Discrete Tree Flows via Tree-Structured Permutations

Mai Elkady [*1] **Jim Lim** [*2] **David I. Inouye** [2]

Abstract

While normalizing flows for continuous data have been extensively researched, flows for discrete data have only recently been explored. These prior models, however, suffer from limitations that are distinct from those of continuous flows. Most notably, discrete flow-based models cannot be straightforwardly optimized with conventional deep learning methods because gradients of discrete functions are undefined or zero. Previous works approximate pseudo-gradients of the discrete functions but do not solve the problem on a fundamental level. In addition to that, backpropagation can be computationally burdensome compared to alternative discrete algorithms such as decision tree algorithms. Our approach seeks to reduce computational burden and remove the need for pseudo-gradients by developing a discrete flow based on decision trees—building upon the success of efficient tree-based methods for classification and regression for discrete data. We first define a tree-structured permutation (TSP) that compactly encodes a permutation of discrete data where the inverse is easy to compute; thus, we can efficiently compute the density value and sample new data. We then propose a decision tree algorithm to build TSPs that learns the tree structure and permutations at each node via novel criteria. We empirically demonstrate the feasibility of our method on multiple datasets.

1. Introduction

Discrete categorical data is abundant in numerous applications and domains, from DNA sequences and medical records to text data and many forms of tabular data. Analyzing discrete data by modelling and inferring its distribution is crucial in many of those applications and can encourage innovation in discrete data utilization such as generating new data that is of similar distribution and properties to our original data. However, discrete data can be inherently hard to model.

Probabilistic graphical models are an important classical way to model discrete distributions such as the Ising model for binary data (Wainwright & Jordan, 2008), multivariate Poisson distributions (Inouye et al., 2017) for count data, mixture models (Ghojogh et al., 2020), admixture models (Inouye et al., 2014), and discrete variational autoencoders (VAE) (van den Oord et al., 2018), (Razavi et al., 2019). However, some of these graphical models lack tractable exact likelihood computation or sampling procedures.

Autoregressive models for discrete data can provide exact likelihood computation (e.g., (Germain et al., 2015)), however, their structures hinder sampling speed as variables are conditionally dependent on one another and must be traversed in a fixed order.

A relatively new approach in tackling this problem is normalizing flows (Rezende & Mohamed, 2016), which leverage invertible models to combine different advantages of latent variable models by having a latent variable representation of the data, while also allowing for exact likelihood computation and fast sampling. These models have mostly focused on continuous random variables (Dinh et al., 2017),(Kingma et al., 2016),(Papamakarios et al., 2018), but have recently been introduced for discrete random variables as well.

The two key components of discrete flows are a base distribution (similar to continuous flows) and a permutation of the discrete configuration values (a discrete version of invertible functions) (Tran et al., 2019). *Discrete flows* have a similar change of variable structure via invertible models as seen in this comparison between continuous change of variables and discrete change of variables:

$$\begin{aligned} P_x(x) &= Q_z(f(x)) \det \left| \frac{df(x)}{dx} \right| \quad \text{(Continuous)} \\ P_x(x) &= Q_z(f(x)) \quad \text{(Discrete)} \end{aligned} \tag{1}$$

where Q_z is some (possibly learnable) base distribution that is usually a simple distribution (e.g., a Gaussian in the continuous flows case or an independent categorical distribution for discrete flows), and f is an invertible function. Impor-

[*]Equal contribution [1]Department of Computer Science, Purdue University [2]Department of Electrical and Computer Engineering, Purdue University. Correspondence to: Mai Elkady <melkady1@purdue.edu>, Jim Lim <lim316@purdue.edu>.

Proceedings of the 39^{th} International Conference on Machine Learning, Baltimore, Maryland, USA, PMLR 162, 2022.

tantly, in the discrete case, the only invertible functions are permutations over the possible discrete configurations of x; thus, there is no Jacobian determinant term because permutations do not change volume. However, given that the number of discrete configurations of d features with k possible discrete values is k^d, the number of all possible permutations of all configurations is $k^d!$. Optimizing over all possible permutations is computationally intractable. Thus, parameterizing and optimizing over a subset of possible permutations is critical for practical algorithms and generalizability.

Different methods were proposed to handle modeling discrete data using normalizing flows, among these are:

Using a straight-through gradient estimator (STE) to approximate the gradients: Tran et al. (2019) introduces Autoregressive and Bipartite flows (AF and BF) to model *categorical* discrete data and proposes to parameterize the permutations using neural networks, and to tackle discrete functions non-differentiability they resorted to using an STE (Bengio et al., 2013) along with a Gumbel-softmax distribution for back-propagation. This work suggests that gradient bias from the discrete gradient approximations may be a key issue, van den Berg et al. (2020) later show that the architecture of the coupling layers is significantly more important than the gradient bias issue. Hoogeboom et al. (2019) introduces integar discrete flows (IDF) and van den Berg et al. (2020) improves upon it in IDF++. Both works also use STEs but focus on *integer* discrete data.

Using a mixture of continuous and discrete training: Lindt & Hoogeboom (2021) introduced Discrete Denoising Flows (DDFs) to model *categorical* discrete data. They proposed to separate learning into a continuous learning step and a discrete step: learn an NN probabilistic classifier on half of the features to predict the other half of the features (i.e., a bipartite coupling layer) and then sort each feature in the second half based on the predicted logits (i.e., the discrete operation). This approach sidesteps the possible gradient bias.

Solving the problem in the continuous space, then projecting the solution in the discrete space: Ziegler & Rush (2019) introduced 3 different flow architectures (which we'll refer to as latent flows (LF)) that can be used as the prior in a VAE model with a variable length latent code which is used to encode the discrete data in a continuous space. In a similar fashion, Lippe & Gavves (2021) introduces categorical normalizing flows (CNF) and proposes to use an encoder to project categorical discrete variables to the continuous space using variational inference and then utilize a continuous flow model to solve the problem. Hoogeboom et al. (2021) introduced ARGMAX flows (ARGMAXF) that use an argmax based transformation, and a probabilistic right inverse to lift the discrete categorical data into the continuous space. These approaches don't allow for a discrete latent space to be achieved, nor for exact likelihood calculation, but overcomes issues that are common for categorical data when using dequantization. A comparison of all these previous works is presented in Table 1.

Table 1: A comparison of different methods for solving the discrete flows problem in terms of whether the method handles categorical data, will produce a discrete latent space and is trained by Exact likelihood Optimization(ELO)

	CATEGORICAL DATA	DISCRETE LATENT SPACE	TRAINING BY ELO
AF, BF	√	√	√
DDF	√	√	√
IDF	×	√	√
IDF++	×	√	√
LF	√	×	×
CNF	√	×	×
ARGMAXF	√	×	×

Most of these prior methods don't address the discrete nature of the problem on a fundamental level and may be computationally expensive (as they usually involve optimizing many neural network parameters which translates to an increase in training times) compared to alternative discrete-oriented algorithms such as those based on decision trees—which have seen wide success in classification and regression for discrete data (e.g., XGBoost (Chen & Guestrin, 2016)). Thus, we seek to answer the following research question: **Can we design a more computationally efficient discrete flow algorithm using decision trees that handles discrete data on a fundamental level?**

To answer this, we propose a novel tree-structured permutation (TSP) model that can compactly represent permutations and a novel decision tree algorithm that optimizes over the space of these permutations. Moreover, for more powerful permutations, we can iteratively build up a sequence of TSPs to form a deep permutation which we refer to as Discrete Tree Flows (DTFs).

We summarize our contributions as follows:

- We define Tree-Structured Permutations (TSP) that compactly encode permutations at each node of the tree and we prove what constraints are required on these permutations such that the whole TSP is efficiently invertible.
- We also propose a novel decision tree algorithm for building TSPs that includes building the tree structure based on a splitting critera, and two passes to learn and apply permutations. We theoretically prove the viability of our algorithm as well.
- Finally, we demonstrate the feasibility of our method

on simulated and real-world categorical datasets.

2. Model: Discrete Tree Flows (DTF)

We introduce a new model for discrete flows called Tree Structured Permutation (TSP) that utilizes trees for compactly parameterizing a set of permutations based on decision trees. Our Discrete Tree Flow (DTF) model is merely a composition of multiple TSPs. First, we briefly define some notation that will be used throughout the paper.

Notation We will denote a discrete dataset as $X \in \mathcal{Z}^{n \times d}$ where n is the number of samples, d is the number of dimensions, and $\mathcal{Z}$ is a set of discrete values, and where the maximum number of possible discrete values per feature (i.e., the number of categories) is k. Let π denote an *independent* permutation, i.e., $\pi(\boldsymbol{x}) = [\pi_0(x_0), \pi_1(x_1), \cdots, \pi_d(x_d)]$ where $\pi_j(x_j)$ is a 1D permutation, and let Π denote the set of independent permutations. Similarly, let σ denote a general (possibly non-independent) permutation (e.g., $\sigma_{\mathcal{T}}$), and let Σ denote a set of general (possibly non-independent) permutations. Given an independent permutation π and a count matrix $c \in \mathbb{Z}^{d \times k}$, let $\pi[c]$ denote the operation of permuting the entries of the count matrix by π (note that this is different than applying the permutation to a category value a as in $\pi(a)$) and can be defined as: $\pi[c] \triangleq [\pi_0[c(0)], \pi_1[c(1)], \cdots, \pi_d[c(d)]]$, where $\pi_j[c(j)] \triangleq [c(j, \pi_j^{-1}(0)), c(j, \pi_j^{-1}(1)), \cdots, c(j, \pi_j^{-1}(k))]$, and where the first entry and the second entry in $c(\cdot, \cdot)$ specify a feature and a category, respectively in the count matrix. We let $\circ$ denote the composition operator, e.g., $\pi_1 \circ \pi_2(x) = \pi_1(\pi_2(x))$. We will denote a binary decision tree by it's set of nodes $\mathcal{T} \triangleq \{N_j\}_{j=0}^{|\mathcal{N}|-1}$ where each node (except a leaf node) has two child nodes and N_0 is the root node. Each node will have an associated split feature s, split value(s) v, and node permutation π_N.

2.1. Tree-Structured Permutations

We define a *tree-structured permutation* to be a binary decision tree where each node N is described by both a permutation π and the usual split information (i.e., a split feature s and split value v). Informally, to evaluate a TSP, an input vector traverses the tree from the root to a leaf node based on the split information and the node permutations are applied as soon as the data reaches the node. Thus, a TSP node will first permute the input data and then forward the data to the left or right node depending on the split information. To ensure the TSP is computationally tractable, we choose to restrict our allowable permutations to the natural and computationally tractable class of independent feature-wise permutations, denoted by Π, which allows each feature to be permuted independently of the other features (i.e., the permutation of one feature cannot depend on the permutations of other features). This class significantly reduces the number of permutations compared to all possible permutations, i.e., $|\Pi| = (k!)^d \ll (k^d)! = |\Sigma|$, where Σ is the set of all possible permutations of k^d possible configurations. We illustrate the forward traversal with an example in Figure 1. We formally define our tree-structured permutation as follows:

Definition 1 (Tree-Structured Permutation)**.** *A* tree-structured permutation *is defined as* $\sigma_{\mathcal{T}}(\boldsymbol{x}) \triangleq f_{N_0}(\boldsymbol{x})$*, where* N_0 *is the root node and*

$$f_N(x) \triangleq \begin{cases} \pi_N(x), & \text{if leaf node} \\ f_{N_{\text{left}}}(\pi_N(x)), & \text{if } \pi_N(x)_s \in v \\ f_{N_{\text{right}}}(\pi_N(x)), & \text{otherwise} \end{cases} \tag{2}$$

where $s \in \{0, 1, \ldots, k-1\}$ *denotes the split feature for the node and* v *are the set of values that determine if the observation goes left in the tree.*

We also define the set of configurations that will reach a particular node as follows:

Definition 2 (Node Domain)**.** *The* node domain *of the root node is all possible configurations, i.e.,* $\mathcal{D}(N_0) \triangleq \mathcal{Z}^d$*, and* node domains *of children nodes are defined recursively as:*

$$\mathcal{D}(N_{\text{left}}) \triangleq \{\pi_N(\boldsymbol{x}) : \boldsymbol{x} \in \mathcal{D}(N), \pi_N(\boldsymbol{x})_s \in v\}$$
$$\mathcal{D}(N_{\text{right}}) \triangleq \{\pi_N(\boldsymbol{x}) : \boldsymbol{x} \in \mathcal{D}(N), \pi_N(\boldsymbol{x})_s \notin v\},$$

where s *and* v *are the split feature and value(s) of node* N*. Similarly, let* $\mathcal{D}_j(N) \triangleq \{x_j : \boldsymbol{x} \in \mathcal{D}(N)\}$ *denote the domain of the* j*-th feature.*

2.2. Invertibility of TSPs

To ensure our TSPs are invertible (and thus applicable to discrete flows), we prove that a simple and intuitive constraint on the node permutations is sufficient as defined next.

Theorem 1. *TSP Invertibility Constraint: A TSP is invertible if the range of the node permutations is equal to the node domain, i.e.,* $\forall \boldsymbol{x} \in \mathcal{D}(N), \pi_N(\boldsymbol{x}) \in \mathcal{D}(N)$.

The proof is constructive and relies on the following lemma that is proved by induction in Appendix E.

Lemma 1. *If the invertibility constraint is satisfied, the TSP tree traversal path for any input can be recovered from the output.*

Given this lemma, the sequence of permutations that was applied to an input point can be recovered and the inverse is then merely the inverse of this sequence of permutations (which are all invertible themselves).

While this gives us understanding about the invertibility of TSPs, the proof of Lemma 1 (in Appendix E) naturally gives

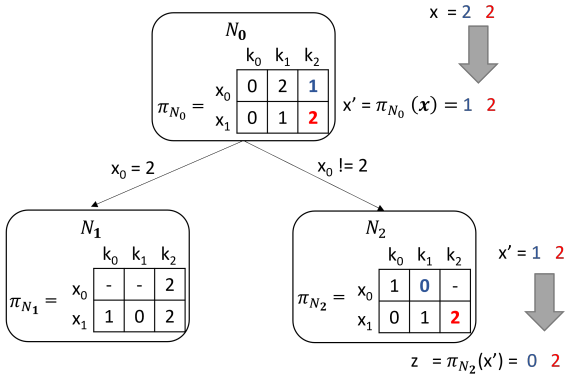


Figure 1: An example of evaluating the forward pass of a TSP on datapoint x yielding datapoint z. At each node the input data will be permuted by the node's permutation π_N. We will represent the independent permutation π by a $d \times k$ matrix (which is different from a regular permutation matrix). The row indices correspond to each feature (x_0 and x_1), the column indices correspond to categories (k_0, k_1, k_2), and the matrix entries correspond to the new category value (i.e., the permuted value). A value of - in the matrix indicates entries that are outside the node's domain $\mathcal{D}(N)$. As an example, consider applying the permutation to the input $\boldsymbol{x}$. The first dimension (x_0) has a category value 2. In the permutation representation matrix the entry at row index 0 and column index 2 is 1, so the datapoint will be permuted to 1 at the 0th dimension after passing through N_0. The data then passes to the left or right nodes depending on the split information which are presented on the arrows.

an efficient algorithm for computing the inverse. We show that we can compute the inverse by traversing the tree in the forward direction (i.e., from root to leaves) without applying the node permutations but keeping track of the path. Once we reach a leaf, we apply the permutations along the path in reverse order to compute the inverse. While we provide a simple sufficient condition for invertibility, we prove a more complex necessary and sufficient condition for invertibility in Appendix E.

2.3. Expressivity of DTFs

We prove in Appendix D that a composition of TSPs does not restrict expressivity, i.e., a sequential composition of TSPs can produce a universal permutation. As a proof sketch, we demonstrate that a single TSP can swap configurations that differ in only one feature value while keeping all other configurations the same. Then, we prove that there exists a snake-like path that connects all possible configurations such that adjacent configurations on the path differ in only one feature value. Finally, we use the fact that all transpositions (i.e., swaps) between two adjacent configurations is a generating set of the whole permutation space. Thus, by composing TSPs into a DTF we can express any possible permutation (see Appendix D for full proof).

3. Algorithm: Learning Discrete Tree Flows

Our goal is to minimize the Negative Log Likelihood (NLL) assuming an independent base distribution $Q_{\boldsymbol{z}}$, i.e.,

$$\arg\min_{\sigma_{\mathcal{T}}} \min_{Q_{\boldsymbol{z}}} -\frac{1}{n}\sum_{i=1}^{n} \log Q_{\boldsymbol{z}}(\sigma_{\mathcal{T}}(\boldsymbol{x}_i)) \,. \tag{3}$$

Note that solving the inner minimization problem over $Q_{\boldsymbol{z}}$ is known in closed-form based merely on the discrete value counts along each dimension independently. While at first it may seem that we must learn the permutations and the splits simultaneously, we show that we can decouple this problem into two subproblems: 1) Estimate decision tree structure and 2) Estimate node permutations.

First, our algorithm estimates the decision tree structure (i.e., the split features and values) via a split criteria. Second, given the decision tree structure, our algorithm globally optimizes the node permutations over all TSPs with equivalent tree structure (the equivalence of tree structure is defined later). Intuitively, this finds the node permutations that will make the different conditional distributions align as much as possible—thereby moving the joint distribution towards independence. We give the pseudocode of our algorithms in Appendix H.

3.1. Splitting Criteria for Structure Learning

As in every decision tree algorithm, we need to define an appropriate splitting criteria. We choose to implement a simple baseline of randomly splitting, i.e., choosing a split feature at random then a split value also at random. We refer to a DTF that uses this approach as DTF$_{RND}$. We also propose another approach for splitting, that is a heuristic we call *greedy local permutation* (GLP), that relies on theorizing about doing a permutation for a specific split and choosing the split that leads to the best decrease in NLL if that hypothetical permutation is to be applied.

Greedy local permutation splitting criteria For this approach, we evaluate the possible splits based on the potential decrease in NLL that will be attained if we were to split the data accordingly and do a local greedy permutation if needed. We define a local greedy permutation to be the permutation of data that sorts the categorical values in ascending order for each dimension. We present a visual example of how this splitting criteria works in Appendix B. More formally, we define the minimum permuted negative

log likelihood criteria as follows

$$\text{MinPermNLL}(s, v) = \min_{Q,\pi} -\sum_{i=1}^{n} \log Q_\pi(\boldsymbol{x}_i)\,, \quad (4)$$

where $Q_\pi(\boldsymbol{x}) \triangleq I(x_s \in v)Q(\boldsymbol{x}) + I(x_s \notin v)Q(\pi(\boldsymbol{x}))$, Q is a shared independent distribution, and the permutation for the split feature is the identity (no permutations), i.e., $\pi_s = \pi_{\text{id}}$ and I is an indicator function that is 1 if the condition is true and 0 otherwise. Because Q is independent and π is a feature-wise permutation, this split criterion can be trivially decomposed into d independent subproblems that can be solved exactly in 1D using sorting. Concretely, without loss of generality, we assume that the counts on the left of the split are already in ascending order. The optimal π is merely the permutation such that the counts on the right are also sorted in ascending order, and the minimum likelihood Q is equal to the empirical frequencies after sorting. We give the pseudocode of our ConstructTree algorithm and our FindBestSplit algorithm below in Alg. 1 and Alg. 2, respectively.

Algorithm 1 ConstructTree: Learn node splits and count data at leaves

Input: Node N, training data at node X, max depth M, min samples to split $n_{\min}$, split score function $\phi(\cdot, \cdot)$
Output: Root node of decision tree structure N_0

if $|N.X| \geq n_{\min}$ **and** $N.\text{depth} < M$ **then**
  $(N.s, N.v, X_{\text{left}}, X_{\text{right}}, \mathcal{D}_{\text{left}}, \mathcal{D}_{\text{right}}) \leftarrow$ FindBestSplit$(X, N.\mathcal{D}, \phi)$
  $N.\text{left} \leftarrow$ CreateNode$(\mathcal{D}_{\text{left}}, N.\text{depth} + 1, \pi = \pi_{\text{Id}})$
  $N.\text{right} \leftarrow$ CreateNode$(\mathcal{D}_{\text{right}}, N.\text{depth} + 1, \pi = \pi_{\text{Id}})$
  ConstructTree$(N.\text{left}, X_{\text{left}})$
  ConstructTree$(N.\text{right}, X_{\text{right}})$
else
  $N.\tilde{c}^{\text{init}} \leftarrow$ CountsPerDimension(X)
end if
return N

Algorithm 2 FindBestSplit: Find best split of data

Input: Node data X, node domain $\mathcal{D}$, split score function $\phi(\cdot, \cdot)$ that will depend on the splitting critera we use
Output: Split feature s, split values v, left and right data $(X_{\text{left}}, X_{\text{right}})$, left and right domains $(\mathcal{D}_{\text{left}}, \mathcal{D}_{\text{right}})$

$\mathcal{S} \leftarrow$ GeneratePossibleSplits$(\mathcal{D})$
$s^*, v^* \leftarrow \arg\max_{(s,v)\in\mathcal{S}} \phi(\text{LeftSplit}(X, s, v), \text{RightSplit}(X, s, v))$
$(X_{\text{left}}, X_{\text{right}}) \leftarrow$ Split(X, s, v)
$(\mathcal{D}_{\text{left}}, \mathcal{D}_{\text{right}}) \leftarrow$ CreateChildDomains$(\mathcal{D}, s, v)$
return $(s^*, v^*, X_{\text{left}}, X_{\text{right}}, \mathcal{D}_{\text{left}}, \mathcal{D}_{\text{right}})$

3.2. Learning Node Permutations

We first explore the theoretically optimal node permutations given the decision tree structure learned in the first step. Then we will present our algorithm for learning node permutations and prove that it learns the theoretically optimal node permutations.

3.2.1. Theoretically Optimal Node Permutations

Before we can provide our optimality theorem, we need to introduce several important definitions. The next two definitions formalize the idea that the conditional distributions at each leaf node should be as close as possible to the marginal distribution over all leaf nodes.

Definition 3 (TSP Node Counts)**.** *Given a training dataset $X \in \mathcal{Z}^{n\times d}$, we define the TSP* node counts *$c_N \in \mathbb{Z}_+^{d\times k}$ for a node N as follows:*

$$c_N(j, a) = \sum_{i=1}^{n} \mathbb{1}(\sigma_{\mathcal{T}}(\boldsymbol{x}_i)_j = a \wedge \sigma_{\mathcal{T}}(\boldsymbol{x}_i) \in \mathcal{D}(N)) \quad (5)$$

where $j \in \{0, 1, \ldots, d-1\}$, $a \in \{0, 1, \ldots, k\}$, and $\mathbb{1}$ is an indicator function that is 1 if the condition is true and 0 otherwise.

Definition 4 (Rank Consistency)**.** *A TSP $\sigma_{\mathcal{T}}$ is rank consistent if and only if there exists an independent permutation π such that $\forall N, \pi[c_N] \in \mathcal{R}(\mathcal{D}(N))$, where $\mathcal{R}(\mathcal{D}(N))$ is the set of* rank consistent *count matrices $\mathcal{R}$ w.r.t. to a node's domain $\mathcal{D}(N)$ defined as $\mathcal{R}(\mathcal{D}) \triangleq \{c \in \mathbb{Z}_+^{d\times k} : \forall j,\ c(j) \in \mathcal{R}(\mathcal{D}_j)\}$, and $\mathcal{R}(\mathcal{D}_j)$ is the set of rank consistent count vectors defined as: $\mathcal{R}(\mathcal{D}_j) \triangleq \{c(j) \in \mathbb{Z}_+^k : \forall a < b, \in \mathcal{D}, c(j, a) \leq c(j, b), \forall \ell \notin \mathcal{D}, c(j, \ell) = 0\}$.*

We also need to formalize the notion that TSPs can have an equivalent decision tree structure even if they have different node permutations.

Definition 5 (TSP Tree Equivalence)**.** *Two TSPs $\sigma_{\mathcal{T}}^A, \sigma_{\mathcal{T}}^B$ are* tree equivalent*, denoted by $\sigma_{\mathcal{T}}^A \equiv_{\text{tr}} \sigma_{\mathcal{T}}^B$ if and only if they have the same graph structure (i.e., same nodes and edges), and $\forall j, \boldsymbol{x}$, $\sigma_{\mathcal{T}}^A(\boldsymbol{x}) \in \mathcal{D}(N_j^A) \Leftrightarrow \sigma_{\mathcal{T}}^B(\boldsymbol{x}) \in \mathcal{D}(N_j^B)$ where j is the index of the node.*

The proof that this is a true equivalence relation and more discussion is in the Appendix G. This means that any input value will traverse the same path through both TSPs and land at an equivalent leaf node in either structure. Note that if the node permutations are different between the two TSPs, then the split values will also be different but equivalent. For example, if previously the node had an identity permutation and split on the value v, an equivalent node with permutation π should split on $\pi(v)$. The tree equivalence definition ensures this is true for the whole tree and not just a single

split. Given these definitions, we can now provide our main optimality result.

Theorem 2 (Optimality of rank consistent TSPs)**.** *Given a TSP tree $\mathcal{T}^*$, rank consistent TSPs attain the optimal negative log-likelihood among TSPs that are tree equivalent, i.e., rank consistent TSPs are optimal solutions to the problem:*

$$\underset{\sigma_{\mathcal{T}} \in \Sigma(\mathcal{T}^*)}{\arg\min} \min_{Q_{\boldsymbol{z}}} -\frac{1}{n} \sum_{i=1}^{n} \log Q_{\boldsymbol{z}}(\sigma_{\mathcal{T}}(\boldsymbol{x}_i)) \,, \tag{6}$$

where $\Sigma(\mathcal{T})$ is the set of TSPs whose trees are tree equivalent to $\mathcal{T}^*$, and $Q_{\boldsymbol{z}}$ is an independent distribution.

The proof by contradiction is presented in Appendix F. Informally, if this property does not hold, there exists a pair of counts at a leaf node that could be switched and decrease the negative log likelihood leading to a contradiction. Intuitively, this theorem states that making the conditional distributions after transformation of each leaf (i.e., local counts of each leaf node) approximately match the marginal distributions (i.e., the global counts) is theoretically optimal—essentially the alignment of conditional distributions is the best way to get approximate independence. This will lead to better log likelihood because we assume the prior distribution is independent.

3.2.2. Two-Pass Algorithm to Learn Optimal Node Permutations

Given the theoretical results of the previous section and the tree structure that we learned via one of our splitting criteria (an example of such tree is presented in the leftmost part of Figure 2), we begin our two pass algorithm. In the first pass (see Alg. 3: LearnLocalPermutations), we traverse the tree in a bottom-up fashion to learn the local node permutations ($\tilde{\pi}$), which are different from the node's final permutation π. These auxiliary local node permutations $\tilde{\pi}$ relate the local counts $\tilde{c}$ to the node counts of the new tree constructed in the next pass (this is formalized in Lemma 6 in Appendix I). $\tilde{\pi}$ is learned by sorting the initial (unsorted) local node counts $\tilde{c}^{\text{init}}$ in each dimension j in ascending order only on the node's domain $\mathcal{D}(N)$. Because the algorithm recursively does this from bottom to top, all local counts would be globally ranked (i.e., rank consistent). We demonstrate this in the middle part of Figure 2 where at the leaf level we sort the counts and permute the data based on the sorting outcome, at the upper levels, we either simply combine the permuted counts from the level below (for all features except the split feature as seen on the left part of the tree) or combine the data, sort again and permute for the split feature (since combining can lead to a distortion in the sorted order, as seen in the right part of the tree). However, this first pass does not construct a valid TSP. So in the second pass (see Alg. 4: ConstructEquivalentTree), we traverse the tree in a top-down fashion to construct an equivalent TSP (i.e., changing the node permutations π_N and split values v) given the auxiliary local permutations $\tilde{\pi}$ learned in the first pass. This new tree will be equivalent in tree structure to the original tree but now will be rank consistent and therefore optimal as per Theorem 2. This is demonstrated in the rightmost side of Figure 2. In this example, we see that if we were to start from the original data (the data at N_0 on the leftmost side of the figure) and apply the new TSP (with updated π and v, we would arrive to the sorted data at the leaves, and our tree would be rank consistent.

Algorithm 3 LearnLocalPermutations: Learn local permutations to sort local counts

Input: Node N
Output: Modified node N with local permutations ($\tilde{\pi}$)

if N is a non-leaf internal node **then**
  N.left $\leftarrow$ LearnLocalPermutations(N.left)
  N.right $\leftarrow$ LearnLocalPermutations(N.right)
  $N.\tilde{c}^{\text{init}} \leftarrow N$.left.$\tilde{c} + N$.right.$\tilde{c}$
end if
$\forall j, N.\tilde{\pi}_j \leftarrow \text{Sort1D}(N.\tilde{c}_j^{\text{init}}, \mathcal{D}(N)_j)$
$N.\tilde{c} \leftarrow \text{PermuteCounts}(N.\tilde{\pi}, N.\tilde{c}^{\text{init}})$
return N

Algorithm 4 ConstructEquivalentTree: Construct a new tree with equivalent splits and permutations based on local permutations

Input: Node N with local permutations $N.\tilde{\pi}$, ancestor permutation π_{anc}
Output: Modified node N with valid but equivalent permutation subtree structure

$N.\pi \leftarrow \pi_{anc} \circ N.\tilde{\pi} \circ \pi_{anc}^{-1}$
if N is internal non-leaf node **then**
  $N.v \leftarrow N.\tilde{\pi} \circ \pi_{anc}(N.v)$
  $\pi_{anc}^{\text{child}} \leftarrow N.\pi \circ \pi_{anc}$
  ConstructEquivalentTree(N.left, π_{anc}^{child})
  ConstructEquivalentTree(N.right, π_{anc}^{child})
end if
return N

We then prove two lemmas regarding this two-pass algorithm (proofs in Appendix I). The first lemma ensures that Alg. 4: ConstructEquivalentTree produces a valid TSP that has equivalent tree structure. The second lemma proves that the permutations learned in Alg. 3: LearnLocalPermutations produce a rank consistent TSP. The proof of algorithm optimality follows easily from these two lemmas and Theorem 2.

Lemma 2. *The ConstructEquivalentTree algorithm produces a new TSP that has equivalent tree structure to the original TSP from the ConstructTree algorithm.*

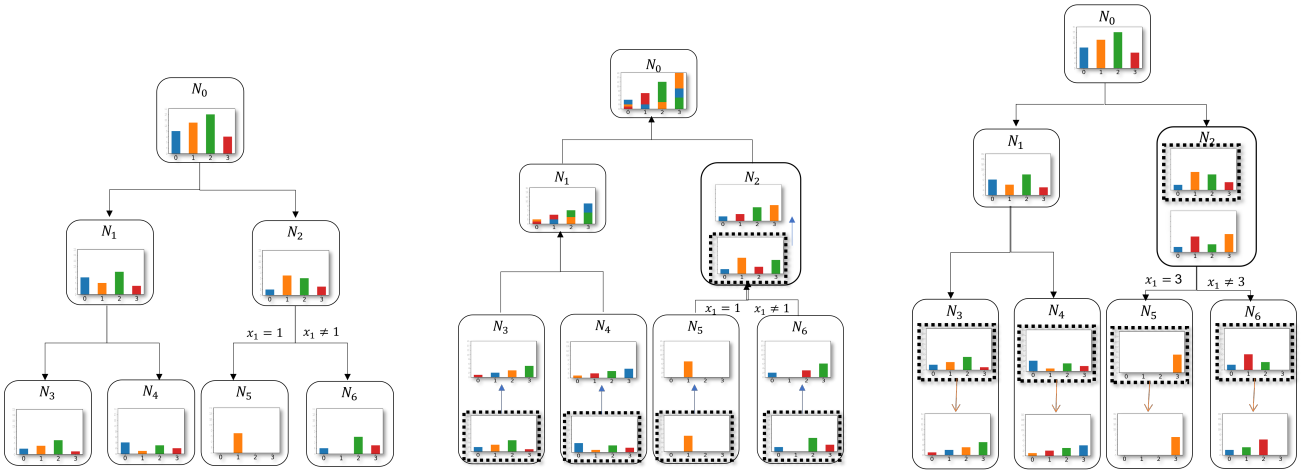


Figure 2: On the left, we demonstrate tree construction. The bar plots in each node indicate the counts of the categories (0,1,2 and 3) for the feature (x_1), we show just x_1's counts for simplicity, but the same idea applies to other features. The arrows are left blank when the data is split based on another feature that's not x_1. The different colors of the bars are to easily track the changes in counts when permutations are applied. In the middle, we demonstrate the operation of pass 1. The dashed lines around borders of the bar plots indicate that these are intermediate steps. The blue arrows indicate that we are learning and applying the local permutations for x_1 at the corresponding node. On the right, we demonstrate the resulting tree after both passes. The orange arrows indicate the new node permutations. Also, note how the split feature is now based on category 3 instead of 1 (i.e., we also updated v). Another more detailed example is presented in Appendix C.

Lemma 3. *The local permutations learned in LearnLocalPermutations ensure that the equivalent TSP will be rank consistent.*

Theorem 3 (Algorithm finds optimal permutations)**.** *Given a tree structure $\mathcal{T}$, our permutation learning algorithm finds the optimal permutations for the nodes in terms of negative log likelihood.*

This result is particularly notable since the LearnLocalPermutations and ConstructEquivalentTree can be computed in closed-form using only sorting operations on count matrices and manipulations of independent permutations. Thus, the overall computational complexity is linear (up to log factors) in all relevant parameters.

4. Experiments

To test our approach, we compare different results including the negative log likelihood (NLL), training time (TT), and the number of model parameters (NP), against models for discrete flows that are suitable for categorical data and that allow for discrete latent spaces to be achieved by training by optimizing the exact likelihood. The models that fit these criteria are the two models that were introduced in (Tran et al., 2019) and implemented in (Bricken, 2021) (the Autoregressive Flow with Independent Base distribution (AF), and Bipartite Flow with independent base distribution (BF)) and the Discrete Denoising flows (DDF) that were introduced in (Lindt & Hoogeboom, 2021).

Our model, Discrete Tree Flows (DTF), utilizes an independent base distribution, and has three variations depending on the splitting criteria that we use. DTF$_{GLP}$, DTF$_{RND}$ is when our models uses the greedy local permutation, and random splitting critera respectively. To calculate the number of parameters in our DTF model, we counted the number of permutations that are not the identity in each node and also added a contribution of two from each node to account for the parameters of the split feature and split value that are encoded in the node. All the timing results we report are from running our experiments on an Intel®Core™ i9-10920X 3.50GHz for CPU, and Nvidia GeForce RTX™ 3090 for GPU. Note that we had to modify the implementation of the BF model in (Bricken, 2021) as it had some bugs, the details of those modifications are presented in Appendix J.1.

4.1. Synthetic data

We carry out a set of experiments on synthetic data and report the results for comparing our DTF$_{GLP}$ model with AF, BF and DDF in Table 2. A full table that includes DTF$_{RND}$ is presented in Appendix J.4. In this table we report the results for the best performing model among those we have tried (We list all models we tried in Appendix J.3 and the best models that we report the results for in Appendix J.5). The training times for the different models vary depending on the model's hyperparamteres, the size of the data $n \times d$ and the number of categories k.

Table 2: Average of NLL and Training Time on CPU (TTC), and GPU (TTG) across 5 folds (±std) for synthetic datasets, the full table is available in Appendix J.4

	AF	BF	DDF	DTF$_{GLP}$
8GAUSSIAN				
NLL	6.92 (± 0.06)	7.21 (± 0.09)	**6.42 (± 0.03)**	6.5 (± 0.03)
TTC	155.9 (± 2.2)	231.6 (± 5.2)	119.8 (± 0.8)	**7.3 (± 0.1)**
TTG	42.2 (± 3.5)	135.8 (± 0.2)	79.7 (± 0.8)	NA
COP-H				
NLL	1.53 (± 0.02)	1.47 (± 0.06)	1.46 (± 0.1)	**1.33 (± 0.02)**
TTC	10.7 (± 0.2)	13.2 (± 0.2)	58.1 (± 1.0)	**≤0.1 (± 0.0)**
TTG	14.6 (± 0.4)	20.9 (±0.3)	48.8 (±0.9)	NA
COP-M				
NLL	1.76 (± 0.1)	1.62 (± 0.05)	1.51 (± 0.16)	**1.4 (± 0.02)**
TTC	10.6 (± 0.02)	13.3 (± 0.06)	77.9 (± 1.8)	**≤0.1(± 0.0)**
TTG	14.6 (± 0.43)	20.8 (± 0.8)	66.9 (±0.5)	NA
COP-W				
NLL	2.42 (± 0.02)	2.35 (± 0.03)	2.29 (± 0.07)	**2.22 (± 0.02)**
TTC	10.5 (± 0.01)	13.2 (± 0.1)	77.3 (± 1.7)	**≤0.1 (± 0.0)**
TTG	13.9(± 0.2)	19.2 (± 0.2)	67.5(± 0.1)	NA

Discretized Gaussian mixture model (8 Gaussian) Following the experiments in the (Tran et al., 2019) paper that was also reproduced in (Lindt & Hoogeboom, 2021), we generate data from a mixture of 8 Gaussian distributions and then *discretize* each feature into 91 categorical bins so that $d = 2$ and $k = 91$ dataset. We choose to draw 12,800 samples in total, reserving 10,240 samples for training and 2,560 for testing. Visualizations of samples trained on this discretized Gaussian mixture model dataset can be found in Appendix J.7.

Gaussian Copula Synthetic Data We also created synthetic data with pairwise dependencies using a Gaussian copula model with discrete marginals (as a reference for copula models see (Nelsen, 2007)). The generating process (summarized in Appendix J.2) is used to generate 3 datasets with varying degrees of dependency among the features. For all distributions, we set $d = 4$ and sampled $n = 10000$ and used an inverse CDF of a Bernoulli distribution to generate binary data with $k = 2$. The first dataset has a very high correlation (COP-H) where the feature are very dependant, the second with moderate correlations (COP-M), the third with weak correlations (COP-W).

Synthetic Data Experimental Results: As can be observed from Table 2, Our method with the GLP splitting criteria outperforms all other methods in terms of NLL and training time (even when comparing training times on GPU for the other models), with the exception of the 8 Gaussian dataset where our model gives very comparable result to the best model (DDF) but at a fraction of the training time.

4.2. Real Data

We follow the same settings used in the synthetic data experiments, and proceed to investigate the performance of our model on real datasets including some that are high dimensional (MNIST and Genetic).

Mushroom Dataset This dataset includes different attributes of mushrooms[1],and has $n = 8,124$, $d = 22$ and the maximum number of categories in any column is $k = 12$.

Vectorized binary MNIST Dataset We then investigate the performance of our algorithm for the the vectorized and binarized MNIST dataset (Deng, 2012), where "vectorized" means that we do not leverage the image structure of the dataset but merely treat the data as a 784-dimensional vector.

Genetic Dataset Given that our algorithm is a general purpose discrete flow algorithm, we also investigate the performance of DTF on a realistic high-dimensional discrete genetic dataset to emphasize the generality of our method. Specifically, we use a dataset of $n = 2504$ individuals with $d = 805$ sampled single nucleotide polymorphism (SNPs) that was made available by (Yelmen et al., 2021). Where the SNPs were encoded as binary data ($k = 2$).

Real Data Results We present results for the real-world datasets averaged across 3 folds for MNIST and Genetic data and across 5 folds for the mushroom data in Table 3. We notice some interesting results, our DTF$_{RND}$ model always have the fastest training time, and its NLL results are either very comparable to the other model and sometimes even gives better results than the other models (it always exceeds the performance of AF and BF but sometimes falls behind DDF). This suggests that even with a very simple splitting criteria such as random splitting, our two-pass algorithm is efficient enough to learn interesting aspects about the data. Our DTF$_{GLP}$ usually gives the best NLL results (with the exception of MNIST), but tends to take more training time, as finding the best splits dominate the training time in this case. This was not an issue with lower dimensional datasets (like the synthetic data), but becomes more prominent when the dimension of the data increases (as in MNIST and Genetic), especially when comparing our model's CPU training times to the other model's GPU training times. The only case where our model doesn't seem in par with the best NLL result is for MNIST. While our model gives better results than AF and BF, it fails to out-preform DDF. This suggests that our model is better suited for tabular data, unlike MNIST which presume a more dimension-by-dimension dependent vectorized dataset. Our models also has a smaller number of parameters, and they're not a fixed

[1] https://archive.ics.uci.edu/ml/datasets/Mushroom

Table 3: Experiment results for real datasets. The lower the number the better. TTC, and TTG are the training times in seconds on CPU and GPU respectively.

	AF	BF	DDF	DTF$_{GLP}$	DTF$_{RND}$
Mushroom Dataset					
NLL	24.87 (± 2.28)	23.02 (± 2.3)	19.18 (± 3.48)	**14.15 (± 2.44)**	16.66 (± 2.98)
TTC	29.3 (± 2.0)	20.9 (± 2.7)	175.8 (± 1.9)	9.9 (± 0.2)	**0.5 (± 0.0)**
TTG	7.7 (± 1.0)	**6.0 (± 1.3)**	75.2 (± 1.0)	NA	NA
Number of Parameters	337952	522720	3290452	7604 (± 578)	13544 (± 2352)
MNIST Dataset					
NLL	206.014 (± 0.32)	205.94 (± 0.26)	**144.78 (± 10.52)**	177.75 (± 0.56)	187.44 (± 1.17)
TTC	12104.6 (± 359.2)	3290.5 (± 13.3)	2909.3 (± 45.4)	5213.7 (± 204.9)	**105.6 (± 0.1)**
TTG	**305.6 (± 31.4)**	308.7 (± 4.1)	334.3 (± 8.7)	NA	NA
Number of Parameters	44283456	42591578	2679408	89583 (± 2846)	19549 (± 2061)
Genetic Dataset					
NLL	490.55 (± 0.69)	471.54 (± 1.87)	446.86 (± 8.64)	**437.19 (± 1.02)**	470.9 (± 6.1)
TTC	834.0(± 2.1)	251.6 (± 0.5)	209.4 (± 0.6)	411.5 (± 2.3)	**5.9 (± 0.0)**
TTG	**23.8 (± 0.9)**	38.0 (± 5.4)	29.5 (±1.1)	NA	NA
Number of Parameters	46686780	4484964	1205630	9014 (± 454)	14174 (± 568)

set of parameters as with the other models, our parameters can grow depending on the data and what permutations are deemed necessary, which makes it more flexible than a typical flow model.

5. Discussion and Conclusion

We presented a novel framework for discrete normalizing flows called DTF that relies on tree-structured permutations (TSPs), which we define and develop. We prove that our learning algorithm finds the optimal node permutations given a decision tree structure. Empirically, our model results demonstrate that DTF outperforms prior approaches in terms of NLL while being substantially faster for most experiments.

We note that our model does have some limitations. First, while we do guarantee permutations optimality given the tree structure, the tree structure itself is not guaranteed to be optimal since the tree is grown greedily as with most decision tree algorithms. Moreover, our splitting is required to be axis aligned in our current framework and thus cannot perform complex split operations. We believe this could be addressed by constructing coupling-like TSPs, similar to coupling layers in continuous normalizing flows. Specifically, the split functions could be arbitrarily complex functions of half of the features while the node permutations only permute the other half of the features. This would allow deep models to be used for the split functions. This idea of coupling-like TSP can also address another issue of handling very high dimensional data (e.g., $d > 1000$) because our non random split algorithms are naively $O(d^2)$. This can also be handled in our current approach by drawing from techniques used in decision tree algorithms. Second, similar to other methods, large values of k may be challenging for the split criteria. For random splitting, this should be straightforward by selecting a random set of categorical values to go to the left (possibly based on counts for each category). For GLP, we implemented the simplest case of choosing only one value to go left. However, we could greedily select the next best feature to go to the left but this would increase the computational complexity by the max number of values that go to the left. Third, we have focused on tabular categorical discrete data (even MNIST is vectorized and treated as a 784-dimensional vector). Extending to discrete image data or text-based data is non-trivial. Prior gradient-based works such as AF and DDF were able to leverage CNNs or NLP-based transformer architectures. Thus, we do not expect our current tabular-focused DTF approach to be competitive on these tasks. As we mentioned earlier, it may be possible to extend our framework to use arbitrary NNs for the split function if we partition the features into fixed and free sets (as is done in standard flow coupling layers). Or, for NLP-based applications, our framework may be extended to have an autoregressive-like structure to our TSPs. Ultimately, we hope that our paper lays the groundwork for developing practical and effective discrete flows using decision tree algorithms.

Acknowledgements

All authors acknowledge support from the Army Research Lab through contract number W911NF-2020-221.

References

Bengio, Y., Léonard, N., and Courville, A. Estimating or propagating gradients through stochastic neurons for conditional computation, 2013.

Bricken, T. TrentBrick/PyTorchDiscreteFlows, 2021. URL https://github.com/TrentBrick/PyTorchDiscreteFlows. original-date: 2020-01-31T03:54:04Z.

Chen, T. and Guestrin, C. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pp. 785–794, 2016.

Conrad, K. Generating sets, 2018. https://kconrad.math.uconn.edu/blurbs/grouptheory/genset.pdf.

Deng, L. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

Dinh, L., Sohl-Dickstein, J., and Bengio, S. Density estimation using real nvp, 2017.

Germain, M., Gregor, K., Murray, I., and Larochelle, H. Made: Masked autoencoder for distribution estimation, 2015.

Ghojogh, B., Ghojogh, A., Crowley, M., and Karray, F. Fitting a mixture distribution to data: Tutorial, 2020.

Hoogeboom, E., Peters, J. W. T., van den Berg, R., and Welling, M. Integer discrete flows and lossless compression, 2019.

Hoogeboom, E., Nielsen, D., Jaini, P., Forré, P., and Welling, M. Argmax flows and multinomial diffusion: Learning categorical distributions, 2021.

Inouye, D. I., Ravikumar, P., and Dhillon, I. S. Admixture of poisson mrfs: A topic model with word dependencies. In *International Conference on Machine Learning (ICML)*, jun 2014.

Inouye, D. I., Yang, E., Allen, G. I., and Ravikumar, P. A review of multivariate distributions for count data derived from the poisson distribution. *Wiley Interdisciplinary Reviews: Computational Statistics*, 9(3):e1398, 2017.

Kingma, D. P., Salimans, T., Jozefowicz, R., Chen, X., Sutskever, I., and Welling, M. Improved variational inference with inverse autoregressive flow. In Lee, D., Sugiyama, M., Luxburg, U., Guyon, I., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016. URL https://proceedings.neurips.cc/paper/2016/file/ddeebdeefdb7e7e7a697e1c3e3d8ef54-Paper.pdf.

Lindt, A. and Hoogeboom, E. Discrete denoising flows. In *ICML Workshop on Invertible Neural Networks, Normalizing Flows, and Explicit Likelihood Models (INNF+)*, 2021.

Lippe, P. and Gavves, E. Categorical normalizing flows via continuous transformations. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=-GLNZeVDuik.

Nelsen, R. B. *An introduction to copulas*. Springer Science & Business Media, 2007.

Papamakarios, G., Pavlakou, T., and Murray, I. Masked autoregressive flow for density estimation, 2018.

Razavi, A., van den Oord, A., and Vinyals, O. Generating diverse high-fidelity images with vq-vae-2, 2019.

Rezende, D. J. and Mohamed, S. Variational inference with normalizing flows, 2016.

Tran, D., Vafa, K., Agrawal, K. K., Dinh, L., and Poole, B. Discrete flows: Invertible generative models of discrete data, 2019.

van den Berg, R., Gritsenko, A. A., Dehghani, M., Sønderby, C. K., and Salimans, T. Idf++: Analyzing and improving integer discrete flows for lossless compression, 2020.

van den Oord, A., Vinyals, O., and Kavukcuoglu, K. Neural discrete representation learning, 2018.

Wainwright, M. J. and Jordan, M. I. Graphical models, exponential families, and variational inference. *Foundations and Trends® in Machine Learning*, 1(1-2):1–305, 2008.

Yelmen, B., Decelle, A., Ongaro, L., Marnetto, D., Tallec, C., Montinaro, F., Furtlehner, C., Pagani, L., and Jay, F. Creating artificial human genomes using generative neural networks. *PLOS Genetics*, 17(2):1–22, 02 2021. doi: 10.1371/journal.pgen.1009303. URL https://doi.org/10.1371/journal.pgen.1009303.

Ziegler, Z. M. and Rush, A. M. Latent normalizing flows for discrete sequences, 2019.

A. Overview

We have organized our appendix as follows:

- **Appendix B** provides a visual explanation of the GLP splitting criteria.
- **Appendix C** provides a visual example of our two pass algorithm.
- **Appendix D** describes the expressivity of DTF models by proving they are universal permutation models.
- **Appendix E** ultimately proves invertibility of TSPs by including a proof of Lemma 1, proof of Theorem 1, and a necessary and sufficient condition for invertibility with its proof.
- **Appendix F** proves the optimality of rank consistent of TSPs with the proof of Theorem 2.
- **Appendix G** proves that Definition 5 is an equivalence relation.
- **Appendix H** provides pseudo code for all algorithms.
- **Appendix I** includes proofs of the algorithms by proving Theorem 3 with multiple lemmas and definitions.
- **Appendix J** provides more experimental details and results (i.e. modification and details of codes and architecture, extra details of dataset configuration, and more table results and figures).

We give a reference table of the notations used throughout the paper and appendix below on Table 4:

Table 4: Notation

indices:	
$\boldsymbol{x}$	input data
d	number of dimension/features
n	number of samples
k	number of categories
s	split feature
j	denotes feature that usually excludes the split feature (e.g. $j \neq s$)
v	split value
$\mathcal{T}$	tree
N	node
π	independent permutation usually on a specific node
$\sigma_{\mathcal{T}}$	a tree-structured permutation based on tree $\mathcal{T}$
$\mathcal{P}$	tree traversal path of a specific input $\boldsymbol{x}$
$\mathcal{D}$	categorical domain (e.g. $\mathcal{D}(N)$ would denote all categorical domain of the specific node)
$Q_{\boldsymbol{z}}$	independent base distribution
Σ	an exhaustive set of equivalent trees (e.g. $\Sigma(\mathcal{T})$ would be set of all TSPs that are equivalent to $\mathcal{T}$)
$c(j, a)$	count value of feature j for categorical value a usually for a specific node (can be represented by a matrix)
$\mathcal{R}$	denotes rank consistency for count matrices w.r.t. to a node domain (e.g. $\mathcal{R}(\mathcal{D}(N))$)
subscript:	
$[\cdot]$	the level on the tree usually a subscript for N (e.g. $N_{[n]}$ would represent the nodes on level n of the tree)
$(\cdot)$	the max depth of tree usually a subscript for $\mathcal{T}$ (e.g. $\mathcal{T}_{(n)}$ would represent a tree with max depth n)
left	represents the left child (e.g. N_{left} would denote the left child node of that specific node)
right	represents the right child (e.g. N_{right} would denote the right child node of that specific node)
anc	denotes all the ancestral nodes on the current node and is usually a subscript for π (see Eqn. 49, Eqn. 50, Eqn. 58)
leaf	the leaf nodes (e.g. $\mathcal{T}_{\text{leaf}}$ would be the leaf nodes of this specific tree)
superscript:	
new	represents the new tree constructed after Alg 4 (e.g. π^{new} represents new tree permutation and c^{new} represents new counts)
init	represents unsorted local counts only used with the local counts notation (e.g. c^{init})
others:	
$\pi[\cdot]$	applies permutation to the values of vectors (i.e. $\pi[c] \triangleq [\pi_0[c(0)], \pi_1[c(1)], \cdots, \pi_d[c(d)]]$)
$\tilde{\cdot}$	represents "local" auxiliary objects (e.g. $\tilde{\pi}$ represents local permutation and $\tilde{c}$ represents local counts at each node computed in Alg 1 and 3)
$\equiv_{\text{tr}}$	represents equivalence relation between two sets in this case we use it for trees (e.g. $\sigma^A_{\mathcal{T}} \equiv_{\text{tr}} \sigma^B_{\mathcal{T}}$)

B. A visual explanation of the GLP splitting criteria

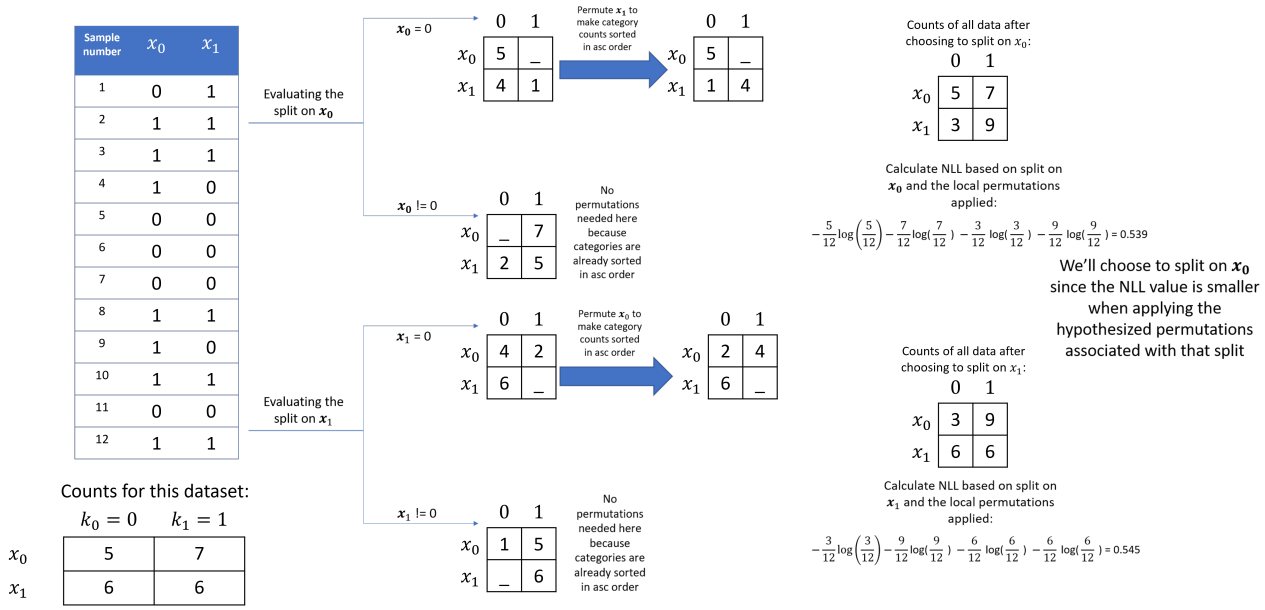


Figure 3: An example for how the GLP criteria is evaluated.

C. A visual example of the two pass algorithm

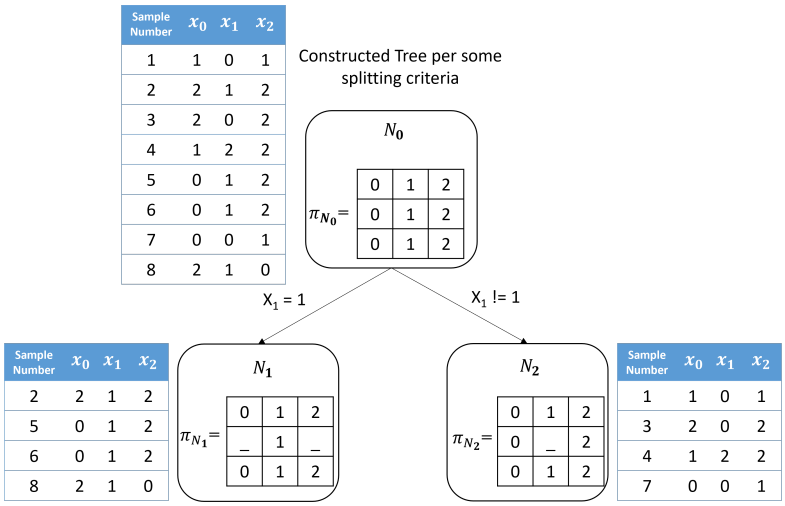


Figure 4: An example of the tree constructed using some splitting criteria of choice, the permutation matrices are just the identity since we only learn the split information here.

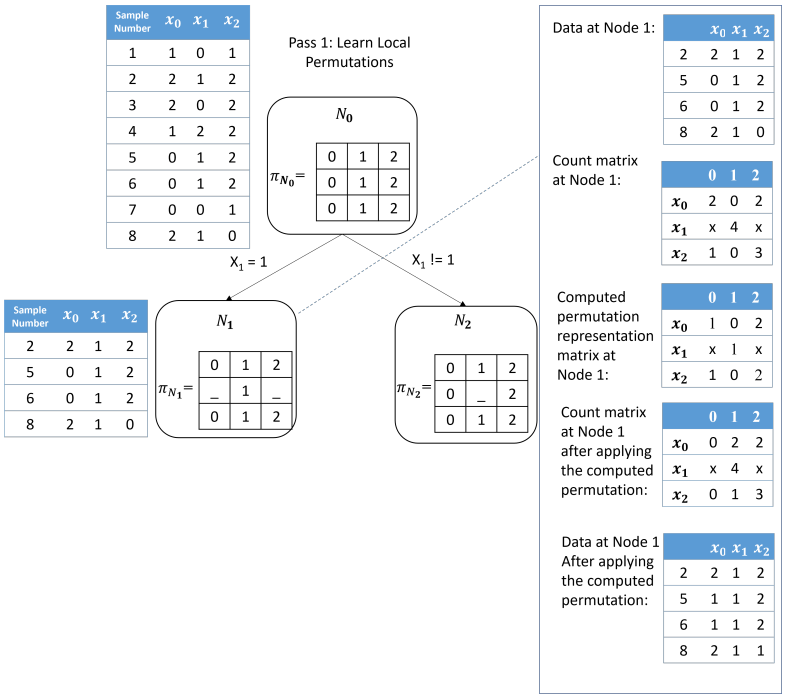


Figure 5: Following tree construction, we start pass 1 at the bottom of the tree, the figure shows the steps done at node 1 in full details until the permutation matrix is calculated.

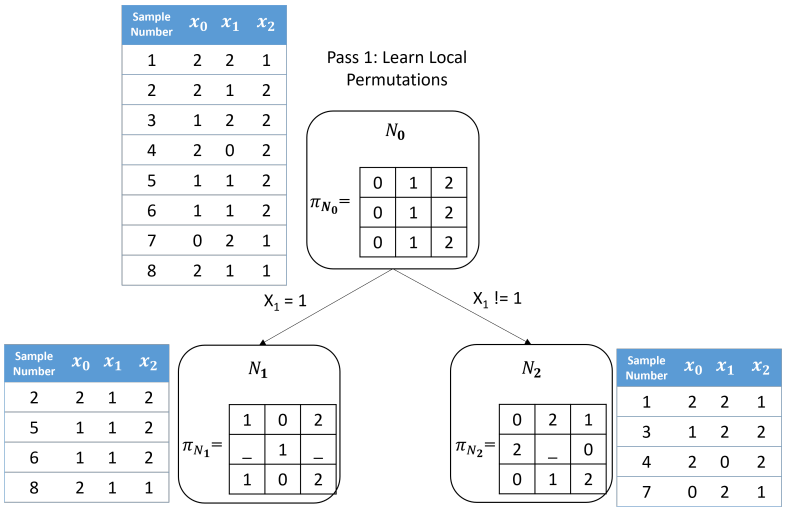


Figure 6: The same logic is applied to Node 2, and we propagate the new permuted data up to node 0.

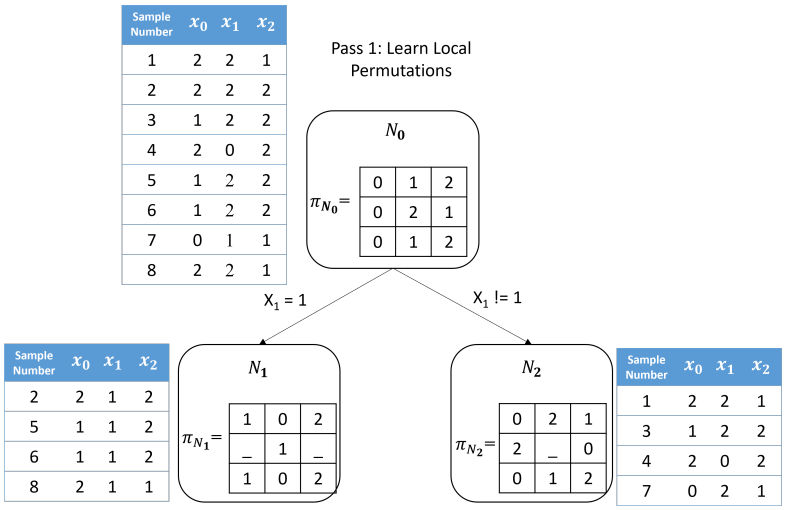


Figure 7: At node 0, we use the same logic to compute the permutation matrix at that node – notice that only the split feature can have a permutation since the others will be already in order.

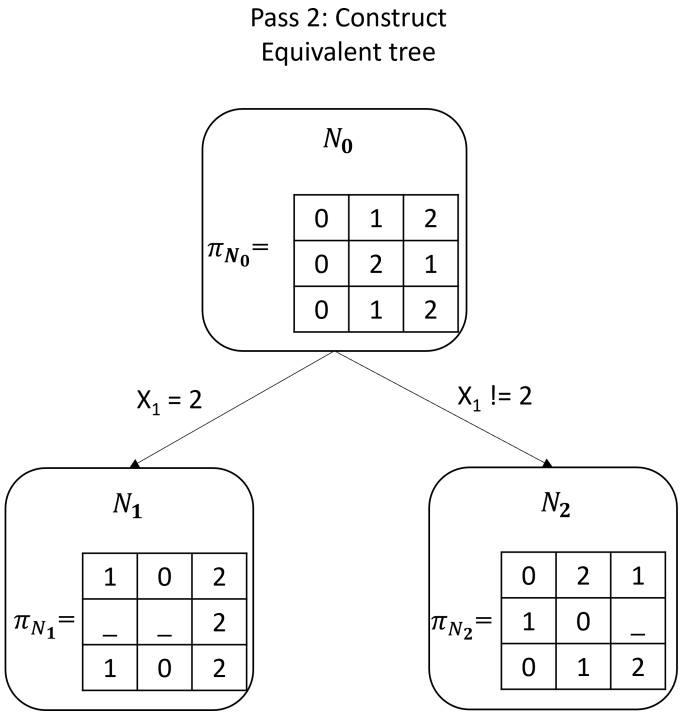


Figure 8: In pass 2, we traverse the tree from top to bottom to fix it, "fixing it" includes applying the ancestor's node permutations to the split values, and to the permutation matrices as well of the children nodes.

D. Proof of Expressivity of DTF

We will prove that our DTF (i.e. a sequential combination of TSPs) can theoretically produce a universal permutation (i.e. an n element permutation is universal if it contains all permutations of length n) on its domain. To prove this universality we must demonstrate that our DTF is a generating set of the symmetric group S_n where S_n is an exhaustive collection of permutation subsets of an n elements set. n is the total number of elements in our sample space (i.e. $n = 144$ for a 2 feature 12 category domain, and S_n would have a total of $n!$ subsets).

Our proof will rely on the theorem below:

Theorem 4 (from (Conrad, 2018))**.** *For $n \geq 2$, S_n is generated by the $n-1$ transpositions/bijective permutations in this cyclic notation form $(1\ 2), (2\ 3), ..., (n-1\ n)$.*

Definition 6. *Given a configuration $\boldsymbol{x}$ and a new value for the j-th feature denoted $\tilde{x}_j \neq x_j$, a* single feature swap *is a permutation that permutes $\boldsymbol{x} = x_1 x_2 \cdots x_d$ and $\tilde{\boldsymbol{x}} = x_1 x_2 \cdots \tilde{x}_j \cdots x_d$, $\forall j\ \exists x_j \in \mathcal{D}(N)$, while keeping all other configurations the same, which could be denoted in cyclic notation as $(\boldsymbol{x}, \tilde{\boldsymbol{x}})$.*

We will show that a combination of DTFs can express each adjacent transposition corresponding to Theorem 4. Let us first illustrate that a single TSP can implement a single feature swap (e.g. $(15342_7\ 15344_7)$ where the element has a base of 7 and only a single category on the fifth feature permutes $2 \leftrightarrow 4$ and the other features and categories do not change).

Suppose there is a TSP with a max depth equal to d where d is the total number of features. Let us focus on the left most non-leaf nodes on each level of the tree where each left node fixes a single category on the non-permuting feature of our desired permuting configuration and let all node permutations by default be an identity. Next, if we store the permuting categories on the second to the leftmost leaf node, we can apply the desired permutation on this leaf node. Thus, obtaining a single feature swap. An example of this can be seen below.

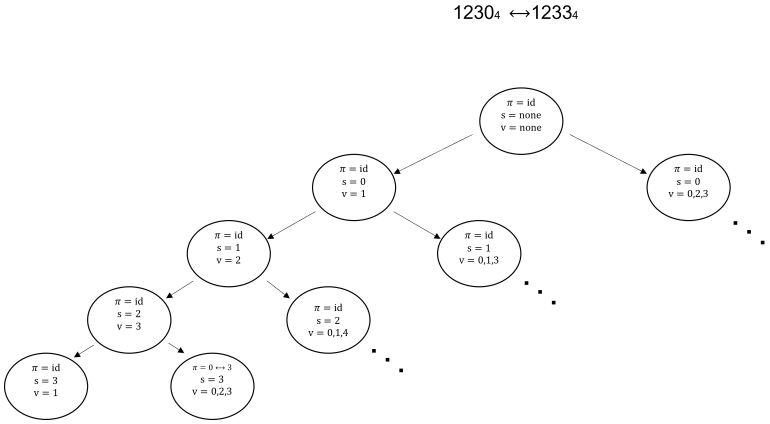


Figure 9: An example of a single feature swap.

As seen in Fig. 9, $1230_4 \leftrightarrow 1233_4$ has 4 features, therefore we have a TSP of max depth 5. The non-permuting features are the 0, 1, 2 features, so we fix the categorical values of these features on the leftmost none-leaf nodes. We want to permute on feature 3, thus we store these categorical on the 2nd leaf node and apply our single feature swap on this node (i.e. $0 \leftrightarrow 3$).

Above, we demonstrated that we can permute any pair of categories on a single feature and restrict the others. We will use this condition to show that a sequential combination of single feature swaps can traverse the whole permutation space by showing that there is a snake-like path that can fill the permutation space.

Lemma 4. *There exists a snake-like path that reaches all discrete configurations where adjacent configurations on the path differ in only one feature value.*

Proof. We will prove Lemma 4 by induction. We will prove that a n-dimensional discrete space can be reduced to have the same cardinality as a single dimensional discrete space (i.e. $\mathbb{Z}_+^n \to \mathbb{Z}$). Thus, we show that there is a full adjacent single feature varying path that can traverse the whole n-dimensional space.

Base case $(n = 1)$ Let the space have a base number of $b + 1$ where $b \geq 0$. Obviously, We can use a traversal path of $0 \leftrightarrow 1 \leftrightarrow 2 \leftrightarrow \cdots \leftrightarrow b$ for the single dimension, thus proving there is a 1-D discrete path.

Induction step$(n = k \Rightarrow n = k + 1)$ Let us assume for up to a k-dimensional permutation space with a base number of $b + 1$, the k-dimensional discrete space can be traversed with a 1-D discrete path. Now, let us look at the $k + 1$ dimensional discrete space. We know that this traversal path is valid $0\underbrace{0\cdots0}_{k} \leftrightarrow \cdots \leftrightarrow 0\underbrace{b\cdots b}_{k}$ with our assumption that the k-dimensional space can be traversed with a 1-D discrete path. Rather than traversing the path $0\underbrace{b\cdots b}_{k} \leftrightarrow \cdots \leftrightarrow 1\underbrace{0\cdots0}_{k}$, we traverse the path $0\underbrace{b\cdots b}_{k} \leftrightarrow 1\underbrace{b\cdots b}_{k}$. Again, we know that we can traverse the path $1\underbrace{b\cdots b}_{k} \leftrightarrow \cdots \leftrightarrow 1\underbrace{0\cdots0}_{k}$, and thus we can traverse the path $1\underbrace{0\cdots0}_{k} \leftrightarrow 2\underbrace{0\cdots0}_{k}$. A repetition of this traversal process shows us that the $k + 1$ discrete permutation space has a snake-like 1-D path traversal (i.e. $\underbrace{0\cdots0}_{k+1} \leftrightarrow \cdots \leftrightarrow \underbrace{b\cdots b}_{k+1}$). Note, this is just one path of many that can be taken to traverse the space.

Therefore, there exists a snake-like path that traverses the whole discrete permutation space. □

Now by using the snake-like path from Lemma 4 we can put all the configurations into an 1D sequence of categorical values (which has k^d unique values). And from above, we know that a single TSP can swap any two of these adjacent categorical values. Thus, these TSPs that swap any two adjacent values is a generating set for all possible permutations of the k^d configurations by Theorem 4. Therefore, a composition of these TSPs can express any possible permutation (albeit possibly an exponentially large number of TSPs) and thus our TSPs with independent node permutations do not hinder the expressivity of DTFs.

E. Proofs for invertibility of TSPs

As a helpful idea, we first define the range of a node to simplify the proofs in certain cases.

Definition 7 (Range of a Node)**.** *We define the* range of a node*, denoted $\mathcal{R}(N)$, as the image of $\mathcal{D}(N)$ under π_N, i.e., $\mathcal{R}(N) \triangleq \{\pi_N(\boldsymbol{x}) : \boldsymbol{x} \in \mathcal{D}(N)\}$.*

E.1. Proof of Lemma 1

Lemma 1. *If the invertibility constraint is satisfied, the TSP tree traversal path for any input can be recovered from the output.*

Proof. First, note that because π is a permutation (i.e., one-to-one mapping) and $\pi_N(x) = x, \forall x \notin \mathcal{D}(N)$ (ie. configurations outside the domain will remain unchanged), then $\forall \boldsymbol{x} \in \mathcal{D}(N), \pi_N(x) \in \mathcal{D}(N)$, i.e., all node permutations π_N do not permute configurations from in the domain to outside the domain.

Without loss of generality, we consider trees where all leaf nodes are at the max depth of M.

We will denote the tree traversal path of an input $\boldsymbol{x}$ up to tree level i to be $\mathcal{P}_{(i)}(\boldsymbol{x}) \triangleq (N_{[0],\boldsymbol{x}}, N_{[1],\boldsymbol{x}}, \cdots, N_{[i],\boldsymbol{x}})$, where $N_{[i]}$ is the tree node that $\boldsymbol{x}$ reaches at i-th level of the tree , and where $N_{[0]}$ is the root node. We will usually suppress the dependence on $\boldsymbol{x}$ if this is clear from the context and merely write $\mathcal{P}_{(i)}(\boldsymbol{x}) \triangleq (N_{[0]}, N_{[1]}, \cdots, N_{[i]})$.

Let $\boldsymbol{y} \triangleq \sigma_{\mathcal{T}_{(i)}}(\boldsymbol{x})$ denote the output of our TSP forward evaluation.

For all $\boldsymbol{x} \in \mathcal{Z}^d$, let $\boldsymbol{y} \triangleq \sigma_{\mathcal{T}_{(M)}}(\boldsymbol{x}) \equiv \pi_{N_{[M]}} \circ \cdots \circ \pi_{N_{[1]}} \circ \pi_{N_{[0]}}(\boldsymbol{x})$ denote the output of our TSP forward evaluation. We want to prove that $\mathcal{P}_{(M)}(\boldsymbol{x})$ can be recovered from $\boldsymbol{y}$ and $\sigma_{\mathcal{T}}$. Let $\mathcal{P}'_{(i)}(\boldsymbol{y}) \triangleq N'_{[0]}, N'_{[1]}, \cdots, N'_{[i]}$ where we traverse the decision tree of $\mathcal{T}$ *without* applying node permutations If we can prove $\mathcal{P}'_{(M)}(\boldsymbol{y}) = \mathcal{P}_{(M)}(\boldsymbol{x})$ then we are done.

Inductive hypothesis For all $i \in \{0, 1, \cdots, M\}$, $\mathcal{P}'_{(i)}(\boldsymbol{y}) = \mathcal{P}_{(i)}(\boldsymbol{x})$.

Base case ($i = 0$) Since both $\boldsymbol{y}$ and $\boldsymbol{x}$ start at the root node, then the path up to $i = 0$ is the same.

Induction step We need to prove that if $\mathcal{P}'_{(i)}(\boldsymbol{y}) = \mathcal{P}_{(i)}(\boldsymbol{x})$, then $\mathcal{P}'_{(i+1)}(\boldsymbol{y}) = \mathcal{P}_{(i+1)}(\boldsymbol{x})$.

Proof From assumption of inductive hypothesis, we know that $N'_{[i]} = N_{[i]}$ (i.e., $N_{[i]}$ and the corresponding $N'_{[i]}$ are the same node for level i).

Let $\boldsymbol{x}^{(i)} = \sigma_{\mathcal{T}(i)}(\boldsymbol{x}) = \pi_{N_{[i]}} \circ \cdots \circ \pi_{N_{[1]}} \circ \pi_{N_{[0]}}(\boldsymbol{x})$ where $\boldsymbol{x}^{(M)} \equiv \boldsymbol{y}$

Note that only the value v of the split feature s for both $\boldsymbol{x}^{(i)}$ and $\boldsymbol{y}$ is relevant for determining whether to go left or right.

We prove that the chosen nodes are the same using *contradiction*.

Suppose $x_s^{(i)} = v$ such that $\boldsymbol{x}$ goes left and suppose $y_s \neq v$ such that $\boldsymbol{y}$ would go right.

We know that the s-th part of the domain of the left child has only v in it, i.e., $\mathcal{D}_s(N_{\text{left}}) = \{v\}$. Also, we know that domains of children are always smaller disjoint subsets of the parent, i.e., $\mathcal{D}_s(N_{[l]}) \subseteq \mathcal{D}_s(N_{\text{left}})$ for all $l > i$.

Thus, $x_s^{(l)} = v$ for all $l > i$ because the invertibility constraint ensures that we cannot permute a value inside the domain to a value outside the domain. However, this is a contradiction to our assumption that $x_s^{(M)} \equiv y_s \neq v$. Therefore, if $\boldsymbol{x}$ goes left, then $\boldsymbol{y}$ will also go left.

In a similar way, now suppose $x_s^{(i)} \neq v$ such that $\boldsymbol{x}$ goes right and suppose $y_s = v$ such that $\boldsymbol{y}$ would go left. The s-th part of the domain of the right child has $\mathcal{D}_s(N_{\text{right}}) = \{a : a \neq v, a \in \mathcal{D}_s(N)\}$. Again, $\mathcal{D}_s(N_{[l]}) \subseteq \mathcal{D}_s(N_{\text{right}})$ for all $l > i$ because every child is a subset of the parent domain.

Therefore, $\boldsymbol{x}_s^{(l)} \in \mathcal{D}_s(N_{\text{right}})$ for all $l > i$, and thus in particular $x_s^{(M)} \in \mathcal{D}_s(N_{\text{right}})$, where $x_s^{(M)} \equiv y_s$ by definition. However, this contradicts our assumption that $y_s = v$ (i.e., goes left) because $v \notin \mathcal{D}_s(N_{\text{right}})$.

Hence, if x goes right, y will also go right.

Combining these two we get that $N'_{[i+1]} = N_{[i+1]}$ (i.e., they will both go left or both go right), and thus we can recover the path for $i+1$ by adding the child node to the path for i, i.e., $\mathcal{P}'_{(i+1)}(\boldsymbol{y}) = \mathcal{P}_{(i+1)}(\boldsymbol{x})$. This proves our inductive step and concludes the proof of the lemma. □

E.2. Proof of Theorem 1

Proof. Lemma 1 (proven above) states that the TSP traversal path for any input can be recovered from the output. Thus, the output path is identical to the input path $\mathcal{P}'_{(i)}(\boldsymbol{y}) = \mathcal{P}_{(i)}(\boldsymbol{x})$. Therefore, $\boldsymbol{x} = \sigma_{\mathcal{T}}^{-1}(\boldsymbol{y}) \equiv \pi_{N'_{[0]}}^{-1} \circ \cdots \circ \pi_{N'_{[m-1]}}^{-1} \circ \pi_{N'_{[m]}}^{-1}(\boldsymbol{y})$ because each node permutation is itself invertible by the definition of a permutation. This can be seen as traversing the tree from the corresponding leaf node to the root node and applying the inverse node permutations along the path.

□

E.3. Necessary and Sufficient Condition for Invertibility with Proof

For better understanding of TSPs, we now present a condition (i.e., disjoint ranges of leaf nodes) that is both necessary and sufficient for invertibility. Note that this theorem can be easily used to prove Theorem 1 as a corollary because the original invertibility constraint set is a subset of the disjoint range of leaf nodes constraint. However, the proof here does not provide an efficient algorithm for determining the leaf node for the inverse, while the proof of Lemma 1 does provide an efficient algorithm (i.e., merely traverse the tree as described in the lemma proof to determine the path).

Theorem 5 (TSP Necessary and Sufficient Invertibility Constraint)**.** *A TSP is invertible if and only if the range of each leaf node is disjoint from all other leaf nodes, i.e.,* $\mathcal{R}(N) \cap \mathcal{R}(N') = \emptyset, \forall N, N' \in \mathcal{T}_{\text{leaf}}$ *such that* $N \neq N'$*, where* $\mathcal{T}_{\text{leaf}}$ *are the set of leaves in the TSP tree.*

Proof. **We use a constructive proof for the if direction (sufficiency).** Because each of the permutations themselves are invertible, the primary challenge is *showing that we can find the right path through the tree* (as there could be multiple paths if we don't consider domain related constraints)

If the disjoint range condition is satisfied, then each possible output can be mapped to one of the leaves, i.e. $N_{[M]}$ is the leaf node such that $\boldsymbol{y} \in \mathcal{R}(N)$. Given the leaf node, there is only one possible path through the decision tree back to the root.

Thus, the inverse can be computed by traversing from the leaf node to the root node and applying the inverse of each node's permutation.

To prove the only-if direction (necessity), we will use a proof by contradiction.

Suppose a TSP is invertible but the disjoint range condition is not satisfied, then $\mathcal{R}(N) \cap \mathcal{R}(N') \neq \emptyset$. Therefore, there exists an output $\boldsymbol{y}$ that is in the range of two leaf nodes, i.e., $\exists \boldsymbol{y}$ such that $\boldsymbol{y} \in \mathcal{R}(N)$ and $\boldsymbol{y} \in \mathcal{R}(N')$ where $N \neq N'$. Yet, each input traverses the TSP tree in a deterministic way and thus each unique input will always arrive at the same leaf node. Therefore, there must exist two distinct inputs $\boldsymbol{x} \neq \boldsymbol{x}'$ such that $\sigma_{\mathcal{T}}(\boldsymbol{x}) = \sigma_{\mathcal{T}}(\boldsymbol{x}') = \boldsymbol{y}$. This means that two distinct inputs map to the same output (i.e., not one-to-one mapping) and violates invertibility. However, this contradicts our assumption that the TSP is invertible. □

F. Proof of optimality of rank consistency

F.1. Proof of Theorem 2

Theorem 2 (Optimality of rank consistent TSPs)**.** *Given a TSP tree* $\mathcal{T}^*$*, rank consistent TSPs attain the optimal negative log-likelihood among TSPs that are tree equivalent, i.e., rank consistent TSPs are optimal solutions to the problem:*

$$\arg\min_{\sigma_{\mathcal{T}} \in \Sigma(\mathcal{T}^*)} \min_{Q_{\boldsymbol{z}}} -\frac{1}{n} \sum_{i=1}^{n} \log Q_{\boldsymbol{z}}(\sigma_{\mathcal{T}}(\boldsymbol{x}_i)) \,, \tag{6}$$

where $\Sigma(\mathcal{T})$ *is the set of TSPs whose trees are tree equivalent to* $\mathcal{T}^*$*, and* $Q_{\boldsymbol{z}}$ *is an independent distribution.*

Proof. The proof is by *contradiction*.

We will prove that we can construct another equivalent TSP from this TSP that will yield a better log likelihood which will lead to a contradiction of optimality. In particular, there exists a pair of counts that can be switched to be consistent with the global rank that will yield a better TSP.

Suppose a TSP is optimal among TSPs that are tree equivalent but rank consistency was not satisfied. Without loss of generality, we will assume the global rank permutations are the identity (i.e., $\pi_j = \pi_{\text{Id}}, \forall j$) so we can simplify notation. This would mean that there exists a rank inconsistent (RI) node, i.e., $\exists \tilde{N}, j \in \{0, \cdots, d-1\}, (a,b) \in \{(a,b) : a < b, a \in \mathcal{D}(\tilde{N}), b \in \mathcal{D}(\tilde{N})\}$ such that

$$c_{\tilde{N}}(j,a) > c_{\tilde{N}}(j,b)\,. \tag{7}$$

Let $\tilde{\pi}$ be the permutation that switches a and b of the j-th dimension from the above but leaves all other discrete values untouched. Now, we alter the current node's permutation and split values as such:

$$\pi_{\tilde{N}}^{\text{new}} \triangleq \tilde{\pi} \circ \pi_{\tilde{N}} \tag{8}$$

$$v_{\tilde{N}}^{\text{new}} \triangleq \{\tilde{\pi}(v) : v \in v_{\tilde{N}}\} \tag{9}$$

and we alter the permutations and split values at descendant nodes as follows, i.e., $\forall N \in \text{desc}(\tilde{N})$

$$\pi_{N}^{\text{new}} \triangleq \tilde{\pi} \circ \pi_{N} \circ \tilde{\pi}^{-1} \tag{10}$$

$$v_{N}^{\text{new}} \triangleq \{\tilde{\pi}(v) : v \in v_{N}\}\,. \tag{11}$$

First, we show that this modified TSP is tree equivalent by *induction* on the depth of the descendant subtree after the $\tilde{N}$ node, which is at depth m. Note that any nodes that are not $\tilde{N}$ or its descendants are unchanged and thus already satisfy the tree equivalence property.

The **base case** is then simply to determine if a configuration that would go left in the original TSP will go left in the new tree for the current $\tilde{N}$.

Let $\boldsymbol{x} \in \mathcal{D}(\tilde{N})$ go left in the original tree, i.e., $\pi_{\tilde{N},s}(\boldsymbol{x}) \in v_{\tilde{N}}$. From this we have that:

$$\pi_{\tilde{N},s}(\boldsymbol{x}) \in v_{\tilde{N}} \tag{12}$$

$$\Leftrightarrow \tilde{\pi} \circ \pi_{\tilde{N},s}(\boldsymbol{x}) \in v_{\tilde{N}}^{\text{new}} \tag{13}$$

$$\Leftrightarrow \pi_{\tilde{N},s}^{\text{new}}(\boldsymbol{x}) \in v_{\tilde{N}}^{\text{new}}\,, \tag{14}$$

where the second line is by the definition of $v_{\tilde{N}}^{\text{new}}$, and the third is by the definition of $\pi_{\tilde{N}}^{\text{new}} \equiv \tilde{\pi} \circ \pi_{\tilde{N}}$. We also note that for these leaf nodes, the domain and range are the same. Thus, we have that $\forall \boldsymbol{x}, \sigma_{\mathcal{T}(m+1)}(\boldsymbol{x}) \in \mathcal{D}(\tilde{N}_{\text{left}}) \Leftrightarrow \sigma_{\mathcal{T}(m+1)}^{\text{new}}(\boldsymbol{x}) \in \mathcal{D}(\tilde{N}_{\text{left}}^{\text{new}})$ where the size of the new domain is equivalent $|\mathcal{D}(N_{\text{left}})| = |\mathcal{D}(N_{\text{left}}^{\text{new}})|$ (and similarly for the right node), i.e., any configuration that went left in the original TSP will go left in the new TSP where the depth of the subtree of $\tilde{N}$ is only 1 (i.e., a single split).

For the **induction** step, suppose the induction hypothesis holds for up to depth $m+k$, we will prove that it holds for depth $m+k+1$. Let N be a node at depth k away from $\tilde{N}$ which itself is at depth m, and let $\sigma_{\mathcal{T}(m)}$ be the TSP permutation up to depth m and $\sigma_{\mathcal{T}(m+k)}$ is the TSP permutation up to depth $m+k$.

By the inductive hypothesis, we have that the domains of the nodes are equivalent:

$$\sigma_{\mathcal{T}(m+k)}(\boldsymbol{x}) \in \mathcal{D}(N) \tag{15}$$

$$\Leftrightarrow \sigma_{\mathcal{T}(m+k)}^{\text{new}}(\boldsymbol{x}) \in \mathcal{D}^{\text{new}}(N)\,. \tag{16}$$

First, let's show that $\sigma_{\mathcal{T}(m+k)}^{\text{new}} \equiv \tilde{\pi} \circ \sigma_{\mathcal{T}(m+k)}$ as follows. We will denote the tree traversal path of an input $\boldsymbol{x}$ to be $\mathcal{P}(\boldsymbol{x}) \triangleq (N_{[0]}, N_{[1]}, \cdots, N_{[M]})$, where $N_{[M]}$ is the leaf node at max depth M that the input reaches and $N_{[0]}$ is the root node. Also, note that $\tilde{N} \equiv N_{[m]}$.

We define $\pi_{\text{anc}(N_{[m]})}(x) \triangleq \sigma_{\mathcal{T}(m-1)}(\boldsymbol{x})$ (i.e. $\pi_{\text{anc}(N_{[m]})}(x)$ is the evaluation of permutations of all ancestral nodes of $N_{[m]}$) to add intuition to the proofs.

$$\sigma_{\mathcal{T}(m+k)}(\boldsymbol{x}) \equiv \pi_{N_{[m+k]}} \circ \cdots \circ \pi_{N_{[1]}} \circ \pi_{N_{[0]}}(\boldsymbol{x}) \tag{17}$$

$$\equiv \pi_{N_{[m+k]}} \circ \cdots \circ \pi_{N_{[m]}} \circ \pi_{\text{anc}(\tilde{N})}(\boldsymbol{x}) \tag{18}$$

$$\sigma^{\text{new}}_{\mathcal{T}(m+k)}(\boldsymbol{x}) \equiv \pi^{\text{new}}_{N_{[m+k]}} \circ \cdots \circ \pi^{\text{new}}_{N_{[m]}} \circ \pi_{\text{anc}(\tilde{N})}(\boldsymbol{x})\,. \tag{19}$$

Now let's expand the new part:

$$\sigma^{\text{new}}_{\mathcal{T}(m+k)}(\boldsymbol{x}) = \pi^{\text{new}}_{N_{[m+k]}} \circ \cdots \circ \pi^{\text{new}}_{N_{[m+1]}} \circ \pi^{\text{new}}_{N_{[m]}} \circ \pi_{\text{anc}(\tilde{N})}(\boldsymbol{x}) \tag{20}$$

$$= (\tilde{\pi} \circ \pi_{N_{[m+k]}} \circ \tilde{\pi}^{-1}) \circ \cdots \circ (\tilde{\pi} \circ \pi_{N_{[m+1]}} \circ \tilde{\pi}^{-1}) \circ (\tilde{\pi} \circ \pi_{N_{[m]}}) \circ \pi_{\text{anc}(\tilde{N})}(\boldsymbol{x}) \tag{21}$$

$$= \tilde{\pi} \circ \pi_{N_{[m+k]}} \circ \cdots \circ \pi_{N_{[m+1]}} \circ \pi_{N_{[m]}} \circ \pi_{\text{anc}(\tilde{N})}(\boldsymbol{x}) \tag{22}$$

$$= \tilde{\pi} \circ \sigma_{\mathcal{T}(m+k)}\,, \tag{23}$$

where the second line is by the definition of the new node permutations, the third line is by noticing that this composition is a telescoping composition where the inner $\tilde{\pi}^{-1} \circ \tilde{\pi}$ terms cancel out, and the last line is by the definition of $\sigma_{\mathcal{T}(m+k)}$.

Next, we show that given our definitions of the descendant nodes above, the same configuration will go left in the new tree,

$$\sigma_{\mathcal{T}(m+k)}(\boldsymbol{x}_s) \in v \tag{24}$$

$$\tilde{\pi} \circ \sigma_{\mathcal{T}(m+k)}(\boldsymbol{x}_s) \in v^{\text{new}} \tag{25}$$

$$\sigma^{\text{new}}_{\mathcal{T}(m+k)}(\boldsymbol{x}_s) \in v^{\text{new}}\,, \tag{26}$$

where the first line is by assumption, the second line is by the definition of v^{new} and the third line is by our derivation above about the relationship between the new and old permutations. Thus, again, the same inputs will go left and right and thus the inductive hypothesis holds for $m+1$, i.e., $\forall N$ at a depth of $m+k+1$:

$$\sigma_{\mathcal{T}(m+k+1)}(\boldsymbol{x}) \in \mathcal{D}(N) \Leftrightarrow \sigma^{\text{new}}_{\mathcal{T}(m+k+1)}(\boldsymbol{x}) \in \mathcal{D}^{\text{new}}(N)\,. \tag{27}$$

For the next part of the proof, we need to prove that this equivalent TSP has better negative log likelihood. Similar to the above derivation we can know that $\sigma^{\text{new}}_{\mathcal{T}}(\boldsymbol{x}) = \tilde{\pi} \circ \sigma_{\mathcal{T}}(\boldsymbol{x}), \forall \boldsymbol{x} \in \mathcal{D}(\tilde{N})$. And because $\tilde{\pi}$ only swaps two configurations, then the node counts are all equivalent except that $c^{\text{new}}_N(s,a) = c_N(s,b)$ and $c^{\text{new}}_N(s,b) = c_N(s,a)$ for all nodes that are equal to $\tilde{N}$ or descendants of $\tilde{N}$. Now we also note that the maximum likelihood solution is actually equivalent to minimizing the sum of feature-wise empirical entropies:

$$\min_Q \frac{1}{n} \sum_{i=1}^{n} -\log Q(\boldsymbol{z}_i) \tag{28}$$

$$= \min_Q \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{k} -\log Q_j(z_{i,j}) \tag{29}$$

$$= \sum_{j=1}^{k} \min_{Q_j} \frac{1}{n} \sum_{i=1}^{n} -\log Q_j(z_{i,j}) \tag{30}$$

$$= \sum_{j=1}^{k} \min_{Q_j} \mathbb{E}_{\hat{Q}_j}[-\log Q_j(z_{i,j})] \tag{31}$$

$$= \sum_{j=1}^{k} \mathbb{E}_{\hat{Q}_j}[-\log \hat{Q}_j(z_{i,j})] \tag{32}$$

$$= \sum_{j=1}^{k} H(\hat{Q}_j)\,, \tag{33}$$

where $\boldsymbol{z} = \sigma_{\mathcal{T}}(\boldsymbol{x})$ and $\hat{Q}_j$ is the empirical distribution of the j-th dimension of $\boldsymbol{z}$ (i.e., merely the empirical probabilities based on normalizing the counts). The first equals is by the assumption of independence, the second is by noting that the minimization is decomposable, the third is by the definition of an empirical expectation, the fourth is by the optimal solution (i.e., the empirical probabilities is the best MLE estimate given the empirical distribution) and the last is by the definition of entropy. Note that Eqn. 31 can actually be seen as another definition of entropy. Thus, we only need to compare the entropies for feature s (since all others are equal) $H(\hat{Q}_s^{\text{new}}) < H(\hat{Q}_s)$. Without loss of generality, let us suppose that $k = 2$ (i.e., $\hat{Q}_s$ is a Bernoulli distribution). Now let $p_Q \triangleq \hat{Q}_s(a)$ and $p_Q^{\text{new}} \triangleq \hat{Q}_s^{\text{new}}(a)$, dropping the notation on s because this is the only variable the changes. Because the empirical counts for a are larger in the new distribution, we know that $p_Q^{\text{new}} > p_Q$ and additionally we know that $p_Q > (1 - p_Q)$ because $c_{\tilde{N}}(s, a) > c_{\tilde{N}}(s, b)$ by assumption. Given that $p_Q > (1 - p_Q)$, the derivative of entropy is negative, i.e., $\frac{dH(p_Q)}{dp} = -\log p_Q + \log(1 - p_Q) < 0$ because $\log$ is a monotonically increasing function (i.e., $\log p_Q > \log(1 - p_Q)$). We can form a linear upper bound on H by the first order Taylor series expansion around p_0:

$$H(p) \leq H(p_0) + \frac{dH(p_0)}{dp}(p - p_0) \tag{34}$$

and derive that the entropy of the new is lower:

$$H(p_Q^{\text{new}}) \leq H(p_Q) + \frac{dH(p_Q)}{dp}(p_Q^{\text{new}} - p_Q) < H(p_Q)\,, \tag{35}$$

where the first line is by the concavity of $H(p)$ and the second is by the fact that $p_Q^{\text{new}} - p_Q > 0$ while $\frac{dH(p_Q)}{dp} < 0$. Thus, the newly constructed TSP is tree equivalent yet it has a lower negative log likelihood (or equivalently lower feature-wise entropy). Yet, this is a contradiction to our assumption that the original TSP was optimal. □

G. Tree Equivalence

Definition 5 (TSP Tree Equivalence)**.** *Two TSPs $\sigma_{\mathcal{T}}^A, \sigma_{\mathcal{T}}^B$ are* tree equivalent*, denoted by $\sigma_{\mathcal{T}}^A \equiv_{\text{tr}} \sigma_{\mathcal{T}}^B$ if and only if they have the same graph structure (i.e., same nodes and edges), and $\forall j, \boldsymbol{x},\ \sigma_{\mathcal{T}}^A(\boldsymbol{x}) \in \mathcal{D}(N_j^A) \Leftrightarrow \sigma_{\mathcal{T}}^B(\boldsymbol{x}) \in \mathcal{D}(N_j^B)$ where j is the index of the node.*

Proof that $\equiv_{\text{tr}}$ is indeed an equivalence relation.

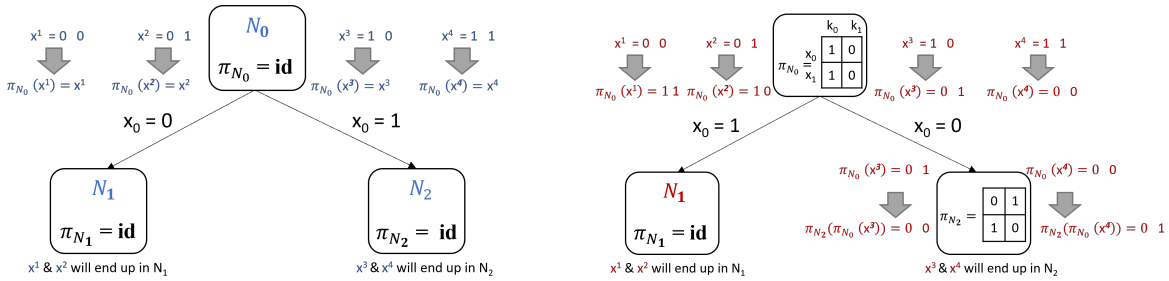
Reflexive property ($\forall \mathcal{T}, \mathcal{T} \equiv_{\text{tr}} \mathcal{T}$) This property is trivial by inspection of the definition. For any $\mathcal{T}$, clearly any tree has the same graph structure as itself and $\forall j, \boldsymbol{x},\ \sigma_{\mathcal{T}}(\boldsymbol{x}) \in \mathcal{D}(N_j) \Leftrightarrow \sigma_{\mathcal{T}}(\boldsymbol{x}) \in \mathcal{D}(N_j)$, where both statements are almost trivial.

Symmetric property ($\forall \mathcal{T}_A, \mathcal{T}_B, \mathcal{T}_A \equiv_{\text{tr}} \mathcal{T}_B \Leftarrow \mathcal{T}_B \equiv_{\text{tr}} \mathcal{T}_A$) Again, this is easy to prove. Suppose $\mathcal{T}_A \equiv_{\text{tr}} \mathcal{T}_B$, this means that $\mathcal{T}_A$ and $\mathcal{T}_B$ have the same graph structure and that $\forall j, \boldsymbol{x},\ \sigma_{\mathcal{T}_A}(\boldsymbol{x}) \in \mathcal{D}(N_j^{(A)}) \Leftrightarrow \sigma_{\mathcal{T}_B}(\boldsymbol{x}) \in \mathcal{D}(N_j^{(B)})$. Because the structure is the same, then $\mathcal{T}_B$ has the same structure as $\mathcal{T}_A$ and we know that $\forall j, \boldsymbol{x},\ \sigma_{\mathcal{T}_B}(\boldsymbol{x}) \in \mathcal{D}(N_j^{(B)}) \Leftrightarrow \sigma_{\mathcal{T}_A}(\boldsymbol{x}) \in \mathcal{D}(N_j^{(A)})$. Thus, $\mathcal{T}_B \equiv_{\text{tr}} \mathcal{T}_A$.

Transitive property ($\forall \mathcal{T}_A, \mathcal{T}_B, \mathcal{T}_C,$ if $\mathcal{T}_A \equiv_{\text{tr}} \mathcal{T}_B$ and $\mathcal{T}_B \equiv_{\text{tr}} \mathcal{T}_C,$ then $\mathcal{T}_A \equiv_{\text{tr}} \mathcal{T}_C$) First, we check the transitivity of property (1): If $\mathcal{T}_A$ has the same graph structure as $\mathcal{T}_B$ and $\mathcal{T}_B$ has the same graph structure as $\mathcal{T}_C$, then $\mathcal{T}_A$ and $\mathcal{T}_C$ have the same graph structure. Second, we check the transitivity of property (2): From $\mathcal{T}_A \equiv_{\text{tr}} \mathcal{T}_B$, we know that $\forall j, \boldsymbol{x},\ \sigma_{\mathcal{T}_A}(\boldsymbol{x}) \in \mathcal{D}(N_j^{(A)}) \Leftrightarrow \sigma_{\mathcal{T}_B}(\boldsymbol{x}) \in \mathcal{D}(N_j^{(B)})$ and from $\mathcal{T}_B \equiv_{\text{tr}} \mathcal{T}_C$, we know that $\forall j, \boldsymbol{x},\ \sigma_{\mathcal{T}_B}(\boldsymbol{x}) \in \mathcal{D}(N_j^{(B)}) \Leftrightarrow \sigma_{\mathcal{T}_C}(\boldsymbol{x}) \in \mathcal{D}(N_j^{(C)})$. Because each of these statements are if and only if (i.e., $\Leftrightarrow$), we can clearly conclude that $\forall j, \boldsymbol{x},\ \sigma_{\mathcal{T}_A}(\boldsymbol{x}) \in \mathcal{D}(N_j^{(A)}) \Leftrightarrow \sigma_{\mathcal{T}_C}(\boldsymbol{x}) \in \mathcal{D}(N_j^{(C)})$, which proves the second property is transitive. Thus, we can conclude that $\mathcal{T}_A \equiv_{\text{tr}} \mathcal{T}_C$.

□

This equivalence relation sheds additional interpretation of our algorithm. The splitting criteria determines one tree in the equivalence class of TSPs defined by the tree equivalence relation (specifically one in the equivalence class where all the permutations are the identity). Our other algorithms then finds the optimal *equivalent* tree in the equivalence class that maintains this split structure, but changes the permutations to be the optimal with respect to NLL.



(a) Tree A (b) Tree B

Figure 10: An example of two equivalent trees a) Tree A b) Tree B

Informally, this equivalence relation means that any configuration will reach the same node for any tree in the equivalence class. However, the permutations and split criteria might be different for each TSP in the equivalence class—and in particular, the choice of which permutations and split criteria will affect the NLL of the model.

We present an example of two tree equivalent trees in Figures G

H. Algorithms pseudo code

We give the additional pseudo-code for the complete algorithm here. At a high level, **??**:LearnTSP is the complete algorithm and calls all the other algorithms, it creates a root node, then calls Alg. 1:ConstructTree which is for constructing the tree, then learn the local permutations (Alg. 3: LearnLocalPermutations) discussed in section 3.2.2), then find the equivalent tree structure (Alg. 4: ConstructEquivalentTree discussed in section 3.2.2). Finally returning the node of the equivalent tree, which is the node to the final "correct" TSP. **??**:LearnTSP builds the tree recursively given the specified maximum depth of the tree. It utilizes Alg. 2:FindBestSplit to find the best split and it also outputs the counts of the categorical values at each node (which will be used by the later algorithms). Alg. 2:FindBestSplit simply iterates over all possible splits (that is all dimensions and category values) calculates the scores for each split according to the scoring function (which can be GLP, or RND. For RND it will just randomly select a dimension to split on and a split value without the need to calculate any scores), following that it shows the pair of split feature and split value that resulted in the best score, and splits the data accordingly and also calculates the domains of the children that are created by this split.

I. Proofs for Algorithms

Before we prove Theorem 3, we first prove several important lemmas.

I.1. Proof that Alg. 4 constructs an equivalent tree

Lemma 2. *The ConstructEquivalentTree algorithm produces a new TSP that has equivalent tree structure to the original TSP from the ConstructTree algorithm.*

Proof. We will prove by **induction** on the tree depth.

Base case (depth $= 1$) For the base case of depth equal to one, we only have three nodes: the parent node N_0 and two children nodes, where each has an identity permutation $\pi_{N_i} \triangleq \pi_{\text{Id}}, \forall i$ (by construction from Algorithm 1) but also a local permutation $\tilde{\pi}_{N_i}$ (by Algorithm 4) . We will distinguish the new nodes created by ConstructEquivalentTree with superscript new for example, the root node will be N_0^{new}. Each of these nodes has a $\pi \triangleq \pi_{\text{anc}} \circ \tilde{\pi} \circ \pi_{\text{anc}}^{-1}$ (where $\pi_{\text{anc}(N_0)} \triangleq \pi_{\text{Id}}$), and $v_N^{\text{new}} = \{\tilde{\pi} \circ \pi_{\text{anc}(N)}(v) : v \in v_N\}$ It is sufficient to verify that the same input configurations that went left in the original tree will go to the left in the equivalent tree.

First, note that for depth $= 1$, $\pi_{\text{anc}(N_0)} = \pi_{\text{Id}}$ and thus $\pi_{N_0}^{\text{new}} = \tilde{\pi}_{N_{[0]}}$. Let $\boldsymbol{x}$ be a configuration that went left in the original

TSP. This means that

$$x_j \in v_N \tag{36}$$
$$\Leftrightarrow \tilde{\pi}_{N_0}(x_j) \in v_N^{\text{new}}\,, \tag{37}$$

where the first implications is by the construction of v_N^{new}. To ensure invertibility, we must also verify that the domain and range are equal for each new node. The root node has a domain of the entire space and thus any permutation will ensure that the domain and range are equal. Thus, we only need to verify that the new child permutations only permute their new domains, i.e., $\boldsymbol{x} \in \mathcal{D}(N^{\text{new}}) \Leftrightarrow \pi^{\text{new}}(\boldsymbol{x}) \in \mathcal{D}(N^{\text{new}})$. First, we note that $\mathcal{R}(\tilde{\pi}) \equiv \mathcal{D}$ by construction of $\tilde{\pi}$, i.e., $\boldsymbol{x} \in \mathcal{D} \Leftrightarrow \tilde{\pi}(\boldsymbol{x}) \in \mathcal{D}$. We also note that the same input $\boldsymbol{x}$ reach this node (where all the permutations were the identity), i.e., $\forall \boldsymbol{x}, \sigma_{\mathcal{T}}(\boldsymbol{x}) \equiv \boldsymbol{x} \in \mathcal{D}(N) \Leftrightarrow \sigma_{\mathcal{T}}^{\text{new}}(\boldsymbol{x}) \in \mathcal{D}(N^{\text{new}})$, but now there are permutations applied at the ancestors of the node N, we have the following:

$$\mathcal{D}(N^{\text{new}}) = \{\pi_{\text{anc}}(\boldsymbol{x}) : \boldsymbol{x} \in \mathcal{D}(N)\}\,. \tag{38}$$

We prove for the left child as the right child can be proved in the same way.

$$\boldsymbol{x} \in \mathcal{D}(N^{\text{new}}) \tag{39}$$
$$\Leftrightarrow \pi_{\text{anc}}^{-1}(\boldsymbol{x}) \in \mathcal{D}(N) \tag{40}$$
$$\Leftrightarrow \tilde{\pi} \circ \pi_{\text{anc}}^{-1}(\boldsymbol{x}) \in \mathcal{D}(N) \tag{41}$$
$$\Leftrightarrow \pi_{\text{anc}} \circ \tilde{\pi} \circ \pi_{\text{anc}}^{-1}(\boldsymbol{x}^{\text{new}}) \in \mathcal{D}(N^{\text{new}}) \tag{42}$$

where the first line is from the properties of one-to-one mappings and Eqn. 38, the second line is by the property of $\tilde{\pi}$ such that the domain and range are the same , and the third line is also by Eqn. 38. Thus, the new TSP has equivalent tree structure and is a valid TSP.

Inductive step The key step is to prove that if the property is true all nodes of till depth m, any nodes at depth $m+1$ also have equivalent tree structure. Without loss of generality, we choose the split of one node at depth m and show that the same points would go left (all other nodes at depth m can be proved similarly). Suppose that $\boldsymbol{x} \in \mathcal{D}(N)$ went to the left child in our original TSP, i.e., $x_s \in v$. By our inductive hypothesis $\pi_{\text{anc}}(\boldsymbol{x}) \in \mathcal{D}(N^{\text{new}})$.

Let $\boldsymbol{x}^{\text{new}} \equiv \tilde{\pi} \circ \pi_{\text{anc}}(\boldsymbol{x}_s)$ be the equivalent configuration in the new TSP, we can derive that:

$$x_s \in v_N \tag{43}$$
$$\Leftrightarrow \tilde{\pi} \circ \pi_{\text{anc}}(\boldsymbol{x}_s) \in v_N^{\text{new}} \tag{44}$$
$$\Leftrightarrow x_s^{\text{new}} \in v_N^{\text{new}}\,, \tag{45}$$

where the second line is by the definition of v^{new} and the third line is by the definition of $\boldsymbol{x}^{\text{new}}$. Thus, the samples that went left in the original TSP will also go left in this new TSP. For this step, the domains of the node can be validated in the same way as in the base case since there is nothing special in the inductive case.

□

I.2. Proof that new TSP is equal to applying local permutations in reverse order

We will need the following simple lemma about the structure of the new TSP permutation to prove our result in Lemma 3.

Lemma 5. *The new TSP constructed from our algorithms is equal to applying the local permutations in reverse order:*

$$\forall M > 0, \quad \sigma_{\mathcal{T}_{(M)}}^{\text{new}} = \pi_{N_{[M]}}^{\text{new}} \circ \pi_{N_{[M-1]}}^{\text{new}} \circ \cdots \circ \pi_{N_{[0]}}^{\text{new}} = \tilde{\pi}_{N_{[0]}} \circ \tilde{\pi}_{N_{[1]}} \circ \cdots \circ \tilde{\pi}_{N_{[M]}}\,.$$

Also, one specific case of this is for ancestor permutations:

$$\pi_{\text{anc}(N_{[M]})}^{\text{new}} = \tilde{\pi}_{N_{[0]}} \circ \tilde{\pi}_{N_{[1]}} \circ \cdots \circ \tilde{\pi}_{N_{[M-1]}}\,. \tag{46}$$

Proof. We will prove by induction. We will first recall some definitions:

$$\pi \triangleq \pi_{\text{anc}} \circ \tilde{\pi} \circ \pi_{\text{anc}}^{-1} \tag{47}$$

$$\pi^{-1} \triangleq \pi_{\text{anc}} \circ \tilde{\pi}^{-1} \circ \pi_{\text{anc}}^{-1} \tag{48}$$

$$\pi_{\text{anc}(N_{[m]})} \triangleq \pi_{N_{[m-1]}} \circ \pi_{N_{[m-2]}} \circ \cdots \circ \pi_{N_{[1]}} \circ \pi_{N_{[0]}} \tag{49}$$

$$\pi^{-1}_{\text{anc}(N_{[m]})} \triangleq \pi^{-1}_{N_{[0]}} \circ \pi^{-1}_{N_{[1]}} \circ \cdots \circ \pi^{-1}_{N_{[m-2]}} \circ \pi^{-1}_{N_{[m-1]}} \tag{50}$$

Base case (TSP depth $= 1$) We define the new TSP for depth of 1

$$\sigma^{\text{new}}_{\mathcal{T}_{(1)}} = \pi^{\text{new}}_{N_{[1]}} \circ \pi^{\text{new}}_{N_{[0]}} \tag{51}$$

$$= (\pi^{\text{new}}_{\text{anc}(N_{[1]})} \circ \tilde{\pi}_{N_{[0]}} \circ \pi^{\text{new}}_{\text{anc}(N_{[1]})}{}^{-1}) \circ \pi^{\text{new}}_{N_{[0]}} \tag{52}$$

$$= \pi^{\text{new}}_{N_{[0]}} \circ \tilde{\pi}_{N_{[1]}} \circ \pi^{\text{new}}_{N_{[0]}}{}^{-1} \circ \pi^{\text{new}}_{N_{[0]}} \tag{53}$$

$$= \pi^{\text{new}}_{N_{[0]}} \circ \tilde{\pi}_{N_{[1]}} \tag{54}$$

$$= (\pi^{\text{new}}_{\text{anc}(N_{[0]})} \circ \tilde{\pi}_{N_{[0]}} \circ \pi^{\text{new}}_{\text{anc}(N_{[0]})}{}^{-1}) \circ \tilde{\pi}_{N_{[1]}} \tag{55}$$

$$= \tilde{\pi}_{N_{[0]}} \circ \tilde{\pi}_{N_{[1]}}\,, \tag{56}$$

where the second equality is just the definition of our permutation for $\pi^{\text{new}}_{N_{[1]}}$, the third equality is the expansion of the ancestor permutation definitions, the fourth equality is because $\pi^{\text{new}}_{N_{[0]}}{}^{-1} \circ \pi^{\text{new}}_{N_{[0]}}$ cancel each other. The equation is simplified to the last equation since there exists no ancestor nodes for $N_{[0]}$, thus we can assign $\pi^{\text{new}}_{\text{anc}(N_{[0]})}$ as an identity. Therefore, Lemma 5 holds for the base case.

Furthermore, we can define the new TSP at depth 1 as the new ancestor permutations for $N_{[2]}$:

$$\pi^{\text{new}}_{\text{anc}(N_{[2]})} = \sigma^{\text{new}}_{\mathcal{T}_{(1)}} = \tilde{\pi}_{N_{[0]}} \circ \tilde{\pi}_{N_{[1]}}\,, \tag{57}$$

and we can give a more general definition:

$$\pi^{\text{new}}_{\text{anc}(N_{[m]})} = \sigma^{\text{new}}_{\mathcal{T}_{(m-1)}} = \tilde{\pi}_{N_{[0]}} \circ \tilde{\pi}_{N_{[1]}} \circ \cdots \tilde{\pi}_{N_{[m-1]}}\,, \tag{58}$$

Induction step (TSP depth $= m \Rightarrow TSP$ depth $= m+1$) Let us assume for depth of m, Lemma 5 holds true. Then the new TSP for depth of m using definition from Eqn. 58 can be written as below:

$$\sigma^{\text{new}}_{\mathcal{T}_{(m)}} = \pi^{\text{new}}_{\text{anc}(N_{[m+1]})} \tag{59}$$

$$= \pi^{\text{new}}_{N_{[m]}} \circ \pi^{\text{new}}_{N_{[m-1]}} \circ \cdots \circ \pi^{\text{new}}_{N_{[0]}} \tag{60}$$

$$= \tilde{\pi}_{N_{[0]}} \circ \tilde{\pi}_{N_{[1]}} \circ \cdots \tilde{\pi}_{N_{[m]}}\,. \tag{61}$$

Let us now observe our new TSP permutation for depth of $m+1$:

$$\sigma^{\text{new}}_{\mathcal{T}(m+1)} = \pi^{\text{new}}_{N_{[m+1]}} \circ \pi^{\text{new}}_{N_{[m]}} \circ \cdots \circ \pi^{\text{new}}_{N_{[0]}}\,. \tag{62}$$

Referring back to Eqn. 59, we can rewrite the equality as:

$$\sigma^{\text{new}}_{\mathcal{T}_{(m+1)}} = \pi^{\text{new}}_{N_{[m+1]}} \circ \pi^{\text{new}}_{\text{anc}(N_{[m+1]})} \tag{63}$$

$$= \pi^{\text{new}}_{\text{anc}(N_{[m+1]})} \circ \tilde{\pi}_{N_{[m+1]}} \circ \pi^{\text{new}}_{\text{anc}(N_{[m+1]})}{}^{-1} \circ \pi^{\text{new}}_{\text{anc}(N_{[m+1]})} \tag{64}$$

$$= \pi^{\text{new}}_{\text{anc}(N_{[m+1]})} \circ \tilde{\pi}_{N_{[m+1]}} \tag{65}$$

$$= \tilde{\pi}_{N_{[0]}} \circ \tilde{\pi}_{N_{[1]}} \circ \cdots \tilde{\pi}_{N_{[m]}} \circ \tilde{\pi}_{N_{[m+1]}}\,, \tag{66}$$

where the third equality is given by the cancellation of the ancestor and the inverse ancestor permutation on the previous line. The final equality is the expansion of $\pi^{\text{new}}_{\text{anc}(N_{[m+1]})}$ on Eqn. 61. Therefore, we prove that

$$\sigma^{\text{new}}_{\mathcal{T}_{(M)}} = \tilde{\pi}_{N_{[0]}} \circ \tilde{\pi}_{N_{[1]}} \circ \cdots \circ \tilde{\pi}_{N_{[M]}}. \tag{67}$$

□

I.3. Proof that new counts are a permutation of the local counts

Lemma 6 (Relation between local and new node counts)**.** *The new node counts $c_{N^{\text{new}}}$ are equal to the local counts $\tilde{c}_N$ after permutation by the ancestor permutation $\pi^{\text{new}}_{\text{anc}(N)}$, i.e., $c_{N^{\text{new}}} = \pi^{\text{new}}_{\text{anc}(N)}[\tilde{c}_N]$.*

Proof. Consider any **leaf node** $N \equiv N_{[M]}$ at depth M, where we suppress the dependence on M for now. At the leaf node, the local counts are originally equal to the raw unpermuted counts based on Line 8 in Alg. 1, i.e.,

$$\forall j, a, \quad \tilde{c}^{\text{init}}_N(j,a) = \sum_{i=1}^{n} \mathbb{1}(\sigma_{\mathcal{T}}(\boldsymbol{x}_i)_j = a \wedge \sigma_{\mathcal{T}}(\boldsymbol{x}_i) \in \mathcal{D}(N)) = \sum_{i=1}^{n} \mathbb{1}(x_{i,j} = a \wedge \boldsymbol{x}_i \in \mathcal{D}(N)) \tag{68}$$

where the first equality is by the fact that only inputs that land in the node are counted and the second equality is because the original has all identity node permutations so $\sigma_{\mathcal{T}} \equiv \pi_{\text{id}}$. We now note that an ancestor permutation is a composition of independent permutations so each ancestor permutation can be split into an independent permutation over each feature, i.e.,

$$\pi_{\text{anc}(N)}(\boldsymbol{x})_j = \pi_{\text{anc}(N),j}(x_j). \tag{69}$$

We also note that:

$$\sigma^{\text{new}}_{\mathcal{T}} = \tilde{\pi}_{N_{[0]}} \circ \tilde{\pi}_{N_{[1]}} \circ \cdots \circ \tilde{\pi}_{N_{[M-1]}} \circ \tilde{\pi}_{N_{[M]}} = \pi^{\text{new}}_{\text{anc}(N_{[M]})} \circ \tilde{\pi}_{N_{[M]}} = \pi^{\text{new}}_{\text{anc}(N)} \circ \tilde{\pi}_N\,, \tag{70}$$

where the first and third equality is due to Lemma 5, and the last is by suppressing notational dependence on $[M]$. Combining the facts above, we can derive the result for leaf nodes:

$$\begin{aligned}
\forall j, a, \quad c_{N^{\text{new}}}(j,a) &= \sum_{i=1}^{n} \mathbb{1}(\sigma^{\text{new}}_{\mathcal{T}}(\boldsymbol{x}_i)_j = a \wedge \sigma^{\text{new}}_{\mathcal{T}}(\boldsymbol{x}_i) \in \mathcal{D}(N^{\text{new}})) && (71)\\
&= \sum_{i=1}^{n} \mathbb{1}(\sigma^{\text{new}}_{\mathcal{T}}(\boldsymbol{x}_i)_j = a \wedge \boldsymbol{x}_i \in \mathcal{D}(N)) && (72)\\
&= \sum_{i=1}^{n} \mathbb{1}(\pi^{\text{new}}_{\text{anc}(N)} \circ \tilde{\pi}_N(\boldsymbol{x}_i)_j = a \wedge \boldsymbol{x}_i \in \mathcal{D}(N)) && (73)\\
&= \sum_{i=1}^{n} \mathbb{1}(\pi^{\text{new}}_{\text{anc}(N),j} \circ \tilde{\pi}_{N,j}(x_{i,j}) = a \wedge \boldsymbol{x}_i \in \mathcal{D}(N)) && (74)\\
&= \sum_{i=1}^{n} \mathbb{1}(x_{i,j} = \tilde{\pi}^{-1}_{N,j} \circ \pi^{\text{new}-1}_{\text{anc}(N),j}(a) \wedge \boldsymbol{x}_i \in \mathcal{D}(N)) && (75)\\
&= \pi^{\text{new}}_{\text{anc}(N),j}[\tilde{\pi}_{N,j}[\tilde{c}^{\text{init}}_N(j,a)]] && (76)\\
&= \pi^{\text{new}}_{\text{anc}(N),j}[\tilde{c}_N(j,a)] && (77)
\end{aligned}$$

where the first equality is by definition, the second is by TSP tree equivalence, the third is by (70), the fourth is by (69), the fifth is by invertibility, the sixth is by the definition of the $\pi[\cdot]$ operator and (68), and the last is by the PermuteCounts line in Alg. 3.

The above derivation proves the statement for leaf nodes. We now want to prove by induction that the statement holds for **non-leaf nodes**. Without loss of generality, we will prove for one path between root and leaf by induction on a depth where $N_{[M]}$ is the leaf node and $N_{[0]}$ is the root node. Specifically, we will prove that this holds for $N_{[M-k]}$ where $k \in \{0, 1, \ldots, M\}$. The base case of $k = 0$ is proved above as $N_{[M]}$ is a leaf node so we only need to prove the inductive

case. Let $N \equiv N_{[M-(k+1)]}$ and let N_{left} and N_{right} be it's left and right nodes which are both at depth $M-k$ (i.e., where the inductive hypothesis is assumed to hold). First, we establish that the counts of a parent node are merely equal to the sum of the counts of children nodes, i.e.,

$$\begin{aligned}\forall j, a, \quad c_N(j,a) &= \sum_{i=1}^{n} \mathbb{1}(\sigma_{\mathcal{T}}(\boldsymbol{x}_i)_j = a \wedge \sigma_{\mathcal{T}}(\boldsymbol{x}_i) \in \mathcal{D}(N)) \\ &= \sum_{i=1}^{n} \mathbb{1}(\sigma_{\mathcal{T}}(\boldsymbol{x}_i)_j = a \wedge \sigma_{\mathcal{T}}(\boldsymbol{x}_i) \in \mathcal{D}(N_{\text{left}})) + \sum_{i=1}^{n} \mathbb{1}(\sigma_{\mathcal{T}}(\boldsymbol{x}_i)_j = a \wedge \sigma_{\mathcal{T}}(\boldsymbol{x}_i) \in \mathcal{D}(N_{\text{right}})) \\ &= c_{N_{\text{left}}}(j,a) + c_{N_{\text{right}}}(j,a)\,,\end{aligned} \tag{78}$$

where the second equality comes from the fact that $\mathcal{D}(N_{\text{left}}) \cap \mathcal{D}(N_{\text{right}}) = \emptyset$ and $\mathcal{D}(N_{\text{left}}) \cup \mathcal{D}(N_{\text{right}}) = \mathcal{D}(N)$. Second, we show a following fact that will be useful in the proof:

$$\begin{aligned}\pi^{\text{new}}_{\text{anc}(N_{\text{left}})} \circ \tilde{\pi}_N^{-1} &= \pi^{\text{new}}_{\text{anc}(N_{[M-k]})} \circ \tilde{\pi}^{-1}_{N_{[M-(k+1)]}} && (79)\\ &= (\tilde{\pi}_{N_{[0]}} \circ \tilde{\pi}_{N_{[1]}} \circ \cdots \circ \tilde{\pi}_{N_{[M-(k+2)]}} \circ \tilde{\pi}_{N_{[M-(k+1)]}}) \circ \tilde{\pi}^{-1}_{N_{[M-(k+1)]}} && (80)\\ &= \tilde{\pi}_{N_{[0]}} \circ \tilde{\pi}_{N_{[1]}} \circ \cdots \circ \tilde{\pi}_{N_{[M-(k+2)]}} && (81)\\ &= \pi^{\text{new}}_{\text{anc}(N_{[M-(k+1)]})} = \pi^{\text{new}}_{\text{anc}(N)}\,, && (82)\end{aligned}$$

where the (79) and (82) are by Lemma 5 Finally, we can prove the inductive step using the two facts above:

$$\begin{aligned}c_{N^{\text{new}}} &= c_{N^{\text{new}}_{\text{left}}} + c_{N^{\text{new}}_{\text{right}}} && (83)\\ &= \pi^{\text{new}}_{\text{anc}(N_{\text{left}})}[\tilde{c}_{N_{\text{left}}}] + \pi^{\text{new}}_{\text{anc}(N_{\text{right}})}[\tilde{c}_{N_{\text{right}}}] && (84)\\ &= \pi^{\text{new}}_{\text{anc}(N_{\text{left}})}[\tilde{c}_{N_{\text{left}}}] + \pi^{\text{new}}_{\text{anc}(N_{\text{left}})}[\tilde{c}_{N_{\text{right}}}] && (85)\\ &= \pi^{\text{new}}_{\text{anc}(N_{\text{left}})}[\tilde{c}_{N_{\text{left}}} + \tilde{c}_{N_{\text{right}}}] && (86)\\ &= \pi^{\text{new}}_{\text{anc}(N_{\text{left}})}[\tilde{c}^{\text{init}}_N] && (87)\\ &= \pi^{\text{new}}_{\text{anc}(N_{\text{left}})}[\tilde{\pi}_N^{-1}[\tilde{c}_N]] && (88)\\ &= \pi^{\text{new}}_{\text{anc}(N)}[\tilde{c}_N] && (89)\end{aligned}$$

where the first is by (78), the second is by the inductive hypothesis, the third is by noting that the left and right nodes have the same ancestors and defining $\pi^{\text{new}}_{\text{anc}(N_{\text{left}})} \equiv \pi^{\text{new}}_{\text{anc}(N_{\text{right}})}$, the fourth is by the linearity of a permutation function, the fifth is by the definition $\tilde{c}^{\text{init}}$ from Line 4 of Alg. 3, the sixth is by the count permutation in Line 7 of Alg. 3, and the last is by the composition of permutations and (82).

□

I.4. Proof that algorithm produces a rank consistent TSP

Lemma 3. *The local permutations learned in LearnLocalPermutations ensure that the equivalent TSP will be rank consistent.*

Proof. From Lemma 6, we know that $c_{N^{\text{new}}} = \pi^{\text{new}}_{\text{anc}(N)}[\tilde{c}_N]$. We also note that the local counts are rank consistent by construction via Lines 6 and 7 of Alg. 3, i.e., $\tilde{c}_N \in \mathcal{R}$. Thus, we will show that applying $\pi^{\text{new}}_{\text{anc}(N)}$ maintains the rank consistency property of $\tilde{c}_N$ (similar to how a merge sort will retain the ordering when merging disjoint sets).

First, we can split the ancestor permutation into an independent permutation per feature:

$$\pi^{\text{new}}_{\text{anc}(N)}[\tilde{c}_N] = [\pi^{\text{new}}_{\text{anc}(N),0}[\tilde{c}_N(0)], \pi^{\text{new}}_{\text{anc}(N),1}[\tilde{c}_N(1)], \cdots] \tag{90}$$

Without loss of generality, we can focus on the j-th feature and expend the ancestor permutation based on Lemma 5:

$$\pi^{\text{new}}_{\text{anc}(N),j}[\tilde{c}_N(j)] = \tilde{\pi}_{N_{[0]},j} \circ \tilde{\pi}_{N_{[1]},j} \circ \cdots \circ \tilde{\pi}_{N_{[M-1]},j}[\tilde{c}_N(j)] \tag{91}$$

We will prove that the local permutations at every level for the feature j will maintain rank consistency. Without loss of generality, let's denote this by $N \equiv N_m$, where $0 \leq m \leq M-1$ (this means all nodes are non-leaves). We note that $\tilde{c}_N^{\text{init}} = \tilde{c}_{N_{\text{left}}} + \tilde{c}_{N_{\text{right}}}$ from Line 4 of Alg. 3.

We will first consider the case where the node split feature is not j, i.e., $j \neq s$. If $j \neq s$, then $\mathcal{D}_j(N) = \mathcal{D}_j(N_{\text{left}}) = \mathcal{D}_j(N_{\text{right}})$. Thus, because the local counts are rank consistent on their domains, i.e., $\tilde{c}_{N_{\text{left}}}(j) \in \mathcal{R}(\mathcal{D}_j(N))$ and $\tilde{c}_{N_{\text{right}}}(j) \in \mathcal{R}(\mathcal{D}_j(N))$, then $\tilde{c}_N^{\text{init}}(j) = \tilde{c}_{N_{\text{left}}}(j) + \tilde{c}_{N_{\text{right}}}(j) \in \mathcal{R}(\mathcal{D}_j(N))$ because adding two vectors that are already ranked ordered will still be ranked ordered even when restricted to a subset of the vector based on the domain. Now because $\tilde{c}_N^{\text{init}}(j)$ is already rank ordered, the learned permutation will be the identity, i.e., $\tilde{\pi}_{N,j} = \pi_{\text{id}}$. Clearly, the identity permutation will maintain rank consistency.

Now we consider the case where j is the split feature, i.e., $j = s$. If $j = s$, then the children domains are a partition of the parent's domain, i.e., $\mathcal{D}(N_{\text{left}}) \cap \mathcal{D}(N_{\text{right}}) = \emptyset$ and $\mathcal{D}(N_{\text{left}}) \cup \mathcal{D}(N_{\text{right}}) = \mathcal{D}(N)$. Therefore, the initial local counts are either from the left or right but not both, i.e.,

$$\tilde{c}_N^{\text{init}}(j,a) = \begin{cases} \tilde{c}_{N_{\text{left}}}(j,a), & \text{if } a \in \mathcal{D}_j(N_{\text{left}}) \\ \tilde{c}_{N_{\text{right}}}(j,a), & \text{if } a \in \mathcal{D}_j(N_{\text{right}}) \end{cases} \tag{92}$$

We will now prove by contradiction that $\tilde{\pi}_N$ maintains rank consistency. Suppose $\tilde{\pi}_N$ did not maintain rank consistency of its children, i.e., $\tilde{c}_{N_{\text{left}}}(j, \tilde{\pi}_N^{-1}(a)) > \tilde{c}_N(j, \tilde{\pi}_N^{-1}(b))$ for $a < b$ and $a, b \in \mathcal{D}(N_{\text{left}})$ where we can focus on the left node without loss of generality. By the assumption on $\tilde{\pi}_N$ and the fact that $\tilde{\pi}_N$ sorts the counts from Line 6 of Alg. 3, we know that $\tilde{c}_N^{\text{init}}(j,a) > \tilde{c}_N^{\text{init}}(j,b)$. And by combining this with the fact that $a, b \in \mathcal{D}_j(N_{\text{left}})$ and the (92), we can infer that $\tilde{c}_{N_{\text{left}}}(j,a) > \tilde{c}_{N_{\text{left}}}(j,b)$. However, this contradicts the fact that we know $\tilde{c}_{N_{\text{left}}}(j,a) \leq \tilde{c}_{N_{\text{left}}}(j,b)$ because the local counts of the children are rank consistent bye construction, i.e., $\tilde{c}_{N_{\text{left}}}(j) \in \mathcal{R}(\mathcal{D}(N_{\text{left}}))$. Thus, an ancestor permutation on a split feature $j = s$ will also maintain rank consistency of its children.

Putting it all together, we know that $c_N^{\text{new}} = \pi_{\text{anc}(N)}^{\text{new}}[\tilde{c}_N]$ from Lemma 6, we know that $\tilde{c}_N \in \mathcal{R}$ by Lines 6 and 7 of Alg. 3, and we have just proven that $\pi_{\text{anc}(N)}^{\text{new}}$ maintains the rank consistency of the input. Therefore, $\forall N,\ c_N^{\text{new}} \in \mathcal{R}$. □

I.5. Proof of Theorem 3

Theorem 3 (Algorithm finds optimal permutations)**.** *Given a tree structure $\mathcal{T}$, our permutation learning algorithm finds the optimal permutations for the nodes in terms of negative log likelihood.*

Proof. The proof follows directly by combining the lemmas, i.e., that the algorithm will produce a TSP that has equivalent tree structure and that this TSP is rank consistent. Then, we apply Theorem 2 to prove that this is indeed optimal given the tree structure. □

J. More experimental details and results

J.1. Modification of the Bipartite flow code

Note that the BF model implemented in (Bricken, 2021), had errors when training dataset with odd dimensions and dimensions greater than 2. Thus, modifications were made to fix the bipartite model code (that are discussed in more details in the appendix). The model's initial embedding flow layers (commonly used for NLP sequence data) were replaced by a 5 linear layer network with batch normalization and ReLU activation functions on every intermediate output node. A skip connection was implemented between the input and the output for every coupling layer. The BF model implemented in (Bricken, 2021), had some bugs in the code (e.g., running the bipartite model for data dimensions $d > 2$ would yield a runtime error).Thus, modifications were made to fix the bipartite model code. The model's initial embedding flow layers (commonly used for NLP sequence data) were replaced by a single hidden layer network with an activation function. A ReLU activation function was incorporated into the hidden layer to add non-linearity. The size of the hidden layer was proportional to the feature size times half the dimension ($\frac{1}{2} * k * d$), since only half of the dimension would be mapped after the split. Each flow layer does a transformation on one of the splits, therefore, at least a paired flow layers (even number of flow layers) is required.

J.2. More details about Gaussian Copula Synthetic Data Experiments

The generating process can be summarized as follows: We first generated data from a multivariate normal distribution, which can have strong dependencies between features, and normalized each feature by subtracting the mean and dividing by the standard deviation. Then, we applied the CDF of a standard normal distribution to all features independently—which creates uniform marginal distributions. Finally, we applied the inverse CDF of a discrete distribution—which will generate discrete data; the discrete distribution could be a Bernoulli distributions with different bias parameters p ($k = 2$) or more generally a categorical distribution ($k \geq 2$). We then generated four datasets using the Gaussian copula model, with varying degrees of dependency among the dataset's features, where the underlying Gaussian graphical model is a simple cycle graph. Specifically, we set the total correlation (a generalization of mutual information) of the Gaussian distribution (underlying the Gaussian copula), which can be solved in closed form for Gaussian distributions, to be $\{0, 1, 10, 100\}$. For all experiments, we used the inverse CDF of a Bernoulli distribution with parameters $p \in \{0.5, 0.3, 0.5, 0.2\}$ applied to each of the 4 dimensions separately. or a balanced dimensional transformation.

J.3. Model space

Below we list the models that we tried for DTF$_{GLP}$, DTF$_{RND}$ grouped under DTF, as well as the models we tried for AF and BF and DDF in an attempt to find the "best" model for all the experiments of section 4.

Table 5: Models tried for different exps.

Exp	DTF	AF	BF	DDF
synthetic	TSPs $\in \{1, 2.., 10\}$ M $\in \{2, 3, ..., 8\}$	hidden layers $\in \{64, 128\}$	$\alpha \in \{1\}$ $\beta \in \{2, 16\}$	hidden layers $\in \{128, 256, 512\}$ coupling layers $\in \{1, 2, ..5\}$
Mushroom	TSPs $\in \{1, 2.., 10\}$ M $\in \{2, 3, ..., 8\}$	hidden layers $\in \{128\}$	$\alpha \in \{2, 4\}$ $\beta \in \{2, 4, 8\}$	hidden layers $\in \{128, 256, 512\}$ coupling layers $\in \{1, 2, ..5\}$
MNIST	TSPs $\in \{1, 2.., 30\}$ M $\in \{3, 7, 10\}$	hidden layers $\in \{1568\}$	init $\in \{id$,w$\backslash o$ id$\}$ $\alpha \in \{2, 4, 8\}$ $\beta \in \{1, 1/4, 1/16\}$	hidden layers $\in \{128, 256, 512\}$ coupling layers $\in \{1, 2, ..5\}$
Genetic	TSPs $\in \{1, 2.., 20\}$ M $\in \{3\}$	hidden layers $\in \{1610\}$	init $\in \{id$,w$\backslash o$ id$\}$ $\alpha \in \{2, 4, 8\}$ $\beta \in \{1, 1/4, 1/16\}$	hidden layers $\in \{128, 256, 512\}$ coupling layers $\in \{1, 2, ..5\}$

DDF's details: We used the mlp architecture for all experiments of that we tested for the DDF model, and varied the number of coupling layers and hidden layers across the different experiments. We trained all models using 10 epochs for both the prior and the neural network training since increasing this number further didn't lead to a significant improvement in the results.

AF and BF details: For the AF and BF models, we use a linear layer architecture where 4 flow layers were used for the synthetic data experiments, and 6 flow layers were used for the mushroom, MNIST, and genetic data experiments experiments. We ran for a total of 200 epochs sampling 250 samples in each epoch.

AF's model architecture The AF model was constructed based on a Masked Autoencoder for Distribution Estimation (MADE) architecture, and a fixed number of hidden layers were decided for each experiment. By default, all small dimensional dataset ($d \leq 10$) was set to 64 hidden layers and medium dimensional dataset ($10 \leq d \leq 100$) was set to 128 hidden layers. Very large dimensional dataset (MNIST & Genetic) had hidden layers twice the size of its dimension.

BF's model architecture: The BF model had two different 5 linear layer architectural structures. The first BF model scales the initial input size I by a constant β and retains both the input and outputs size to $\beta * I$ for all the intermediate linear layers and reduces the size back to its input size at the last linear layer. The second model followed an autoencoder structure where the output of the first two linear layers reduces the input by a constant factor of α. The third linear layer retains the size of the input and the last 2 linear layers increases the inputs by a constant factor of α, thus rescaling the final layer output size back to its original input size. Moreover, for MNIST and Genetic datasets experiments the models had an extra preprocessing stage where the layers could be initial set to behave as an identity function seen in Table 4 (init $\in \{id$,w$\backslash o$ id$\}$).

Other notes: Our DTF model's independent base distribution $Q_{\boldsymbol{z}}$ is the marginal distribution calculated using the counts of each category and for each dimension of our output. The baseline model's prior distribution, however, is a randomized marginal distribution parameter that is optimized as the NLL is minimized in the deep learning model. For a better comparison with our DTF model -where Q is initialized to be the marginal on the input data- we include an initialization stage for the BF model that modifies the model parameter weights to make the first forward pass resemble an identity function. We also initialize the marginal distribution as the frequency counts of the input dataset. This helps the baseline model have the same NLL starting value as that of our DTF model. We include both the identity initialization and the random initialization for the BF baseline experiments for the MNIST dataset and the genetic dataset.

J.4. The full table for synthetic datasets

Below we present the results of Table 2 in more details.

Table 6: The full table version of Table 2

	AF	BF	DDF	DTF$_{GLP}$	DTF$_{RND}$
8GAUSSIAN					
NLL	6.92 ($\pm$ 0.06)	7.21 ($\pm$ 0.09)	6.42 ($\pm$ 0.03)	6.5 ($\pm$ 0.03)	6.94 ($\pm$ 0.04)
TT	155.9 ($\pm$ 2.2)	231.6 ($\pm$ 5.2)	119.8 ($\pm$ 0.8)	7.3 ($\pm$ 0.1)	0.4 ($\pm$ 0.0)
NUMPARAMS	253656	571116	114796	5650 ($\pm$ 38)	23693 ($\pm$ 1428)
COP-H					
NLL	1.53 ($\pm$ 0.02)	1.47 ($\pm$ 0.06)	1.46 ($\pm$ 0.1)	1.33 ($\pm$ 0.02)	1.33 ($\pm$ 0.02)
TT	10.7 ($\pm$ 0.2)	13.2 ($\pm$ 0.2)	58.1 ($\pm$ 1.0)	$\leq$0.1 ($\pm$ 0.0)	0.1 ($\pm$ 0.0)
NUMPARAMS	21024	14112	541720	34 ($\pm$ 1)	119 ($\pm$ 8)
COP-M					
NLL	1.76 ($\pm$ 0.1)	1.62 ($\pm$ 0.05)	1.51 ($\pm$ 0.16)	1.4 ($\pm$ 0.02)	1.4 ($\pm$ 0.02)
TT	10.6 ($\pm$ 0.02)	13.3 ($\pm$ 0.06)	77.9 ($\pm$ 1.8)	$\leq$0.1 ($\pm$ 0.0)	0.1 ($\pm$ 0.0)
NUMPARAMS	21024	14112	677148	43 ($\pm$ 1)	132 ($\pm$ 11)
COP-W					
NLL	2.42 ($\pm$ 0.02)	2.35 ($\pm$ 0.03)	2.29 ($\pm$ 0.07)	2.22 ($\pm$ 0.02)	2.22 ($\pm$ 0.02)
TT	10.5 ($\pm$ 0.01)	13.2 ($\pm$ 0.1)	77.3 ($\pm$ 1.7)	$\leq$0.1 ($\pm$ 0.0)	0.1 ($\pm$ 0.0)
NUMPARAMS	21024	14112	677148	40 ($\pm$ 0)	162 ($\pm$ 5)

J.5. The model that was used for the results

In Table 7 we report the model that corresponded to the results reported in Tables 2 and 3.

Table 7: The model parameters for the results presented in tables 2 and 3. nH refers to the number of hidden layers, nC refers to the number of coupling layers, nTSP referes to the number of TSPs and M is the maximum depth

	AF	BF	DDF	DTF$_{GLP}$	DTF$_{RND}$
8 Gaussian	nH = 128	$\alpha = 1$ $\beta = 2$	nC = 2 nH = 128	nTSP = 9 M = 2	nTSP = 10 M = 8
COP-H	nH = 64	$\alpha = 1$ $\beta = 16$	nC = 4 nH = 256	nTSP = 2 M = 2	nTSP = 10 M = 2
COP-M	nH = 64	$\alpha = 1$ $\beta = 16$	nC = 5 nH = 256	nTSP = 2 M = 2	nTSP = 10 M = 2
COP-W	nH = 64	$\alpha = 1$ $\beta = 16$	nC = 5 nH = 256	nTSP = 2 M = 2	nTSP = 10 M = 2
Mushroom	nH = 128	$\alpha = 4$ $\beta = 8$	nC = 5 nH = 512	nTSP = 8 M = 6	nTSP = 10 M = 7
MNIST	nH = 1568	$\alpha = 4$ $\beta = 1/16$	nC = 5 nH = 256	nTSP = 21 M = 10	nTSP = 30 M = 3
Genetic	nH = 1610	$\alpha = 4$ $\beta = 1/16$	nC = 5 nH = 128	nTSP = 6 M = 3	nTSP = 20 M = 3

J.6. GPU training times comparison

In section 4 of the main paper, we presented all training times on CPU for our model vs AF or BF. In Table 8, we present our model's training time on CPU vs the AF and BF training times on GPU.

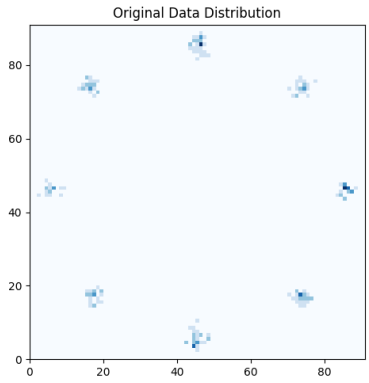
Table 8: Training times for baselines (GPU) and our approach (DTF). Our random DTF (DTF$_{RND}$) trained on a CPU is faster than other methods *even when baselines are trained on a GPU*.

Model	8Gaussian	COPH	Mushroom	Genetic	MNIST
AF (GPU)	42.2 $\pm$ 3.5	14.6 $\pm$ 0.4	7.7 $\pm$ 1.0	23.8 $\pm$ 0.9	305.6 $\pm$ 31.4
BF (GPU)	135.8 $\pm$ 0.2	20.9 $\pm$0.3	6.0 $\pm$ 1.3	38.0 $\pm$ 5.4	308.7 $\pm$ 4.1
DDF (GPU)	79.7 $\pm$ 0.8	48.8 $\pm$0.9	75.2 $\pm$ 1.0	29.5 $\pm$1.1	334.3 $\pm$ 8.7
DTF$_{GLP}$	7.3 $\pm$ 0.1	$\leq$ **0.1** $\pm$ **0.0**	9.9$\pm$ 0.2	411.5 $\pm$ 2.3	5213.7 $\pm$ 204.9
DTF$_{RND}$	**0.4** $\pm$ **0.0**	$\leq$ **0.1** $\pm$ **0.0**	**0.5** $\pm$ **0.0**	**5.9** $\pm$ **0.0**	**105.6** $\pm$ **0.1**

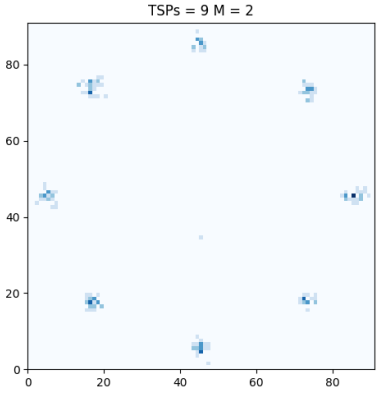
J.7. Exploration of the samples obtained from sampling the learned distribution

Discretized Gaussian Mixture Model To visualize samples that can be generated from the distribution using our model, we trained our best DTF models on the 8Gaussian dataset. We then generated 1024 samples that were plotted in Figure 11b for DTF$_{GLP}$ and in Figure 11c for DTF$_{RND}$. The original data distribution is plotted in Figure 11a. We also sampled 1024 samples generated from the distributions that was learned from the best AF, BF, and DDF models and the results of these are plotted in Figures 11d, 11e and 11f respectively.

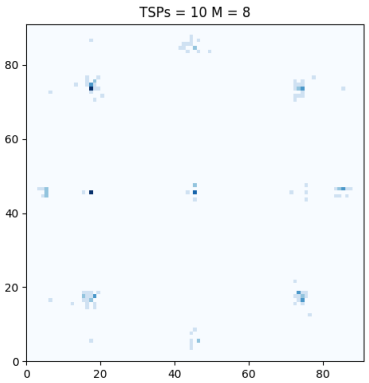
We notice that the samples that were generated from the distribution we learned from the DTF$_{GLP}$ model mirror the original samples of data better than the samples generated from the learned distribution of the AF or BF models and is of similar quality compared to those obtained from DDF.



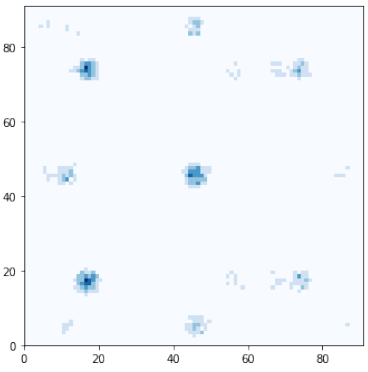
(a) Original data distribution.



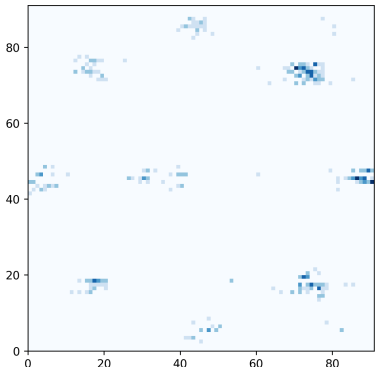
(b) Samples from learned distribution for DTF$_{GLP}$.



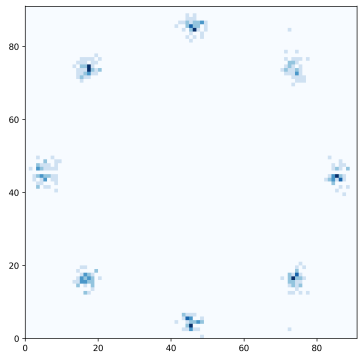
(c) Samples from learned distribution for DTF$_{RND}$.



(d) Samples from learned distribution for AF.



(e) Samples from learned distribution for BF.



(f) Samples from learned distribution for DDF.