Shortest Two Disjoint Paths in Conservative Graphs

Ildikó Schlotter
Centre for Economic and Regional Studies, Budapest, Hungary
Budapest University of Technology and Economics, Hungary

Abstract

We consider the following problem that we call the Shortest Two Disjoint Paths problem: given an undirected graph $G = (V, E)$ with edge weights $w : E \to \mathbb{R}$, two terminals s and t in G, find two internally vertex-disjoint paths between s and t with minimum total weight. As shown recently by Schlotter and Sebő (2022), this problem becomes NP-hard if edges can have negative weights, even if the weight function is conservative, i.e., there are no cycles in G with negative total weight. We propose a polynomial-time algorithm that solves the Shortest Two Disjoint Paths problem for conservative weights in the case when the negative-weight edges form a constant number of trees in G.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis; Theory of computation → Shortest paths; Theory of computation → Dynamic programming

Keywords and phrases Shortest paths, disjoint paths, conservative weights, graph algorithm

Digital Object Identifier 10.4230/LIPIcs.STACS.2024.57

Related Version *Full Version*: `https://arxiv.org/abs/2307.12602` [12]

Funding *Ildikó Schlotter*: supported by the Hungarian Academy of Sciences under its Momentum Programme (LP2021-2) and its János Bolyai Research Scholarship.

1 Introduction

Finding disjoint paths between given terminals is a fundamental problem in algorithmic graph theory and combinatorial optimization. Besides its theoretical importance, it is also motivated by numerous applications in transportation, VLSI design, and network routing. In the Disjoint Paths problem, we are given k terminal pairs (s_i, t_i) for $i \in \{1, \dots, k\}$ in an undirected graph G, and the task is to find pairwise vertex-disjoint paths $P_1, \dots, P_k$ so that P_i connects s_i with t_i for each $i \in \{1, \dots, k\}$. This problem was shown to be NP-hard by Karp [7] when k is part of the input, and remains NP-hard even on planar graphs [10]. Robertson and Seymour [11] proved that there exists an $f(k)n^3$ algorithm for Disjoint Paths with k terminal pairs, where n is the number of vertices in G and f some computable function; this celebrated result is among the most important achievements of graph minor theory. In the Shortest Disjoint Paths problem we additionally require that $P_1, \dots, P_k$ have minimum total length (in terms of the number of edges). For fixed k, the complexity of this problem is one of the most important open questions in the area. Even the case for $k = 2$ had been open for a long time, until Björklund and Husfeldt [3] gave a randomized polynomial-time algorithm for it in 2019. For directed graphs the problem becomes much harder: the Directed Disjoint Paths problem is NP-hard already for $k = 2$. The Disjoint Paths problem and its variants have also received considerable attention when restricted to planar graphs [6, 5, 8, 1, 4, 14, 9].

The variant of Disjoint Paths when $s_1 = \cdots = s_k = s$ and $t_1 = \cdots = t_k = t$ is considerably easier, since one can find k pairwise (openly vertex- or edge-) disjoint paths between s and t using a max-flow computation. Applying standard techniques for computing a minimum-cost flow (see e.g. [15]), one can even find k pairwise disjoint paths between s and t with minimum total weight, given non-negative weights on the edges. Notice that if negative weights are allowed, then flow techniques break down for undirected graphs: in order to construct an appropriate flow network based on our undirected graph G, the standard technique is to direct each edge of G in both directions; however, if edges can have negative weight, then this operation creates negative cycles consisting of two arcs, an obstacle for computing a minimum-cost flow. Recently, Schlotter and Sebő [13] have shown that this issue is a manifestation of a complexity barrier: finding two openly disjoint paths with minimum total weight between two vertices in an undirected edge-weighted graph is NP-hard, even if weights are *conservative* (i.e., no cycle has negative total weight) and each edge has weight in $\{-1, 1\}$.[1] Note that negative edge weights occur in network problems due to various reasons: for example, they might arise as a result of some reduction (e.g., deciding the feasibility of certain scheduling problems with deadlines translates into finding negative-weight cycles), or as a result of data that is represented on a logarithmic scale. We remark that the Single-Source Shortest Paths problem is the subject of active research for the case when negative edges are allowed; see Bernstein et al. [2] for an overview of the area and their state-of-the-art algorithm running in near-linear time on directed graphs.

Our contribution

We consider the following problem which concerns finding paths between two fixed terminals (as opposed to the classic Shortest Disjoint Paths problem):

Shortest Two Disjoint Paths:

Input:	An undirected graph $G = (V, E)$, a weight function $w\colon E \to \mathbb{R}$ that is conservative on G, and two vertices s and t in G.
Task:	Find two paths P_1 and P_2 between s and t with $V(P_1) \cap V(P_2) = \{s, t\}$ that minimizes $w(P_1) + w(P_2)$.

A *solution* for an instance (G, w, s, t) of Shortest Two Disjoint Paths is a pair of (s, t)-paths that are openly disjoint, i.e., do not share vertices other than their endpoints.

From the NP-hardness proof for Shortest Two Disjoint Paths by Schlotter and Sebő [13] it follows that the problem remains NP-hard even if the set of negative-weight edges forms a perfect matching. Motivated by this intractability, we focus on the "opposite" case when the subgraph of G spanned by the set $E^- = \{e \in E : w(e) < 0\}$ of negative-weight edges, denoted by $G[E^-]$, has only few connected components.[2] Note that since w is conservative on G, the graph $G[E^-]$ is acyclic. Hence, if c denotes the number of connected components in $G[E^-]$, then $G[E^-]$ in fact consists of c trees.

We can think of our assumption that c is constant as a compromise for allowing negative-weight edges but requiring that they be confined to a small part of the graph. For a motivation, consider a network where negative-weight edges arise as some rare anomaly. Such an anomaly may occur when, in a certain part of a computer network, some information can be collected while traversing the given edge. If such information concerns, e.g., the detection of (possibly) faulty nodes or edges in the network, then it is not unreasonable to assume that these faults are concentrated to a certain part of the network, due to underlying physical causes that are responsible for the fault.

[1] In fact, Schlotter and Sebő use an equivalent formulation of the problem where, instead of finding two openly disjoint paths between s and t, the task is to find two vertex-disjoint paths between $\{s_1, s_2\}$ and $\{t_1, t_2\}$ for four vertices $s_1, s_2, t_1, t_2 \in V$.

[2] See Section 2 for the precise definition of a subgraph spanned by an edge set.

Ideally, one would aim for an algorithm that is fixed-parameter tractable (FPT) when parameterized by c; however, already the case $c = 1$ turns out to be challenging. We prove the following result, which can be thought of as a first step towards an FPT algorithm:

▶ **Theorem 1.** *For each constant $c \in \mathbb{N}$, Shortest Two Disjoint Paths can be solved in polynomial time on instances where the set of negative edges spans c trees in G.*

Our algorithm first applies standard flow techniques to find minimum-weight solutions among those that have a simple structure in the sense that there is no negative tree in $G[E^-]$ used by both paths. To deal with more complex solutions where there is at least one tree T in $G[E^-]$ used by both paths, we use recursion to find two openly disjoint paths from s to T, and from T to t; to deal with the subpaths of the solution that heavily use negative edges from T, we apply an intricate dynamic programming method that is based on significant insight into the structural properties of such solutions.

Organization

We give all necessary definitions in Section 2. In Section 3 we make initial observations about optimal solutions for an instance (G, w, s, t) of Shortest Two Disjoint Paths, and we also present a lemma of key importance that will enable us to create solutions by combining partial solutions that are easier to find (Lemma 10). We present the algorithm proving our main result, Theorem 1, in Section 4. In Section 4.1 we give a general description of our algorithm, and explain which types of solutions can be found using flow-based techniques. We proceed in Section 4.2 by establishing structural observation that we need to exploit in order to find those types of solutions where more advanced techniques are necessary. Section 4.3 contains our dynamic programming method for finding partial solutions which, together with Lemma 10, form the heart of our algorithm. We conclude with some questions for further research in Section 5. All proofs are deferred to the full version of our paper [12].

2 Notation

For a positive integer ℓ, we use $[\ell] := \{1, 2, \dots, \ell\}$.

Let a graph G be a pair (V, E) where V and E are the set of vertices and edges, respectively. For two vertices u and v in V, an edge connecting u and v is denoted by uv or vu.

For a set of X of vertices (or edges), let $G - X$ denote the subgraph of G obtained by deleting the vertices (or edges, respectively) of X; if $X = \{x\}$ then we may simply write $G - x$ instead of $G - \{x\}$. Given a set $F \subseteq E$ of edges in G, we denote by $V(F)$ the vertices incident to some edge of F. The subgraph of G *spanned by* F is the graph $(V(F), F)$; we denote this subgraph as $G[F]$.

A *walk* W in G is a series $e_1, e_2, \dots, e_\ell$ of edges in G for which there exist vertices $v_0, v_1, \dots, v_\ell$ in G such that $e_i = v_{i-1}v_i$ for each $i \in [\ell]$; note that both vertices and edges may appear repeatedly on a walk. We denote by $V(W)$ the set of vertices *contained by* or *appearing on* W, that is, $V(W) = \{v_0, v_1, \dots, v_\ell\}$. The *endpoints* of W are v_0 and v_ℓ, or in other words, it is a (v_0, v_ℓ)-*walk*, while all vertices on W that are not endpoints are *inner vertices*. If $v_0 = v_\ell$, then we say that W is a *closed walk*.

A *path* is a walk on which no vertex appears more than once. By a slight abuse of notation, we will usually treat a path as a *set* $\{e_1, e_2, \dots, e_\ell\}$ of edges for which there exist distinct vertices $v_0, v_1, \dots, v_\ell$ in G such that $e_i = v_{i-1}v_i$ for each $i \in [\ell]$. For any i and j with $0 \le i \le j \le \ell$ we will write $P[v_i, v_j]$ for the *subpath* of P between v_i and v_j, consisting of edges $e_{i+1}, \dots, e_j$. Note that since we associate no direction with P, we have

$P[v_i, v_j] = P[v_j, v_i]$. Given two vertices s and t, an (s,t)-path is a path whose endpoints are s and t. Similarly, for two subsets S and T of vertices, an (S,T)-path is a path with one endpoint in S and the other endpoint in T.

We say that two paths are *vertex-* or *edge-disjoint*, if they do not share a common vertex or edge, respectively. Two paths are *openly disjoint*, if they share no common vertices apart from possibly their endpoints. Given vertices s_1, s_2, t_1, and t_2, we say that two $(\{s_1,s_2\},\{t_1,t_2\})$-paths are *permissively disjoint*, if a vertex v can only appear on both paths if either $v = s_1 = s_2$ or $v = t_1 = t_2$. Two paths properly intersect, if they share at least one edge, but neither is the subpath of the other.

A *cycle* in G is a set $\{e_1, e_2, \ldots, e_\ell\}$ of distinct edges in G such that $e_1, e_2, \ldots, e_{\ell-1}$ form a path in $G - e_\ell$ whose endpoints are connected by e_ℓ. A set $T \subseteq E$ of edges in G is *connected*, if for every pair of edges e and e' in T, there is a path contained in T containing both e and e'. If T is connected and *acyclic*, i.e., contains no cycle, then T is a *tree* in G. Given two vertices a and b in a tree T, we denote by $T[a,b]$ the unique path contained in T whose endpoints are a and b. For an edge $uv \in T$ and a path P within T such that $uv \notin P$, we say that v *is closer to* P in T than u, if $v \in V(T[u,p])$ for some vertex $p \in V(P)$.

Given a weight function $w \colon E \to \mathbb{R}$ on the edge set of G, we define the *weight* of any edge set $F \subseteq E$ as $w(F) = \sum_{e \in F} w(e)$. We extend this notion for any pair $\mathcal{F} = (F_1, F_2)$ of edge sets by letting $w(\mathcal{F}) = w(F_1) + w(F_2)$. The restriction of w to an edge set $F \subseteq E$, i.e., the function whose domain is F and has value $w(f)$ on each $f \in F$, is denoted by $w_{|F}$. We say that w (or, to make the dependency on G explicit, the weighted graph (G,w)) is *conservative*, if no cycle in G has negative total weight.

3 Structural Observations

Let $G = (V,E)$ be an undirected graph with a conservative weight function $w : E \to \mathbb{R}$. Let $E^- = \{e \in E : w(e) < 0\}$ denote the set of negative edges, and $\mathcal{T}$ the set of negative trees they form. More precisely, let $\mathcal{T}$ be the set of connected components in the subgraph $G[E^-]$; the acyclicity of each $T \in \mathcal{T}$ follows from the conservativeness of w. For any subset $\mathcal{T}'$ of $\mathcal{T}$, we use the notation $E(\mathcal{T}') = \bigcup_{T \in \mathcal{T}'} E(T)$ and $V(\mathcal{T}') = \bigcup_{T \in \mathcal{T}'} V(T)$.

In Section 3.1 we gather a few useful properties of conservative weight functions. In Section 3.2 we collect observations on how an optimal solution can use different trees in $\mathcal{T}$. We close the section with a lemma of key importance in Section 3.3 that enables us to compose solutions by combining two path pairs without violating our requirement of disjointness.

3.1 Implications of Conservative Weights

The next two lemmas establish implications of the conservativeness of our weight function. Lemma 2 concerns closed walks, while Lemma 3 considers paths running between two vertices on some negative tree in $\mathcal{T}$. These lemmas will be useful in proofs where a given hypothetical solution is “edited” – by removing certain subpaths from it and replacing them with paths within some negative tree – in order to obtain a specific form without increasing its weight.

▶ **Lemma 2.** *If W is a closed walk that does not contain any edge with negative weight more than once, then $w(W) \geq 0$.*

▶ **Lemma 3.** *Let x, y, x', y' be four distinct vertices on a tree T in $\mathcal{T}$.*

(1) *If Q is an (x,y)-walk in G using each edge of E^- at most once, then $w(Q) \geq w(T[x,y])$.*

(2) *If Q is an (x,y)-path and Q' is an (x',y')-path vertex-disjoint from Q, then $w(Q) + w(Q') \geq w(T[x,y] \setminus T[x',y']) + w(T[x',y'] \setminus T[x,y])$.*

3.2 Solution Structure on Negative Trees

We first observe a simple property of minimum-weight solutions.

▶ **Definition 4 (Locally cheapest path pairs).** *Let s_1, s_2, t_1, t_2 be vertices in G, and let P_1 and P_2 be two permissively disjoint $(\{s_1, s_2\}, \{t_1, t_2\})$-paths. A path $T[u, v]$ in some $T \in \mathcal{T}$ is called a* shortcut *for P_1 and P_2, if u and v both appear on the same path, either P_1 or P_2, and there is no inner vertex or edge of $T[u, v]$ contained in $P_1 \cup P_2$. We will call P_1 and P_2* locally cheapest*, if there is no shortcut for them.*

The idea behind this concept is the following. Suppose that P_1 and P_2 are permissively disjoint $(\{s_1, s_2\}, \{t_1, t_2\})$-paths, and $T[u, v]$ is a shortcut for P_1 and P_2. Suppose that u and v both lie on P_i (for some $i \in [2]$), and let P_i' be the path obtained by replacing $P_i[u, v]$ with $T[u, v]$; we refer to this operation as *amending* the shortcut $T[u, v]$. Then P_i' is also permissively disjoint from P_{3-i} and, since Lemma 2 implies $w(P_i[u, v]) \geq -w(T[u, v]) > 0$, has weight less than $w(P_i)$. Hence, we have the following observation.

▶ **Observation 5.** *Let P_1 and P_2 be two permissively disjoint $(\{s_1, s_2\}, \{t_1, t_2\})$-paths admitting a shortcut $T[z, z']$. Suppose that z and z' are on the path, say, P_1. Let P_1' be the path obtained by amending $T[z, z']$ on P_1. Then P_1' and P_2 are permissively disjoint $(\{s_1, s_2\}, \{t_1, t_2\})$-paths and $w(P_1') < w(P_1)$.*

▶ **Corollary 6.** *Any minimum-weight solution for (G, w, s, t) is a pair of locally cheapest paths.*

For convenience, for any (s, t)-path P and vertices $u, v \in V(P)$ we say that u *precedes* v on P, or equivalently, v *follows* u on P, if u lies on $P[s, v]$. When defining a vertex as the "first" (or "last") vertex with some property on P or on a subpath P' of P then, unless otherwise stated, we mean the vertex on P or on P' that is closest to s (or farthest from s, respectively) that has the given property.

The following lemma shows that if two paths in a minimum-weight solution both use negative trees T and T' for some $T, T' \in \mathcal{T}$ then, roughly speaking, they must traverse T and T' in the same order; otherwise one would be able to replace the subpaths of the solution running between T and T' by two paths, one within T and one within T', of smaller weight.

▶ **Lemma 7.** *Let P_1 and P_2 be two openly disjoint (s, t)-paths of minimum total weight, and let T and T' be distinct trees in $\mathcal{T}$. Suppose that v_1, v_2, v_1' and v_2' are vertices such that $v_i \in V(T) \cap V(P_i)$ and $v_i' \in V(T') \cap V(P_i)$ for $i \in [2]$, with v_1 preceding v_1' on P_1. Then v_2 precedes v_2' on P_2.*

The following lemma is a consequence of Corollary 6 and Lemma 7, and considers a situation when one of the paths in an optimal solution visits a negative tree $T \in \mathcal{T}$ at least twice, and visits some $T' \in \mathcal{T} \setminus \{T\}$ in between.

▶ **Lemma 8.** *Let P_1 and P_2 be two openly disjoint (s, t)-paths of minimum total weight, and let T and T' be distinct trees in $\mathcal{T}$. Suppose that v_1, v_2', and v_3 are vertices appearing in this order on P_1 when traversed from s to t, and suppose $v_1, v_3 \in V(T)$ while $v_2' \in V(T')$. Then*

- *$V(P_2) \cap V(T) \neq \emptyset$;*
- *$V(P_2) \cap V(T') = \emptyset$;*
- *no vertex of $V(P_1) \cap V(T')$ precedes v_1 or follows v_3 on P_1.*

We say that two paths P_1 and P_2 are *in contact* at T, if there is a tree $T \in \mathcal{T}$ and two distinct vertices v_1 and v_2 in T such that v_1 lies on P_1, and v_2 lies on P_2. Lemmas 7 and 8 imply the following fact that will enable us to use recursion in our algorithm to find solutions that consist of two paths in contact.

▶ **Lemma 9.** *Let P_1 and P_2 be two openly disjoint (s,t)-paths of minimum total weight, and assume that they are in contact at some tree $T \in \mathcal{T}$. For $i \in [2]$, let a_i and b_i denote the first and last vertices of P_i on T when traversed from s to t. Then we can partition $\mathcal{T} \setminus \{T\}$ into $(\mathcal{T}_s, \mathcal{T}_0, \mathcal{T}_t)$ such that for each $T' \in \mathcal{T} \setminus \{T\}$ and $i \in [2]$ it holds that*

(i) *if $P_i[s, a_i]$ contains a vertex of T', then $T' \in \mathcal{T}_s$*

(ii) *if $P_i[b_i, t]$ contains a vertex of T', then $T' \in \mathcal{T}_t$*

(iii) *if $P_i[a_i, b_i]$ contains a vertex of T', then $T' \in \mathcal{T}_0$.*

Given a solution (P_1, P_2) whose paths are in contact at some tree $T \in \mathcal{T}$, a partition of $\mathcal{T} \setminus \{T\}$ is *T-valid* with respect to (P_1, P_2), if it satisfies the conditions of Lemma 9.

3.3 Combining Path Pairs

The following lemma will be a crucial ingredient in our algorithm, as it enables us to combine "partial solutions" without violating the requirement of vertex-disjointness.

▶ **Lemma 10.** *Let p_1, p_2, q_1, q_2 be vertices in G, and let $T \in \mathcal{T}$ contain vertices v_1 and v_2 with $v_1 \neq v_2$. Let P_1 and P_2 be two permissively disjoint $(\{p_1, p_2\}, \{v_1, v_2\})$-paths in G, and let Q_1 and Q_2 be two permissively disjoint $(\{v_1, v_2\}, \{q_1, q_2\})$-paths in G that are locally cheapest. Assume also that we can partition $\mathcal{T}$ into two sets $\mathcal{T}_1$ and $\mathcal{T}_2$ with $T \in \mathcal{T}_2$ such that*

(i) *$V(T) \cap V(P_1 \cup P_2) = \{v_1, v_2\}$, and*

(ii) *$P_1 \cup P_2$ contains no edge of $E(\mathcal{T}_2)$, and $Q_1 \cup Q_2$ contains no edge of $E(\mathcal{T}_1)$.*

Then we can find in linear time two permissively disjoint $(\{p_1, p_2\}, \{q_1, q_2\})$-paths S_1 and S_2 in G such that $w(S_1) + w(S_2) \leq w(P_1) + w(P_2) + w(Q_1) + w(Q_2)$.

We remark that Lemma 10 heavily relies on the condition that v_1 and v_2 are both on T: to construct the desired paths S_1 and S_2, we not only use the path pairs (P_1, P_2) and (Q_1, Q_2) but, if necessary, remove certain subpaths from them and stitch together the remainder with a path running within T.

4 Polynomial-Time Algorithm for Constant $|\mathcal{T}|$

This section contains the algorithm proving our main result, Theorem 1. Let (G, w, s, t) be our instance of SHORTEST TWO DISJOINT PATHS with input graph $G = (V, E)$, and assume that the set E^- of negative edges spans c trees in G for some constant c. We present a polynomial-time algorithm that computes a solution for (G, w, s, t) with minimum total weight, or correctly concludes that no solution exists for (G, w, s, t). The running time of our algorithm is $O(n^{2c+9})$ where $n = |V|$, so in the language of parameterized complexity, our algorithm is in XP with respect to the parameter c.

In Section 4.1 we present the main ideas and definitions necessary for our algorithm, and provide its high-level description together with some further details. We will distinguish between so-called *separable* and *non-separable* solutions. Finding an optimal and separable solution will be relatively easy, requiring extensive guessing but only standard techniques for computing minimum-cost flows. By contrast, finding an optimal but non-separable solution is much more difficult. Therefore, in Section 4.2 we collect useful properties of optimal, non-separable solutions. These observations form the basis for an important subroutine necessary for finding optimal, non-separable solutions; this subroutine is presented in Section 4.3.

4.1 The Algorithm

We distinguish between two types of solutions for our instance (G, w, s, t) of Shortest Two Disjoint Paths.

▶ **Definition 11 (Separable solution).** *Let (P_1, P_2) be a solution for (G, w, s, t). We say that P_1 and P_2 are* separable*, if either*

- *they are* not in contact*, i.e., there is no tree $T \in \mathcal{T}$ that shares distinct vertices with both P_1 and P_2, or*
- *there is a* unique *tree $T \in \mathcal{T}$ such that P_1 and P_2 are in contact at T, but the intersection of T with both P_1 and P_2 is a path, possibly containing only a single vertex;*

otherwise they are non-separable*.*

In Section 4.1.1 we show how to find an optimal, separable solution, whenever such a solution exists for (G, w, s, t). Section 4.1.2 deals with the case when we need to find an optimal, non-separable solution. Our algorithm for the latter case is more involved, and relies on a subroutine that is based on dynamic programming and is developed throughout Sections 4.2 and 4.3. The existence of this subroutine is stated in Corollary 33.

4.1.1 Finding Separable Solutions

Suppose that (P_1, P_2) is a minimum-weight solution for (G, w, s, t) that is separable. The following definition establishes conditions when we are able to apply a simple strategy for finding a minimum-weight solution using well-known flow techniques.

▶ **Definition 12 (Strongly separable solution).** *Let (P_1, P_2) be a solution for (G, w, s, t). We say that P_1 and P_2 are* strongly separable*, if they are separable and they are either not in contact at any tree of $\mathcal{T}$, or they are contact at a tree of $\mathcal{T}$ that contains s or t.*

Suppose that P_1 and P_2 are either not in contact, or they are in contact at a tree of $\mathcal{T}$ that contains s or t. Let $\mathcal{T}^{\not\ni s,t}$ denote the set of all trees in $\mathcal{T}$ that contain neither s nor t. For each $T \in \mathcal{T}^{\not\ni s,t}$ that shares a vertex with P_i for some $i \in [2]$, we define a_T as the first vertex on P_i (when traversed from s to t) that is contained in T; note that T cannot share vertices with both P_1 and P_2 as they are strongly separable, so P_i is uniquely defined.

Our approach is the following: we guess the vertex a_T for each $T \in \mathcal{T}^{\not\ni s,t}$, and then compute a minimum-cost flow in an appropriately defined network. More precisely, for each possible choice of vertices $Z = \{z_T \in V(T) : T \in \mathcal{T}^{\not\ni s,t}\}$, we build a network N_Z as follows.

▶ **Definition 13 (Flow network N_Z for strongly separable solutions).** *Given a set $Z \subseteq V$ such that $Z \cap V(T) = \{z_T\}$ for each $T \in \mathcal{T}^{\not\ni s,t}$, we create N_Z as follows. We direct each non-negative edge in G in both directions. Then for each $T \in \mathcal{T}^{\not\ni s,t}$, we direct the edges of T away from z_T.*[3] *If some $T \in \mathcal{T}$ contains s, then we direct all edges of T away from s; similarly, if some $T \in \mathcal{T}$ contains t, then we direct all edges of T towards t. We assign a capacity of 1 to each arc and to each vertex*[4] *in the network except for s and t, and we retain the cost function w (meaning that we define $w(\overrightarrow{e})$ as $w(e)$ for any arc $\overrightarrow{e}$ obtained by directing some edge e). We let s and t be the source and the sink in N_Z, respectively.*

---

[3] Directing a tree T away from a vertex $z \in V(T)$ means that an edge uv in T becomes an arc (u, v) if and only if $T[z, u]$ has fewer edges than $T[z, v]$; directing T towards z is defined analogously.

[4] The standard network flow model can be adjusted by well-known techniques to allow for vertex capacities.

▶ **Lemma 14.** *If there exists a strongly separable solution for (G, w, s, t) with weight k, then there exists a flow of value 2 having cost k in the network N_Z for some choice of $Z \subseteq V$ containing exactly one vertex from each tree in $\mathcal{T}^{\not\ni s,t}$. Conversely, a flow of value 2 and cost k in the constructed network N_Z for some Z yields a solution for (G, w, s, t) with weight at most k.*

Next, we show how to deal with the case when a minimum-weight solution (P_1, P_2) is separable, but not strongly separable; then there is a unique tree $T \in \mathcal{T}$ at which P_1 and P_2 are in contact. In such a case, we can simply delete an edge from T in a way that paths P_1 and P_2 cease to be in contact in the resulting instance. This way, we can reduce our problem to the case when there is a strongly separable optimal solution; note, however, that the number of trees spanned by the negative edges (our parameter c) increases by 1.

▶ **Lemma 15.** *If there exists a separable, but not strongly separable solution for (G, w, s, t) with weight k, then there exists an edge $e \in E^-$ such that setting $G' = G - e$ and $E' = E \setminus \{e\}$, the instance $(G', w_{|E'}, s, t)$ admits a strongly separable solution with weight at most k. Conversely, a solution for $(G', w_{|E'}, s, t)$ is also a solution for (G, w, s, t) with the same weight.*

Thanks to Lemmas 14 and 15, if there exists a minimum-weight solution for (G, w, s, t) that is separable, then we can find some minimum-weight solution using standard algorithms for computing minimum-cost flows. In Section 4.1.2 we explain how we can find a minimum-weight non-separable solution for (G, w, s, t).

4.1.2 Finding Non-separable Solutions

To find a minimum-weight solution for (G, w, s, t) that is not separable, we need a more involved approach. We now provide a high-level presentation of our algorithm for finding a non-separable solution of minimum weight. We remark that Algorithm STDP contains a pseudocode; however, we believe that it is best to read the following description first.

Step 1. We guess certain properties of a minimum-weight non-separable solution (P_1, P_2): First, we guess a tree $T \in \mathcal{T}$ that shares distinct vertices both with P_1 and P_2, i.e., a tree at which P_1 and P_2 are in contact. Second, we guess a partition $(\mathcal{T}_s, \mathcal{T}_0, \mathcal{T}_t)$ of $\mathcal{T} \setminus \{T\}$ that is T-valid with respect to (P_1, P_2).

Step 2. If $\mathcal{T}_s \neq \emptyset$, then we guess the first vertex of P_1 and of P_2 contained in $V(T)$, denoted by a_1 and a_2, respectively. We use recursion to compute two permissively disjoint $(s, \{a_1, a_2\})$-paths using only the negative trees in $\mathcal{T}_s$, and to compute two permissively disjoint $(\{a_1, a_2\}, t)$-paths using only the negative trees in $\mathcal{T} \setminus \mathcal{T}_s$. Observe that by $\mathcal{T}_s \neq \emptyset$ and $T \in \mathcal{T} \setminus \mathcal{T}_s$, we search for these paths in graphs that contain only a strict subset of the negative trees in $\mathcal{T}$. Thus, our parameter c strictly decreases in both constructed sub-instances. We combine the obtained pairs of paths into a solution by using Lemma 10. We proceed in a similar fashion when $\mathcal{T}_t \neq \emptyset$.

Step 3. If $\mathcal{T}_s = \mathcal{T}_t = \emptyset$, then for both $i \in [2]$ we guess the first and last vertex of P_i contained in $V(T)$, denoted by a_i and b_i, respectively. We apply standard flow techniques to compute two $(s, \{a_1, a_2\})$-paths and two $(\{b_1, b_2\}, t)$-paths with no inner vertices in $V(\mathcal{T})$ that are pairwise permissively disjoint. Then, we apply the polynomial-time algorithm we devise for computing a pair of permissively disjoint $(\{a_1, a_2\}, \{b_1, b_2\})$-paths in G. This algorithm is the cornerstone of our method, and is based on important structural observations that allow for efficient dynamic programming. We combine the obtained pairs of paths into a solution by applying Lemma 10.

Step 4. We output a solution of minimum weight among all solutions found in Steps 2 and 3.

Let us now provide more details about these steps; see also Algorithm STDP.

Step 1: Initial guesses on $\mathcal{T}$

There are $c = |\mathcal{T}|$ possibilities to choose T from $\mathcal{T}$, and there are 3^{c-1} further possibilities to partition $\mathcal{T} \setminus \{T\}$ into $(\mathcal{T}_s, \mathcal{T}_0, \mathcal{T}_t)$, yielding $c3^{c-1}$ possibilities in total for our guesses.

Step 2: Applying recursion

To apply recursion in Step 2 when $\mathcal{T}_s \neq \emptyset$, we guess two distinct vertices a_1 and a_2 in T, and define the following sub-instances.

▶ **Definition 16 (Sub-instances for $\mathcal{T}_s \neq \emptyset$).** *For some $\emptyset \neq \mathcal{T}_s \subseteq \mathcal{T}$ and distinct vertices a_1 and a_2, we create a graph G_s by adding a new vertex $a^\star$, and connecting it to both a_1 and a_2 with an edge of weight $w_{a^\star} = |w(T[a_1, a_2])|/2$. We create instances I_s^1 and I_s^2 as follows:*

- *to get I_s^1, we delete $E(\mathcal{T} \setminus \mathcal{T}_s)$ and $V(\mathcal{T} \setminus \mathcal{T}_s) \setminus \{a_1, a_2\}$ from G_s, and designate s and $a^\star$ as our two terminals;*
- *to get I_s^2, we delete $V(\mathcal{T}_s)$ from G_s, and designate $a^\star$ and t as our two terminals.*

We use recursion to solve Shortest Two Disjoint Paths on sub-instances I_s^1 and I_s^2. Note that conservativeness is maintained for both I_s^1 and I_s^2, due to our choice of $w_{a^\star}$ and statement (1) of Lemma 3. If both sub-instances admit solutions, we obtain two permissively disjoint $(s, \{a_1, a_2\})$-paths $Q_1^\nearrow$ and $Q_2^\nearrow$ from our solution to I_s^1 by deleting the vertex $a^\star$. Similarly, we obtain two permissively disjoint $(\{a_1, a_2\}, t)$-paths $Q_1^\searrow$ and $Q_2^\searrow$ from our solution to I_s^2 by deleting the vertex $a^\star$. Next we use Lemma 10 to create a solution for our original instance (G, w, s, t) using paths $Q_1^\nearrow, Q_2^\nearrow, Q_1^\searrow$, and $Q_2^\searrow$.

If $\mathcal{T}_t \neq \emptyset$, then we proceed similarly: after guessing two distinct vertices b_1 and b_2 in T, we use a construction analogous to Definition 16.

▶ **Definition 17 (Sub-instances for $\mathcal{T}_t \neq \emptyset$).** *For some $\emptyset \neq \mathcal{T}_s \subseteq \mathcal{T}$ and distinct vertices b_1 and b_2, we create a graph G_t by adding a new vertex $b^\star$, and connecting it to both b_1 and b_2 with an edge of weight $w_{b^\star} = |w(T[b_1, b_2])|/2$. We create instances I_t^1 and I_t^2 as follows:*

- *to get I_t^1, we delete $V(\mathcal{T}_t)$ from G_t, and designate s and $b^\star$ as our two terminals;*
- *to get I_t^2, we delete $E(\mathcal{T} \setminus \mathcal{T}_t)$ and $V(\mathcal{T} \setminus \mathcal{T}_t) \setminus \{b_1, b_2\}$ from G_t and designate $b^\star$ and t as our two terminals.*

We use recursion to solve Shortest Two Disjoint Paths on sub-instances I_t^1 and I_t^2; again, conservativeness is ensured for I_t^1 and I_t^2 by our choice of $w_{b^\star}$ and statement (1) of Lemma 3. If both sub-instances admit solutions, we obtain two permissively disjoint $(s, \{b_1, b_2\})$-paths $Q_1^\nearrow$ and $Q_2^\nearrow$ from our solution to I_t^1 by deleting the vertex $b^\star$. Similarly, we obtain two permissively disjoint $(\{b_1, b_2\}, t)$-paths $Q_1^\searrow$ and $Q_2^\searrow$ from our solution to I_t^2 by deleting the vertex $b^\star$. Again, we use Lemma 10 to create a solution for our original instance (G, w, s, t) using paths $Q_1^\nearrow, Q_2^\nearrow, Q_1^\searrow$, and $Q_2^\searrow$.

We state the correctness of Step 2 in the following lemma:

▶ **Lemma 18.** *Suppose that (P_1, P_2) is a minimum-weight solution for (G, w, s, t) such that*

- *P_1 and P_2 are in contact at some $T \in \mathcal{T}$,*
- *a_i and b_i are the first and last vertices of P_i contained in T, respectively, for $i \in [2]$, and*
- *$(\mathcal{T}_s, \mathcal{T}_0, \mathcal{T}_s)$ is a T-valid partition w.r.t. (P_1, P_2).*

Then instances I_s^1 and I_s^2 admit solutions whose total weight (summed over all four paths) is $w(P_1) + w(P_2) + 4w_{a^\star}$. Furthermore, given a solution $\mathcal{S}_i$ for I_s^i for both $i \in [2]$, we can compute in linear time a solution for (G, w, s, t) of weight at most $w(\mathcal{S}_1) + w(\mathcal{S}_2) - 4w_{a^\star}$. The same holds when substituting I_s^1, I_s^2, and $w_{a^\star}$ with I_t^1, I_t^2, and $w_{b^\star}$ in these claims.

Algorithm STDP Solving Shortest Two Disjoint Paths with conservative weights.

```
Input: An instance $(G, w, s, t)$ where $w$ is conservative on $G$.
Output: A solution for $(G, w, s, t)$ with minimum weight, or $\varnothing$ if no solution exist.
 1: Let $\mathcal{S} = \emptyset$.
 2: for all $E' \subseteq E$ such that $E \setminus E' \subseteq E^-$ and $|E \setminus E'| \leq 1$ do        ▷ Separable solutions.
 3:     Create the instance $(G[E'], w_{|E'}, s, t)$.
 4:     Let $\mathcal{T}^{\not\ni s,t} = \{T : T$ is a maximal tree in $G[E' \cap E^-]$ with $s, t \notin V(T)\}$.
 5:     for all $Z \subseteq V$ such that $|Z \cap V(T)| = 1$ for each $T \in \mathcal{T}^{\not\ni s,t}$ do
 6:         Create the network $N_Z$ from instance $(G[E'], w_{|E'}, s, t)$.        ▷ Use Def. 13.
 7:         if $\exists$ a flow $f$ of value 2 in $N_Z$ then
 8:             Compute a minimum-cost flow $f$ of value 2 in $N_Z$.
 9:             Construct a solution $(S_1, S_2)$ from $f$ using Lemma 14.
10:             $\mathcal{S} \leftarrow (S_1, S_2)$.
11: for all $T \in \mathcal{T}$ do        ▷ Non-separable solutions.
12:     for all partitions $(\mathcal{T}_s, \mathcal{T}_0, \mathcal{T}_t)$ of $\mathcal{T} \setminus \{T\}$ do
13:         if $\mathcal{T}_s \neq \emptyset$ then
14:             for all $a_1, a_2 \in V(T)$ with $a_1 \neq a_2$ do
15:                 Create sub-instances $I_s^1$ and $I_s^2$.        ▷ Use Def. 16.
16:                 Compute $(Q_1^{\nearrow}, Q_2^{\nearrow}) = \mathrm{STDP}(I_s^1)$.
17:                 Compute $(Q_1^{\searrow}, Q_2^{\searrow}) = \mathrm{STDP}(I_s^2)$.
18:                 if $(Q_1^{\nearrow}, Q_2^{\nearrow}) \neq \varnothing$ and $(Q_1^{\searrow}, Q_2^{\searrow}) \neq \varnothing$ then
19:                     Create a solution $(S_1, S_2)$ from $(Q_1^{\nearrow}, Q_2^{\nearrow})$ and $(Q_1^{\searrow}, Q_2^{\searrow})$ using Lemma 18.
20:                     $\mathcal{S} \leftarrow (S_1, S_2)$.
21:         else if $\mathcal{T}_t \neq \emptyset$ then
22:             for all $b_1, b_2 \in V(T)$ with $b_1 \neq b_2$ do
23:                 Create sub-instances $I_t^1$ and $I_t^2$.        ▷ Use Def. 17.
24:                 Compute $(Q_1^{\nearrow}, Q_2^{\nearrow}) = \mathrm{STDP}(I_t^1)$.
25:                 Compute $(Q_1^{\searrow}, Q_2^{\searrow}) = \mathrm{STDP}(I_t^2)$.
26:                 if $(Q_1^{\nearrow}, Q_2^{\nearrow}) \neq \varnothing$ and $(Q_1^{\searrow}, Q_2^{\searrow}) \neq \varnothing$ then
27:                     Create a solution $(S_1, S_2)$ from $(Q_1^{\nearrow}, Q_2^{\nearrow})$ and $(Q_1^{\searrow}, Q_2^{\searrow})$ using Lemma 18.
28:                     $\mathcal{S} \leftarrow (S_1, S_2)$.
29:         else        ▷ $\mathcal{T}_s = \mathcal{T}_t = \emptyset$.
30:             for all $a_1, a_2, b_1, b_2 \in V(T)$ that constitute a reasonable guess do
31:                 if $\exists$ a flow $f$ of value 4 in $N_{(a_1,b_1,a_2,b_2)}$ then        ▷ Use Def. 19.
32:                     if $\exists$ two permissively disjoint $(\{a_1, a_2\}, \{b_1, b_2\})$-paths in $G$ then
33:                         Compute a minimum-cost flow $f$ of value 4 in $N_{(a_1,b_1,a_2,b_2)}$.
34:                         Compute permissively disjoint $(\{a_1, a_2\}, \{b_1, b_2\})$-paths $Q_1$ and $Q_2$.
35:                                                         ▷ Use Corollary 33 in Section 4.3
36:                         Construct a solution $(S_1, S_2)$ from $f$, $Q_1$, and $Q_2$ using Lemma 20.
37:                         $\mathcal{S} \leftarrow (S_1, S_2)$.
38: if $\mathcal{S} = \emptyset$ then return $\varnothing$.
39: else Let $S^\star$ be the cheapest pair among those in $\mathcal{S}$, and return $S^\star$.
```

Step 3: Applying flow techniques and dynamic programming

We now describe Step 3 in more detail, which concerns the case when $\mathcal{T}_s = \mathcal{T}_t = \emptyset$.

First, we guess vertices a_1, b_1, a_2, and b_2; the intended meaning of these vertices is that a_i and b_i are the first and last vertices of P_i contained in $V(T)$, for both $i \in [2]$. We only consider guesses that are *reasonable*, meaning that they satisfy the following conditions:

- if $s \in V(T)$, then $s = a_1 = a_2$, otherwise $a_1 \neq a_2$;
- if $t \in V(T)$, then $t = b_1 = b_2$, otherwise $b_1 \neq b_2$;
- $T[a_1, b_1]$ and $T[a_2, b_2]$ share at least one edge.

Then we compute four paths from $\{s, t\}$ to $\{a_1, b_1, a_2, b_2\}$ in the graph $G - E(\mathcal{T}) - (V(\mathcal{T}) \setminus \{a_1, a_2, b_1, b_2\})$ with minimum weight such that two paths have s as an endpoint, the other two have t as an endpoint, and no other vertex appears on more than one path. To this end, we define the following network and compute a minimum-cost flow of value 4 in it.

▶ **Definition 19 (Flow network $N_{(a_1,b_1,a_2,b_2)}$ for non-separable solutions.).** *Given vertices a_1, b_1, a_2, and b_2, we create $N_{(a_1,b_1,a_2,b_2)}$ as follows. First, we delete all edges in E^- and all vertices in $V(\mathcal{T}) \setminus \{a_1, a_2, b_1, b_2\}$ from G, and direct each edge in G in both directions. We then add new vertices $s^\star$ and $t^\star$, along with arcs $(s^\star, s)$ and $(s^\star, t)$ of capacity 2, and arcs $(a_i, t^\star)$ and $(b_i, t^\star)$ for $i = 1, 2$ with capacity 1.*[5] *We assign capacity 1 to all other arcs, and also to each vertex of $V(G) \setminus \{s, t\}$. All newly added arcs will have cost 0, otherwise we retain the cost function w. We let $s^\star$ and $t^\star$ be the source and the sink in $N_{(a_1,b_1,a_2,b_2)}$, respectively.*

Next, we compute two $(\{a_1, a_2\}, \{b_1, b_2\})$-paths Q_1 and Q_2 in G with minimum total weight that are permissively disjoint. A polynomial-time computation for this problem, building on structural observations from Section 4.2, is provided in Section 4.3 (see Corollary 33). Finally, we apply Lemma 10 (in fact, twice) to obtain a solution to our instance (G, w, s, t) based on a minimum-cost flow of value 4 in $N_{(a_1,b_1,a_2,b_2)}$ and paths Q_1 and Q_2. We finish this section with Lemma 20, stating the correctness of Step 3.

▶ **Lemma 20.** *Suppose that (P_1, P_2) is a minimum-weight solution for (G, w, s, t) such that*

- *P_1 and P_2 are non-separable, and in contact at some $T \in \mathcal{T}$,*
- *a_i and b_i are the first and last vertices of P_i contained in T, respectively, for $i \in [2]$, and*
- *$(\emptyset, \mathcal{T} \setminus \{T\}, \emptyset)$ is a T-valid partition w.r.t. (P_1, P_2).*

Let Q_1 and Q_2 be two permissively disjoint $(\{a_1, a_2\}, \{b_1, b_2\})$-paths in G with minimum total weight. Then the following holds:
If $w^\star$ is the minimum cost of a flow of value 4 in the network $N_{(a_1,b_1,a_2,b_2)}$, then $w^\star + w(Q_1) + w(Q_2) \leq w(P_1) + w(P_2)$. Conversely, given a flow of value 4 in the network $N_{(a_1,b_1,a_2,b_2)}$ with cost $w^\star$, together with paths Q_1 and Q_2, we can find a solution for (G, w, s, t) with cost at most $w^\star + w(Q_1) + w(Q_2)$ in linear time.

4.2 Properties of a Non-separable Solution

Let us now turn our attention to the subroutine lying at the heart of our algorithm for Shortest Two Disjoint Paths: an algorithm that, given two source terminals and two sink terminals on some tree $T \in \mathcal{T}$, computes two permissively disjoint paths from the two source terminals to the two sink terminals, with minimum total weight. It is straightforward to see that any non-separable solution whose paths are in contact at T contains such a pair

[5] In the degenerate case when $s = a_1 = a_2$ or $t = b_1 = b_2$ this yields two parallel arcs from s or t to $t^\star$.

of paths. Therefore, as described in Section 4.1.2, finding such paths is a necessary step to computing an optimal, non-separable solution for our instance (G, w, s, t). This section contains observations about the properties of such paths.

Let us now formalize our setting. Let a_1, a_2, b_1, and b_2 be vertices on a fixed tree $T \in \mathcal{T}$ such that $T[a_1, b_1]$ and $T[a_2, b_2]$ intersect in a path X (with at least one edge), with one component of $T \setminus X$ containing a_1 and a_2, and the other containing b_1 and b_2. Let the vertices on X be $x_1, \ldots, x_r$ with x_1 being the closest to a_1 and a_2. We will use the notation $A_i = T[a_i, x_1]$ and $B_i = T[b_i, x_r]$ for each $i \in [2]$. For each $i \in [r]$, let T_i be the maximal subtree of T containing x_i but no other vertex of X. We also define $T_{(i,j)} = \bigcup_{i \le h \le j} T_h$ for some i and j with $1 \le i \le j \le r$.

For convenience, for any path Q that has $a_i \in \{a_1, a_2\}$ as its endpoint, we will say that Q *starts* at a_i and *ends* at its other endpoint. Accordingly, for vertices $u, v \in V(Q)$ we say that u *precedes* v on Q, or equivalently, v *follows* u on Q, if u lies on $Q[a_i, v]$. When defining a vertex as the "first" (or "last") vertex with some property on Q or on a subpath Q' of Q then, unless otherwise stated, we mean the vertex on Q or on Q' that is closest to a_i (or farthest from a_i, respectively) that has the given property.

Using Lemma 2 and (an extended version of) Lemma 3, we can establish the following properties of a non-separable minimum-weight solution.

▶ **Definition 21** (**X-monotone path**). *A path Q starting at a_1 or a_2 is* X-monotone *if for any vertices u_1 and u_2 on Q such that $u_1 \in V(T_{j_1})$ and $u_2 \in V(T_{j_2})$ for some $j_1 < j_2$ it holds that u_1 precedes u_2 on Q.*

▶ **Definition 22** (**Plain path**). *A path Q is* plain*, if whenever Q contains some $x_i \in V(X)$, then the vertices of Q in T_i induce a path in T_i. In other words, if vertices $x_j \in V(X)$ and $u \in V(T_j)$ both appear on Q, then $T[u, x_j] \subseteq Q$.*

▶ **Lemma 23.** *Let Q_1 and Q_2 be two permissively disjoint $(\{a_1, a_2\}, \{b_1, b_2\})$-paths in G with minimum total weight. Then both Q_1 and Q_2 are X-monotone and plain.*

The following observation summarizes our understanding on how an optimal solution uses paths A_1, A_2, B_1, and B_2.

▶ **Lemma 24.** *If Q_1 and Q_2 are two permissively disjoint $(\{a_1, a_2\}, \{b_1, b_2\})$-paths that are locally cheapest and also plain, then one of them contains A_1 or A_2, and one of them contains B_1 or B_2.*

4.3 Computing Partial Solutions

In this section we design a dynamic programming algorithm that computes two permissively disjoint $(\{a_1, a_2\}, \{b_1, b_2\})$-paths of minimum total weight (we keep all definitions introduced in Section 4.2, including our assumptions on vertices a_1, b_1, a_2, and b_2). In Section 4.2 we have established that two permissively disjoint $(\{a_1, a_2\}, \{b_1, b_2\})$-paths of minimum total weight are necessarily X-monotone, plain, and they form a locally cheapest pair. A natural approach would be to require these same properties from a partial solution that we aim to compute. However, it turns out that the property of X-monotonicity is quite hard to ensure when building subpaths of a solution. The following relaxed version of monotonicity can be satisfied much easier, and still suffices for our purposes:

▶ **Definition 25** (**Quasi-monotone path**). *A path P starting at a_1 or a_2 is* quasi-monotone*, if the following holds: if $x_i \in V(P)$ for some $i \in [r]$, then all vertices in $\bigcup_{h \in [i-1]} V(T_h) \cap V(P)$ precede x_i on P, and all vertices in $\bigcup_{h \in [r] \setminus [i]} V(T_h) \cap V(P)$ follow x_i on P.*

▶ **Definition 26 (Well-formed path pair).** *Two paths P_1 and P_2 form a* well-formed pair*, if they are locally cheapest, and both are plain and quasi-monotone.*

We are now ready to define partial solutions, the central notion that our dynamic programming algorithm relies on.

▶ **Definition 27 (Partial solution).** *Given vertices $u \in V(T_i)$ and $v \in V(T_j)$ for some $i \leq j$ and a set $\tau \subseteq \mathcal{T} \setminus \{T\}$, two paths Q_1 and Q_2 form a* partial solution (Q_1, Q_2) for (u, v, τ)*, if*

(a) *Q_1 and Q_2 are permissively disjoint $(\{a_1, a_2\}, \{u, v\})$-paths;*
(b) *Q_1 and Q_2 are a well-formed pair;*
(c) *Q_1 ends with the subpath $T[x_i, u]$;*
(d) *$V(T_{(i+1,r)}) \cap V(Q_2) \subseteq \{v\}$;*
(e) *if $Q_1 \cup Q_2$ contains a vertex of some $T' \in \mathcal{T} \setminus \{T\}$, then $T' \in \tau$;*
(f) *there exists no tree $T' \in \mathcal{T} \setminus \{T\}$ such that Q_1 and Q_2 are in contact at T'.*

We will say that the vertices of $V(\mathcal{T} \setminus (\tau \cup \{T\}))$ are *forbidden* for (τ, T); then condition (e) asks for $Q_1 \cup Q_2$ not to contain vertices forbidden for (τ, T).

Before turning our attention to the problem of computing partial solutions, let us first show how partial solutions enable us to find two permissively disjoint $(\{a_1, a_2\}, \{b_1, b_2\})$-paths.

▶ **Lemma 28.** *Paths P_1 and P_2 are permissively disjoint $(\{a_1, a_2\}, \{b_1, b_2\})$-paths of minimum weight in G if and only if they form a partial solution for $(b_h, b_{3-h}, \mathcal{T} \setminus \{T\})$ of minimum weight for some $h \in [2]$.*

We now present our approach for computing partial solutions using dynamic programming.

Computing partial solutions: high-level view

For each $u \in V(T_i)$ and $v \in V(T_j)$ for some $i \leq j$, and each $\tau \subseteq \mathcal{T} \setminus \{T\}$, we are going to compute a partial solution for (u, v, τ) of minimum weight, denoted by $F(u, v, \tau)$, using dynamic programming; if there exists no partial solution for (u, v, τ), we set $F(u, v, \tau) = \varnothing$.

To apply dynamic programming, we fix an ordering $\prec$ over $V(T)$ fulfilling the condition that for each $i' < i$, $u \in V(T_i)$ and $u' \in V(T_{i'})$ we have $u' \prec u$. We compute the values $F(u, v, \tau)$ based on the ordering $\prec$ in the sense that $F(u', v', \tau')$ is computed before $F(u, v, \tau)$ whenever $u' \prec u$. This computation is performed by Algorithm PartSol which determines a partial solution $F(u, v, \tau)$ based on partial solutions already computed.

To compute $F(u, v, \tau)$ in a recursive manner, we use an observation that either the partial solution has a fairly simple structure, or it strictly contains a partial solution for (u', v', τ') for some vertices u' and v' with $u' \in V(T_{i'})$ and $i' < i$, and some set $\tau' \subseteq \tau$. We can thus try all possible values for u', v' and τ', and use the partial solution (Q_1', Q_2') we have already computed and stored in $F(u', v', \tau')$. To obtain a partial solution for (u, v, τ) based on Q_1' and Q_2', we append paths to Q_1' and to Q_2' so that they fulfill the requirements of Definition 27 – most importantly, that Q_1 ends with $T[x_i, u]$, that Q_2 ends at v, and that $Q_1 \cup Q_2$ contains no vertex of $V(\mathcal{T} \setminus (\tau \cup \{T\}))$. To this end, we create a path $P_1 = Q_2' \cup T[v', u]$ and a path $P_2 = Q_1' \cup R$ where R is a shortest (u', v)-path in a certain auxiliary graph. Essentially, we use the tree T for getting from v' to u, and we use the "remainder" of the graph for getting from u' to v; note that we need to avoid the forbidden vertices and ensure condition (f) as well. The precise definition of the auxiliary subgraph of G that we use for this purpose is provided in Definition 29. If the obtained path pair (P_1, P_2) is indeed a partial solution for (u, v, τ), then we store it. After trying all possible values for u', v', and τ', we select a partial solution that has minimum weight among those we computed.

▶ **Definition 29 (Auxiliary graph).** *For some $T \in \mathcal{T}$, let $P \subseteq T$ be a path within T, let u and v be two vertices on T, and let $\tau \subseteq \mathcal{T} \setminus \{T\}$. Then the* auxiliary graph *$G\langle P, u, v, \tau\rangle$ denotes the graph defined as*

$$G\langle P, u, v, \tau\rangle = G - \left(\bigcup\{V(T_h) : V(T_h) \cap V(P) = \emptyset\} \cup V(P) \setminus \{u, v\} \cup V(\mathcal{T} \setminus (\tau \cup \{T\}))\right).$$

In other words, we obtain $G\langle P, u, v, \tau\rangle$ from G by deleting all trees T_h that do not intersect P, and deleting P itself as well, while taking care not to delete u or v, and additionally deleting all vertices forbidden for (τ, T).

Working towards explaining the main ideas behind Algorithm PartSol, we start with two simple observations. The first one, stated by Lemma 30 below, essentially says that a path in a partial solution that uses a subtree T_h of T for some $h \in [r]$ should also go through the vertex x_h whenever possible, that is, unless the other path uses x_h.

▶ **Lemma 30.** *Let Q_1 and Q_2 be two permissively disjoint, locally cheapest $(\{a_1, a_2\}, \{x_i, v\})$-paths for some $v \in V(T_j)$ where $1 \leq i \leq j \leq r$. Let $z \in V(T_h)$ for some $h \leq j$ such that $h < j$ or $x_h \in V(T[z, v])$. If $z \in V(Q_1 \cup Q_2)$, then $x_h \in V(Q_1 \cup Q_2)$.*

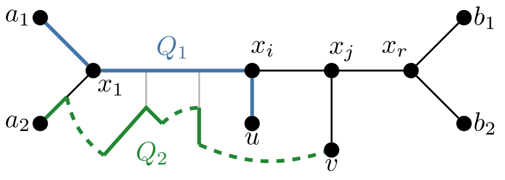
As a consequence of Lemma 30, applied with a_1 taking the role of z and x_1 taking the role of x_h, we get that every partial solution for some (u, v, τ) must contain x_1; using that both paths in a partial solution must be plain, we get the following fact.

▶ **Observation 31.** *Let (Q_1, Q_2) be a partial solution for (u, v, τ) for vertices $u \in V(T_i)$ and $v \in V(T_j)$ for some $i \leq j$ and for $\tau \subseteq \mathcal{T} \setminus \{T\}$. Then either Q_1 or Q_2 contains A_1 or A_2.*

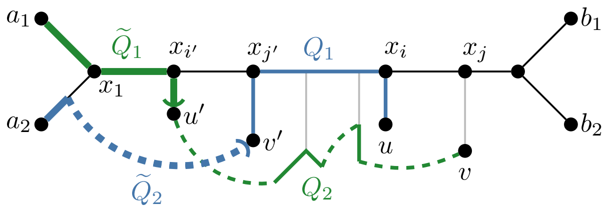
Let us now give some insight on Algorithm PartSol that computes a minimum-weight partial solution (Q_1, Q_2) for (u, v, τ), if it exists, for vertices $u \in V(T_i)$ and $v \in V(T_j)$ with $i \leq j$ and trees $\tau \subseteq \mathcal{T} \setminus \{T\}$. It distinguishes between two cases based on whether Q_2 contains a vertex of $T[x_1, x_i]$ or not; see Figure 1 for an illustration. In both cases it constructs candidates for a partial solution, and then chooses one among these with minimum weight.

Case A: Q_2 does not contain any vertices from $T[x_1, x_i]$. In this case, due to Observation 31, we know that Q_1 contains A_h for some $h \in [2]$; let us fix this value of h. Since Q_1 and Q_2 are locally cheapest and Q_1 ends with $T[x_i, u]$, we also know that Q_1 must contain $T[x_1, x_i]$. Therefore, we obtain $Q_1 = A_h \cup T[x_1, u]$. In this case, we can also prove that Q_2 is a shortest (a_{3-h}, v)-path in the auxiliary graph $G\langle A_h \cup T[x_1, u], a_{3-h}, v, \tau\rangle$. Hence, Algorithm PartSol computes such a path R and constructs the pair $(A_h \cup T[x_1, u], R)$ as a candidate for a partial solution for (u, v, τ).

Case B: Q_2 contains a vertex from $T[x_1, x_i]$. In this case, let $x_{i'}$ be the vertex on $T[x_1, x_{i-1}]$ closest to x_i that appears on Q_2, and let u' be the last vertex of Q_2 in $T_{i'}$; since Q_2 is plain, we know $T[x_{i'}, u'] \subseteq Q_2$. Let $x_{j'}$ denote the vertex on $T[x_{i'}, x_i]$ closest to $x_{i'}$ that appears on Q_1; then $i' < j' \leq i$. As Q_1 and Q_2 are locally cheapest, $T[x_{j'}, x_i] \subseteq Q_1$ follows. Let v' denote the first vertex of Q_1 in $T_{j'}$. Since Q_1 is plain, we know $T[v', x_{j'}] \subseteq Q_1$.
Define $\widetilde{Q}_1 = Q_2 \setminus Q_2[u', v]$ and $\widetilde{Q}_2 = Q_1 \setminus Q_1[v', u]$. Let also τ' denote those trees in $\mathcal{T} \setminus \{T\}$ that share a vertex with $\widetilde{Q}_1 \cup \widetilde{Q}_2$. We can then prove that $(\widetilde{Q}_1, \widetilde{Q}_2)$ is a partial solution for (u', v', τ'); moreover, $Q_2[u', v]$ is a path in the auxiliary graph $G\langle T[v', u], u', v, \tau \setminus \tau'\rangle$. Thus, Algorithm PartSol takes a partial solution (Q'_1, Q'_2) for (u', v', τ'), already computed, and computes a shortest (u', v)-path R in $G\langle T[v', u], u', v, \tau \setminus \tau'\rangle$. It then creates the path pair $(Q'_2 \cup T[v', u], Q'_1 \cup R)$ as a candidate for a partial solution for (u, v, τ).



(a) Case A. The figure assumes $Q_1 = A_1 \cup T[x_1, u]$ (so $h = 1$).



(b) Case B. The subpaths $\widetilde{Q}_2$ and $\widetilde{Q}_1$ of Q_1 and Q_2, respectively, form a partial solution for (u', v', τ') and are depicted in bold, with their endings marked by a parenthesis-shaped delimiter.

Figure 1 Illustration for Algorithm PartSol. Edges within T are depicted using solid lines, edges not in T using dashed lines. Paths Q_1 and Q_2 are shown in **blue** and in **green**, respectively (see the online version for colored figures).

Algorithm PartSol Computes a partial solution $F(u, v, \tau)$ of minimum weight for (u, v, τ) where $u \in V(T_i)$ and $v \in V(T_j)$ with $i \leq j$, and $\tau \subseteq \mathcal{T} \setminus \{T\}$.

```
Input: Vertices $u$ and $v$ where $u \in V(T_i)$ and $v \in V(T_j)$ for some $i \leq j$, and a set $\tau \subseteq \mathcal{T} \setminus \{T\}$.
Output: A partial solution $F(u, v, \tau)$ for $(u, v, \tau)$ of minimum weight, or $\varnothing$ if not existent.
 1: Let $\mathcal{S} = \emptyset$.
 2: for all $h \in [2]$ do
 3:     if $A_h \cup T[x_1, u]$ is a path then
 4:         if $v$ is reachable from $a_{3-h}$ in $G\langle A_h \cup T[x_1, u], a_{3-h}, v, \tau\rangle$ then
 5:             Compute a shortest $(a_{3-h}, v)$-path $R$ in $G\langle A_h \cup T[x_1, u], u, a_{3-h}, v, \tau\rangle$.
 6:             if $(A_h \cup T[x_1, u], R)$ is a partial solution for $(u, v, \tau)$ then
 7:                 $\mathcal{S} \leftarrow (A_h \cup T[x_1, u], R)$.
 8: for all $i' \in [i-1]$ and $u' \in V(T_{i'})$ do
 9:     for all $j' \in [i] \setminus [i']$ and $v' \in V(T_{j'})$ such that $T[x_{j'}, v'] \cap T[x_i, u] = \emptyset$ do
10:         for all $\tau' \subseteq \tau$ do
11:             if $F(u', v', \tau') = \varnothing$ then continue;
12:             Let $(Q'_1, Q'_2) = F(u', v', \tau')$.
13:             if $v$ is not reachable from $u'$ in $G\langle T[v', u], u', v, \tau \setminus \tau'\rangle$ then continue;
14:             Compute a shortest $(u', v)$-path $R$ in $G\langle T[v', u], u', v, \tau \setminus \tau'\rangle$.
15:             Let $P_1 = Q'_2 \cup T[v', u]$ and $P_2 = Q'_1 \cup R$.
16:             if $(P_1, P_2)$ is a partial solution for $(u, v, \tau)$ then
17:                 $\mathcal{S} \leftarrow (P_1, P_2)$.
18: if $\mathcal{S} = \emptyset$ then return $\varnothing$.
19: else Let $S^\star$ be the cheapest pair among those in $\mathcal{S}$, and return $F(u, v, \tau) := S^\star$.
```

The following lemma guarantees the correctness of Algorithm PartSol. Formally, we say that $F(u, v, \tau)$ is *correctly computed* if either it contains a minimum-weight partial solution for (u, v, τ), or no partial solution for (u, v, τ) exists and $F(u, v, \tau) = \varnothing$.

▶ **Lemma 32.** *Let $i, j \in [r]$ with $i \leq j$, $u \in V(T_i)$, $v \in V(T_j)$ and $\tau \subseteq \mathcal{T} \setminus \{T\}$. Assuming that the values $F(u', v', \tau')$ are correctly computed for each $u' \in V(T_{i'})$ with $i' < i$, Algorithm PartSol correctly computes $F(u, v, \tau)$.*

Using the correctness of Algorithm PartSol, as established by Lemma 32, and the observation in Lemma 28 on how partial solutions can be used to find two permissively disjoint $(\{a_1, a_2\}, \{b_1, b_2\})$-paths of minimum total weight, we obtain the following.

▶ **Corollary 33.** *For each constant $c \in \mathbb{N}$, there is a polynomial-time algorithm that finds two permissively disjoint $(\{a_1, a_2\}, \{b_1, b_2\})$-paths of minimum total weight in G (if such paths exist), where the set of negative edges in G spans c trees.*

5 Conclusion

We have presented a polynomial-time algorithm for solving the Shortest Two Disjoint Paths problem on undirected graphs G with conservative edge weights, assuming that the number of connected components in the subgraph $G[E^-]$ spanned by all negative-weight edges is a fixed constant c. The running time of our algorithm is $O(n^{2c+9})$ on an n-vertex graph. Is it possible to give a substantially faster algorithm for this problem? In particular, is it possible to give a fixed-parameter tractable algorithm for Shortest Two Disjoint Paths on undirected conservative graphs when parameterized by c?

More generally, is it possible to find in polynomial time k openly disjoint (s, t)-paths with minimum total weight for some fixed $k \geq 3$ in undirected conservative graphs with constant values of c?

References

1 Isolde Adler, Stavros G. Kolliopoulos, Philipp Klaus Krause, Daniel Lokshtanov, Saket Saurabh, and Dimitrios M. Thilikos. Irrelevant vertices for the Planar Disjoint Paths problem. *Journal of Combinatorial Theory, Series B*, 122:815–843, 2017. doi:10.1016/j.jctb.2016.10.001.

2 Aaron Bernstein, Danupon Nanongkai, and Christian Wulff-Nilsen. Negative-weight single-source shortest paths in near-linear time. In *FOCS 2022: Proceedings of the 63rd Annual IEEE Symposium on Foundations of Computer Science*, pages 600–611. IEEE Computer Society, 2022. doi:10.1109/FOCS54457.2022.00063.

3 Andreas Björklund and Thore Husfeldt. Shortest two disjoint paths in polynomial time. *SIAM Journal on Computing*, 48(6):1698–1710, 2019. doi:10.1137/18M1223034.

4 Marek Cygan, Dániel Marx, Marcin Pilipczuk, and Michal Pilipczuk. The planar directed k-vertex-disjoint paths problem is fixed-parameter tractable. In *FOCS 2013: Proceedings of the 54th Annual IEEE Symposium on Foundations of Computer Science*, pages 197–206. IEEE Computer Society, 2013. doi:10.1109/FOCS.2013.29.

5 Éric Colin de Verdière and Alexander Schrijver. Shortest vertex-disjoint two-face paths in planar graphs. *ACM Trans. Algorithms*, 7(2):19:1–19:12, 2011. doi:10.1145/1921659.1921665.

6 Guoli Ding, A. Schrijver, and P. D. Seymour. Disjoint paths in a planar graph – a general theorem. *SIAM Journal on Discrete Mathematics*, 5(1):112–116, 1992. doi:10.1137/0405009.

7 Richard M. Karp. On the computational complexity of combinatorial problems. *Networks*, 5(1):45–68, 1975.

8 Yusuke Kobayashi and Christian Sommer. On shortest disjoint paths in planar graphs. *Discrete Optimization*, 7(4):234–245, 2010. doi:10.1016/j.disopt.2010.05.002.

9 Daniel Lokshtanov, Pranabendu Misra, Michał Pilipczuk, Saket Saurabh, and Meirav Zehavi. An exponential time parameterized algorithm for planar disjoint paths. In *STOC 2020: Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, pages 1307–1316. Association for Computing Machinery, 2020. `doi:10.1145/3357713.3384250`.

10 James F. Lynch. The equivalence of theorem proving and the interconnection problem. *ACM SIGDA Newsletter*, 5:31–36, 1975. `doi:10.1145/1061425.1061430`.

11 Neil Robertson and P.D. Seymour. Graph minors. XIII. The disjoint paths problem. *Journal of Combinatorial Theory, Series B*, 63(1):65–110, 1995. `doi:10.1006/jctb.1995.1006`.

12 Ildikó Schlotter. Shortest two disjoint paths in conservative graphs, 2023. `arXiv:2307.12602`.

13 Ildikó Schlotter and András Sebő. Odd paths, cycles and T-joins: Connections and algorithms, 2022. `arXiv:2211.12862`.

14 Alexander Schrijver. Finding k disjoint paths in a directed planar graph. *SIAM Journal on Computing*, 23(4):780–788, 1994. `doi:10.1137/S0097539792224061`.

15 Alexander Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*. Springer, Berlin, 2003.