BEATING THE WORST-CASE $O(n^2)$ PER-ITERATION UPDATE TIME: A NOVEL DYNAMIC NETWORK SIMPLEX METHOD FOR OPTIMAL TRANSPORT

Anonymous authors
Paper under double-blind review

ABSTRACT

Optimal Transport is widely applied in fields such as computer vision, machine learning, and operations research. It seeks a minimum-cost transport plan from source to target domain under supply and demand constraints. However, optimal transport problems in dynamic scenarios face the challenge of high complexity in real-time adjustments to transport plans. To overcome this computational barrier, we aim to develop an efficient dynamic algorithm for the seminal NETWORK SIMPLEX method. We find the primary difficulty lies in the rapid and effective maintenance of the reduced cost matrix, with up to n^2 entries ("n" is the number of input points). To address this, we propose a novel "Bi-resolution Matrix Decomposition Tree" technique for maintaining matrix operations. By indexing the reduced cost matrix with Euler tour sequences and maintaining a hierarchical bi-resolution block decomposition, it reduces the worst-case time per simplex iteration to $\mathrm{O}(n)$. Finally, we demonstrate our method's effectiveness through experiments on synthetic and real-world datasets.

1 INTRODUCTION

Optimal Transport (OT) provides a principled mathematical framework for comparing probability distributions by explicitly incorporating the geometry of the underlying space (Villani et al., 2008; Peyré & Cuturi, 2019). OT has applications in many fields, such as computer vision (Rubner et al., 2000; Courty et al., 2016), generative models (Arjovsky et al., 2017; Gulrajani et al., 2017), graphics (Li et al., 2025a;b), medical imaging (Kim et al., 2025; Lei et al., 2024). Given two point sets $\mathcal{A} = \{a_1, \ldots, a_{n_\mathcal{A}}\}$ and $\mathcal{B} = \{b_1, \ldots, b_{n_\mathcal{B}}\}$, we define the function $cost(\cdot, \cdot)$ as the pairwise distance between any two points of them. Suppose $C = (c_{ij} \triangleq cost(a_i, b_j))_{n_\mathcal{A} \times n_\mathcal{B}} \in \mathbb{R}_+^{n_\mathcal{A} \times n_\mathcal{B}}$ is the **cost matrix** between $\mathcal{A}$ and $\mathcal{B}$. Let $\boldsymbol{\alpha} = [\alpha_1, \cdots, \alpha_{n_\mathcal{A}}] \in \mathbb{R}_+^{n_\mathcal{A}}, \boldsymbol{\beta} = [\beta_1, \cdots, \beta_{n_\mathcal{B}}] \in \mathbb{R}_+^{n_\mathcal{B}}$ be their corresponding nonnegative weight vectors, normalized so that $\sum_{i=1}^{n_\mathcal{A}} \alpha_i = \sum_{j=1}^{n_\mathcal{B}} \beta_j = 1$. Any balanced instance with common total mass $M > 0$ can be normalized by dividing both weight vectors by M. The OT problem seeks an optimal **coupling matrix** $W = (w_{ij})_{n_\mathcal{A} \times n_\mathcal{B}}$ that minimizes the **transportation cost**:

$$P(\mathcal{A}, \mathcal{B}, \boldsymbol{\alpha}, \boldsymbol{\beta}) \triangleq \min_{W \in \mathbb{R}^{n_\mathcal{A} \times n_\mathcal{B}}} \sum_{i=1}^{n_\mathcal{A}} \sum_{j=1}^{n_\mathcal{B}} c_{ij} w_{ij} \tag{1}$$

with the constraints: 1) $\sum_{j=1}^{n_\mathcal{B}} w_{ij} = \alpha_i, \forall 1 \le i \le n_\mathcal{A}$; 2) $\sum_{i=1}^{n_\mathcal{A}} w_{ij} = \beta_j, \forall 1 \le j \le n_\mathcal{B}$; 3) $w_{ij} \ge 0, \forall 1 \le i \le n_\mathcal{A}, 1 \le j \le n_\mathcal{B}$. Without loss of generality, we assume both $n_\mathcal{A}$ and $n_\mathcal{B}$ are $\Theta(n)$. We can interpret OT as a min-cost flow problem. For every $a_i \in \mathcal{A}$, there is a flow of α_i to be transmitted to points in $\mathcal{B}$; for every $b_j \in \mathcal{B}$, there is a flow of β_j to be received. Several algorithmic families have become central to optimal transport computation. The seminal NETWORK SIMPLEX method (Cunningham, 1976; Orlin, 1997; Lahn et al., 2019) provides an **exact**, **combinatorial** solution with the ability to exploit sparsity through its tree-based structure. Another line of work, starting from Cuturi (2013), introduces entropic regularization to the objective equation 1, followed by its variations such as those proposed by Chakrabarty & Khanna (2018); Dvurechensky et al. (2018); Altschuler et al. (2017). The SINKHORN algorithm is differentiable and scalable (Eisenberger et al., 2022), while the NETWORK SIMPLEX method outperforms the entropic regularization method when precision is important (Dong et al., 2020; Lin et al., 2022).

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Modern applications of OT involve dynamic environments where data arrives continuously (Mensch & Peyré, 2020; Nguyen, 2025). Take single-cell studies for instance, OT must cope with source and target distributions that evolve over time, forcing the transport map itself to change accordingly (Halmos et al., 2025). When the cost matrix or some weights are modified, traditional methods like the NETWORK SIMPLEX algorithm incur a time complexity of at least $\Omega(n^2)$ to re-optimize. Motivated by the observation that dynamic modifications rarely alter the global structure of the solution significantly, Xu & Ding (2024) propose a novel data structure—the 2D SKIP ORTHOGONAL LIST—combined with dynamic tree techniques. This approach facilitates simplex iterations in expected linear time, offering a scalable alternative to static re-solvers. However, their approach is a randomized algorithm with only an expected time complexity guarantee, with the worst-case complexity still being $\Theta(n^2)$.

Recently, a promising line of research (Sharathkumar & Agarwal, 2012; Lahn et al., 2025) has explored geometric methods, such as Bichromatic Closest Pair (BCP) algorithms (Eppstein, 1995; Chan, 2020), to accelerate optimal transport computations. These elegant approaches leverage spatial data structures and proximity to efficiently approximate the transport cost. However, while these geometric techniques offer significant theoretical speedups, they are typically designed to provide approximate rather than exact solutions. Additionally, because they rely heavily on the properties of metric spaces, it is difficult to generalize these methods to arbitrary cost structures, and their complexity is often bottlenecked by spatial properties such as the spread parameter Δ and dimensionality. Furthermore, directly adapting these structures to dynamic scenarios presents inherent challenges. It is difficult to straightforwardly apply them to the complex, localized structural updates required during exact dynamic iterations.

In addition, several combinatorial optimization problems can be reduced to special cases of optimal transport. For example, the *Euclidean bi-chromatic matching* problem can be cast as an OT instance with uniform marginals. In this setting, Goranci et al. (2025) proposed the first fully dynamic algorithm with *sub-linear update time* ($\mathrm{O}(n^{\varepsilon})$), where $0 < \varepsilon < 1$ controls the trade-off between update speed and approximation quality. Their nested-grid framework, built upon implicit representations of matchings, achieves asymptotically faster updates than any prior static recomputation. Nevertheless, this approach provides an expected $\mathrm{O}(1/\varepsilon)$-approximation guarantee, that is, the obtained solution can be as large as $\mathrm{O}(1/\varepsilon)$ of the optimal cost. Furthermore, the branching factor p^d of the grid structure grows exponentially with dimension, where p is the number of subdivisions along each coordinate axis and d is the dimension of the point space, rendering the method difficult to scale to high-dimensional regimes.

1.1 OUR CONTRIBUTIONS

We aim to develop an efficient solution for dynamic modifications of OT. In particular, we consider the following two **basic operations** and their composition (note that any operation should not violate the constraint that the total weight remains 1 for both $\mathcal{A}$ and $\mathcal{B}$):

1. **Weight modification:** Transferring a certain amount of weight from one node to another in $\mathcal{A}$ or $\mathcal{B}$ (e.g., the weight of a_{i_1} is increased by $0 \leq \delta \leq \alpha_{i_2}$ and the weight of a_{i_2} is decreased by δ, so that the total weight of $\mathcal{A}$ remains 1).
2. **Position modification:** Move a node to another position (therefore, the corresponding column or row in the cost matrix C of Objective equation 1 may change).

Moreover, to enable node insertion and deletion while preserving the total weight constraint, we maintain a pool of zero-weight **auxiliary nodes**. We call these two modifications "basic operations" since other updates can be reduced to a combination of them. Specifically, inserting a node involves retrieving a placeholder from the pool, repositioning it via position modification, and assigning mass via weight modification; deleting a node entails zeroing its weight and returning it to the pool. Node insertions and deletions are thus naturally subsumed within this framework.

Why is dynamic OT challenging? Developing efficient and exact algorithms for dynamic OT presents a fundamental trade-off. For combinatorial methods like the NETWORK SIMPLEX algorithm, the primary challenge does not lie in reducing the number of iterations—since the starting point of iteration is typically close to the new optimum after a dynamic update, the iteration count is already low, leaving little room for significant improvement through iterative strategies. **The real bottleneck lies in the high per-iteration complexity**: to maintain global optimality in the linear programming duality sense, each iteration necessitates a global inspection and feasibility correction of the solution.

This operation entails an intrinsic computational cost that is difficult to reduce, making it the principal efficiency constraint in dynamic settings. While the SINKHORN algorithm enjoys low per-iteration cost, its number of iterations is highly sensitive to the regularization parameter and suffers from unstable convergence in dynamic environments, constituting a distinct and complementary challenge. Furthermore, spatial data structures are generally difficult to adapt to dynamic scenarios. Although there have been some attempts to adapt them to dynamic settings, **a single-point dynamic adjustment can trigger batch changes in the transport plan**. Consequently, dynamic spatial data structures often struggle to handle such extensive variations.

To address these challenges, we propose a novel solution within the traditional NETWORK SIMPLEX method. The NETWORK SIMPLEX method solves optimal transport problems algebraically, whose core operation revolves around maintaining a "reduced cost matrix". Through appropriate transformations, we reformulate the iterative process of the simplex method as a matrix maintenance problem, with detailed procedures presented in Section 2.

However, maintaining an $n \times n$ matrix presents significant challenges. We need to support *matrix splitting, merging, global value addition, and minimum value queries*, with splitting and merging operations being particularly difficult to maintain efficiently. To overcome these obstacles, we design a block decomposition algorithm. Specifically, we partition the reduced cost matrix according to Euler tour sequences, constructing a hierarchical block structure that facilitates efficient splitting and merging through divide-and-conquer techniques. To accelerate the maintenance of single-layer block structures, we design a **Bi-resolution Matrix Decomposition Tree (BMD Tree)** by adding a special layer of blocks between the two layers to reduce maintenance complexity. By dynamically adjusting block sizes, we prevent extreme block length variations, thereby maintaining favorable complexity bounds. Overall, the algorithm achieves $\mathrm{O}(\mathbf{n})$ **complexity** per simplex iteration.

Compared with previous methods for dynamic OT (Xu & Ding, 2024; Goranci et al., 2025; Lahn et al., 2025), our BMD Tree achieves strict worst-case linear reduced-cost maintenance per simplex iteration while maintaining exact optimality, without the dimensional limitations of spatial partitioning or the uncertainty of randomized data structures.

2 PRELIMINARIES

Due to the space limit, we leave the full introduction on related works to appendix A. In this section, we present some important notations and properties of OT, and the NETWORK SIMPLEX method, which will be used throughout this paper.

2.1 NOTATIONS AND PROPERTIES OF OT

The Optimal Transport problem aims to find the minimal total cost required to transmit all flows from $\mathcal{A}$ to $\mathcal{B}$ according to the given constraints as the objective (1). Connecting all points in $\mathcal{A}$ to all points in $\mathcal{B}$ forms a **bipartite graph**, and we denote it as $G(\mathcal{A}, \mathcal{B})$. For each edge (a_i, b_j) in this graph, $c_{ij} \in \mathbb{R}_{\geq 0}$ denotes the cost of transmitting a unit flow from point a_i to point b_j. For any matrix $W = (w_{ij}) \in \mathbb{R}_{\geq 0}^{n_{\mathcal{A}} \times n_{\mathcal{B}}}$ satisfying $w_{ij} \geq 0$ for all $a_i \in \mathcal{A}$ and $b_j \in \mathcal{B}$, we refer to W as a *transportation plan*. An edge (a_i, b_j) is termed an *active edge* if $w_{ij} > 0$. We rely on the following Proposition from Peyré & Cuturi (2019, Proposition 3.4) on the existence of tree solution for OT:

Proposition 2.1. *Given an instance of the OT problem, there always exists a feasible transportation plan achieving the optimal cost, whose active edges reside on a spanning tree of the original bipartite graph $G(\mathcal{A}, \mathcal{B})$; that is, these edges form no cycle when edge directions are disregarded.*

We place the proof for this proposition in Section D. With this result, we can simply focus on the case that all active edges are on a spanning tree. Further, we introduce the duality structure of OT as a linear programming (LP) problem:

Definition 2.2 (**Dual Solution of the OT Problem**).

$$\begin{aligned} (\mathcal{A}, \mathcal{B}, \boldsymbol{\alpha}, \boldsymbol{\beta}) \triangleq \max_{h^{\mathcal{A}} \in \mathbb{R}^{n_{\mathcal{A}}}, h^{\mathcal{B}} \in \mathbb{R}^{n_{\mathcal{B}}}} & \sum_{i=1}^{n_{\mathcal{A}}} \alpha_i h_i^{\mathcal{A}} - \sum_{j=1}^{n_{\mathcal{B}}} \beta_j h_j^{\mathcal{B}} \\ \text{s.t. } h_i^{\mathcal{A}} - h_j^{\mathcal{B}} \leq c_{ij} \ \forall 1 \leq i \leq n_{\mathcal{A}}, 1 \leq j \leq n_{\mathcal{B}} & \end{aligned} \tag{2}$$

The two vectors $h^{\mathcal{A}} = [h_1^{\mathcal{A}}, \cdots, h_{n_{\mathcal{A}}}^{\mathcal{A}}]$ and $h^{\mathcal{B}} = [h_1^{\mathcal{B}}, \cdots, h_{n_{\mathcal{B}}}^{\mathcal{B}}]$ are dual variables.

Definition 2.3 (T_{ot}**-induced solutions**). For any given spanning T_{ot} of the bipartite graph $G(\mathcal{A}, \mathcal{B})$, if there is a feasible primal solution supported on T_{ot} (i.e. only take active edge from T_{ot}), we call it a T_{ot}-induced feasible primal solution; if there is a feasible dual solution satisfying $h_i^{\mathcal{A}} - h_j^{\mathcal{B}} = c_{ij}$, $\forall (a_i, b_j) \in T_{ot}$, we call it a T_{ot}-induced feasible dual solution.

The following proposition bounds the time we need to find a solution from a spanning tree, proof of which is shown in Section E.

Proposition 2.4. *For any spanning tree T_{ot}, if it can induce a feasible primal (resp., dual) solution, one can explicitly construct this solution in* $\mathrm{O}(n)$.

By the strong duality theorem and the Karush-Kuhn-Tucker conditions of LP, we have the following result from Peyré & Cuturi (2019):

Proposition 2.5. *If a spanning tree T_{ot} simultaneously induces feasible primal and feasible dual solutions, then these two solutions are optimal for the primal and dual problems, respectively.*

2.2 Network Simplex Method

We briefly introduce the Network Simplex method (Cunningham, 1976) here. The dual optimal transport problem (2) requires $c_{ij} - h_i^{\mathcal{A}} + h_j^{\mathcal{B}} \geq 0$. The left side of this inequality can then be considered a function of a_i and b_j, which quantifies the slack of the dual solution, leading to the following definition.

Definition 2.6 (**Reduced cost matrix** R_{rc}). We define the reduced cost between any $1 \leq i \leq n_{\mathcal{A}}$ and $1 \leq j \leq n_{\mathcal{B}}$ as $r_{ij} = c_{ij} - h_i^{\mathcal{A}} + h_j^{\mathcal{B}}$. The matrix $R_{rc} \triangleq (r_{ij})_{n^{\mathcal{A}} \times n^{\mathcal{B}}}$ composed of these reduced costs is called the reduced cost matrix induced by the dual variables $h^{\mathcal{A}}$ and $h^{\mathcal{B}}$.

According to Definition 2.2, a dual solution is feasible if and only if all entries in its reduced cost matrix R_{rc} are non-negative. The Network Simplex method is built upon this key property, which is an efficient algorithm specifically designed to the minimum cost flow problem (Ahuja et al., 2001), of which the optimal transport problem is a special case. Prior to a detailed description of the algorithm, we need to introduce several key related concepts first.

Definition 2.7 (**Residual Network**). We define a directed edge (a_i, b_j) from a source $a_i \in \mathcal{A}$ to a sink $b_j \in \mathcal{B}$ as a **forward edge**. Conversely, a directed edge (b_j, a_i) from a sink $b_j \in \mathcal{B}$ to a source $a_i \in \mathcal{A}$ is called a **reverse edge**. A **residual network** consists of forward and reverse edges, where:

- A forward edge (a_i, b_j) has a cost of c_{ij} and a capacity of ∞.
- A reverse edge (b_j, a_i) has a cost of $\tilde{c}_{ji} = -c_{ij}$ and a capacity equal to the current flow on the corresponding forward edge (a_i, b_j).

Within a residual network, transferring w units of flow along a reverse edge (b_j, a_i) corresponds to reducing the flow on the forward edge (a_i, b_j) by w units in the original network—a mechanism that facilitates the dynamic adjustment of an optimal transport solution. This framework also provides a clearer interpretation of the dual variables in equation 2: for any source $a_i \in \mathcal{A}$ and sink $b_j \in \mathcal{B}$ on a spanning tree, $h_i^{\mathcal{A}} - h_j^{\mathcal{B}}$ represents the cost of transporting goods from a_i to b_j, while $h_j^{\mathcal{B}} - h_i^{\mathcal{A}}$ gives the cost of transporting them back. Consequently, the reduced cost $r_{ij} = c_{ij} - h_i^{\mathcal{A}} + h_j^{\mathcal{B}}$ reflects the net cost of sending goods directly from a_i to b_j and then returning an equal quantity via the tree path. When $r_{ij} < 0$, a negative-cost cycle exists, allowing flow reduction to lower total cost; conversely, if $r_{ij} \geq 0$ for all $a_i \in \mathcal{A}, b_j \in \mathcal{B}$, no further cost-reducing adjustments are possible, indicating an optimal solution and satisfying the dual feasibility condition.

The Network Simplex method is an iterative optimization algorithm that operates on a spanning tree structure (due to Proposition 2.1). The algorithm begins with a spanning tree T_{ot} which can induce a feasible primal solution (as described in Definition 2.3), and computes the corresponding reduced cost matrix R_{rc}. Then repeat the following steps until no item in R_{rc} is negative:

Select an edge $(a_i, b_j) \notin T_{ot}$ with the minimum reduced cost. Identify a cycle in the residual network formed by the entering edge (a_i, b_j) and the tree path. Send the maximum possible flow along this cycle until the capacity of a reverse edge is reached. Remove that reverse edge from the spanning tree T_{ot} and replace it with (a_i, b_j), forming a new spanning tree. Update the dual variable and recompute the reduced cost matrix R_{rc}.

Each execution of these steps is a simplex iteration (the pseudocode is shown in Algorithm 1). We use Cunningham's leaving-arc rule to prevent cycling (Section G). Obviously, the process of finding the **minimum reduced cost** and **recomputing** R_{rc} both take $\Theta(n^2)$ time. In Section 3, we show how our new technique reduces this complexity to $\mathrm{O}(n)$ in dynamic scenarios.

3 BI-RESOLUTION MATRIX DECOMPOSITION TREE TECHNIQUE

We propose a **Bi-resolution Matrix Decomposition Tree (BMD Tree)** technique that supports splitting, merging, uniform additive updates, and global minimum queries on R_{rc}. By hierarchically partitioning the matrix and employing lazy tag, our method avoids explicit quadratic updates while preserving exact optimality. We begin in Section 3.1 by presenting the overall framework. The details of our core technique BMD Tree are elaborated in Section 3.2. Finally, Section 3.3 provides a rigorous complexity analysis, establishing an $\mathrm{O}(n)$ worst-case update time per simplex iteration.

3.1 THE OVERALL FRAMEWORK

We assume that the optimal solution before the modification is induced by a spanning tree T_{ot}. Now, we consider how to design our dynamic algorithm based on T_{ot} and a coming update operation $\mathcal{O}$. As mentioned in Section 1.1, we can assume that $\mathcal{O}$ is one of the two basic operations: position modification and weight modification. After obtaining the modified spanning tree, through Proposition 2.4, we can directly achieve the updated optimal primal solution for the objective (1). We discuss the two cases separately.

Case 1: $\mathcal{O}$ is a position modification, that is, the cost matrix C in objective equation 1 changes while the weights $\boldsymbol{\alpha}, \boldsymbol{\beta}$ remain unchanged. Then, the T_{ot}-induced primal solution is still feasible, but the dual solution may not be feasible. In this case, we only need to run the NETWORK SIMPLEX method (Algorithm 1) with T_{ot} as the initial spanning tree.

Case 2: $\mathcal{O}$ is a weight modification, then the problem becomes more complicated. In this case, the weights α and β are changed, so the T_{ot}-introduced primal solution becomes not feasible, but the dual solution is still feasible. Without loss of generality, let G_{res} denote the residual network corresponding to the optimal transportation plan before the modification. If $\mathcal{O}$ reduces the weight of a_{i_1} and increases that of a_{i_2} by δ, this corresponds to pushing δ units of flow from a_{i_2} to a_{i_1} in G_{res}. Symmetrically, reducing the weight of b_{j_1} and increasing that of b_{j_2} by δ corresponds to pushing δ units from b_{j_1} to b_{j_2}. Without loss of generality, assume that we need to transport flow from a_{i_2} to a_{i_1}. If this transportation can be completed, the algorithm terminates. Otherwise, this implies that there exists a reverse edge (b_{j_0}, a_{i_0}) on the path whose capacity has been reduced to zero. Removing this edge splits the tree T_{ot} into two sub-trees, T_1 and T_2, where $a_{i_1} \in T_1$ and $a_{i_2} \in T_2$. Then the remaining task is to find a pair $a_{i'} \in T_2$ and $b_{j'} \in T_1$ so that $r_{i'j'}$ is the minimum among all possible such pairs, add the corresponding edge, and continue transporting flow until the required transportation is completed. This procedure is detailed in Algorithm 2.

In our Algorithm 1 and Algorithm 2, we temporarily ignore two technical issues: **how to cut or link the spanning tree T_{ot} and how to maintain the reduced cost matrix R_{rc}**. For the first issue, the spanning tree T_{ot} can be maintained in $\mathrm{O}(n)$ time while using depth first search, and can be accelerated to amortized $\mathrm{O}(\log n)$ using Link-Cut Trees (LCT) (Sleator & Tarjan, 1981), as discussed in the Section C. In either case, the complexity of maintaining T_{ot} is not dominant in the whole algorithm. However, maintaining the reduced cost matrix R_{rc} is challenging and constitutes the primary computational bottleneck of our algorithm. To resolve this obstacle, we propose a novel **Bi-resolution Matrix Decomposition Tree** technique. This technique uniquely employs a bi-resolution representation of the residual network, maintained through a hybrid strategy of lightweight matrix decomposition and efficient divide-and-conquer updates. Specifically, the decomposition provides a compressed yet information-preserving model, while the divide-and-conquer logic ensures that updates affect only relevant sub-structures. This design enables $\mathrm{O}(n)$ worst-case reduced-cost matrix maintenance per simplex iteration, while the total re-optimization time after a dynamic modification additionally depends on the number of simplex iterations required.

Before illustrating this technique, we need to introduce the *Euler Tour Tree* (ETT) technique (Tarjan, 1997), and the operations that we will impose on it. ETT represents a tree with n nodes as a sequence of length $3n-2$ by replacing each undirected edge with two directed edges and adding a self-loop for every node (an example is shown in Figure 4). The advantage of ETT is that when the tree undergoes an update, the corresponding sequence can be maintained in $\mathrm{O}(1)$ time. A detailed description of this technique is provided in Section B.

Drawing inspiration from Xu & Ding (2024)'s idea, we use the Euler tour sequence to design a matrix $\Gamma \in (\mathbb{R} \cup \{+\infty\})^{(3n-2)\times(3n-2)}$: the rows and columns of the maintained matrix are indexed by the Euler Tour Sequence of the spanning tree T_{ot}. For the items in Γ, only the elements corresponding to row (a_i, a_i) and column (b_j, b_j) (where $a_i \in \mathcal{A}$, $b_j \in \mathcal{B}$) have the value r_{ij}, while all other elements are set to $+\infty$ to mark them as invalid. In fact, only $n_{\mathcal{A}}$ rows and $n_{\mathcal{B}}$ columns corresponding to the self-loop of a vertex in $\mathcal{A}$ and $\mathcal{B}$ can be finite. Under this condition, the valid finite elements in Γ have a one-to-one correspondence with the elements in R_{rc}, meaning Γ is essentially a reordered representation of R_{rc}. Therefore, **the minimum value of Γ corresponds exactly to the minimum value of the R_{rc} matrix.**

Furthermore, according to Tarjan (1997), splitting a tree T_{ot} can be achieved by applying $\mathrm{O}(1)$ split and merge operations to its Euler tour sequence, which yields the independent Euler tour sequences of the two resulting trees, T_1 and T_2. Since the rows and columns of the matrix Γ are strictly indexed by the Euler tour sequence, the matrix itself can also be correspondingly partitioned via $\mathrm{O}(1)$ operations. Correspondingly, the matrix Γ is partitioned into four blocks: Γ_{11}, Γ_{12}, Γ_{21}, Γ_{22}, as shown in Figure 1, where Γ_{lr} contains all r_{ij}s with $a_i \in T_l$ and $b_j \in T_r$ for $l, r \in \{1, 2\}$. Symmetrically, the link operation between two disjoint trees follows identical topological principles. Merging two independent trees into a single spanning tree necessitates only $\mathrm{O}(1)$ split and merge operations on their respective Euler tour sequences. Consequently, the matrix Γ accomplishes the merge through corresponding block recombinations, effectively reversing the partition process with the same $\mathrm{O}(1)$ theoretical efficiency.

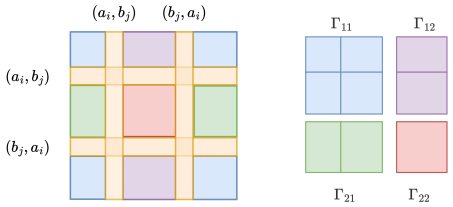


Figure 1: Matrix transformation when cutting edge (a_i, b_j).

To update the dual solution, the entries of the reduced cost matrix R_{rc} are modified as follows:

$$r'_{ij} = \begin{cases} r_{ij} + \delta, & \text{if } a_i \in T_1,\ b_j \in T_2, \\ r_{ij} - \delta, & \text{if } a_i \in T_2,\ b_j \in T_1, \\ r_{ij}, & \text{otherwise}, \end{cases}$$

where δ is the value that rebalances the dual solution. This update is equivalent to adding $\pm\delta$ globally to the corresponding off-diagonal blocks of Γ.

In summary, the data structure we designed needs to maintain a matrix and support the following four operations:

- SPLIT: Partition the matrix along a row or column, splitting it into two submatrices.
- MERGE: Concatenate two matrices with the same number of rows (or columns) into a single matrix by concatenating the rows (or columns) of one matrix after those of the other.
- UPDATE: Add a given value x to all elements of the matrix.
- QUERY: retrieve the minimum value together with an attaining index pair.

3.2 The Construction for BMD Tree

After introducing ETT as data structure to transform the reduced cost matrix R_{rc} into Γ, we now demonstrate how to operate Γ efficiently. In this section, we introduce the fundamental structure of the block decomposition algorithm and the querying of global minima of Γ.

For the matrix Γ, we partition it in the following manner: we perform cuts along the rows and columns at intervals of size $\sqrt{n}$, resulting in $\mathrm{O}(\sqrt{n}) \times \mathrm{O}(\sqrt{n}) = \mathrm{O}(n)$ sub-blocks. Once the minimum value within each sub-block is explicitly maintained, the global minimum can be queried in $\mathrm{O}(n)$ time via a simple sequential search. Consequently, for any structural SPLIT or MERGE operation, at most $\mathrm{O}(\sqrt{n})$ sub-blocks are affected, meaning only $\mathrm{O}(\sqrt{n})$ block minima require updates.

While a single-level partition bounds the number of affected blocks, naively maintaining the internal minimum of each $\mathrm{O}(\sqrt{n}) \times \mathrm{O}(\sqrt{n})$ block still demands $\mathrm{O}(\sqrt{n})$ time, yielding an $\mathrm{O}(n)$ overhead for this layer alone. To systematically reduce this per-block maintenance cost, a natural intuition is to treat each sub-block as a new grand block and recursively partition it in the same manner. For instance, a sub-block of size $\sqrt{n} \times \sqrt{n}$ can be further partitioned into $n^{1/4} \times n^{1/4}$ sub-sub-blocks. We

repeat this recursive decomposition until the block size becomes constant. This recursive partitioning yields a **hierarchical tree structure**. At the k-th level (defining the root grand block as the 0-th level), the individual block size is $n^{1/2^k} \times n^{1/2^k}$. Within this level, the maintenance complexity per block is $\mathrm{O}(n^{1/2^k})$, and the total number of affected blocks across the matrix is $\mathrm{O}(n^{(2^k-1)/2^k})$. Therefore, the aggregate maintenance complexity strictly remains $\mathrm{O}(n)$ per hierarchical level.

Lemma 3.1. *Following the uniform recursive decomposition strategy described above, the block size reduces to at most 4 when the hierarchical depth reaches* $\lceil \log_2 \log_4 n \rceil$.

Lemma 3.1 can be verified via straightforward algebraic manipulation. Assume that at the k-th level, the block size drops below 4. We then have $n^{1/2^k} < 4$, which implies $n < 4^{2^k}$, and subsequently $\log_4 n \leq 2^k$. Taking the base-2 logarithm yields $k \geq \log_2 \log_4 n$.

By directly applying Lemma 3.1, we observe a critical theoretical barrier: since each of the $\mathrm{O}(\log \log n)$ levels incurs an $\mathrm{O}(n)$ maintenance cost, the total time complexity of maintaining the matrix Γ amounts to $\mathrm{O}(n \log \log n)$. This complexity is insufficient to achieve our ultimate goal of a strict $\mathrm{O}(n)$ update time. To formally break this $\mathrm{O}(n \log \log n)$ barrier, we introduce a simple bi-resolution flattening technique, which we detail below.

Efficiently calculate the minimum value of all sub-blocks in a single level. We employ a hierarchical nesting technique. Specifically, at a given level, in addition to partitioning the structure into base blocks of size $\sqrt{n}$, **we insert an intermediate level of *medium blocks***. These medium blocks are partitioned at intervals of n^ρ, where ρ can be any value in $(\frac{1}{2}, 1)$. Under this scheme, each dimension yields $n^{1-\rho}$ medium blocks, and each medium block contains $n^{\rho-1/2}$ *small blocks* per dimension. Consequently, the total number of medium blocks is $\mathrm{O}\left(n^{1-\rho}\right) \times \mathrm{O}\left(n^{1-\rho}\right) = \mathrm{O}\left(n^{2(1-\rho)}\right)$, and each medium block encompasses $\mathrm{O}\left(n^{\rho-1/2}\right) \times \mathrm{O}\left(n^{\rho-1/2}\right) = \mathrm{O}\left(n^{2\rho-1}\right)$ small blocks in total, as Figure 2 shown.

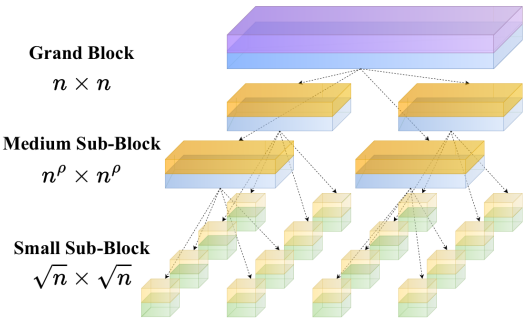


Figure 2: The basic structure of BMD Tree. The grand block is cut into $n^{1-\rho} \times n^{1-\rho}$ medium sub-blocks, then cut into $n^{1/2} \times n^{1/2}$ small sub-blocks. These small sub-blocks will then serve as the grand blocks for the next level and be further cut.

However, as SPLIT and MERGE operations proceed, the number of blocks in one layer may grow too large due to dynamically changing sub-blocks, degrading overall efficiency. For example, if certain blocks are continuously SPLIT multiple times, their side lengths become excessively small, leading to a proliferation in the total number of blocks. To prevent this, we implement a **dynamic side-length adjustment mechanism**. At recursive level ℓ (with the root at level 0), let $k = n^{2^{-\ell}}$ be the fixed standard scale. The standard side lengths are $S_m = k^\rho$ for medium blocks and $S_s = \sqrt{k}$ for small blocks; they depend on the level and global n, not on the current physical dimensions of the parent block. If a sub-block's dimension exceeds twice its standard length, we bisect it and recompute minima. Conversely, if adjacent sub-blocks fall short of the standard, we MERGE them. These adjustments prevent asymptotic degradation—where unchecked growth inflates processing costs and fragmentation multiplies the block count per dimension beyond $\Theta(k^{1-\rho})$—ensuring the hierarchy preserves its efficiency bounds dynamically. Details appear in Section H.2.

To handle UPDATE operations, we introduce the **Lazy Tag** deferred update strategy (Tarjan, 1997; Sleator & Tarjan, 1981). Each block maintains a lazy tag $\texttt{tag}_v$ alongside its minimum $\texttt{min}_v$, making the true minimum $\texttt{min}_v + \texttt{tag}_v$. Global modifications (e.g., adding x) simply increment $\texttt{tag}_v$ by x. Before SPLIT or MERGE, we recursively propagate lazy tags to subordinates, update $\texttt{min}_v$, and reset $\texttt{tag}_v$ to 0. Since tag propagation strictly follows the block accesses required for minimum maintenance, it preserves asymptotic complexity. The details of lazy tag are provided in Section H.1.

Finally, we summarize the operational workflow of SPLIT and MERGE procedures within the BMD Tree structure as follows:

1. **Propagate lazy tags** from the global block to all medium blocks, then from each affected medium block to all its small blocks.
2. **Execute recursive SPLIT/MERGE** operations on the underlying data structures within each affected small block.
3. **Maintain block side lengths:** using the fixed standard scale $k = n^{2^{-\ell}}$ at the current recursive level, split a small interval if its length exceeds $2\sqrt{k}$ and merge adjacent small intervals if their total length is below $\sqrt{k}$; apply the same rules to medium intervals with thresholds $2k^\rho$ and k^ρ.
4. **Recalculate minima** upward: first for each medium block from its small blocks, then for the global block from all medium blocks.

This maintenance procedure ensures that after each structural modification, the hierarchical structure is fully restored to numerical correctness through lazy tag propagation and minimum recomputation. Detailed pseudocode appears in Algorithm 3.

3.3 Complexity Analysis

Before analyzing the overall time complexity, we first establish the $\mathrm{O}(1)$ bounds for the QUERY and UPDATE operations. Because our maintenance procedure explicitly recalculates and stores the global minimum after every SPLIT, MERGE, and UPDATE, a QUERY operation retrieves this value in $\mathrm{O}(1)$ time. Similarly, the lazy tag technique decouples global numerical updates from element-wise modifications. By transforming an UPDATE into a single addition to the global lazy tag, its complexity is strictly bounded to $\mathrm{O}(1)$. Next, we analyze the complexity of the structural adjustments: the SPLIT and MERGE operations. We first analyze the complexity of maintaining a single block.

Lemma 3.2. *Let $k = n^{2^{-\ell}}$ be the fixed standard scale at the current recursive level ℓ. Recall we set the standard side length of its internal medium sub-blocks as $S_m = k^\rho$ (for any $\rho \in (1/2, 1)$), and its small sub-blocks as $S_s = k^{1/2}$. The non-recursive maintenance complexity for a block whose side lengths are $\mathrm{O}(k)$ is $\mathrm{O}(k^{\max\{2-2\rho,\rho,1/2\}})$. In particular, if we fix $\rho = \frac{2}{3}$, the non-recursive overhead becomes $\mathrm{O}(k^{2/3})$.*

Due to the space limit, we leave the proof to Section J. The high level idea is to analyze the non-recursive complexity level by level (Grand block $\rightarrow$ Medium blocks $\rightarrow$ Small blocks). Finally, we take the maximum of these three part as the dominated complexity, i.e., $\mathrm{O}(k^{\max\{2-2\rho,\rho,1/2\}})$. Using Lemma 3.2, we can now derive the total complexity of the algorithm.

Theorem 3.3. *Through BMD Tree, each structural matrix operation,* SPLIT *or* MERGE*, can be supported in $\mathrm{O}(n)$ worst-case time. Moreover,* QUERY *and global additive* UPDATE *operations take $\mathrm{O}(1)$ time. Hence the reduced-cost matrix maintenance cost per simplex iteration is $\mathrm{O}(n)$.*

The formal proof is placed in Section K, and we briefly introduce the sketch here. Following the notation of Lemma 3.2, k denotes the fixed standard scale of the current recursive level, and let $T(k)$ denote its worst-case maintenance cost. By Lemma 3.2, fixing $\rho = 2/3$ gives a non-recursive overhead of $\mathrm{O}(k^{2/3})$. For the idealized equal-length decomposition, a structural SPLIT or MERGE affects $\mathrm{O}(k^{1/2})$ small sub-blocks, each of side length $k^{1/2}$. Thus, for some constant $C > 0$, $T(k) \le k^{1/2}T(k^{1/2}) + Ck^{2/3}$. After t levels, the number of subproblems is denoted as $\lambda_t = k^{1-2^{-t}}$, and the overhead at level t is $C\lambda_t\,(k/\lambda_t)^{2/3} = Ck^{2/3}\lambda_t^{1/3} = Ck^{1-2^{-t}/3}$. As shown in Section K, the sum of these overheads before reaching the constant-size base cases is $\mathrm{O}(k)$, and the total base-case contribution is also $\mathrm{O}(k)$. Hence $T(k) = \mathrm{O}(k)$. Applying this bound to the root block with $k = n$ gives $T(n) = \mathrm{O}(n)$.

4 Experiments

We conduct three experiments to evaluate our algorithm's performance. (1) On the synthetic data, Gaussian random point sets in 2D, we compare against Network Simplex (Orlin, 1997), Sinkhorn (Cuturi, 2013), implemented using the Python POT library (Flamary et al., 2021), and the Skip Lists algorithm (Xu & Ding, 2024) for dynamic OT, demonstrating superior efficiency in dynamic settings. (2) Using real-world datasets MNIST (28×28 grayscale digits) (LeCun et al., 2010) and CIFAR-10 (32×32 colored objects) (Krizhevsky, 2009), our algorithm maintains consistent advantages over the exact baselines. (3) For the Dynamic Bi-Chromatic Matching problem,

we outperform both the static KUHN–MUNKRES (KM) algorithm (Kuhn, 1955), a classical exact algorithm for minimum-cost bipartite matching implemented using the SciPy library (Virtanen et al., 2020), and Goranci et al. (2025)'s dynamic method EUCLMATCH. All results were obtained on a server with 1024 GB of memory, an Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40 GHz, and the Ubuntu 22.04.3 operating system. For each problem size, we generate 10 independent instances. For each instance, we perform 100 position modifications and 100 weight modifications, and report the average runtime over these updates.

(1) Experiments on synthetic datasets. The results in Figure 3(a) show that our algorithm substantially outperforms the exact static NETWORK SIMPLEX solver and the dynamic algorithm based on SKIP LISTS for both types of operations across all tested scales. Compared with SINKHORN, our method computes exact OT solutions rather than entropically regularized approximations, while still achieving competitive or better runtime. For example, compared with SINKHORN using $\epsilon = 0.01$, our algorithm is about $4\times$ faster for position modifications and about $7\times$ faster for weight modifications on the tested synthetic instances.

(2) Experiments on real-world datasets. For the MNIST and CIFAR-10 datasets, we use Manhattan distance as the ground metric. When sampling points from the datasets, we impose uniform distribution across all labels. As shown in Figure 3(b)&(c), our dynamic algorithm demonstrates similar efficiency advantage as the experiments on synthetic datasets.

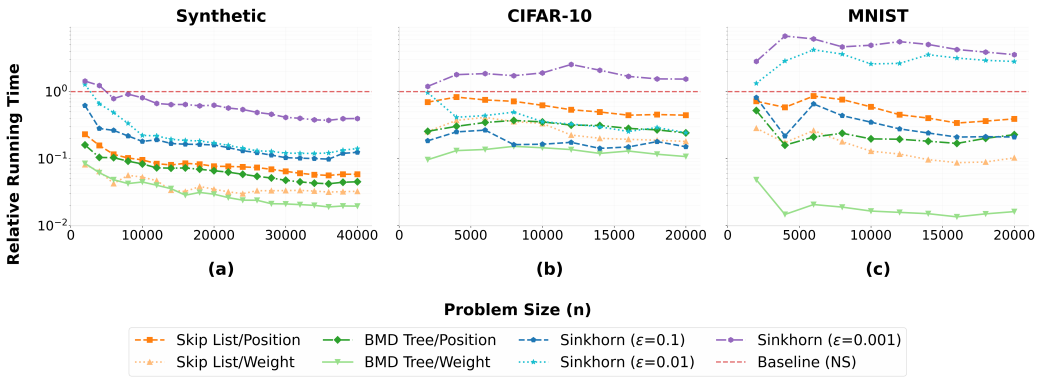


Figure 3: Runtime relative to NETWORK SIMPLEX (NS) versus problem size n, shown on a logarithmic y-axis; lower values indicate faster execution. BMD Tree is our method. SINKHORN uses $\epsilon \in \{0.1, 0.01, 0.001\}$. "Position" and "Weight" denote the modification types.

(3) Dynamic Bi-Chromatic Matching. The Bi-Chromatic Matching problem involves finding a minimum-cost perfect matching in a bipartite graph formed by two sets of points of different colors. As mentioned in Section 1, this problem can be formulated as an OT problem, and its dynamic version can therefore be solved by our proposed algorithm. Due to the space limit, we place the detailed experimental results in Section L.3. Our algorithm demonstrates a clear time advantage over the static KM algorithm. EUCLMATCH is a sublinear approximate dynamic algorithm. It shows superior efficiency compared to our method in 2-dimensional settings or when their approximation factor is not required to be low (note that their complexity is a trade-off between running time and approximation factor). However, under high-dimensional data or with stricter precision settings, our dynamic algorithm exhibits a significantly more pronounced advantage. For instance, on the 20-dimensional synthetic dataset, our algorithm achieves almost a $200\times$ speedup.

Additional experimental results. Section L presents (1) runtime and speedup comparisons on two-dimensional Gaussian point sets across problem sizes and both update types; (2) additional MNIST and CIFAR-10 results, including standard deviations of update runtimes; and (3) average simplex iteration counts per modification across datasets and update types.

5 CONCLUSION

We propose an efficient matrix maintenance structure that enables the NETWORK SIMPLEX method to handle dynamic optimal transport problems. While our approach is designed for discrete OT, extending it to continuous settings remains a future research direction. Further directions include establishing worst-case bounds on the number of simplex iterations, adapting the method to other OT variants, and exploring its applications in machine learning.

AI USE STATEMENT

The research ideas, experimental design, and core algorithm implementation were independently developed by the authors. AI tools assisted with language polishing and auxiliary scripts for experiment execution and result aggregation. The authors reviewed the adopted AI-assisted material and take responsibility for the final manuscript, code, and results.

ETHICS STATEMENT

This work studies algorithms for discrete optimal transport using synthetic data and the public MNIST and CIFAR-10 benchmarks. It involves no newly recruited human participants or newly collected personal data.

REPRODUCIBILITY STATEMENT

Proofs of the structural properties are provided in Sections D and E, and proofs of the complexity bounds in Sections J and K. The supplementary material provides the solver source code and instructions for compilation and input preparation. Section 4 and Section L describe the datasets, sampling procedures, and experimental settings, including the construction of synthetic Gaussian point sets and the use of MNIST and CIFAR-10.

REFERENCES

Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Minimum cost flow problem Minimum Cost Flow Problem*, pp. 1382–1392. Springer US, Boston, MA, 2001. ISBN 978-0-306-48332-5. doi: 10.1007/0-306-48332-7_283. URL https://doi.org/10.1007/0-306-48332-7_283.

Mokhtar Z Alaya, Maxime Bérar, Gilles Gasso, and Alain Rakotomamonjy. Screening sinkhorn algorithm for regularized optimal transport, 2019.

Jason Altschuler, Jonathan Niles-Weed, and Philippe Rigollet. Near-linear time approximation algorithms for optimal transport via sinkhorn iteration. *Advances in neural information processing systems*, 30, 2017.

Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein generative adversarial networks. In *International conference on machine learning*, pp. 214–223. Pmlr, 2017.

Phil Bagwell. Ideal hash trees. *Es Grands Champs*, 1195:5–2, 2001.

Philip Bagwell and Tiark Rompf. Rrb-trees: efficient immutable vectors. *Tech. Rep.*, 2011.

Hans-J Boehm, Russ Atkinson, and Michael Plass. Ropes: An alternative to strings. *Software: Practice and Experience*, 25(12):1315–1330, 1995.

Deeparnab Chakrabarty and Sanjeev Khanna. Better and Simpler Error Analysis of the Sinkhorn-Knopp Algorithm for Matrix Scaling. In Raimund Seidel (ed.), *1st Symposium on Simplicity in Algorithms (SOSA 2018)*, volume 61 of *Open Access Series in Informatics (OASIcs)*, pp. 4:1–4:11, Dagstuhl, Germany, 2018. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-064-4. doi: 10.4230/OASIcs.SOSA.2018.4. URL https://drops.dagstuhl.de/entities/document/10.4230/OASIcs.SOSA.2018.4.

Timothy M Chan. Dynamic generalized closest pair: Revisiting eppstein's technique. In *Symposium on Simplicity in Algorithms*, pp. 33–37. SIAM, 2020.

Nicolas Courty, Rémi Flamary, Devis Tuia, and Alain Rakotomamonjy. Optimal transport for domain adaptation. *IEEE transactions on pattern analysis and machine intelligence*, 39(9):1853–1865, 2016.

W. H. Cunningham. A network simplex method. *Mathematical Programming*, 11(1):105–116, 1976.

Marco Cuturi. Sinkhorn distances: Lightspeed computation of optimal transport. In C.J. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K.Q. Weinberger (eds.), *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc., 2013. URL https://proceedings.neurips.cc/paper_files/paper/2013/file/af21d0c97db2e27e13572cbf59eb343d-Paper.pdf.

Yihe Dong, Yu Gao, Richard Peng, Ilya Razenshteyn, and Saurabh Sawlani. A study of performance of optimal transport. *arXiv preprint arXiv:2005.01182*, 2020.

James R Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. Making data structures persistent. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pp. 109–121, 1986.

Pavel Dvurechensky, Alexander Gasnikov, and Alexey Kroshnin. Computational optimal transport: Complexity by accelerated gradient descent is better than by sinkhorn's algorithm. In Jennifer Dy and Andreas Krause (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 1367–1376. PMLR, 10–15 Jul 2018. URL https://proceedings.mlr.press/v80/dvurechensky18a.html.

Marvin Eisenberger, Aysim Toker, Laura Leal-Taixé, Florian Bernard, and Daniel Cremers. A unified framework for implicit sinkhorn differentiation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 509–518, 2022.

David Eppstein. Dynamic euclidean minimum spanning trees and extrema of binary functions. *Discrete & Computational Geometry*, 13(1):111–122, 1995.

Rémi Flamary, Nicolas Courty, Alexandre Gramfort, Mokhtar Z. Alaya, Aurélie Boisbunon, Stanislas Chambon, Laetitia Chapel, Adrien Corenflos, Kilian Fatras, Nemo Fournier, Leo Gautheron, Nathalie T. H. Gayraud, Hicham Janati, Alain Rakotomamonjy, Ievgen Redko, Antoine Rolet, Antony Schutz, Vivien Seguy, Danica J. Sutherland, Romain Tavenard, Alexander Tong, and Titouan Vayer. Pot: Python optimal transport. *Journal of Machine Learning Research*, 22(78):1–8, 2021.

Gramoz Goranci, Peter Kiss, Neel Patel, Martin P. Seybold, Eva Szilagyi, and Da Wei Zheng. Fully dynamic euclidean bi-chromatic matching in sublinear update time. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=up21Rwj5Fo.

Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron C Courville. Improved training of wasserstein gans. *Advances in neural information processing systems*, 30, 2017.

Peter Halmos, Xinhao Liu, Julian Gold, Feng Chen, Li Ding, and Benjamin J Raphael. Dest-ot: Alignment of spatiotemporal transcriptomics data. *Cell Systems*, 16(2), 2025.

Ralf Hinze and Ross Paterson. Finger trees: a simple general-purpose data structure. *Journal of functional programming*, 16(2):197–217, 2006.

Haim Kaplan and Robert E Tarjan. Persistent lists with catenation via recursive slow-down. In *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, pp. 93–102, 1995.

Boah Kim, Yan Zhuang, Tejas Sudharshan Mathai, and Ronald M. Summers. Otmorph: Unsupervised multi-domain abdominal medical image registration using neural optimal transport. *IEEE Trans. Medical Imaging*, 44(1):165–179, January 2025. URL https://doi.org/10.1109/TMI.2024.3437295.

Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

Harold W Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

Nathaniel Lahn, Deepika Mulchandani, and Sharath Raghvendra. A graph theoretic additive approximation of optimal transport. *Advances in Neural Information Processing Systems*, 32, 2019.

Nathaniel Lahn, Sharath Raghvendra, Emma Saarinen, and Pouyan Shirzadian. Scalable approximation algorithms for p-wasserstein distance and its variants. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=5YXzrcchMu.

Yann LeCun, Corinna Cortes, and Christopher J C Burges. Mnist handwritten digit database. http://yann.lecun.com/exdb/mnist/, 2010.

Bohan Lei, Yueting Zhuang, Xiaoyin Xu, and Min Zhang. An optimal transport-based method for medical image generation. *2024 IEEE International Conference on Image Processing (ICIP)*, pp. 3640–3646, 2024.

Mingzhe Li, Xinyuan Yan, Lin Yan, Tom Needham, and Bei Wang. Flexible and probabilistic topology tracking with partial optimal transport. *IEEE Transactions on Visualization and Computer Graphics*, 2025a.

Zezeng Li, Weimin Wang, Yuming Zhao, Wenhai Li, Na Lei, and Xianfeng Gu. Hyper-spherical optimal transport for semantic alignment in text-to-3d end-to-end generation. *IEEE Transactions on Visualization and Computer Graphics*, 2025b.

Tianyi Lin, Nhat Ho, and Michael I Jordan. On the efficiency of entropic regularized algorithms for optimal transport. *Journal of Machine Learning Research*, 23(137):1–42, 2022.

Arthur Mensch and Gabriel Peyré. Online sinkhorn: Optimal transport distances from sample streams. *Advances in Neural Information Processing Systems*, 33:1657–1667, 2020.

Khai Nguyen. Streaming sliced optimal transport. *arXiv preprint arXiv:2505.06835*, 2025.

James B. Orlin. A polynomial time primal network simplex algorithm for minimum cost flows. *Mathematical Programming*, 78(2):109–129, 1997.

Gabriel Peyré and Marco Cuturi. Computational optimal transport. *Found. Trends Mach. Learn.*, 11 (5–6):355–607, 2019. doi: 10.1561/2200000073.

Yossi Rubner, Carlo Tomasi, and Leonidas J Guibas. The earth mover's distance as a metric for image retrieval. *International journal of computer vision*, 40(2):99–121, 2000.

R Sharathkumar and Pankaj K Agarwal. Algorithms for the transportation problem in geometric settings. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, pp. 306–317. SIAM, 2012.

Daniel D Sleator and Robert Endre Tarjan. A data structure for dynamic trees. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pp. 114–122, 1981.

Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM (JACM)*, 32(3):652–686, 1985.

Robert E Tarjan. Dynamic trees as search trees via euler tours, applied to the network simplex algorithm. *Mathematical Programming*, 78(2):169–177, 1997.

Robert Endre Tarjan and Renato Fonseca F Werneck. Self-adjusting top trees. In *SODA*, volume 5, pp. 813–822, 2005.

Alexis Thibault, Lénaïc Chizat, Charles Dossal, and Nicolas Papadakis. Overrelaxed sinkhorn–knopp algorithm for regularized optimal transport. *Algorithms*, 14(5):143, 2021.

Cédric Villani et al. *Optimal transport: old and new*, volume 338. Springer, 2008.

Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stefan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C. J. Carey, Ilhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, and Paul van Mulbregt. Scipy 1.0: fundamental algorithms for scientific computing in python. *Nature Methods*, 17:261–272, 2020. doi: 10.1038/s41592-019-0686-2.

Xiaoyang Xu and Hu Ding. A novel skip orthogonal list for dynamic optimal transport problem. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(18):20838–20846, 2024.

A Related Work

Optimal Transport The Network Simplex method (Cunningham, 1976), grounded in duality theory, provides a direct approach for solving the linear programming problem after dynamic adjustments. This class of methods guarantees solution optimality and, following numerous refinements by various researchers, has achieved considerable efficiency (Orlin, 1997). However, while these methods perform excellently in static environments, their performance in dynamic scenarios remains suboptimal. The principal challenge in adapting the simplex method to dynamic settings lies in maintaining adjustment costs, which necessitates the design of efficient data structures. Currently, Xu & Ding (2024) have devised a data structure capable of maintaining dynamic adjustment costs with an expected complexity of $\mathrm{O}(n)$. Furthermore, while approximate algorithms for optimal transport exist, such as the Sinkhorn algorithm (Thibault et al., 2021; Alaya et al., 2019; Cuturi, 2013), they similarly face significant challenges in dynamic adaptation.

Dynamic data structure Research on dynamic data structures forms the core foundation for efficiently handling mutable data. In the area of dynamic sequence operations, studies have evolved from the engineering-efficient Rope (Boehm et al., 1995) and the functionally general-purpose finger trees (Hinze & Paterson, 2006) to the RRB-Tree (Bagwell & Rompf, 2011) that supports fast concatenation, enabling flexible maintenance of sequences. Persistent data structures, through general techniques such as path copying (Driscoll et al., 1986), allow data structures to maintain full historical versions with low overhead, giving rise to key building blocks like persistent lists (Kaplan & Tarjan, 1995) and persistent vectors (Bagwell, 2001), which lay the groundwork for version control and concurrency safety. For dynamic tree structures, the self-adjusting top tree (Tarjan & Werneck, 2005), based on Rake–Compress decomposition, supports efficient path queries and aggregation while the tree topology changes dynamically. Together, these works form a solid foundation for processing dynamic data.

Euler Tour Tree (ETT) The Euler Tour Tree (Tarjan, 1997) is a technique that represents a tree as an ordered sequence. For a given tree, we define a directed edge set E' that includes self-loops for all nodes in the tree, as well as directed edges in both directions for each tree edge. The edges in this set form an Eulerian circuit, which constitutes the Euler tour of the tree. The Euler Tour of a tree possesses two key properties: first, operations such as tree cutting and merging can generate a new Euler tour in $\mathrm{O}(1)$ time complexity; second, any sub-tree within the tree corresponds to a contiguous segment in the Euler tour. Leveraging these properties facilitates efficient maintenance of sub-tree information. A detailed exposition of the Euler Tour Tree is provided in Section B.

Link-Cut Tree (LCT) The Link-Cut Tree (Sleator & Tarjan, 1981) is another dynamic tree data structure that enables rapid tree cutting and merging operations. In contrast to the Euler Tour Tree, the LCT is not particularly suited for maintaining sub-tree information but excels at handling path-related queries and modifications, including updates and queries along paths between nodes. The amortized time complexity for its common operations is $\mathrm{O}(\log n)$. Comprehensive details regarding the Link-Cut Tree are elaborated in Section C.

B EULER TOUR TREE

This section introduces the definition and properties of the ETT.

The ETT is a technique that transforms a tree structure into an ordered sequence. We define the Euler tour sequence of a tree as follows:

Definition B.1. For a tree $T = (V, E)$, construct an edge set E' as follows:

For every vertex $v \in V$, add the self-loop (v, v) to E';

For every tree edge from u to v, $u, v \in V$, add both directed edges (u, v) and (v, u) to the edge set E'.

Specifically, in this appendix, (u, v) denotes a directed edge from u to v.

The Euler tour of the directed graph $G = (V, E')$ is called the Euler tour sequence of the tree.

Clearly, for a tree with n nodes, the length of its Euler tour sequence is $3n - 2$, which is of order $\mathrm{O}(n)$.

It should be noted that a tree can have multiple valid Euler tour sequences depending on the traversal order.

The Euler tour sequence possesses the following properties:

Theorem B.2. *During Cut and Link operations, the Euler tour sequence after the operation can be obtained with* $\mathrm{O}(1)$ *sequence splits and merges.*

Here, the Cut operation splits a tree into two by removing an edge, while the Link operation combines two trees into one by adding an edge.

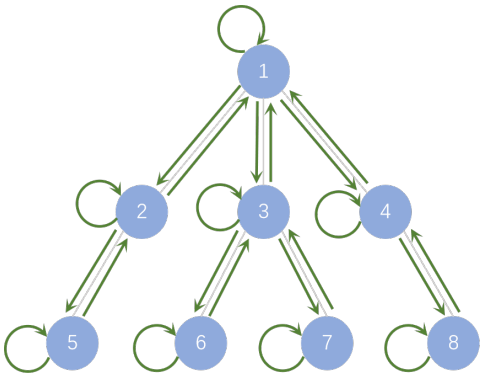


Figure 4: This is an example of ETT. The edge sequence of an ETT's Eulerian tour is called the Euler tour sequence.

Proof. Consider a Cut operation on the edge between u and v. Without loss of generality, assume that in the original sequence, (u, v) precedes (v, u). The Euler tour sequence can then be partitioned as $L_1, (u, v), L_2, (v, u), L_3$. Subsequently, the concatenation of L_1 and L_3 (denoted as $L_1 \oplus L_3$) and the sequence L_2 form the Euler tour sequences of the two resulting trees after the split, respectively.

For two trees with sequences L_1 and L_2, consider a Link operation on edge (u, v), where u belongs to the first tree and v belongs to the second tree. The sequence L_1 can be split at the self-loop (u, u) into L_{11} and L_{12}, and similarly, L_2 can be split at the self-loop (v, v) into L_{21} and L_{22}. The Euler tour sequence of the merged tree is then $L_{12}, L_{11}, (u, v), L_{22}, L_{21}, (v, u)$.

□

In the Euler tour sequence of a tree, the entries corresponding to any entire subtree are not necessarily a single contiguous block; however, they are mathematically guaranteed to form **at most two contiguous segments**. Specifically, as demonstrated in the Cut operation, a subtree's sequence might

consist of a prefix L_1 and a suffix L_3 of the original sequence. Through $\mathrm{O}(1)$ sequence split and merge operations (equivalent to a cyclic shift), these segments can be easily concatenated into a single contiguous sequence.

This property provides an effective methodology for maintaining subtree information. By integrating with various splittable data structures, such as segment trees and balanced binary search trees, efficient maintenance of subtree information can be readily achieved. In this paper, since we require batch modifications of information between node pairs from two distinct trees (for instance, performing batch updates on all pairs $[a_i, b_j]$—where we use square brackets to represent point pairs to avoid confusion—such that $a_i \in T_1$ and $b_j \in T_2$), we employ the Euler Tour Tree to index the rows and columns of a matrix, utilizing the self-loops within the coordinate system as node representatives. For example, the node pair $[a_i, b_j]$ can be represented by the coordinate $[(a_i, a_i), (b_j, b_j)]$. Because a subtree corresponds to at most two contiguous intervals in the ETT sequence, mapping the subtree to the matrix simply divides the target area into at most $2 \times 2 = 4$ contiguous sub-matrix blocks. This decomposition retains the $\mathrm{O}(1)$ overhead for topological matrix partitioning.

Finally, it is worth noting that while the Cut and Link topological adjustments require $\mathrm{O}(1)$ sequence splits and merges at the theoretical level, the actual time complexity to implement these adjustments depends on the underlying splittable data structures, which is typically $\mathrm{O}(\log n)$ per operation.

C LINK-CUT TREE

Link-Cut Tree (LCT) is a data structure for maintaining dynamic trees, supporting operations such as linking, cutting, and changing the root. It was also proposed by Sleator & Tarjan (1981). In this chapter, LCT technology provides a convenient solution for designing the steps of tree flow maintenance in our algorithm.

The key difference between LCT and the ETT mentioned in Section B of this paper is that ETT treats subtrees as continuous sequences, making it suitable for maintaining subtree information (such as synchronously cutting the R matrix after tree cuts in this paper); whereas LCT is based on information along paths in the tree (such as finding the minimum flow value on a path in the NETWORK SIMPLEX method of this paper).

LCT utilizes the idea of tree path decomposition. Tree path decomposition is a method of dividing a tree into several chains, primarily including heavy-light decomposition, long-chain decomposition, and preferred path decomposition. The LCT algorithm requires the use of preferred path decomposition, and its complexity proof utilizes heavy-light decomposition for assistance.

C.1 SPLAY TREE

The Splay Tree, introduced by Robert Sleator & Tarjan (1985), is a self-adjusting binary search tree that provides amortized $\mathrm{O}(\log n)$ complexity for fundamental operations including insertion, search, and deletion. It serves as the underlying auxiliary tree structure in the Link-Cut Tree algorithm required for maintaining path flows in our methodology.

Definition C.1 (Binary Search Tree)**.** A tree is defined as a binary search tree if each node possesses a key value, and for any given node, all keys in its left subtree are strictly less than its own key, while all keys in its right subtree are strictly greater.

Splay Tree is a special binary search tree. The distinctive feature of the Splay Tree is the **splay operation**, which is applied to any accessed node, moving it to the root through a sequence of rotations. This process simultaneously restructures the tree to improve balance and facilitate future accesses. The rotation strategy during splaying is categorized into three cases based on the node's positional relationship with its parent and grandparent:

- Zig: Applied when the node's parent is the root. Involves a single rotation (left or right) around the parent.
- Zig-zig: Applied when the node and its parent are both left children or both right children. Involves two consecutive rotations in the same direction, first on the parent and then on the node.

- Zig-zag: Applied when the node is a left child and its parent is a right child, or vice versa. Involves two rotations in opposite directions, first rotating the node over its parent, then over its former grandparent.

This case-based rotation strategy ensures that frequently accessed nodes migrate towards the root, reducing average access paths. The splay operation itself, while potentially involving multiple rotations per access, maintains an amortized time complexity of $\mathrm{O}(\log n)$ over a sequence of operations, as established by Sleator and Tarjan.

C.2 Maintaining Sequence Operations with Splay Trees

As a binary search tree, the Splay Tree can effectively maintain sequence operations, particularly supporting range reversal operations. In our context, it needs to handle the following operations on a given real-valued sequence:

- Update: Add a constant value x to all elements within a specified interval.
- Reverse: Reverse the order of elements within a specified interval.
- Query: Find the minimum value within a specified interval.

When using Splay Trees for sequence maintenance, elements are sorted by their positional indices in the sequence, while their actual values are stored as auxiliary attributes.

Specifically, each node X maintains the following data:

- key_X: The key value (position index)
- val_X: The element value
- min_X: The minimum value in the subtree rooted at X
- tagu_X: Lazy tag for update operations
- tagr_X: Lazy tag for reversal operations
- l_X, r_X: The left and right boundaries of the sequence segment represented by the subtree

For range minimum queries, the algorithm traverses from the root. Given query interval $[ql, qr]$ and current node x:

1. If $[l_x, r_x] \subseteq [ql, qr]$, return min_x
2. If $ql < \text{key}_x$, search left subtree
3. If $qr > \text{key}_x$, search right subtree
4. If $\text{key}_x \in [ql, qr]$, include val_x in comparison
5. Return the minimum among all candidate values

Range updates follow a similar traversal pattern, modifying tagu_X for affected segments.

For range reversal operations:

1. If l isn't the global minimum index, splay its predecessor to root
2. If r isn't the global maximum index, splay its successor to the right subtree's root
3. The target segment now corresponds to the left subtree of the right child
4. Swap left/right children of this subtree's root and set tagr_X
5. Perform splay operations to update segment information

This approach efficiently maintains sequence operations while preserving the Splay Tree's amortized logarithmic complexity.

C.3 LINK-CUT TREE

In this section, we will introduce the detailed structure of LCT.

Definition C.2 (Preferred Path Decomposition)**.** For a rooted tree, edges are artificially designated as either solid or dashed, ensuring that except for leaf nodes, every node has at most one solid edge connecting to a child.

A chain consisting entirely of solid edges and non-extensible is called a preferred path. The process of decomposing a tree into several preferred paths is called preferred path decomposition.

Preferred path decomposition is a flexible tree decomposition; the assignment of solid and dashed edges has no additional conditions and can be modified at any time. In preferred path decomposition, each preferred chain is precisely a sequence ordered by depth from shallow to deep.

The idea of LCT is to build an auxiliary tree (AuxTree) for each preferred path. Each auxiliary tree is a Splay Tree, sorted by depth, with shallower nodes on the left and deeper nodes on the right.

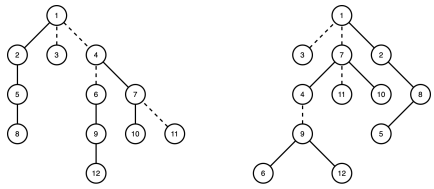


Figure 5: Original tree and an AuxTree of it.

To maintain the relationships of dashed edges, the parent node of the root node of each auxiliary tree is defined as the parent node of the shallowest node in the preferred path.

For example, nodes A, B, C form a preferred path, node A has the shallowest depth, and its parent node is D. However, in the auxiliary tree for this preferred path, B is the root. Then, B's parent node is artificially defined as D.

It should be noted that although we artificially define B's parent as D, D itself resides in another Splay Tree, and its children are also in another Splay Tree; it is not aware that it has descendants like A or B. That is, for dashed edges, only the child recognizes the parent, but the parent does not recognize the child. Nevertheless, a child can determine whether it is a solid child based on whether the parent recognizes it.

A very important operation in LCT is **Access**, whose function is to make the path from a specified node X to the root node consist of exactly one solid path.

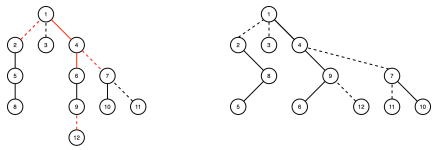


Figure 6: Perform an Access operation on node 9 on the tree in Figure 5

The Access operation is straightforward. First, splay node X to the root of its current auxiliary tree and disconnect its right subtree (i.e., in the original tree, change the edge from X to its solid child into a dashed edge). Then, if the current node is not connected to the root, splay X's parent node in the auxiliary tree (i.e., the parent node of the starting node of X's preferred path) to the root of its auxiliary tree, disconnect its right subtree (i.e., change its original solid child into a dashed edge), and attach X's auxiliary tree as its new right child. Repeat this process until X is connected to the root node.

The correctness of the Access operation is straightforward. Its complexity stems from two parts: one part is the Splay Tree operations, whose complexity is easily proven. The other part comes from the conversion between solid and dashed edges between Splay Trees. In each Access operation, except for the first time when the edge from X to its child is changed to a dashed edge, every subsequent

change of an edge to a dashed edge is necessarily accompanied by changing an edge to a solid edge. Therefore, it is only necessary to prove the upper bound on the number of times edges are changed to solid edges.

Before discussing the complexity of Access, we need to introduce another tree decomposition—heavy-light decomposition—for assistance. Note that introducing heavy-light decomposition here is only for complexity proof and is unrelated to the original algorithm.

Definition C.3 (Heavy-light Decomposition)**.** For a rooted tree, edges are artificially designated as solid (heavy) and dashed (light), ensuring that except for leaf nodes, every node has exactly one heavy edge to a child, and the subtree connected by this heavy edge has the largest number of nodes among all its subtrees.

A chain consisting entirely of heavy edges and non-extensible is called a heavy chain. The process of decomposing a tree into several heavy chains is called heavy-light decomposition.

The difference between heavy-light decomposition and preferred path decomposition is that a tree's heavy-light decomposition is unique (unless a node has two subtrees of equal size). Take any heavy-light decomposition of the tree. Regarding heavy-light decomposition, there is an obvious conclusion:

Lemma C.4. *If an edge from node X is a light edge, then the number of nodes in the connected subtree is less than or equal to half of the number of nodes in the subtree rooted at X.*

From this lemma, another conclusion can be drawn:

Theorem C.5. *For a tree with n nodes, any path from a node to the root contains at most $\log_2 n$ light edges.*

Proof. Each time a light edge is traversed, the subtree size decreases by at least half. The subtree rooted at any node contains at least 1 node, so $\frac{n}{2^k} \geq 1$, implying $k \leq \log_2 n$. □

Theorem C.6. *The amortized complexity of the Access operation is* $\mathrm{O}(\log n)$.

Proof. Take any heavy-light decomposition of the tree. For each operation, the number of times a light dashed edge is changed to a solid edge is at most $\log_2 n$.

For any heavy edge, the number of times it becomes solid is obviously less than or equal to the number of times it becomes dashed plus one. A heavy solid edge becomes dashed only in two scenarios: one is when one of its sibling edges becomes solid, but this sibling edge must be light; the other is during the first disconnection of a child node in an Access operation, but such changes occur only once per operation.

In summary, across all operations, the number of times heavy edges become solid is less than or equal to the number of times light edges become solid plus the number of Access operations plus the number of heavy edges.

Further, assume the original tree has n nodes and m operations are performed. Then the total number of times Access changes an edge to solid is less than or equal to $m \log n + n + m \log n + m = O(n + m \log n)$, resulting in an amortized complexity of $O(\log n)$. □

Besides Access, another important operation is **Make Root**, which changes the root of the tree to a specified node X. The Make Root operation essentially reverses the order of all edges along the path from X to the original root R. Therefore, it suffices to perform an Access on X and then perform a range reversal on the entire auxiliary tree.

Performing a **Cut** operation (splitting the tree into two) simply involves clearing the parent-child relationship at the edge to be cut. To perform a **Link** operation, Make Root the roots of the two trees to be linked, and then make one node the preferred child of the other.

D Proof of Proposition 2.1

Proof. Suppose, for the sake of contradiction, there exists an optimal transport plan containing a cycle. Let this cycle be denoted by $a_1 \to b_1 \to a_2 \to b_2 \to \ldots \to b_t \to a_1$.

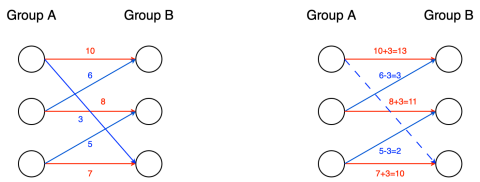


Figure 7: Cycle Removal in OT Solutions. The left panel illustrates the optimal transport plan prior to cycle elimination, while the right panel corresponds to the new optimal transport plan after cycle removal. Since the original plan is optimal, the transport plan composed of red edges and that composed of blue edges exhibit identical per-unit transportation costs.

Define $k = \min_{i\in[1,t]} w_{a_i b_i}$, the minimum flow along any edge in the cycle. We then construct a new transport plan by decreasing $w_{a_i b_i}$ by k for all $i \in [1, t]$, increasing $w_{a_{i+1} b_i}$ by k for all $i \in [1, t]$ (with the convention that $a_{t+1} = a_1$), and leaving all other edge flows unchanged. Figure 7 illustrates this process.

This transformation preserves both the supply and demand constraints while maintaining the same total transportation cost. However, it eliminates at least one edge from the cycle (specifically, the edge with minimum flow k).

By iteratively applying this procedure to all cycles in the transport plan, we eventually obtain an acyclic optimal transport plan whose active edges form a spanning tree of the underlying bipartite graph. □

E Proof of Proposition 2.4

Proof. Constructing the primal solution W from a given spanning tree T:

Build the set of leaf nodes Perform one depth-first search to count the initial degree (i.e., the number of adjacent nodes) of every node, then create a set S and insert all degree-1 nodes (i.e., the leaves of the tree) into S.

Leaf-to-root propagation: Arbitrarily take a leaf node from S and process it.

Mass balance: For a leaf node v (which is either a source $a \in \mathcal{A}$ or a sink $b \in \mathcal{B}$), the flow w_{ab} on its unique incident edge in T is determined solely by that node's supply α_a or demand β_b.

Iterative update: Once the flow on a leaf edge is fixed, subtract that flow from the remaining supply/demand of its neighbor and "remove" the leaf node. At the same time, update the degree of that neighbor; if the neighbor's degree becomes 1, insert it into S (meaning it becomes a new leaf).

Complexity: Because a spanning tree on n nodes has exactly $n - 1$ edges, and the flow on each edge is computed exactly once, the overall complexity is $\mathrm{O}(n)$.

2. Dual Solution Construction: The dual solution $h \in \mathbb{R}^{|\mathcal{A}|+|\mathcal{B}|}$ is determined by the complementary slackness conditions for the edges in the base T:

$$h_a - h_b = c_{ab}, \quad \forall (a, b) \in T \tag{3}$$

To construct this solution:

- Select an arbitrary root node r and set $h_r = 0$.
- Perform a tree traversal (e.g., BFS or DFS) starting from r.
- For each visited edge between $a \in \mathcal{A}$ and $b \in \mathcal{B}$ in T, calculate the dual variable of the child node: if the parent is a and the child is b, set $h_b = h_a - c_{ab}$; if the parent is b and the child is a, set $h_a = h_b + c_{ab}$.

Since each node is visited exactly once, the dual variables are constructed in $\mathrm{O}(n)$.

It should be noted that the $\mathrm{O}(n)$ traversal described above only constructs the T-induced dual potentials. To further verify whether this tree-induced dual solution is globally feasible, one must

check if the inequality $h_a - h_b \leq c_{ab}$ (i.e., $r_{ab} \geq 0$) holds for all non-tree edges. In our dynamic framework, this verification is efficiently handled by querying the global minimum reduced cost from the BMD Tree.

□

F ALGORITHM OF DYNAMIC NETWORK SIMPLEX

In this section we illustrate the algorithms mentioned in Section 3.1. Let R_{rc} be the reduced cost matrix, and r_{ij} be the reduced cost from $a_i \in \mathcal{A}$ to $b_j \in \mathcal{B}$. If e be an edge from point u to v, let $\tilde{e}$ be the edge from point v to u. $e_i.cap$ is defined to be the capacity flow of e_i. Let T_{ot} be the spanning tree corresponding to the current scheme.

For relocation operations, the pre-operation optimal transport plan remains primal feasible but may violate dual feasibility. Starting from the original plan, we employ the NETWORK SIMPLEX method to restore dual feasibility, as detailed in Algorithm 1.

In particular, for the static simplex method, we can generate the original feasible solution at will and solve it through Algorithm 1.

Algorithm 1: NETWORK SIMPLEX Method for Optimal Transport

Input: A spanning tree T_0 that yields a feasible primal solution, and the reduced cost matrix R_{rc}.
Output: A spanning tree T_{OPT} that yields the optimal solution.

Find indices i, j such that $r_{ij} = \min_{(a_i,b_j)\notin T} R_{rc}$;
Let $min_r \leftarrow r_{ij}$;
while $min_r < 0$ **do**
  Let $e_{\min}$ be the edge with minimum capacity on the unique path from b_j to a_i in the tree, resolving ties by Cunningham's leaving-arc rule (Section G);
  $\Delta \leftarrow e_{\min}$.cap;
  foreach *edge* e_i *on the path from* b_j *to* a_i **do**
    e_i.cap $\leftarrow e_i$.cap $- \Delta$;
    $\tilde{e}_i$.cap $\leftarrow \tilde{e}_i$.cap $+ \Delta$;
  end
  Create a new edge e_{new};
  e_{new}.from $\leftarrow a_i$;
  e_{new}.to $\leftarrow b_j$;
  e_{new}.cap $\leftarrow +\infty$;
  Let $\tilde{e}_{\text{new}}$.cap $\leftarrow \Delta$;
  Delete $e_{\min}$ from the tree, splitting it into two subtrees T_1 and T_2 (assume $a_i \in T_1$);
  Compute $add_h \leftarrow h_j^{\mathcal{B}} + c_{ij} - h_i^{\mathcal{A}}$;
  foreach *node pot in* T_1 **do**
    if $pot \in \mathcal{A}$ **then**
      $h_{pot}^{\mathcal{A}} \leftarrow h_{pot}^{\mathcal{A}} + add_h$;
    end
    else
      $h_{pot}^{\mathcal{B}} \leftarrow h_{pot}^{\mathcal{B}} + add_h$;
    end
  end
  Add e_{new} to reconnect T_1 and T_2, obtaining the updated tree;
  Update the reduced cost matrix R_{rc};
  Find indices i, j such that $r_{ij} = \min_{(a_i,b_j)\notin T} R_{rc}$;
  Let $min_r \leftarrow r_{ij}$;
end

For the weight modification, we have provided pseudocode for the algorithm in Algorithm 2. If the modification reduces the weight of a_{i_1} and increases that of a_{i_2} by δ, this corresponds to pushing δ units of flow from a_{i_2} to a_{i_1} in G_{res}. Symmetrically, reducing the weight of b_{j_1} and increasing

that of b_{j_2} by δ corresponds to pushing δ units from b_{j_1} to b_{j_2}. Without loss of generality, assume that we need to transport flow from a_{i_2} to a_{i_1}. If this transportation can be completed, the algorithm terminates. Otherwise, this implies that there exists a reverse edge (b_{j_0}, a_{i_0}) on the path whose capacity has been reduced to zero. Removing this edge splits the tree T_{ot} into two sub-trees, T_1 and T_2, where $a_{i_1} \in T_1$ and $a_{i_2} \in T_2$. Then the remaining task is to find a pair $a_{i'} \in T_2$ and $b_{j'} \in T_1$ so that $r_{i'j'}$ is the minimum among all possible such pairs, add the corresponding edge, and continue transporting flow until the required transportation is completed.

Without loss of generality, we only provide algorithms for modifying $\boldsymbol{\alpha}$.

Algorithm 2: Update after Weight Modification of $\boldsymbol{\alpha}$

Input: Indices i_1, i_2; the weight change δ; a spanning tree T_0 that yields the optimal solution before the modification; the reduced cost matrix R_{rc}.
Output: A spanning tree T_{OPT} that yields the optimal solution after the modification.
`/* Note: Weight modification for target vector` $\boldsymbol{\beta}$ `can be handled symmetrically. */`
while $\delta > 0$ **do**
  Let $e_{\min}$ be the edge with minimum capacity on the unique path from a_{i_2} to a_{i_1} in the current tree, resolving ties by Cunningham's leaving-arc rule (Section G);
  $\delta' \leftarrow \min\{\delta,\ e_{\min}.\text{cap}\}$;
  foreach *directed residual edge* e_i *on the path from* a_{i_2} *to* a_{i_1} **do**
    $e_i.\text{cap} \leftarrow e_i.\text{cap} - \delta'$;
    $\tilde{e}_i.\text{cap} \leftarrow \tilde{e}_i.\text{cap} + \delta'$;
  end
  $\delta \leftarrow \delta - \delta'$;
  if $\delta = 0$ **then**
    break;
  end
  Delete $e_{\min}$ from the tree, splitting it into two subtrees T_1 and T_2, where $a_{i_1} \in T_1$;
  Find i', j' such that $r_{i'j'}$ is minimum among all pairs with $a_{i'} \in \mathcal{A} \cap T_2$ and $b_{j'} \in \mathcal{B} \cap T_1$;
  Compute $add_h \leftarrow h^{\mathcal{B}}_{j'} + c_{i'j'} - h^{\mathcal{A}}_{i'}$;
  foreach *node* $pot \in T_2$ **do**
    if $pot \in \mathcal{A}$ **then**
      $h^{\mathcal{A}}_{pot} \leftarrow h^{\mathcal{A}}_{pot} + add_h$;
    end
    else
      $h^{\mathcal{B}}_{pot} \leftarrow h^{\mathcal{B}}_{pot} + add_h$;
    end
  end
  Add a new edge e_{new} connecting $a_{i'}$ and $b_{j'}$ with capacity $+\infty$ (and its reverse edge $\tilde{e}_{\text{new}}$ with capacity 0);
  Update the reduced cost matrix R_{rc}: subtract add_h from block $T_2 \times T_1$ and add add_h to block $T_1 \times T_2$;
end

G Degeneracy and Deterministic Tie-Breaking

Degeneracy may arise when several entering edges have the same reduced cost or when several reverse edges attain the minimum residual capacity on the fundamental cycle formed by an entering edge and the corresponding tree path. In this case, an iteration can change the spanning-tree basis without strictly decreasing the primal objective, and an unspecified tie-breaking rule may allow the Network Simplex method to cycle.

To guarantee finite termination, we use the standard strongly feasible tree rule of Cunningham (1976). Fix an arbitrary root of the spanning tree and maintain a strongly feasible tree basis in the sense of Cunningham: infinitesimal flow can be sent from every node toward the root along its tree path

without violating the residual-capacity constraints. When several reverse edges on a fundamental cycle attain the same minimum residual capacity, we apply Cunningham's leaving-arc rule and choose a blocking edge whose removal preserves strong feasibility. A fixed edge order is used only to make any remaining choice deterministic.

Cunningham's rule preserves strong feasibility after every simplex iteration and prevents a tree basis from recurring, thereby excluding cycling. Importantly, it places no restriction on the entering-edge rule, so we may continue to choose an edge of minimum negative reduced cost using the BMD Tree. The rule also does not perturb the numerical costs, weights, or objective value: every iteration preserves feasibility for the original OT instance, and termination still occurs exactly when all original reduced costs are nonnegative. Hence the exact-optimality guarantee is unchanged.

The leaving-arc rule is applied in both Algorithms 1 and 2 whenever several blocking edges are tied. It changes only the tie resolution in the dynamic-tree path query; the first eligible tied edge can be located using the maintained path order without changing the amortized $\mathrm{O}(\log n)$ tree-maintenance bound. This cost remains dominated by the $\mathrm{O}(n)$ reduced-cost matrix maintenance per simplex iteration.

H Details of block maintenance

H.1 Lazy Tags and Global Modification of Blocks

This section formalizes the lazy-tag mechanism used for global additive modifications. In the BMD Tree, a grand block is partitioned into medium blocks, and each medium block is further partitioned into small blocks. For each block at every level, we maintain a recorded minimum value and a lazy tag. The lazy tag stores a pending additive update that has not yet been propagated to its descendants.

Let D denote a grand block, E a medium block, and F a small block. Their recorded minimum values are denoted by $\min_D$, $\min_E$, and $\min_F$, respectively, and their lazy tags are denoted by $\mathtt{tag}_D$, $\mathtt{tag}_E$, and $\mathtt{tag}_F$. Let $\mathrm{Sub}(D)$ be the set of medium blocks contained in D, and let $\mathrm{Sub}(E)$ be the set of small blocks contained in E. The true minimum of a block is the sum of its recorded minimum and its lazy tag. For example, the true minimum of D is $\min_D + \mathtt{tag}_D$.

The recorded minimum of each internal block is defined after excluding its own lazy tag but including the lazy tags of its children. Thus, the following invariants are maintained $\min_D = \min_{E \in \mathrm{Sub}(D)} (\min_E + \mathtt{tag}_E)$, and similarly, $\min_E = \min_{F \in \mathrm{Sub}(E)} (\min_F + \mathtt{tag}_F)$. These invariants ensure that the true minimum of any block can be recovered by adding its own lazy tag to its recorded minimum.

A global additive UPDATE on a grand block D, which adds a value x to all entries in D, is implemented by $\mathtt{tag}_D \leftarrow \mathtt{tag}_D + x$. No child block needs to be visited. Therefore, a global additive UPDATE takes $\mathrm{O}(1)$ time.

Before a structural SPLIT or MERGE is performed on a grand block D, its lazy tag is transferred to every direct medium child $E \in \mathrm{Sub}(D)$, including children not intersected by the structural operation: $\mathtt{tag}_E \leftarrow \mathtt{tag}_E + \mathtt{tag}_D$. Only after all direct children have received this value do we reset $\mathtt{tag}_D \leftarrow 0$. For each affected medium block E, we likewise transfer its tag to every direct small child $F \in \mathrm{Sub}(E)$ before resetting $\mathtt{tag}_E \leftarrow 0$. Unaffected medium blocks retain their tags without further propagation. This is the push-down order used in Algorithm 3. Recorded minima of structurally modified blocks are recomputed after the operation, as described below.

After the required split, merge, or rebalancing operations are completed, the recorded minima are recomputed upward using the above invariants. In particular, a modified medium block recomputes its recorded minimum from the small blocks it contains, and a modified grand block recomputes its recorded minimum from its medium blocks.

The lazy-tag mechanism does not introduce additional asymptotic overhead. At a level with standard scale k, propagating the grand-block tag visits $\mathrm{O}(k^{2-2\rho})$ medium children. Propagating tags from the $\mathrm{O}(k^{1-\rho})$ affected medium blocks visits $\mathrm{O}(k^{2\rho-1})$ small children per affected medium block, for a total of $\mathrm{O}(k^{\rho})$ visits. Further propagation within recursively processed small blocks is charged to their recursive calls. These costs are included in the non-recursive maintenance bound in Lemma 3.2.

H.2 Dynamic Block-Size Maintenance

This section formalizes the dynamic block-size invariant used in the SPLIT and MERGE procedures. Index recursive grand-block levels by $\ell \geq 0$, with the root at level 0, and define the fixed standard scale $k_\ell = n^{2^{-\ell}}$. Throughout this subsection, $k = k_\ell$ denotes this standard scale, rather than the current physical side length of an individual block. We set $S_m = k_\ell^\rho$ and $S_s = k_\ell^{1/2} = k_{\ell+1}$, where $\rho \in (1/2, 1)$.

These standards depend only on the level and the global size parameter n, which is held fixed during the update sequence. For example, the third recursive level (counting the root as the first) always has standard scale $n^{1/4}$, even when its parent has actual side length different from $n^{1/2}$. Splitting or merging a parent does not redefine the standard scales of its children. Actual interval lengths may vary, and the local repair rules below control this variation between adjacent levels of the hierarchy. The idealized decomposition uses the standard lengths exactly; in the dynamically maintained structure, block-count estimates use the interval bounds below. These estimates are asymptotic bounds, not exact counts of recursive calls.

Definition H.1. We say that the current partition of a grand block is reasonable if the following conditions hold along both the row and column dimensions:

1. every medium interval has length at most $2S_m$, and every small interval has length at most $2S_s$;
2. the total length of any two adjacent medium intervals is at least S_m, and the total length of any two adjacent small intervals within the same medium interval is at least S_s.

An interval violating the first condition is called overly long, and a pair of adjacent intervals violating the second condition is called overly short.

The above invariant implies the desired block-count bounds.

Theorem H.2. *Consider a reasonable partition of a grand block at standard scale k, with actual side lengths $\mathrm{O}(k)$. Then the number of medium blocks is $\mathrm{O}(k^{1-\rho})$ per dimension and $\mathrm{O}(k^{2-2\rho})$ in total. Moreover, each medium block contains $\mathrm{O}(k^{\rho-1/2})$ small blocks per dimension and $\mathrm{O}(k^{2\rho-1})$ small blocks in total. Consequently, a structural row or column* SPLIT/MERGE *intersects $\mathrm{O}(k^{1-\rho})$ medium blocks and $\mathrm{O}(k^{1/2})$ small blocks.*

Proof. By reasonableness, every medium interval has length at most $2S_m$, and any two adjacent medium intervals have total length at least S_m. Pairing adjacent intervals bounds the number of medium intervals along a dimension of length $L = \mathrm{O}(k)$ by $\mathrm{O}(1 + L/S_m) = \mathrm{O}(k^{1-\rho})$. Therefore, the total number of two-dimensional medium blocks is $\mathrm{O}(k^{1-\rho}) \cdot \mathrm{O}(k^{1-\rho}) = \mathrm{O}(k^{2-2\rho})$. Similarly, every medium interval has length $\mathrm{O}(S_m)$, so pairing adjacent small intervals bounds their number by $\mathrm{O}(1+S_m/S_s) = \mathrm{O}(k^{\rho-1/2})$. Thus each medium block contains $\mathrm{O}(k^{\rho-1/2})\cdot\mathrm{O}(k^{\rho-1/2}) = \mathrm{O}(k^{2\rho-1})$ small blocks. Finally, a row or column cut crosses one interval in the cut dimension and all intervals in the other dimension. Hence it intersects $\mathrm{O}(k^{1-\rho})$ medium blocks and, equivalently, since the small intervals have standard length $k^{1/2}$, it intersects $\mathrm{O}(k^{1/2})$ small blocks. □

Remark H.3 (Rectangular blocks). The BMD Tree and the preceding counting argument do not require a block to be square. Consider a block with p rows and q columns, where $p, q = \mathrm{O}(k)$. Its numbers of medium intervals along the two dimensions are respectively $\mathrm{O}(1 + p/S_m)$ and $\mathrm{O}(1 + q/S_m)$, and the analogous small-interval bounds inside a medium block are $\mathrm{O}(1 + S_m/S_s)$ in each dimension. A row split intersects one row interval and all column intervals, while a column split is symmetric. Thus all bounds used above follow by multiplying the two dimension-wise counts, and remain upper-bounded by their square-block counterparts at scale k. At the root, $n_{\mathcal{A}}, n_{\mathcal{B}} = \Theta(n)$; recursively, choosing k as an upper bound on both side lengths therefore preserves the $\mathrm{O}(n)$ worst-case bound even when the generated blocks are rectangular.

We next describe how the invariant is maintained. Because the level-dependent standards remain fixed, a structural SPLIT or MERGE only changes $\mathrm{O}(1)$ row or column boundaries at the current grand-block level. Therefore, violations of the reasonable-partition invariant can only occur along the affected rows or columns. If a medium interval becomes overly long, we SPLIT it near its midpoint;

if two adjacent medium intervals become overly short, we MERGE them. The same rule is applied to small intervals inside each affected medium block.

Lemma H.4 (Local repair does not cascade)**.** *Along either dimension, restoring the reasonable-partition invariant after a structural* SPLIT *or* MERGE *requires only a constant number of repairs around each newly created or removed boundary. These repairs do not propagate to intervals farther from that boundary.*

Proof. Let S denote S_m for medium intervals or S_s for small intervals. Before the structural operation, every interval has length at most $2S$, and every adjacent pair has total length at least S. Therefore only intervals incident to a changed boundary can violate the invariant.

If a boundary operation creates an overly long interval, that interval is the union of at most two previously valid neighboring intervals and hence has length at most $4S$. Bisecting it produces two intervals of length at most $2S$; when its length exceeds $2S$, both halves have length at least S under an exact midpoint split. The new pair and the two pairs with their unchanged outer neighbors consequently satisfy the adjacent-pair condition.

If a split creates a short boundary interval and this interval together with its neighbor has total length below S, merging that pair creates an interval of length below S and hence below $2S$. Let the unchanged interval on the other side be z. The old neighboring pair already had total length at least S, and replacing one member by its merge with a nonnegative-length interval can only increase that sum. Thus the merge cannot create a new violation at the next boundary. The argument is symmetric on the two sides of a cut and for row and column partitions. Hence each changed boundary causes only constant local work and no cascading sequence of repairs. □

The cost of restoring the invariant is already covered by the non-recursive maintenance bound in Lemma 3.2. Indeed, a structural operation intersects $\mathrm{O}(k^{1-\rho})$ medium blocks. Recomputing the minimum of one affected medium block requires aggregating the minima of $\mathrm{O}(k^{2\rho-1})$ small blocks. Hence the medium-level contribution is $\mathrm{O}(k^{1-\rho}) \cdot \mathrm{O}(k^{2\rho-1}) = \mathrm{O}(k^{\rho})$. At the small-block level, a structural operation intersects $\mathrm{O}(k^{1/2})$ small blocks. For each affected small block, the non-recursive work only consists of constant-size boundary repair, metadata updates, and lazy-tag propagation to the recursive boundary. The actual internal SPLIT or MERGE of the small block is handled as an independent internal processing step within the small block and is not charged to the current non-recursive overhead. Thus the current-level small-block contribution is $\mathrm{O}(k^{1/2})$.

Combining these bounds with the global scan over all medium blocks, whose cost is $\mathrm{O}(k^{2-2\rho})$, the total non-recursive maintenance overhead at the current grand block is $\mathrm{O}\big(k^{\max\{2-2\rho,\rho,1/2\}}\big)$, which is exactly the bound used in Lemma 3.2. In particular, for $\rho = 2/3$, this becomes $\mathrm{O}(k^{2/3})$.

Finally, when an affected small block is processed recursively, it becomes the grand block of the next recursive level. The medium blocks inside that small block belong to the next-level decomposition, not to the current-level medium–small hierarchy. This separation avoids double counting and is the reason why the internal work on affected small blocks is evaluated separately as the internal processing within the small blocks.

I THE ALGORITHMS OF THE SPLIT OPERATION OF BMD TREE

We take the `Split` operation of the Bi-Resolution Matrix Decomposition Tree as an example (the `Merge` operation is analogous) and describe this operation in detail in algorithmic form. Assume we need to operate on a grand block D, whose set of medium sub-blocks is $Sub_m(D)$. For any $E \in Sub_m(D)$, let $Sub_s(E)$ be its set of small sub-blocks. Let O_m be the set of all medium blocks affected by this `Split` operation, and O_s the set of all affected small blocks. For any block B, denote its recorded minimum and lazy tag by $B.\mathtt{min}_v$ and $B.\mathtt{tag}_v$, respectively. The algorithm is shown in Algorithm 3.

Algorithm 3: Split operation of a Bi-Resolution Matrix Decomposition Tree

Input: The block D at recursive level ℓ to be split, with fixed standard scale $k = n^{2^{-\ell}}$; the sets O_m and O_s of affected medium and small blocks
Output: The resulting blocks D_1 and D_2, together with their minima $D_1.\mathtt{min}_v$ and $D_2.\mathtt{min}_v$

foreach $E \in Sub_m(D)$ **do**
  $E.\mathtt{tag}_v \leftarrow E.\mathtt{tag}_v + D.\mathtt{tag}_v$;
end
$D.\mathtt{tag}_v \leftarrow 0$;
foreach $E \in O_m$ **do**
  foreach $F \in Sub_s(E)$ **do**
    $F.\mathtt{tag}_v \leftarrow F.\mathtt{tag}_v + E.\mathtt{tag}_v$;
  end
  $E.\mathtt{tag}_v \leftarrow 0$;
end
foreach $F \in O_s$ **do**
  Execute recursive split on F, obtaining F_1 and F_2;
  Replace F by F_1 and F_2;
  Maintain the intervals of F_1 and F_2: split if length $> 2\sqrt{k}$; merge adjacent intervals if their total length $< \sqrt{k}$;
end
foreach $E \in O_m$ **do**
  Split E into E_1 and E_2;
  Maintain the intervals of E_1 and E_2: split if length $> 2k^{\rho}$; merge adjacent intervals if their total length $< k^{\rho}$;
  $E_1.\mathtt{min}_v \leftarrow +\infty$, $E_2.\mathtt{min}_v \leftarrow +\infty$;
  $E_1.\mathtt{tag}_v \leftarrow 0$, $E_2.\mathtt{tag}_v \leftarrow 0$;
  foreach $F \in Sub_s(E_1)$ **do**
    $E_1.\mathtt{min}_v \leftarrow \min\big(E_1.\mathtt{min}_v,\ F.\mathtt{min}_v + F.\mathtt{tag}_v\big)$;
  end
  foreach $F \in Sub_s(E_2)$ **do**
    $E_2.\mathtt{min}_v \leftarrow \min\big(E_2.\mathtt{min}_v,\ F.\mathtt{min}_v + F.\mathtt{tag}_v\big)$;
  end
end
Split D into D_1 and D_2;
$D_1.\mathtt{min}_v \leftarrow +\infty$, $D_2.\mathtt{min}_v \leftarrow +\infty$;
$D_1.\mathtt{tag}_v \leftarrow 0$, $D_2.\mathtt{tag}_v \leftarrow 0$;
foreach $E \in Sub_m(D_1)$ **do**
  $D_1.\mathtt{min}_v \leftarrow \min\big(D_1.\mathtt{min}_v,\ E.\mathtt{min}_v + E.\mathtt{tag}_v\big)$;
end
foreach $E \in Sub_m(D_2)$ **do**
  $D_2.\mathtt{min}_v \leftarrow \min\big(D_2.\mathtt{min}_v,\ E.\mathtt{min}_v + E.\mathtt{tag}_v\big)$;
end

J Proof of Lemma 3.2

The non-recursive maintenance involves rebalancing the block sizes, propagating lazy tags, and recalculating the local minimums. We must strictly separate these non-recursive repair costs from the recursive calls. We analyze the non-recursive complexity level by level based on the strict hierarchy (Grand block $\rightarrow$ Medium blocks $\rightarrow$ Small blocks):

- **Global operations.** The grand block at standard scale k is partitioned into $\mathrm{O}(k^{1-\rho}) \times \mathrm{O}(k^{1-\rho})$ medium blocks. Global tag propagation and finding the global minimum require iterating over all $\mathrm{O}(k^{2(1-\rho)})$ medium sub-blocks. Hence, the cost at this level is $\mathrm{O}(k^{2-2\rho})$.
- **Medium-level operations.** A global structural cut (split or merge) intersects $\mathrm{O}(k^{1-\rho})$ medium blocks. For each affected medium block, we must recompute its minimum by aggregating the values of its small sub-blocks. Since a medium block of side length k^{ρ}

contains $(k^\rho/k^{1/2})^2 = k^{2\rho-1}$ small blocks, the non-recursive cost per affected medium block is $\mathrm{O}(k^{2\rho-1})$. Therefore, the total non-recursive cost for all affected medium blocks is $\mathrm{O}(k^{1-\rho} \cdot k^{2\rho-1}) = \mathrm{O}(k^\rho)$.

- **Small-block operations.** The global cut intersects $\mathrm{O}(k^{1/2})$ small blocks across the entire block of size k. For each affected small block, the non-recursive overhead (such as adjusting local dynamic boundaries and propagating tags to the recursive boundary) takes constant time $\mathrm{O}(1)$. The actual internal splitting or merging of the small block is a recursive call mapped to $T(k^{1/2})$ and is not counted here. Thus, the total non-recursive repair cost at the small-block level is simply $\mathrm{O}(k^{1/2} \cdot 1) = \mathrm{O}(k^{1/2})$.

The overall non-recursive overhead is dominated by the most expensive of the three stages, which is $\mathrm{O}(k^{\max\{2-2\rho,\rho,1/2\}})$. This bound holds structurally for any $\rho \in (1/2, 1)$. To balance the polynomial exponents, setting $\rho = \frac{2}{3}$ evaluates the three costs to $\mathrm{O}(k^{2/3})$, $\mathrm{O}(k^{2/3})$, and $\mathrm{O}(k^{1/2})$, respectively. The maximum term dictates the overall overhead, yielding $\mathrm{O}(k^{2/3})$.

K Proof of Theorem 3.3

Proof. Let $T(k)$ denote the worst-case cost of maintaining a recursive block of standard scale k during one structural SPLIT or MERGE operation. Here $k = n^{2^{-\ell}}$ is the fixed standard scale at recursive level ℓ; the root scale is n. We assume that the base case takes constant time, i.e., $T(k) = \mathrm{O}(1)$ for $k \le 4$.

By Lemma 3.2, fixing $\rho = 2/3$ gives a non-recursive maintenance overhead of $\mathrm{O}(k^{2/3})$ for a block of side length k. Moreover, one structural operation intersects $\mathrm{O}(k^{1/2})$ small sub-blocks. In the idealized equal-length decomposition, there are $k^{1/2}$ recursive subproblems of standard scale $k^{1/2}$. Under this idealized counting, for some constant $C > 0$,

$$T(k) \le k^{1/2} T(k^{1/2}) + Ck^{2/3}. \tag{4}$$

Expanding the recurrence gives

$$\begin{aligned} T(k) &\le k^{1/2} T(k^{1/2}) + Ck^{2/3} \\ &\le k^{3/4} T(k^{1/4}) + Ck^{5/6} + Ck^{2/3} \\ &\le k^{7/8} T(k^{1/8}) + Ck^{11/12} + Ck^{5/6} + Ck^{2/3} \\ &\le \cdots . \end{aligned}$$

We now formalize this expansion. At recursion level $t \ge 0$, the number of subproblems is

$$\lambda_t = k^{1-2^{-t}},$$

and each subproblem has side length

$$k_t = \frac{k}{\lambda_t} = k^{2^{-t}}.$$

The total non-recursive overhead contributed by level t is therefore

$$\begin{aligned} a_t &= \lambda_t \cdot C \left(\frac{k}{\lambda_t}\right)^{2/3} \\ &= Ck^{2/3} \lambda_t^{1/3}. \end{aligned}$$

By Lemma 3.1, after

$$r = \lceil \log_2 \log_4 k \rceil$$

levels, the subproblem side length satisfies $k_r \le 4$. Hence the total number of base-case subproblems is $\mathrm{O}(k)$, and the total base-case contribution is $\mathrm{O}(k)$. Consequently,

$$T(k) \le \mathrm{O}(k) + \sum_{t=0}^{r-1} a_t. \tag{5}$$

It remains to bound the overhead summation. Consider the surrogate recurrence

$$S(k) = 2S(k/2) + Ck^{2/3}.$$

At level ℓ of this surrogate recurrence, the total overhead is

$$b_\ell = C2^{\ell/3}k^{2/3}.$$

We compare the sequence $\{a_t\}$ with the sequence $\{b_\ell\}$. As long as the original recursion has not reached the base case, $k/\lambda_t > 4$. Therefore,

$$\frac{\lambda_{t+1}}{\lambda_t} = \frac{k^{1-2^{-(t+1)}}}{k^{1-2^{-t}}} = k^{2^{-(t+1)}} = \sqrt{\frac{k}{\lambda_t}} > 2.$$

Let

$$\ell_t = \lfloor \log_2 \lambda_t \rfloor.$$

Then $2^{\ell_t} \le \lambda_t < 2^{\ell_t+1}$. Since $\lambda_{t+1} > 2\lambda_t$, the indices ℓ_t are strictly increasing up to a unit shift; in particular, the indices $\ell_t + 1$ are distinct. Moreover,

$$a_t = Ck^{2/3}\lambda_t^{1/3} \le Ck^{2/3}(2^{\ell_t+1})^{1/3} = b_{\ell_t+1}.$$

Thus the overheads of the original recurrence can be injected into distinct levels of the surrogate recurrence, and hence

$$\begin{aligned}\sum_{t=0}^{r-1} a_t &\le \sum_{t=0}^{r-1} b_{\ell_t+1} \\ &\le \sum_{\ell=0}^{\lfloor \log_2 k \rfloor} b_\ell \\ &\le S(k).\end{aligned}$$

By the Master Theorem, the surrogate recurrence

$$S(k) = 2S(k/2) + Ck^{2/3}$$

satisfies $S(k) = \Theta(k)$, since $\log_2 2 = 1$ and $k^{2/3} = \mathrm{O}(k^{1-\epsilon})$ for $\epsilon = 1/3$. Combining this with equation 5, we obtain

$$T(k) \le \mathrm{O}(k) + S(k) = \mathrm{O}(k).$$

Applying this bound to the root block with side length $k = n$ gives

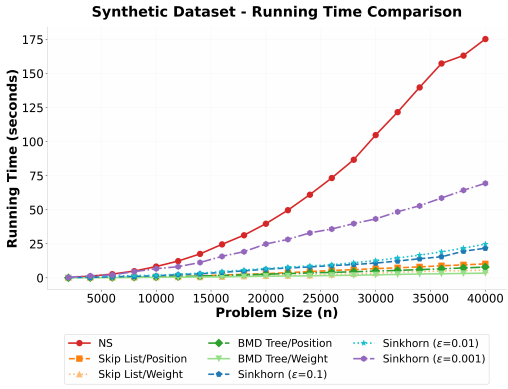
$$T(n) = \mathrm{O}(n).$$

Therefore, each structural SPLIT or MERGE, together with lazy-tag propagation, block-size rebalancing, and minimum maintenance, has $\mathrm{O}(n)$ worst-case reduced-cost matrix maintenance cost. Since QUERY and global additive UPDATE take $\mathrm{O}(1)$ time, the reduced-cost matrix maintenance cost per simplex iteration is $\mathrm{O}(n)$. □

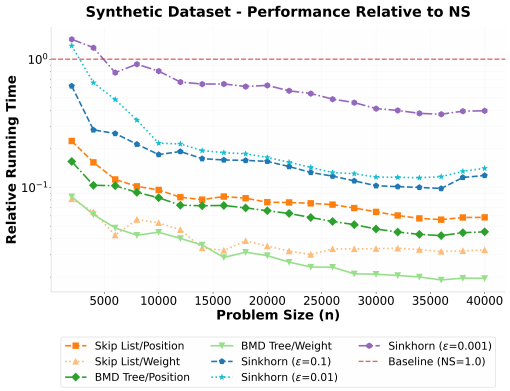
L THE EXPERIMENTAL RESULTS

L.1 DYNAMIC OPTIMAL TRANSPORT

Figure 8a shows the average operation time of two static algorithms, NETWORK SIMPLEX and SINKHORN, as well as two dynamic algorithms, Dynamic with SKIP ORTHOGONAL LISTS and Dynamic with Bi-Resolution Matrix Decomposition Tree, under two different operations—position modify and weight modify—on point sets following a two-dimensional Gaussian distribution. Figure 8b presents the ratio of the average time consumption of the aforementioned algorithms to that of NETWORK SIMPLEX. It can be observed that our algorithm demonstrates considerable advantages over both static algorithms in both types of operations. Moreover, compared to the dynamic algorithm using SKIP ORTHOGONAL LISTS, which has the same expected time complexity as the proposed algorithm, our approach still exhibits certain operational advantages.

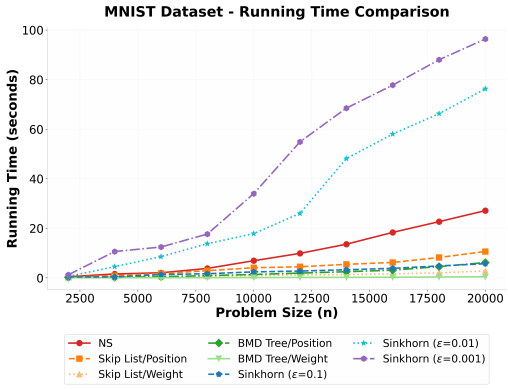


(a) Runtime comparison on synthetic node sets.

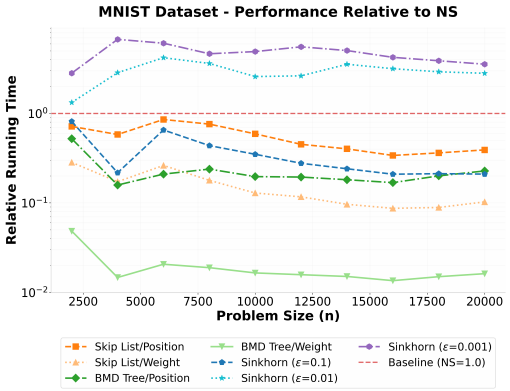


(b) Speedup analysis on synthetic node sets. The vertical axis is on a logarithmic scale.

Figure 8: Performance analysis on synthetic node sets.

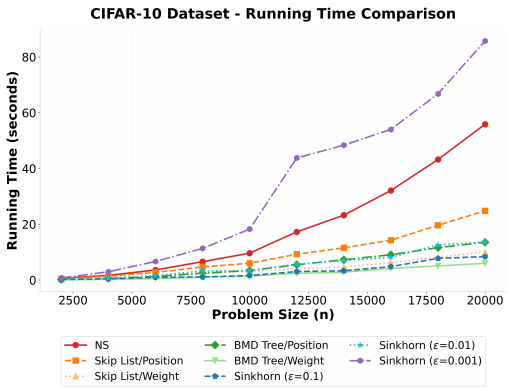


(a) Runtime comparison on MNIST.

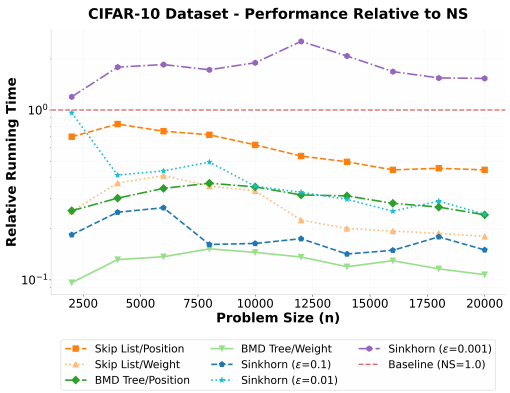


(b) Speedup analysis on MNIST. The vertical axis is on a logarithmic scale.

Figure 9: Performance analysis on MNIST dataset.



(a) Runtime comparison on CIFAR-10.



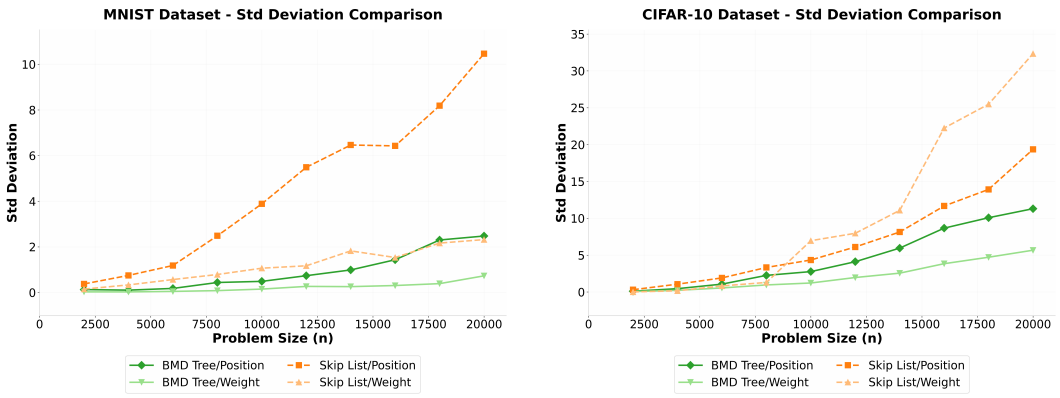
(b) Speedup analysis on CIFAR-10. The vertical axis is on a logarithmic scale.

Figure 10: Performance analysis on CIFAR-10 dataset.

L.2 EXPERIMENTS ON REAL-WORLD DATASET

Figure 9a and Figure 10a show the average operation time of various algorithms under two different operations on the MNIST and CIFAR-10 datasets, while Figure 9b and Figure 10b present the ratio of the average time consumption of the above algorithms to that of NETWORK SIMPLEX. It should be noted that we ensure that at any time, the number of points with the same label and the total weight in the source point set $\mathcal{A}$ and the sink point set $\mathcal{B}$ remain equal. It can be observed that our algorithm demonstrates significant advantages over all other algorithms on MNIST, whereas on CIFAR-10, the position modify operation is slower than the SINKHORN algorithm with $\texttt{reg} = 0.01$. This is partly because the convergence speed of the SINKHORN algorithm is highly dependent on the structure of the dataset (for example, it runs slower than NETWORK SIMPLEX on MNIST), and partly because the number of iterations may increase on datasets with more complex point distributions. However, regardless of the dataset, our algorithm shows a substantial and relatively stable advantage over the static NETWORK SIMPLEX and the dynamic algorithm with SKIP ORTHOGONAL LISTS, which confirms the superiority of our algorithm in obtaining exact solutions for dynamic optimal transport.

Furthermore, by comparing the performance of our algorithm across different datasets, it can be seen that its runtime does not vary significantly, especially in terms of being less affected by the dimensionality of point distributions and the choice of distance metrics. In fact, our algorithm can also operate on general graphs, demonstrating strong stability and broad applicability.



(a) Standard deviation on MNIST.

(b) Standard deviation on cifar10.

Figure 11: Standard deviation analysis on different datasets.

Figure 11a and Figure 11b illustrate the standard deviations of the dynamic NETWORK SIMPLEX algorithm using the BMD Tree structure and that using the SKIP ORTHOGONAL LIST on two real-world datasets. It can be observed that under both basic modification operations, our algorithm exhibits superior robustness. In contrast, as the SKIP ORTHOGONAL LIST is a probabilistic algorithm, its standard deviation is relatively larger.

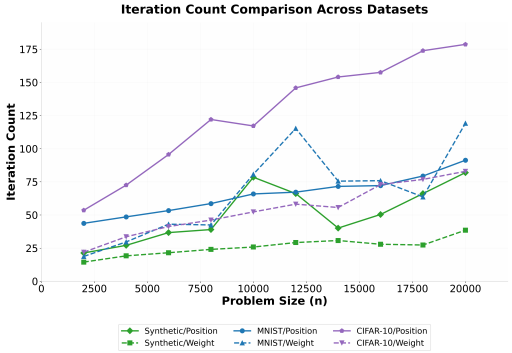
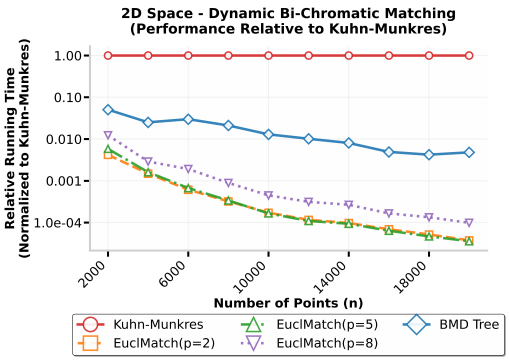


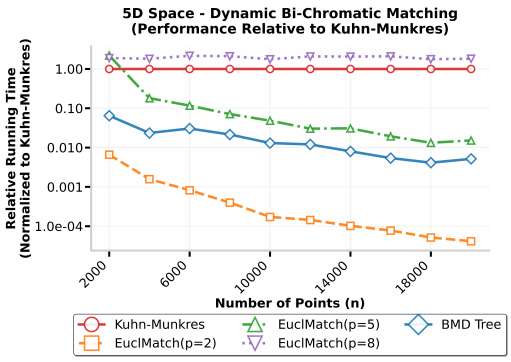
Figure 12: The number of iterations of our dynamic algorithm on different datasets.

Figure 12 shows the number of iterations for our dynamic simplex method under different datasets and modification operations. It can be observed that under all conditions, our dynamic algorithm requires a relatively low number of iterations, with the average number of iterations per modification not exceeding two hundred even for datasets with a scale of 20,000 nodes. On the one hand, this demonstrates that the efficiency of our algorithm does not rely on specific datasets or dimensions, but is broadly applicable across various scenarios; on the other hand, it also indicates that for dynamic algorithms based on the simplex method, the primary optimization objective is not the number of iterations, but rather the time complexity per iteration, which further highlights the significance of our work.

L.3 Dynamic Bi-Chromatic Matching

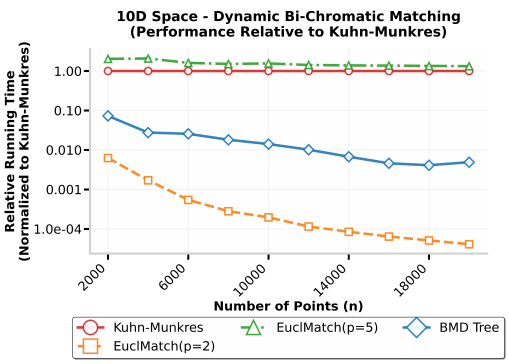


(a) Speedup Analysis on 2D Gaussian Point Sets. The vertical axis is on a logarithmic scale.

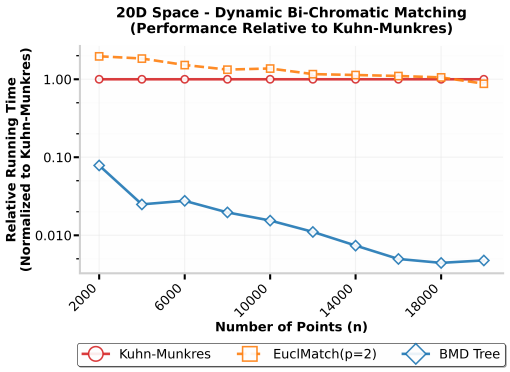


(b) Speedup Analysis on 5D Gaussian Point Sets. The vertical axis is on a logarithmic scale.

Figure 13: Speedup analysis on 2D and 5D Gaussian point sets.



(a) Runtime Comparison on 10D Gaussian Point Sets. The vertical axis is on a logarithmic scale.



(b) Speedup Analysis on 20D Gaussian Point Sets. The vertical axis is on a logarithmic scale.

Figure 14: Performance analysis on 10D and 20D Gaussian point sets.

For the Dynamic Bi-Chromatic Matching problem, our experiments utilized Gaussian point sets in 2-, 5-, 10-, and 20-dimensional spaces. The results demonstrate that our algorithm provides a substantial performance improvement over the standard static baseline, the KM algorithm.

It is important to note, however, that a more nuanced comparison involves the EuclMatch algorithm, whose grid parameter p specifies the number of subdivisions per coordinate axis, giving p^d subcells per cell in dimension d. Increasing p refines the grid but enlarges the local subproblems, producing a trade-off between computational speed and accuracy. Our analysis reveals that dimensionality significantly exacerbates this trade-off. While the running time of our method and the KM algorithm remains largely unaffected by increasing dimensions, the efficiency of EuclMatch degrades sharply

when both the dimension d and the parameter p are high—so much so that the corresponding performance curves for such configurations are omitted from the plots due to impractical runtimes.

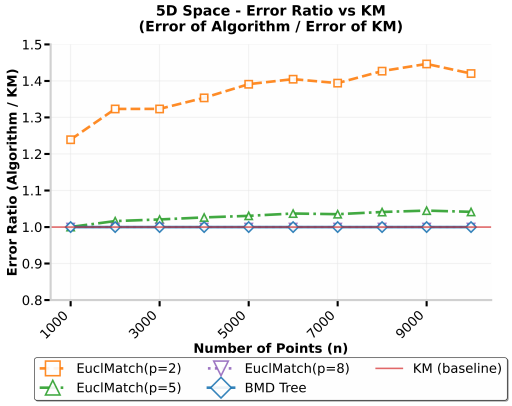


Figure 15: Error ratio Analysis on 5D Gaussian Point Sets. The vertical axis is on a logarithmic scale. The plot for the case where $p = 8$ almost overlaps with the baseline, as does the curve for the BMD Tree.

To quantify this accuracy-efficiency trade-off, we plot the error curve of EUCLMATCH for the case of $d = 5$, as shown in Figure 15. The data indicates that with a small p, EUCLMATCH incurs considerable error relative to the exact solution.

According to Figure 15, we find that while increasing p improves accuracy, it only preserves a competitive advantage in low-dimensional settings before becoming prohibitively slow. In contrast, our algorithm maintains high stability and computes exact solutions efficiently across all tested dimensions, a capability that is crucial for the high-dimensional applications of this problem.

It should be clarified that Dynamic Bi-Chromatic Matching, whose OT formulation has uniform weights, can undergo long sequences of degenerate iterations, increasing the re-optimization time. We prevent cycling with the strongly feasible tree and leaving-arc rule described in Section G. This rule changes only which blocking edge leaves the basis in a tie; it does not modify the OT weights or costs, and the returned matching therefore remains optimal for the original instance.