm 3: Heuristic to reduce the optimality gap

Input: $q \in \mathcal{Q}_2^\dagger \setminus \mathcal{N}_0$, G and f
Output: **TRUE** and updated G and f if the optimality gap is reduced, **FALSE** otherwise

1. **WHILE** $q \neq 0$
2. Arbitrarily choose $e_i = (u_1, v_1) \in E_s(q)$ and $e_j = (u_2, v_2) \in E_t(q)$ that are connected; if no such pair can be found, **RETURN FALSE**. Assume a path from v_1 to u_2 exists; also assume $f(e_i) \leq f(e_j)$.
3. **IF** $v_1 = u_2$ (see Figure 3**(a)**)
4. Add edge $e_k = (u_1, v_2)$ to G with flow value of $f(e_i)$; set $q[k] = 0$; remove e_i; set $q[i] = 0$.
5. Remove e_j and set $q[j] = 0$ if $f(e_i) = f(e_j)$, otherwise set $f(e_j)$ as $f(e_j) - f(e_i)$.
6. **ELSE** (see Figure 3**(b)**)
7. Apply Algorithm 6 (Section 3.3) to make e_i and e_j adjacent. **IF** this succeeds, **GOTO** step 3.
8. Arbitrarily choose a path p from v_1 to u_2 satisfying that all edges in p have a flow value of at least $f(e_i)$. We assume that p does not contain edges in $E_s(q) \cup E_t(q)$, otherwise we switch to merge this closer pair of edges. **IF** such path cannot be found, **RETURN FALSE**.
9. Add edge $e_k = (u_1, v_2)$ to G with flow value of $f(e_i)$; set $q[k] = 0$; remove e_i; set $q[i] = 0$.
10. For edge e_j and every edge in p, subtract their flow value by $f(e_i)$; if their updated flow values become 0, then remove them from the graph and set $q[\cdot] = 0$ for them.
11. **RETURN TRUE**.

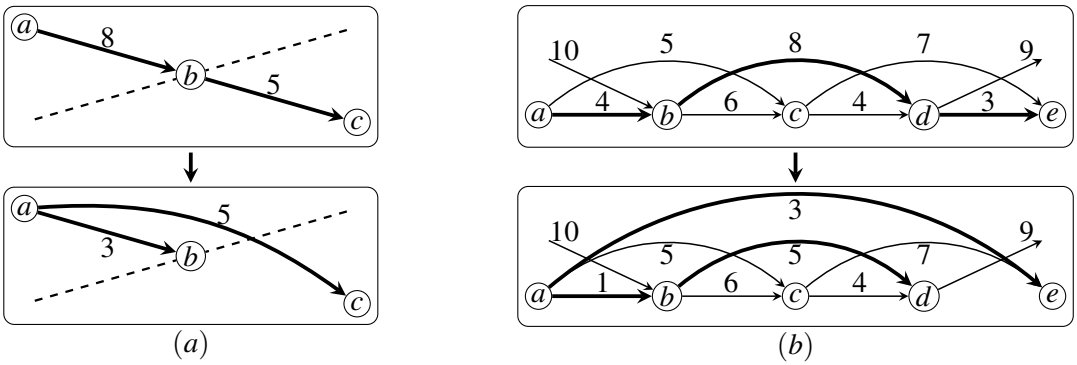


Figure 3: **(a)** Merging two adjacent edges. **(b)** Merging two distant edges (a,b) and (d,e).

Suppose that Algorithm 3 succeeds after n iterations. Let q^k, $G^k = (V^k, E^k)$ and f^k be the updated vector, graph, and flow, respectively, after it finishes k iterations, $1 \leq k \leq n$. Notice that $q^n = 0$. Similar to $\mathcal{N}_0$ and Q_2, we also define $\mathcal{N}_0^k$ and Q_2^k *w.r.t.* G^k and f^k. Let (P^{k*}, w^{k*}) be one optimal decomposition of (G^k, f^k), and also define $\Delta^k = |E^k| - |V^k| + 2$, $1 \leq k \leq n$. Assume that $G^0 = G$ and $f^0 = f$ for sake of simplicity.

Proposition 8 *If Algorithm 3 succeeds, then we have* $q^k \in Q_2^k \setminus \mathcal{N}_0^k$, $1 \leq k \leq n-1$.

Proposition 9 *If Algorithm 3 succeeds, then we have* $n = |E_s(q)| + |E_t(q)| - 1$.

We emphasize that the merging process in Algorithm 3 narrows down the solution space. Specifically, the set of all possible decompositions of (G^k, f^k) is a subset of that of (G^{k-1}, f^{k-1}), $1 \leq k \leq n-1$, i.e., every decomposition of (G^k, f^k) corresponds to a decomposition of (G^{k-1}, f^{k-1}). This implies that $|P^{k*}| \geq |P^{(k-1)*}|$. Notice that it is possible that $|P^{k*}| > |P^{(k-1)*}|$, i.e., optimal solutions get eliminated during Algorithm 3. Figure 4 shows that merging two adjacent edges (lines 3–5 of Algorithm 3) might break optimality.

We now describe our central result, which states that if Algorithm 3 succeeds, then the optimality gap will be reduced by at least 1. The intuition behind the proof is that during the first $(n-1)$ iterations, $(|E| - |V|)$ keeps unchanged or decreases, while at the last iteration, this value shall be reduced by at least 1.

Proposition 10 *If Algorithm 3 succeeds, then we have* $\Delta^n - |P^{n*}| \leq \Delta - |P^*| - 1$.

3.3 Making Two Distant Edges Adjacent

We now describe an algorithm (Algorithm 6) to try to make two given distant edges adjacent while keeping optimality, a key subroutine used in Algorithm 3 (line 7). In this Section, we describe the intuition and idea of this algorithm, and its formal and complete description is in Appendix A. We first characterize the hierarchical structure of the graph. We say a pair of vertices u and v is *closed*, if there does not exist any edge satisfying that exactly one of its ends is in the subgraph induced by these two vertices (see Figure 5 **(a,c)**).

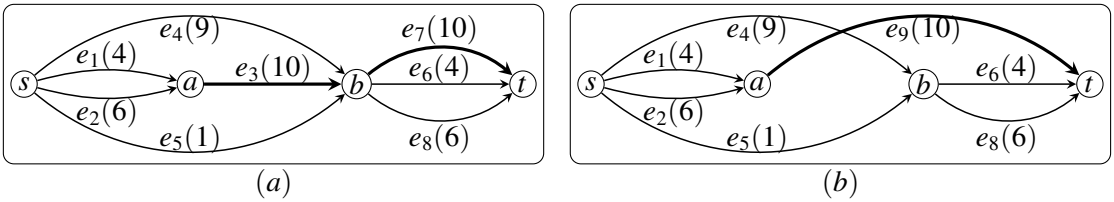


Figure 4: Merging adjacent edges might break optimality. **(a)** Original graph with 4 paths in the optimal decomposition. **(b)** Graph after merging e_3 and e_7, with 5 paths in the optimal decomposition.

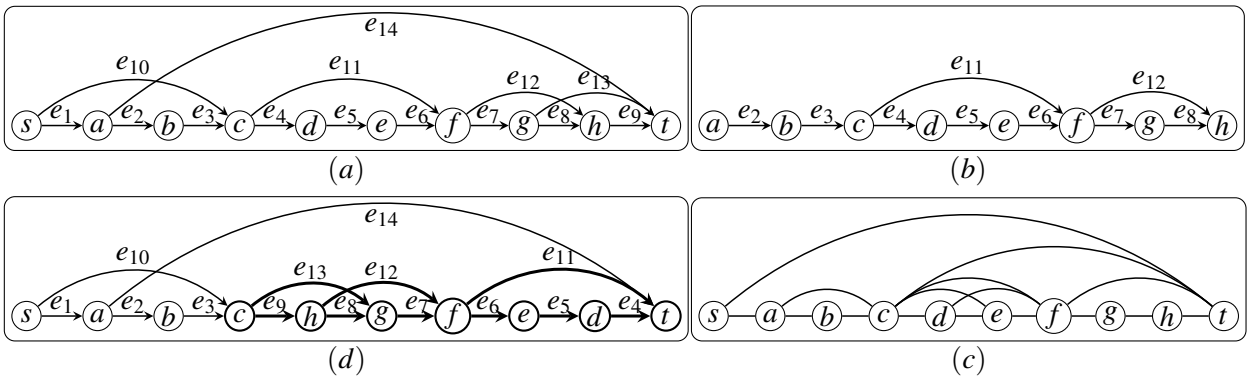


Figure 5: **(a)** Illustration of $G = (V,E)$. **(b)** Illustration of $V(a,h)$ and $E(a,h)$. **(c)** Edges represent all closed pairs (i.e., $\mathcal{C}$). **(d)** Graph after reversing closed pair of (c,t). Notice that edges $\{e_3,e_{10}\}$ and $\{e_9,e_{13}\}$ become adjacent, which are distant before reversing.

With a given pair of closed vertices, we then define a *reverse* operation to modify the graph (Figure 5 **(a,d)**). We can prove that, performing reverse operations will not change the optimality (Proposition 12). Thus, we focus on the problem of deciding whether there exists a series of reverse operations to make two given distant edges adjacent (Problem 3).

We give a polynomail-time exact algorithm to solve Problem 3 (Algorithm 6). The algorithm first use a DAG to organize all closed pairs (Figure 6). Then through traversing the DAG starting from the vertices corresponding to the two given distant edges, we can decide whether these two edges can be made adjacent, and if so the reverse operations can be retrieved by tracing back the DAG.

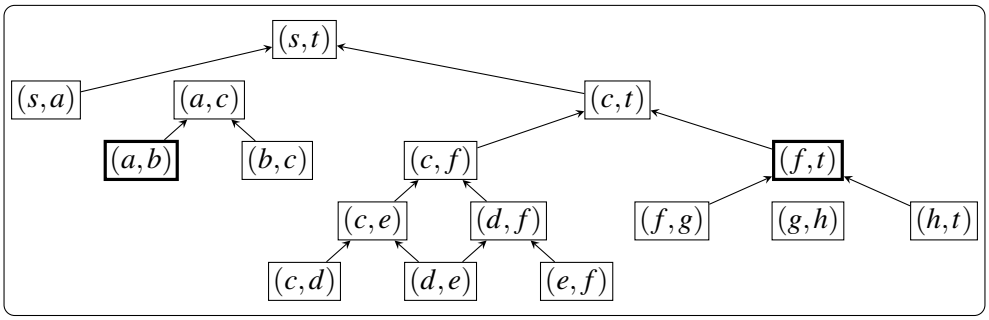


Figure 6: $\overline{H}(\mathcal{C})$ *w.r.t.* the graph in Figure 5 **(a)**. To make edge $e_i = (a,b)$ and edge $e_j = (f,h)$ adjacent, Algorithm 6 first computes their closures in $H(\mathcal{C})$, which are (a,b) and (f,t) respectively. The following traversing procedure gives $S_1 = \{a\}$, $T_1 = \{b,c\}$, $S_2 = \{f,c,s\}$, and $T_2 = \{t\}$. We have $T_1 \cap S_2 = c$, i.e., e_i and e_j can be made adjacent at c, and the operations are reversing (a,c), (f,t) and (c,t).

3.4 The Complete Algorithm

With all above components available, we now formally state our heuristic for Problem 1 in Algorithm 4. The first phase (lines 1 to 3) is to iteratively reduce the optimality gap. It then uses the greedy-width algorithm (proposed in [24]) to fully decompose the remaining graph into paths. Notice that these paths are *w.r.t.* the updated graph, and we need to recover the paths corresponding to the original graph (we maintain the tracing back information when updating the graph).

Algorithm 4: Heuristic for Problem 1

Input: G and f

Output: A decomposition $(P^\dagger, w^\dagger)$ of (G, f)

1. Apply Algorithm 1 and Algorithm 2 to compute $\mathcal{Q}_2^\dagger \setminus \mathcal{N}_0$. **IF** $\mathcal{Q}_2^\dagger \setminus \mathcal{N}_0 = \emptyset$, **GOTO** step 4.
2. **FOREACH** $q \in \mathcal{Q}_2^\dagger \setminus \mathcal{N}_0$
3. Apply Algorithm 3 with q. **IF** it succeeds, **GOTO** step 1.
4. Apply the greedy-width algorithm on the updated graph.
5. For the paths returned in line 4, recover the corresponding paths *w.r.t.* the original graph.

Proposition 11 *Algorithm 4 runs in* $O(|V|^2 \cdot |E|^2 \cdot |f|)$ *time.*

From [24], we know that for an instance with optimality gap of 0, the greedy-width algorithm can guarantee returning one optimal decomposition. This implies that, for any instance, if the first phase of Algorithm 4 closes its optimality gap and does not break the optimality, then Algorithm 4 shall return an optimal decomposition for this instance. We use experimental studies to illustrate its performance in Section 4.

4 Experimental Results

We first describe the experiments in the context of transcript assembly. We use four datasets. The first dataset includes 1000 human RNA-seq samples from SRA (Sequence Reads Archive). For each gene in each sample, we build an instance (i.e., G and f) as follows. We first use Salmon [30] to identify and quantify the expressed transcripts of this gene: the expressed transcripts shall be then used as the ground truth paths, denoted as $P^\ddagger$, while the expression abundances of these transcripts shall be used as the ground truth weights, denoted as $w^\ddagger$. With $(P^\ddagger, w^\ddagger)$, we then construct G and f: vertices of G correspond exons, edges are the union of the edges in the ground truth paths, and the flow value for each edge is the superposition of the ground truth weights of the paths that go through this edge (Figure 7). For the generated instance (G, f), we pipe it into the two algorithms (Algorithm 4 and greedy-width). We denote by $(P^\dagger, w^\dagger)$ as the predicted paths and weights by any of these two algorithms. We say $(P^\dagger, w^\dagger)$ is *correct* if we have $|P^\dagger| \leq |P^\ddagger|$. Notice that $(P^\ddagger, w^\ddagger)$ is not necessarily optimal, but only gives an upper bound, i.e., $|P^\ddagger| \geq |P^*|$. The other three datasets are obtained by simulation using Flux-Simulator [31] on three well-annotated species, human, mouse and zebrafish. For each species, we independently simulate 100 samples. For each gene in each sample, we collect the ground truth $(P^\ddagger, w^\ddagger)$ from Flux-Simulator, and use them to build the instance G and f (Figure 7).

Based on $|P^\ddagger|$, for each dataset, we classify all instances into 10 categories, i.e., $|P^\ddagger| \in \{1, 2, \cdots, 10\}$, and remove those instances with $|P^\ddagger| > 10$ (which is a tiny portion). For each category, we run both algorithms on all instances and compare their *accuracy* (proportion of instances that are predicted correctly), and average ratio between $|P^\dagger|$ and $|P^\ddagger|$ (i.e., mean value of $|P^\dagger|/|P^\ddagger|$ over all instances).

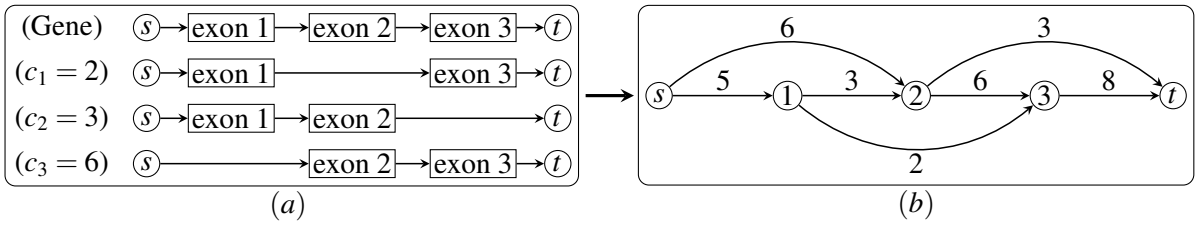


Figure 7: **(a)** Transcripts and their expression abundances (c_k) of a gene. **(b)** The splice graph G and flow f derived from the ground truth in **(a)**.

The results are shown in Figure 8. We can observe that Algorithm 4 gives very high accuracy on all instances, while the accuracy of the greedy-width algorithm decreases rapidly as $|P^{\ddagger}|$ increases. Meanwhile, the decompositions given by the greedy-width algorithm uses more paths and the ratio between the number of predicted paths and the ground truth keeps increasing as $|P^{\ddagger}|$ increases, while for Algorithm 4 such ratio constantly remains closing to 1. These results demonstrate that in the application of transcript assembly, Algorithm 4 significantly outperforms the greedy-width algorithm.

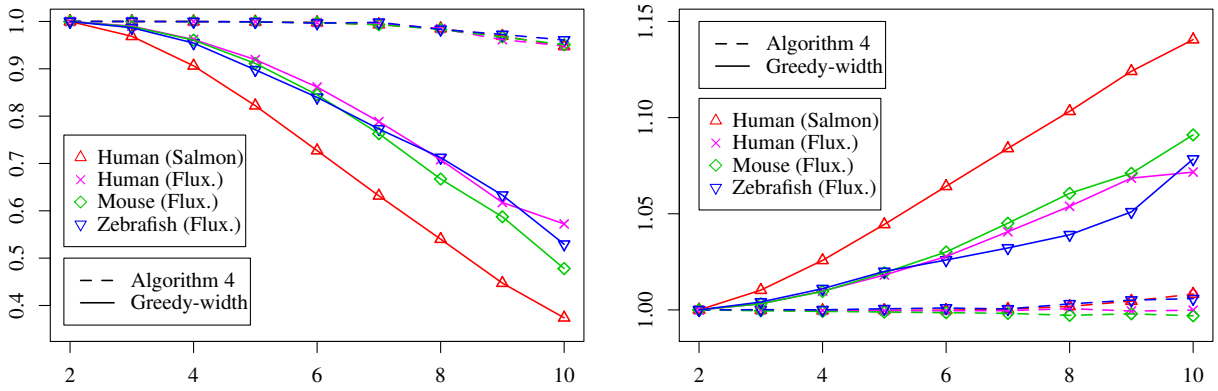


Figure 8: Comparison of the two algorithm on transcript assembly. **Left:** x-axis is $|P^{\ddagger}|$ and y-axis is the accuracy. **Right:** x-axis is $|P^{\ddagger}|$ and y-axis is the average ratio between $|P^{\dagger}|/|P^{\ddagger}|$.

To evaluate the performance of our algorithm on large graphs, we carry out another set of experiments using simulated random instances. The results show that our algorithm can give very high accuracy for a wide spectrum of parameter combinations, while the greedy-width algorithm works well only for easy instances. These results further demonstrate the significant improvement of our algorithm over the greedy-width algorithm. The simulation setup and the detailed results are in Appendix B.

5 Discussion

We give an efficient heuristic for the minimum path flow decomposition problem. The components of this heuristic can be further improved. For example, we shall study how to reduce the optimality gap while keep optimality if a true nontrivial null vector is given, and also study how to identify nontrivial null vectors that are not simple as well as how to use such vectors to reduce the optimality gap. We tackle here only the flow decomposition step of transcript assebmly. Future work also includes extending our algorithm to develop a complete transcript assembler.

Acknowledgements

We thank Meiyue Shao and Guillaume Marçais for helpful discussions and suggestions. This research is funded in part by the Gordon and Betty Moore Foundation's Data-Driven Discovery Initiative through Grant GBMF4554 to Carl Kingsford, by the US National Science Foundation (CCF-1256087, CCF-1319998) and by the US National Institutes of Health (R21HG006913, R01HG007104). C.K. received support as an Alfred P. Sloan Research Fellow.

References

[1] Z. Wang, M. Gerstein, and M. Snyder. RNA-Seq: a revolutionary tool for transcriptomics. *Nat. Rev. Genet.*, 10(1):57–63, 2009.

[2] C. Trapnell, B.A. Williams, G. Pertea, A. Mortazavi, G. Kwan, M.J. Van Baren, S.L. Salzberg, B.J. Wold, and L. Pachter. Transcript assembly and quantification by RNA-Seq reveals unannotated transcripts and isoform switching during cell differentiation. *Nat. Biotechnol.*, 28(5):511–515, 2010.

[3] M. Guttman, M. Garber, J.Z. Levin, J. Donaghey, J. Robinson, X. Adiconis, L. Fan, et al. Ab initio reconstruction of cell type-specific transcriptomes in mouse reveals the conserved multi-exonic structure of lincrnas. *Nat. Biotechnol.*, 28(5):503–510, 2010.

[4] W. Li, J. Feng, and T. Jiang. IsoLasso: a LASSO regression approach to RNA-Seq based transcriptome assembly. *J. Comput. Biol.*, 18(11):1693–1707, 2011.

[5] J.J. Li, C.-R. Jiang, J.B. Brown, H. Huang, and P.J. Bickel. Sparse linear modeling of next-generation mRNA sequencing (RNA-Seq) data for isoform discovery and abundance estimation. *Proc. Natl. Acad. Sci. USA*, 108(50):19867–19872, 2011.

[6] Y.-Y. Lin, P. Dao, F. Hach, M. Bakhshi, F. Mo, A. Lapuk, C. Collins, and S.C. Sahinalp. CLIIQ: Accurate comparative detection and quantification of expressed isoforms in a population. In *Proc. 12th Workshop Algs. in Bioinf. (WABI'12)*, volume 7534 of *Lecture Notes in Comp. Sci.*, pages 178–189, 2012.

[7] W. Li and T. Jiang. Transcriptome assembly and isoform expression level estimation from biased RNA-Seq reads. *Bioinformatics*, 28(22):2914–2921, 2012.

[8] J. Behr, A. Kahles, Y. Zhong, V.T. Sreedharan, P. Drewe, and G. Rätsch. MITIE: Simultaneous RNA-Seq-based transcript identification and quantification in multiple samples. *Bioinformatics*, 29(20):2529–2538, 2013.

[9] A.M. Mezlini, E.J.M. Smith, M. Fiume, O. Buske, G.L. Savich, et al. iReckon: Simultaneous isoform discovery and abundance estimation from RNA-seq data. *Genome Res.*, 23(3):519–529, 2013.

[10] A.I. Tomescu, A. Kuosmanen, R. Rizzi, and V. Mäkinen. A novel min-cost flow method for estimating transcript expression with RNA-Seq. *BMC Bioinformatics*, 14(5):1, 2013.

[11] M. Pertea, G.M. Pertea, C.M. Antonescu, T.-C. Chang, J.T. Mendell, and S.L. Salzberg. StringTie enables improved reconstruction of a transcriptome from RNA-seq reads. *Nat. Biotechnol.*, 33(3):290–295, 2015.

[12] G. Robertson, J. Schein, R. Chiu, R. Corbett, M. Field, S.D. Jackman, K. Mungall, S. Lee, H.M. Okada, J.Q. Qian, et al. De novo assembly and analysis of RNA-seq data. *Nat. Methods*, 7(11):909–912, 2010.

[13] J. Martin, V.M. Bruno, Z. Fang, X. Meng, M. Blow, T. Zhang, G. Sherlock, M. Snyder, and Z. Wang. Rnnotator: an automated de novo transcriptome assembly pipeline from stranded RNA-Seq reads. *BMC Genomics*, 11(1):1, 2010.

[14] M.G. Grabherr, B.J. Haas, M. Yassour, J.Z. Levin, D.A. Thompson, I. Amit, X. Adiconis, et al. Trinity: reconstructing a full-length transcriptome without a genome from RNA-Seq data. *Nat. Biotechnol.*, 29(7):644, 2011.

[15] Y. Xie, G. Wu, J. Tang, R. Luo, J. Patterson, S. Liu, et al. SOAPdenovo-Trans: de novo transcriptome assembly with short RNA-Seq reads. *Bioinformatics*, 30(12):1660–1666, 2014.

[16] D.R. Zerbino and E. Birney. Velvet: algorithms for de novo short read assembly using de Bruijn graphs. *Genome Res.*, 18(5):821–829, 2008.

[17] M.H. Schulz, D.R. Zerbino, M. Vingron, and E. Birney. Oases: robust de novo RNA-seq assembly across the dynamic range of expression levels. *Bioinformatics*, 28(8):1086–1092, 2012.

[18] T. Steijger, J.F. Abril, P.G. Engström, F. Kokocinski, T.J. Hubbard, et al. Assessment of transcript reconstruction methods for RNA-seq. *Nat. Methods*, 10(12):1177–1184, 2013.

[19] K.E. Hayer, A. Pizarro, N.F. Lahens, J.B. Hogenesch, and G.R. Grant. Benchmark analysis of algorithms for determining and quantifying full-length mRNA splice forms from RNA-seq data. *Bioinformatics*, 31(24):3938–3945, 2015.

[20] D. Kim, G. Pertea, C. Trapnell, H. Pimentel, R. Kelley, S.L. Salzberg, et al. TopHat2: accurate alignment of transcriptomes in the presence of insertions, deletions and gene fusions. *Genome Biol.*, 14(4):R36, 2013.

[21] A. Dobin, C.A. Davis, F. Schlesinger, J. Drenkow, C. Zaleski, S. Jha, P. Batut, M. Chaisson, and T.R. Gingeras. STAR: ultrafast universal RNA-seq aligner. *Bioinformatics*, 29(1):15–21, 2013.

[22] D. Kim, B. Langmead, and S.L. Salzberg. HISAT: a fast spliced aligner with low memory requirements. *Nat. Methods*, 12(4):357–360, 2015.

[23] M. Pop. Genome assembly reborn: recent computational challenges. *Brief. Bioinform.*, 10(4):354–366, 2009.

[24] B. Vatinlen, F. Chauvet, P. Chrétienne, and P. Mahey. Simple bounds and greedy algorithms for decomposing a flow into a minimal set of paths. *Eur. J. Oper. Res.*, 185(3):1390–1401, 2008.

[25] T. Hartman, A. Hassidim, H. Kaplan, D. Raz, and M. Segalov. How to split a flow? In *Proc. IEEE INFOCOM 2012*, pages 828–836, 2012.

[26] B. Mumey, S. Shahmohammadi, K. McManus, and S. Yaw. Parity balancing path flow decomposition and routing. In *Proc. IEEE Globecom Workshops*, pages 1–6, 2015.

[27] Y. Hendel and W. Kubiak. Decomposition of flow into paths to minimize their length. `http://sites.google.com/site/yannhendel/Decomposition.pdf`.

[28] G. Baier, E. Köhler, and M. Skutella. On the k-splittable flow problem. In *Proc. 10th Eur. Symp. Algs. (ESA'02)*, pages 101–113, 2002.

[29] Georg Baier, E. Köhler, and M. Skutella. The k-splittable flow problem. *Algorithmica*, 42(3):231–248, 2005.

[30] R. Patro, G. Duggal, and C. Kingsford. Salmon: Accurate, versatile and ultrafast quantification from RNA-seq data using lightweight-alignment. *bioRxiv*, page 021592, 2015.

[31] T. Griebel, B. Zacher, P. Ribeca, E. Raineri, V. Lacroix, R. Guigó, and M. Sammeth. Modelling and simulating generic RNA-seq experiments with the flux simulator. *Nucleic Acids Res.*, 40(20):10073–10083, 2012.

Appendix A Making Two Distant Edges Adjacent

In this Section, we describe an algorithm to make two given distant edges adjacent while keeping optimality. We first give some definitions to characterize the hierarchical structure of G. We say a pair of vertices (u,v) is *connected* (*w.r.t.* graph G), if there exists a path from u to v in G. For a pair of connected vertices (u,v), we define a subset $V(u,v) \subseteq V$ as the set of vertices that are on some path from u to v (see Figure 5 **(a,c)**). Notice that if (u,v) is connected, we always have $\{u,v\} \subseteq V(u,v)$. For a pair of connected vertices (u,v), we also define a subset $E(u,v) \subseteq E$ as follows (see Figure 5): for any $e \in E$ we say $e \in E(u,v)$ if the both ends of e are in $V(u,v)$. We say a pair of connected vertices (u,v) is *closed*, if for every vertex $w \in V(u,v) \setminus \{u,v\}$, all incident edges of w in E are also in $E(u,v)$. We denote by $\mathcal{C}$ the set of all closed pairs. For each pair $(u,v) \in \mathcal{C}$, we define $E_s(u,v) \subseteq E(u,v)$ as the set of edges in $E(u,v)$ that are adjacent to u, and define $E_t(u,v) \subseteq E(u,v)$ as the set of edges in $E(u,v)$ that are adjacent to v.

We define a *reverse* operation to modify G (Figure 5 **(a,d)**). For each pair $(u,v) \in \mathcal{C}$, we can reverse it by first detaching edges in $E_s(u,v)$ from u and attaching them to v, detaching edges in $E_t(u,v)$ from v and attaching them to u, and then switching the direction of all edges in $E(u,v)$. Notice that we treat the reverse operation in a way that the set of edges in G is kept unchanged, but the locations and orientations of edges might be changed. Formally, let G' be the graph after reversing $(u,v) \in \mathcal{C}$. We assume that G' still have the edge set of E. But for each edge $e_i = (u,v)$ in G, it might become $e_i = (u',v')$ in G' with $u' \neq u$ and/or $v' \neq v$. In other words, all edges are recognized by their original identities in E, and their adjacent two vertices might become different after performing reverse operations.

Let G' be the graph after reversing $(u,v) \in \mathcal{C}$. Let (P'^*, w'^*) be one optimal decomposition of (G', f). The following result states that solving Problem 1 on (G,f) is equivalent to solving it on (G',f).

Proposition 12 *There exists a one-to-one correspondence between all decompositions of (G,f) and all decompositions of (G',f). In particular, we have that $|P^*| = |P'^*|$.*

Based on Proposition 12, we address the following problem.

Problem 3 *Given two edges $e_i, e_j \in E$, decide whether we can perform a series of reverse operations, such that in the resulting graph we have $e_i = (u_1, v_1)$ and $e_j = (u_2, v_2)$ with either $v_1 = u_2$ or $v_2 = u_1$.*

To solve Problem 3, we first describe a data structure to organize all the reverse operations. We define a binary relation "$\preceq$" on $\mathcal{C}$: for two pairs $(u_1,v_1),(u_2,v_2) \in \mathcal{C}$, we say $(u_1,v_1) \preceq (u_2,v_2)$ if $V(u_1,v_1) \subseteq V(u_2,v_2)$. It is easy to verify that this relation forms a partial order. Let $H(\mathcal{C})$ be the Hasse diagram on $\mathcal{C}$ *w.r.t.* this partial order. For each edge in $H(\mathcal{C})$ from (u_1,v_1) to (u_2,v_2), we remove it from $H(\mathcal{C})$ if $u_1 \neq u_2$ and $v_1 \neq v_2$. We denote by $\overline{H}(\mathcal{C})$ this updated graph (see Figure 6).

We now describe an exact algorithm to solve Problem 3, which requires constructing $\overline{H}(\mathcal{C})$ and then searching series of reverse operations guided by $\overline{H}(\mathcal{C})$. We first compute the transitive closure of G, which gives us all possible connected pairs. For each connected pair (u,v), we follow the definition to check whether this pair is in $\mathcal{C}$: we compute $V(u,v)$, which can be done in $O(|V|)$ time with the availability of all connected pairs; then for each vertex $w \in V(u,v) \setminus \{u,v\}$ we check whether all adjacent edges of w are still in $V(u,v)$. It takes $O(|E|)$ time to verify each connected pair, and thus the total running time to compute $\mathcal{C}$ is $O(|V|^2 \cdot |E|)$.

We then give an algorithm to compute $H(\mathcal{C})$, i.e., for each pair in $\mathcal{C}$ we need to compute its parents node in $H(\mathcal{C})$. For a subset $V_1 \subseteq V$, $|V_1| \geq 2$, we say $(u,v) \in \mathcal{C}$ is a *closure* of V_1, if we have $V_1 \subseteq V(u,v)$, and there does not exist $(u',v') \in \mathcal{C}$ such that $V_1 \subseteq V(u',v') \subset V(u,v)$. Intuitively, the closure of a given set of vertices is the smallest closed pair whose set of vertices contains the given vertices. As we will see in Proposition 13 such closure is unique. Now we describe an exact algorithm to compute the closure of a given subset, which

Algorithm 5: Exact algorithm to compute closure

Input: $V_1 \subset V$, $\mathcal{C}$ and G
Output: Closure of V_1

1. Compute a topological ordering O of V, i.e., if (u,v) is connected, then $O(u) < O(v)$.
2. Let S be the set containing all vertices in the final closure, initialized as V_1.
3. Let L and R be the smallest and largest vertex in S according to their orderings in O, respectively.
4. Let Q be a queue that stores all (unexamined) vertices in $S \setminus \{L,R\}$, initialized as $S \setminus \{L,R\}$.
5. **REPEAT**
6. **WHILE** Q is not empty
7. Pop element v from Q.
8. **FOREACH** adjacent vertex u of v that is not in S
9. Insert u into S.
10. **IF** $O(u) < O(L)$, push L to Q and set $L = u$.
11. **IF** $O(u) > O(R)$, push R to Q and set $R = u$.
12. **FOREACH** adjacent vertex u of L that is not in S
13. **IF** $(u,R) \in \mathcal{C}$, insert u into S and push u to Q.
14. **IF** Q is empty, **RETURN** (L,R).

shall be used in both constructing $H(\mathcal{C})$ and the following steps (line 2 of Algorithm 6). The idea of this algorithm is to iteratively add necessary vertices until finally it becomes closed.

Proposition 13 *For any $V_1 \subset V$, $|V_1| \geq 2$, Algorithm 5 returns the unique closure of V_1 in $O(|E|)$ time.*

We now describe how to build $H(\mathcal{C})$ using Algorithm 5. Suppose that (u',v') is one of the direct parents of $(u,v) \in \mathcal{C}$ in $H(\mathcal{C})$. We have two cases: either $u \notin \{u',v'\}$, or $v \notin \{u',v'\}$. If it is the first case, then (u',v') is exactly the closure of $V_1 = \{u,v\} \cup \{w \mid (w,u) \in E\} \cup \{w \mid (u,w) \in E\}$. If it is the second case, then (u',v') is exactly the closure of $V_2 = \{u,v\} \cup \{w \mid (w,v) \in E\} \cup \{w \mid (v,w) \in E\}$. In other words, the parents of $(u,v) \in \mathcal{C}$ in $H(\mathcal{C})$ are exactly the closures of V_1 and V_2. For each pair $(u,v) \in \mathcal{C}$, we use this procedure to compute its parents, and thus the whole graph $H(\mathcal{C})$ can be constructed. To further build $\overline{H}(\mathcal{C})$, by definition, we then check each edge of $H(\mathcal{C})$ to see whether the two pairs corresponding to its two adjacent vertices share the same starting or ending vertex. The total running time to build $\overline{H}(\mathcal{C})$ is $O(|V|^2 \cdot |E|)$.

Algorithm 6: Exact algorithm for Problem 3

Input: G, $e_i = (u_1,v_1), e_j = (u_2,v_2) \in E$
Output: **TRUE** and series of reverse operations if they can be made adjacent, **FALSE** otherwise

1. Compute $\overline{H}(\mathcal{C})$.
2. Compute the closures of $\{u_1,v_1\}$ and $\{u_2,v_2\}$; denote them as (u_1',v_1') and (u_2',v_2'), respectively.
3. Initialize $S_1 = \{u_1\}$, $T_1 = \{v_1\}$, $S_2 = \{u_2\}$, $T_2 = \{v_2\}$.
4. Traverse $\overline{H}(C)$ from (u_1',v_1'): for each visited node $(u,v) \in \overline{H}(\mathcal{C})$, insert u to S_1 and insert v to T_1.
5. Traverse $\overline{H}(C)$ from (u_2',v_2'): for each visited node $(u,v) \in \overline{H}(\mathcal{C})$, insert u to S_2 and insert v to T_2.
6. **IF** $S_1 \cap T_2 = \emptyset$ and $S_2 \cap T_1 = \emptyset$, **RETURN FALSE**.
7. Compute arbitrarily $w \in (S_1 \cap T_2) \cup (S_2 \cap T_1)$; assume that $w \in S_2 \cap T_1$.
8. Compute any path from (u_1',v_1') to node $(\cdot,w)$, and traverse this path to construct the reverse operations to make e_i point to w: let (u,v) be the current node (i.e., e_i is either points to v or leaves u) and u be the boundary vertex shared by (u,v) and its parent; reverse (u,v) if e_i is not adjacent to u; move to the parent of (u,v) on this path. In the end reverse $(\cdot,w)$ if e_i is not adjacent to w.
9. Perform the same procedure as line 8 to construct the reverse operations to make e_j leave w.
10. **RETURN TRUE**.

Our complete algorithm to solve Problem 3 is described in Algorithm 6. The idea is first to locate the nodes in $\overline{H}(\mathcal{C})$ for the given two edges, and then traverse $\overline{H}(\mathcal{C})$ starting from these two nodes respectively to compute all the possible positions that the two given edges can be moved to by reverse operations. After that the algorithm examines whether there exists a common vertex where the two edges can be made adjacent. The series of reverse operations can be obtained through tracing back. The correctness and running time of Algorithm 6 is stated as Proposition 14.

Proposition 14 *Algorithm 6 gives an exact solution for Problem 3 in $O(|V|^2 \cdot |E|)$ time.*

Appendix B Experiments with Simulated Random Instances

The simulation procedure first generates the ground truth $(P^\ddagger, w^\ddagger)$, and then builds (G, f) based on them. In the simulation, we assume that $G = (V, E)$ contains 1000 vertices, i.e., $|V| = 1000$, and assume that $(1, 2, \cdots, |V|)$ is one topological sorting of G. To generate a path $p^\ddagger$ in the ground truth, we first fix the length of this path, which is sampled uniformly at random from $\{1, 2, \cdots, L\}$, where $L \leq |V| - 1$ is a parameter governing the maximum length of the paths in the ground truth path. We then sample $(|p^\ddagger| + 1)$ vertices uniformly at random without replacement from $\{1, 2, \cdots, |V|\}$, and the path connects these vertices in an ascending order. The ground truth weight of this path, i.e., $w^\ddagger(p^\ddagger)$, is sampled uniformly at random from $\{1, 2, \cdots, 10000\}$. We repeat this process to generate $|P^\ddagger|$ paths, where $|P^\ddagger|$ is another parameter governing the number of paths in the ground truth. With $(P^\ddagger, w^\ddagger)$, we then construct instance (G, f) by superposition of the weights of the paths that go through the edge (similar to Figure 7). Again, we denote by $(P^\dagger, w^\dagger)$ be the output of the algorithm (Algorithm 4 or greedy-width), and we say $(P^\dagger, w^\dagger)$ is *correct* if we have $|P^\dagger| \leq |P^\ddagger|$.

We perform experiments with two different parameter settings, one is $L = 50, |P^\ddagger| \in \{20, 40, \cdots, 200\}$, the other is $L \in \{10, 20, \cdots, 100\}, |P^\ddagger| = 100$. For each parameter combination $(L, |P^\ddagger|)$, we randomly generate 100 instances using the procedure mentioned above. We run both algorithms on these 100 instances, and for each algorithm we record its *accuracy* (proportion of instances that are predicted correctly), average ratio between $|P^\dagger|$ and $|P^\ddagger|$ (i.e., mean value of $|P^\dagger|/|P^\ddagger|$ over the 100 instances), and total running time.

The results for the first parameter settings are shown in Table 1. First, we can observe that Algorithm 4 gives 100% accuracy when $|P^\ddagger| \leq 140$, and it decreases a few percentage for larger $|P^\ddagger|$. Meanwhile, the accuracy of the greedy-width algorithm decreases rapidly with the increase of $|P^\ddagger|$, and drops to 0 when $|P^\ddagger| \geq 100$. Second, we can see that the number of paths returned by Algorithm 4 is very close to the ground truth for all instances, while greedy-width algorithm uses more paths than the ground truth especially for large $|P^\ddagger|$. Third, both algorithms can finish these instances in a short amount of time, and Algorithm 4 is about 4 times slower than the greedy-width algorithm.

$\lvert P^\ddagger\rvert$		20	40	60	80	100	120	140	160	180	200
Accuracy (%)	Algorithm 4	100	100	100	100	100	100	100	97	93	83
	Greedy-width	94	46	14	1	0	0	0	0	0	0
Mean $\lvert P^\dagger\rvert/\lvert P^\ddagger\rvert$	Algorithm 4	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.01
	Greedy-width	1.01	1.01	1.10	1.21	1.27	1.40	1.51	1.66	1.76	1.83
Time (seconds)	Algorithm 4	4	14	30	51	72	91	146	193	223	255
	Greedy-width	2	5	8	13	17	23	33	44	55	65

Table 1: Comparison of the two algorithms on simulations with $L = 50$.

The results for the second parameter settings are shown in Table 2, which are consistent with that in Table 1. Combining these results we can conclude that Algorithm 4 can give decompositions with the same number of paths as that in the ground truth for a large scope of parameter combinations, while greedy-width algorithm can only give such decompositions for small L and small $|P^{\ddagger}|$, and it requires a significant portion of more paths for large L or large $|P^{\ddagger}|$. This shows a considerable improvement of Algorithm 4 over the greedy-width algorithm, especially for instances with many paths.

L		10	20	30	40	50	60	70	80	90	100
Accuracy (%)	Algorithm 4	100	100	100	100	100	100	100	99	91	81
	Greedy-width	65	20	3	0	0	0	0	0	0	0
Mean $\lvert P^{\dagger}\rvert/\lvert P^{\ddagger}\rvert$	Algorithm 4	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.01
	Greedy-width	1.01	1.02	1.07	1.17	1.29	1.45	1.61	1.82	2.05	2.25
Time (seconds)	Algorithm 4	5	16	32	54	67	89	146	167	195	226
	Greedy-width	3	6	9	13	17	22	31	41	49	58

Table 2: Comparison of the two algorithms on simulations with $|P^{\ddagger}| = 100$.

Appendix C Proofs of Propositions

Proof of Proposition 1. Because (P^*, w^*) forms a decomposition of (G, f), we know that $f = w^* \cdot P^*$. By definition, for any $q \in \mathcal{N}(P^*)$ we also have $P^* \cdot q = 0$. Thus, we have that $f \cdot q = w^* \cdot P^* \cdot q = 0$. □

Proof of Proposition 2. We can write $\Pr(f \cdot q = 0) = \Pr(w^* \cdot P^* \cdot q = 0)$. Since $q \notin \mathcal{N}(P^*)$, we know that $P^* \cdot q \neq 0$. Combining the fact that elements in w^* are sampled independently and uniformly at random from $\mathbb{Z}_k$, using the standard backward analysis, we can conclude that $\Pr(f \cdot q = 0) \leq 1/k$. □

Proof of Proposition 3. We can easily verify that for every row p_i of $\mathcal{P}$ and for every $v \in V_0$ we have that $p_i \cdot q_v = 0$. Particularly, since $P^* \subseteq \mathcal{P}$, we have that $P^* \cdot q_v = 0$. This implies that $q_v \in \mathcal{N}(P^*)$, for every $v \in V_0$. Hence, we have $\mathcal{N}_0 \subseteq \mathcal{N}(P^*)$. □

Proof of Proposition 4. According to the rank-nullity theorem, we have that $\dim(\mathcal{N}(P^*)) = |E| - rank(P^*)$. From [24], we have that rows in the optimal solution are linearly independent, i.e., $rank(P^*) = |P^*|$. Besides, we also have that $\dim(\mathcal{N}_0) = |V| - 2$, since all vectors in $\{q_v \mid v \in V_0\}$ are linearly independent. Combining all these facts, we have that $\dim(\mathcal{N}(P^*)) - \dim(\mathcal{N}_0) = \Delta - |P^*|$. □

Proof of Proposition 5. We first prove that Algorithm 1 returns a subset of $Q_2 \setminus \mathcal{N}_0$. Let $E_1 \subseteq E$ and $E_2 \subseteq E$ be the two balanced subsets through tracing back from $x \in D$. We only need to show that E_1 and E_2 are indivisible. We first prove that E_1 and E_2 are disjoint. Suppose conversely that $e \in E_1 \cap E_2$. Then we have $B[|E|, x - f(e)] = 1$, since there exist two subsets, namely $E_1' = E_1 \setminus \{e\}$ and $E_2' = E_2 \setminus \{e\}$, such that $f(E_1') = f(E_2') = x - f(e)$, a contradiction to the fact that $x \in D$.

Now we show that E_1 and E_2 are indivisible. Suppose conversely that there exist strict subsets $E_1' \subset E_1$ and $E_2' \subset E_2$ such that $f(E_1') = f(E_2') = y$. Let e_k be the edge with the highest index in $E_1 \cup E_2$. Without loss of generality, we assume that $e_k \in E_1 \setminus E_1'$. Let $e_{k'}$ be the edge with the highest index in $E_1' \cup E_2'$. We have that $B[k', y] = 1$, because of the existence of E_1' and E_2'. Following the recursion of B (line 4 of Algorithm 1), we also have that $B[j, y + \sum_{k' < i \leq j} f(e_i)] = 1$, for all $k' < j \leq k$. This implies that $B[k, x - f(e_k)] = 1$, a contradiction to the fact that $x \in D$. □

We can verify that computing A, B, D and tracing back for all elements from D takes $O(|E|\cdot|f|)$ time, while other steps take less than that. Thus, the total running time of Algorithm 1 is $O(|E|\cdot|f|)$.

Proof of Proposition 6. On one side, if there exists $V_1 \subseteq V_0$ such that $q = \sum_{v\in V_1} q_v$ or $q = -\sum_{v\in V_1} q_v$, then by definition, we have that $q \in \mathcal{N}_0$.

Now we prove the other side. Assume that $q \in \mathcal{N}_0$, i.e., $q = \sum_{v\in V_0} c_v \cdot q_v$. Let $V_1 = \{v \in V_0 \mid c_v \neq 0\}$. Let $G_1 = (V_1, E_1)$ be the subgraph of G induced by V_1, i.e., we define $e \in E_1$ if both ends of e is in V_1. We first prove that G_1 is weakly connected. Suppose conversely that G_1 contains $k \geq 2$ connected components, $G_1^i = (V_1^i, E_1^i)$, $i = 1, 2, \cdots, k$. We can write $q = \sum_{1\leq i\leq k} q^i$, where $q^i = \sum_{v\in V_1^i} c_v \cdot q_v$. Notice that q^i contributes to distinct elements of q, i.e., for any edge $e_j \in E_1^i$, we have $q[j] = q^i[j]$. Since $q \in \mathcal{Q}$, we also have that $q^i \in \mathcal{Q}$. Further, we have that $q^i \in \mathcal{Q}_1$, since we have $f \cdot q^i = f \cdot \sum_{v\in V_1^i} c_v \cdot q_v = \sum_{v\in V_1^i} c_v \cdot f \cdot q_v = 0$. This implies that the two balanced subsets corresponding to q can be split into k pairs of balanced subsets, namely, $E_s(q^i)$ and $E_t(q^i)$, $1 \leq i \leq k$, a contradiction to the fact that $q \in \mathcal{Q}_2$.

We now show that $c_v \in \mathbb{Z}, \forall v \in V_1$. Let $E_s(V_1) = \{(u,v) \in E \mid u \in V \setminus V_1, v \in V_1\}$, and $E_t(V_1) = \{(u,v) \in E \mid u \in V_1, v \in V \setminus V_1\}$, Let $E_{st}(V_1) = E_s(V_1) \cup E_t(V_1)$. Let $V_2 \subseteq V_1$ be the subset of vertices that are adjacent to some edge in E_{st}. For any edge $e_j = (u,v) \in E_1$, we have $q[j] = c_v - c_u$. Following this formular, we have that $c_v \in \{-1, +1\}, \forall v \in V_2$, since edges in E_{st} are adjacent to only one vertex in V_1, which is in V_2. Since G' is weakly connected and $q[j] \in \{-1, 0, +1\}$, by propagating from V_2 with the above formular, we can conclude that $c_v \in \mathbb{Z}$ for all $v \in V_1$.

Consider any edge $e_j = (u,v) \in E_1$. If we assume that $c_u \geq 1$, then from the above formular we have that $c_v = c_u + q[j] \geq 0$; and since $c_v \neq 0$ and $c_v \in \mathbb{Z}$, we have that $c_v \geq 1$. Following this reasoning and the fact that G' is weakly connected, through propagating we know that $\{c_v \mid v \in V_1\}$ have the same sign. Without loss of generality, we assume that $c_v \geq 1, \forall v \in V_1$. This implies that $c_v = 1, \forall v \in V_2$, and further implies that $q[j] = 1$ for $e_j \in E_s(V_1)$ and $q[j] = -1$ for $e_j \in E_t(V_1)$. Thus, we have that $E_s(V_1) \subseteq E_s(q)$ and $E_t(V_1) \subseteq E_t(q)$. According to flow conservation, we have that $f(E_s(V_1)) = f(E_t(V_1))$. Combining these two facts and the assumption that q is indivisible, we must have that $E_s(V_1) = E_s(q)$ and $E_t(V_1) = E_t(q)$. This consequently implies that $V_1 = V_2$. As we already have showed, $c_v = 1, \forall v \in V_2$, we have that $q = \sum_{v\in V_1} q_v$. □

Proof of Proposition 7. Suppose that $q \in \mathcal{N}_0$. According to Proposition 6, without loss of generality, we have that there exists a connected subset $V_1 \subseteq V_0$ such that $q = \sum_{v\in V_1} q_v$. It is clear that $q[j] = 1$ if and only if $e_j \in E_s(V_1)$, and $q[j] = -1$ if and only if $e_j \in E_t(V_1)$, since for any edge e_k that are adjacent to two vertices in V_1 we have $q[k] = 0$. This yields that $E_s(q) = E_s(V_1)$ and $E_t(q) = E_t(V_1)$.

On the other side, suppose that there exists a subset $V_1 \subseteq V_0$ such that $E_s(q) = E_s(V_1)$ and $E_t(q) = E_t(V_1)$. We now prove that $q = \sum_{v\in V_1} q_v$. In fact, $\sum_{v\in V_1} q_v$ exactly has value of $+1$ for edges in $E_s(V_1) = E_s(q)$, and has value of -1 for edges in $E_t(V_1) = E_t(q)$. This yields that $q = \sum_{v\in V_1} q_v$. □

Proof of Proposition 8. According to the algorithm, we can easily verify that $q^k \in \mathcal{Q}_1^k$, from the fact that $q \in \mathcal{Q}_1$. We now show that $q^k \in \mathcal{Q}_2^k$. Suppose conversely that q^k is not indivisible, i.e., there exist strict subsets $E_1 \subset E_s(q^k)$ and $E_2 \subset E_t(q^k)$ such that $f^k(E_1) = f^k(E_2)$. Let $e_i \in E_s(q^{k-1})$ and $e_j \in E_t(q^{k-1})$ be the two chosen edges in iteration k. Assume that $f^{k-1}(e_i) \leq f^{k-1}(e_j)$. This implies that $e_i \notin E_s(q^k)$. Let $E_1' = E_1 \cup \{e_i\}$. If we further have $f^{k-1}(e_i) = f^{k-1}(e_j)$, we define $E_2' = E_2 \cup \{e_j\}$, otherwise, we define $E_2' = E_2$. Clearly, we have that $f^{k-1}(E_1') = f^{k-1}(E_2')$ according to the algorithm, and that E_1' and E_2' are strict subsets of $E_s(q^{k-1})$ and $E_t(q^{k-1})$, respectively, following from the fact that E_1 and E_2 are strict subsets of $E_s(q^k)$ and $E_t(q^k)$, respectively. Thus, we conclude that q^{k-1} is not indivisible, a contradiction.

We then show that $q^k \notin \mathcal{N}_0^k$. Suppose conversely that $q^k \in \mathcal{N}_0^k$. According to Proposition 6, there exists $V_1 \subseteq V_0$ such that $E_s(q^k) = E_s(V_1)$ and $E_t(q^k) = E_t(V_1)$. Let $e_i \in E_s(q^{k-1})$ and $e_j \in E_t(q^{k-1})$ be the two chosen

edges in iteration k. Since $q^{k-1} \in Q_4^{k-1}$, we know that either $e_i \in E_s(V_1)$ and $e_j \in E_t(V_1)$, or $e_i \notin E_s(V_1)$ and $e_j \notin E_t(V_1)$. We further show the second case is not possible, since it immediately implies that q^{k-1} is not indivisible. Thus, we have that $e_i \in E_s(V_1)$ and $e_j \in E_t(V_1)$. If q^k is obtained from the first situation (line 3) of Algorithm 3, then this already means that $q^{k-1} \in \mathcal{N}_0^{k-1}$. Otherwise (line 6), since we guarantee that the path p used in line 8 of Algorithm 3 does not contain any edge in $E_s(q^{k-1}) \cup E_t(q^{k-1})$, we can still conclude that $q^{k-1} \in \mathcal{N}_0^{k-1}$, a contradiction, as desired. □

Proof of Proposition 9. Since q^k is indivisible from Proposition 8, we know that during the k-th iteration, the two chosen edges e_i and e_j cannot have that $f(e_i) = f(e_j)$, $1 \le k \le n-1$. According to Algorithm 3, we have that $|E_s(q^k)| + |E_t(q^k)| = |E_s(q^{k-1})| + |E_t(q^{k-1})| - 1$. After $(n-1)$ iterations, we must have that $|E_s(q^{k-1})| = |E_t(q^{k-1})| = 1$, and both the two subsets become empty after the n-th iteration. Thus, after exactly $(|E_s(q)| + |E_t(q)| - 1)$ iterations, Algorithm 3 terminates. □

Proof of Proposition 10. We first prove that, for $1 \le k \le n-1$, we have that $\Delta^k \le \Delta^{k-1}$. Consider the change of number of edges and vertices during the k-th iteration. Let e_i and e_j be the two chosen edges. According to the proof of Proposition 9, we have that only e_i will be removed, and a new edge (namely, e_k in Algorithm 3) will be added. If it is the first situation (line 3 of Algorithm 3), then the middle vertex (namely, $v_1 = u_2$) cannot become isolated, since otherwise it implies that q^{k-1} is not indivisible. If it is the second situation (line 6 of Algorithm 3), some vertices in path p might be become isolated. But if this happens, then at least the same number of edges will also be removed in p. Thus, we always have that $\Delta^k \le \Delta^{k-1}$.

We now show that $\Delta^n \le \Delta^{n-1} - 1$. This is because, in the n-th iteration, both e_i and e_j are removed, and only one edge will be added. Similar to the above analysis, if some vertices become isolated, then at least the same number of additional edges will be removed. Thus, we have that $\Delta^n \le \Delta^{n-1} - 1$.

According to our analysis in the main text, the merging process narrows down the solution space, i.e., we have $|P^{k*}| \ge |P^{(k-1)*}|$, $1 \le k \le n$. By combining these facts, we have that $\Delta^n - |P^{n*}| \le \Delta - |P^*| - 1$. □

Proof of Proposition 11. The running time of Algorithm 4 is dominated by lines 2 and 3. According to Proposition 9, in Algorithm 3, Algorithm 6 will be called at most $|E|$ times. Thus, according to Proposition 14 and the fact that $|Q_2^\dagger \setminus \mathcal{N}_0| \le |f|$, the total running time is $O(|V|^2 \cdot |E|^2 \cdot |f|)$. □

Proof of Proposition 12. Let (P,w) be any decomposition of (G,f). We can obtain a decomposition (P',w') of (G',f) satisfying that $|P'| = |P|$ and that $w' = w$ by only reversing the edges from u to v for each path in P. Since reverse operation is invertible, given a decomposition (P',w') of (G',f), we can also obtain a decomposition (P,w) of (G,f) in the same way. This gives a one-to-one correspondence between the decompositions of (G,f) and that of (G',f). And, in particular we have $|P'^*| = |P^*|$. □

Proof of Proposition 13. The correctness of the algorithm is the consequence of the following facts. First, if $(u,v) \in \mathcal{C}$, then among the vertices in $V(u,v)$, u and v are the smallest and the largest vertices *w.r.t.* any topological ordering. This is simply because all vertices in $V(u,v) \setminus \{u,v\}$ can be reached from u and can reach v. Second, L and R always store the current smallest and largest vertices in S *w.r.t.* the topological ordering. Thus, by definition, vertices added to S in line 9 of Algorithm 5 are guaranteed in the closure of V_1. Third, Q always stores the vertices in $S \setminus \{L,R\}$. Thus, following the first fact, we know that vertices in Q cannot be the boundary vertices of the closure, and consequently, all their adjacent vertices (added in line 9) are guaranteed in the closure of V_1. Fourth, when the algorithm terminates (line 14), we have that $S = V(L,R)$. In fact, let p be any path from L to R. From line 13, we know that the first vertex in p is in S. Then following lines 8 to 11, all vertices in p will be added to S. Finally, all vertices in $S \setminus \{L,R\}$ are examined such that all their adjacent vertices are in S. Thus, by definition, we know that (L,R) is closed. Besides, since all vertices added to S are guaranteed to in the closure of V_1, we have that S is also the smallest subset, implying that (L,R) is the closure of V_1. This reason also proves that the closure of V_1 is unique.

This algorithm is essentially the breadth-first search algorithm, in which edge each is examined at most once. Thus, its running time is $O(|E|)$. □

Proof of Proposition 14. If Algorithm 6 finds $w \in (S_1 \cap T_2) \cup (S_2 \cap T_1)$, then clearly we can perform a series of operations to make e_i and e_j adjacent. On the other side, suppose that there exists a series of reverse operations such that in the resulting graph we have that $e_i = (w_1, w)$ and $e_j = (w, w_2)$. Let c_1 and c_2 be the closures of $\{w_1, w\}$ and $\{w, w_2\}$, respectively. Clearly, all the reverse operations that moves e_i (resp. e_j) must operates on closures that are inside c_1 (resp. c_2). Thus, there must be a path from (u'_1, v'_1) (resp. (u'_2, v'_2)) to c_1 (resp. c_2) in $\overline{H}(\mathcal{C})$, which implies that Algorithm 6 will have $w \in (S_1 \cap T_2) \cup (S_2 \cap T_1)$. This reasoning also implies that when Algorithm 6 traces back to recover the reverse operations (lines 8 and 9), the two paths are disjoint, since they are in two different subtrees (rooted at c_1 and c_2, respectively).

The running time of Algorithm 6 is dominated by building $\overline{H}(\mathcal{C})$, which is $O(|V|^2 \cdot |E|)$. □