OPTIMAL TRANSPORT FOR DISCRETE DISTRIBUTIONS

ADAM B. BLOCK AND PATRIK R. GERBER

Abstract. In this note we describe and implement three algorithms related to computational optimal transport. We give a brief background on the associated problem and describe the network simplex algorithm in detail. We then proceed to consider the case of entropically regularized optimal transport, where we describe the sinkhorn and greenkhorn algorithms. Finally, after implementing these three algorithms, we compare to the benchmark [21] in a thorough performance analysis.

1. Introduction. The study of optimal transport has enjoyed a huge increase in popularity in recent years. On the theoretical side, there have been applications to the study of PDEs, geometry, and concentration of measure in high dimension [19]. On the applied side, there is even greater diversity in the fields that have found the tools of optimal transport invaluable, from operations research [14], to image processing to mathematical statistics. The field has grown sufficiently so as to possess multiple books from those analyzing the theory [19, 17] to those analyzing the practical [16]. In this project, we focus exclusively on the latter.

The first point that must be addressed is the question of what actually is optimal transport. The canonical illustrative example, found in almost any introduction to the topic [16, 19, 17] is to consider two sand piles and a worker with a shovel: the objective of the worker is to transform one sand pile into another with minimal exertion. The formal statement of this is, given two measures μ, ν on a space $\mathcal{X}$ and a cost function $c : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$

$$\min \mathbb{E}_\mu \, c(X, T(X)) \qquad \textbf{such that } T_\# \mu = \nu. \tag{1.1}$$

This formulation has roots as far back as the eighteenth century, with the French mathematician Monge [15]. The problem, realized particularly in the case where μ, ν are discrete measures, is that such transportation maps do not always exist. As such, the necessary reformulation of the question involves finding a coupling Γ, i.e. a measure on $\mathcal{X} \times \mathcal{X}$ such that the marginals of Γ are μ, ν respectively. For measures μ, ν, let $\mathcal{C}(\mu, \nu)$ to be the set of couplings. The new objective then is

$$\min \mathbb{E}_\Gamma \, c(X, Y) \qquad \textbf{such that } \Gamma \in \mathcal{C}(\mu, \nu). \tag{1.2}$$

This new problem, formulated by [11], behaves much better than the former in the discrete setting; in particular, under mild conditions on $\mathcal{X}$ and c (see [19] for details), the minimum is always attained and the theory of optimal transport begins here.

While the theoretical advances regarding optimal transport are certainly useful, the applications often require that one know the optimal transport plan, i.e., the coupling Γ that achieves the minimum in (1.2). While some work has been done in the case where at least one of μ, ν are continuous [13], we focus on the more classical and widely applied case where μ, ν are both discrete measures. In particular, if μ, ν are supported on atoms $x_1, \dots, x_m$ and $y_1, \dots, y_n$ all in $\mathcal{X}$ and have weights $w_1, \dots, w_m$ and $v_1, \dots, v_n$, then a coupling of μ and ν amounts to a matrix $\mathbf{P}$; as we shall see in the following section, the problem (1.2) reduces to one of linear programming and thus can be solved algorithmically.

In addition to the "vanilla" optimal transport problem described above, we also consider the more recent entropically regularized optimal transport. In this framework, described in detail below, a regularization term is added to the problem (1.2) to

penalize certain couplings. As first observed in [8], this penalization leads to significant computational advantages as the number of atoms in μ, ν get large. We consider two algorithms for computing this regularized problem: the sinkhorn algorithm proposed by [8] and the greenkhorn algorithm proposed by [1].

There are several implementations of computational optimal transport algorithms, the most used of which is POT: Python Optimal Transport [10]. In Julia, there are native implementations of some optimal transport algorithms in OptimalTransport.jl [21], although the computation of the vanilla, unregularized cost calls the Python implementation [10]. Using the latter as a benchmark, we compare our own implementations of sinkhorn and greenkhorn as well as our impelementation computing the unregularized cost.

The structure of the note is as follows. In the following section, we describe in greater detail the optimal transport problem as it is applied to discrete measures. We then proceed to discuss in detail the (currently optimal) algorithm for computing the optimal transport plan. After this, we give a brief overview of the entropically regularized optimal transport problem, before describing the two algorithms we implement. Finally, we conduct a thorough performance analysis of our implementations.

2. Computational Optimal Transport. In this section we dive into the details of how one might algorithmically compute the solution to problem (1.2). First, we fix notation. Suppose that μ is supported on m points x_i with weights given by the vector $\mathbf{a}$ and ν is supported on n points y_j with weights given by the vector $\mathbf{b}$. With μ, ν fixed, the optimal transport problem is given by a cost function c. We may restrict c to the domain $\{x_i\} \times \{y_j\}$ and so we form a matrix $\mathbf{C}$ where $\mathbf{C}_{ij} = c(x_i, y_j)$. Consider now that any coupling of μ, ν is supported on $\{x_i\} \times \{y_j\}$ and so can be described by an $m \times n$ matrix $\mathbf{P}$. Moreover, any $m \times n$ matrix with nonnegative entries whose row sums are $\mathbf{a}$ and whose column sums are $\mathbf{b}$ constitutes a coupling. An elementary calculation shows that the cost of a coupling $\mathbf{P}$ is given by $\langle \mathbf{P}, \mathbf{C} \rangle$. Let $\mathcal{U}(\mathbf{a}, \mathbf{b})$ be the set of all couplings of $\mathbf{a}, \mathbf{b}$. Then problem (1.2) becomes

$$\min \langle \mathbf{P}, \mathbf{C} \rangle \qquad \textbf{such that } \mathbf{P} \in \mathcal{U}(\mathbf{a}, \mathbf{b}) \tag{2.1}$$

Note that, because the constraints are expressed as row sum and column sum equalities and elementwise inequalities, problem (2.1) is reduced to a classical linear programming problem as described in detail in, for example, [4, 14, 3]. Naively, we might apply the simplex algorithm to problem (2.1); while this would work, it would be much slower than necessary as it does not take advantage of some of the special structure of the problem. Indeed, as we will see, the transport problem (2.1) (also known as the matching problem when $n = m$) is an instance of the minimum-cost flow problem, for which there exists an efficient adaptation of the simplex method known as the network simplex [3]. One may in fact simplify the network simplex algorithm even further if the problem at hand is (2.1).

To see that (2.1) is a flow-problem, we follow [16, Chapter 3] to introduce the graphical representation of the problem. We may build a bipartite graph from $\mathbf{C}$ as follows. Let the vertices be $1, \dots, m$ and $1', \dots, n'$ corresponding to the source and target distributions. We add an edge with weight $\mathbf{C}_{ij}$ between vertices i, j'. Then the feasibility conditions for $\mathbf{P}$ to be a transport plan are identical to the feasibility conditions for $\mathbf{P}$ to be a flow on the graph, where $\mathbf{P}_{ij}$ denotes the flow from vertex i to vertex j'. We have the following useful result that forms [6, Theorem 8.1.2]:

PROPOSITION 2.1. *Let $\mathcal{G}(\mathbf{P})$ be the bipartite graph with vertices $1, \dots, m$ and $1', \dots, n'$ and an edge between i and j' if and only if $\mathbf{P}_{ij} > 0$. If $\mathbf{P}$ is extremal*

in $\mathcal{U}(\mathbf{a}, \mathbf{b})$ *then* $\mathcal{G}(\mathbf{P})$ *is acyclic.*

The proof of this proposition can be found in [6] as well as in [16, Proposition 3.4]. The key conclusion to be drawn from Proposition 2.1 is one of sparsity: if $\mathcal{G}(\mathbf{P})$ is a graph with $m + n$ vertices but no cycles (also known as a *forest*), then it can have at most $m + n - 1$ edges. Thus the number of nonzero entries in an extremal $\mathbf{P}$ is at most $m + n - 1$. Note that a generic $\mathbf{P}$ has mn entries so, if m and n are of the same order, the number of nonzero entries drops from quadratic in the generic case to linear in the extremal case, a major improvement.

In the following sections we will describe the network simplex algorithm as applied to the transport problem. To do so, let us first remind ourselves of the basic structure of simplex algorithms. It is well known that linear programs achieve their optimum at a vertex of the feasible set [4]. Simplex algorithms proceed according to the following intuitive recipe: 1) Find a vertex of the constraint set. 2) While there exists a neighbouring vertex with better objective, set current vertex to a neighbour that improves the objective. Each vertex of the feasible set is defined by its active constraints, i.e. the inequality constraints where equality is achieved. When moving to a neighbouring vertex, one of these active constraints becomes inactive, and a new constraint becomes active. In terms of the primal and slack variables, this corresponds to a variable equal to zero becoming strictly positive (entering variable) and a previously positive variable becoming zero (leaving variable).

2.1. Initializing the Network Simplex. Before describing the algorithm itself, we are first concerned with the initialization. While it is nice to know that extremal points of the problem (2.1) are quite sparse, this theory does not help us find an extremal point with which to start the algorithm. While there are several initialization schemes, including Vogel's approximation and Russell's approximation (as detailed in [14]), we choose the simplest and the one recommended by [16]: the Northwest corner rule.

The Northwest corner rule works according to the following algorithm:

Algorithm 2.1 NorthWest Corner Rule

Data: $\mathbf{a}, \mathbf{b}$
Result: A vertex of the transport polytope $\mathcal{U}(\mathbf{a}, \mathbf{b})$
Initialize $\mathbf{P} \in \mathbb{R}^{m \times n}$ to all zeros
 Set row_sums $= (0, \ldots, 0) \in \mathbb{R}^m$ and col_sums $= (0, \ldots, 0) \in \mathbb{R}^n$
 Set $i = j = 1$
 while $i \leq m$, $j \leq n$ **do**
  if *row_sums*$[i] = \mathbf{a}_i$ **then**
   $i = i + 1$
  end
  if *col_sums*$[j] = \mathbf{b}_j$ **then**
   $j = j + 1$
  end
  $\mathbf{P}_{ij} = (\mathbf{a}_i - \text{row\_sums}[i]) \wedge (\mathbf{b}_j - \text{col\_sums}[j])$
   row_sums$[i]$ = row_sums$[i] + \mathbf{P}_{ij}$
   col_sums$[j]$ = col_sums$[j] + \mathbf{P}_{ij}$
end
Return $\mathbf{P}$

Intuitively, the algorithm starts at the top left (north west) corner of the matrix

and greedily makes $\mathbf{P}_{ij}$ as large as is feasible before either moving down the row or moving down the column. An easy inductive argument shows that the resulting $\mathbf{P}$ is feasible and it is clear from the construction that $\mathcal{G}(\mathbf{P})$ has no cycles. There are at most $m+n$ iterations so we have a fast algorithm that returns a vertex of the feasible set.

2.2. Network Simplex Iteration. In order to understand the network simplex iteration, we need to describe the dual problem of (2.1). Let $\mathcal{D}(\mathbf{C})$ denote the set of all pairs of vectors $(\mathbf{f}, \mathbf{g})$ such that $\mathbf{f}$ is length m, $\mathbf{g}$ is length n and for all i, j, $\mathbf{f}_i + \mathbf{g}_j \leq \mathbf{C}_{ij}$. Then duality of linear programming [4, 14] tell us that (2.1) is equivalent to solving the problem

$$\max\langle \mathbf{a}, \mathbf{f}\rangle + \langle \mathbf{b}, \mathbf{g}\rangle \qquad \textbf{such that } (\mathbf{f}, \mathbf{g}) \in \mathcal{D}(\mathbf{C}) \tag{2.2}$$

In fact, by the classical theory of linear programming, if $\mathbf{P}^*$ is the solution of (2.1) and $(\mathbf{f}^*, \mathbf{g}^*)$ is a solution of (2.2), then complementary slackness holds:

$$\mathbf{P}^*_{ij} \times (\mathbf{f}^*_i + \mathbf{g}^*_j - \mathbf{C}_{ij}) = 0. \tag{2.3}$$

Conversely, if $\mathbf{P} \in \mathcal{U}(\mathbf{a}, \mathbf{b})$ is a vertex, $(\mathbf{f}, \mathbf{g})$ is feasible for the dual problem and (2.3) is satisfied, then both $\mathbf{P}$ and $(\mathbf{f}, \mathbf{g})$ are optimal. Given a vertex $\mathbf{P}$ of the transport polytope, one may always construct $\mathbf{f} \in \mathbb{R}^m, \mathbf{g} \in \mathbb{R}^n$ such that (2.3) holds, which we call complementary dual variables/potentials. By the above, checking optimality of $\mathbf{P}$ reduces to checking feasibility of the complementary potentials.

Each iteration of the network simplex checks if the current vertex $\mathbf{P}$ is optimal by looking at whether complementary variables $(\mathbf{f}, \mathbf{g})$ are feasible and, if not, modifies $\mathbf{P}$ to make the corresponding new complementary potentials closer to being feasible. The algorithm is given below:

Algorithm 2.2 Network Simplex Update

Data: Current guess $\mathbf{P}$
Result: A better transport plan than $\mathbf{P}$
$(\mathbf{f}, \mathbf{g}) \leftarrow$complementary_dual(P)
if $(\mathbf{f}, \mathbf{g})$ *is not feasible* **then**
  while $\mathcal{G}(\mathbf{P})$ *has no cycles* **do**
    Add Edge (i, j') to $\mathcal{G}(\mathbf{P})$ where $\mathbf{f}_i + \mathbf{g}_j > \mathbf{C}_{ij}$
  end
  Find the unique cycle $L = (i_1, j'_1), (j'_1, i_2), \ldots, (j'_k, i_1)$
  $\theta = \max P_{j_l i_{l+1}}$
  $\mathbf{P}_{i_\ell j_\ell} = \mathbf{P}_{i_\ell j_\ell} + \theta$
  $\mathbf{P}_{j_\ell i_{\ell+1}} = \mathbf{P}_{j_\ell i_{\ell+1}} - \theta$
end
Return $\mathbf{P}$

The above algorithm presents several difficulties: finding complementary dual variables; checking feasibility; finding a cycle. The first is the easiest. We know that $(\mathbf{f}, \mathbf{g})$ is complementary to $\mathbf{P}$ if for all i, j such that $\mathbf{P}_{ij} > 0$, $\mathbf{f}_i + \mathbf{g}_j = \mathbf{C}_{ij}$. We similarly know that there are at most $m + n - 1$ such pairs i, j by the fact that $\mathbf{P}$ is a vertex of $\mathcal{U}(\mathbf{a}, \mathbf{b})$. Thus we have at most $m + n - 1$ linear equations in $m + n$ unknowns. To efficiently find $(\mathbf{f}, \mathbf{g})$, we may arbitrarily pick a vertex i^* and set $\mathbf{f}_{i^*} = 0$. Using a breadth first search, we may then fill in the rest of the $\mathbf{f}_i, \mathbf{g}_j$ in the

connected component of i^* in $\mathcal{G}(\mathbf{P})$ to what their values need to be in order to satisfy the complementarity conditions.

Checking feasibility is also easy in the sense that we may simply compare $\mathbf{f}_i + \mathbf{g}_j$ with $\mathbf{C}_{ij}$ to find a possible offending pair. We may then add this edge to the graph $\mathcal{G}(\mathbf{P})$ and a cycle can be found through an elementary search (see the next section for more details). The entries of $\mathbf{P}$ need to be nonnegative so the constraint is the minimal value in the cycle L of $\mathbf{P}_{i_\ell j_\ell}$ which can be found easily by traversing the cycle.

Putting the two pieces together, we initialize the network simplex with the North-West Corner Rule and then iterate the Network Simplex Update until the optimal solution is found. It was proved in [18] that, up to log factors, the runtime of the network simplex is $\widetilde{\mathcal{O}}\left((m+n)mn\right)$, which, if we take m, n to be of comparable order, is $\widetilde{\mathcal{O}}\left(n^3\right)$. One of the points made in [18] and further elucidated in [3, 16] is that, in order to achieve an efficient implementation, special data structures must be introduced to accelerate the network simplex' update step. This, and other practicalities of implementation are the subject of the following section.

2.3. Implementing Network Simplex. Due to the time constraints of this project, we weren't able to produce a state of the art implementation of network simplex. However, we have a somewhat inefficient implementation that we describe now. Note that due to the issue of degenerate updates and the selection of strongly feasible entering arcs (see [3, page 183]) on occasion our implementation gets stuck in a local minimum and doesn't produce the correct answer.

As explained in subsection 2.2, given an initial, feasible $\mathbf{P}$, the steps of the algorithm are

1. Construct dual solution $(\mathbf{f}, \mathbf{g})$
2. Check feasibility of $(\mathbf{f}, \mathbf{g})$
3. Choose entering variable
4. Choose leaving variable
5. Update $\mathbf{P}$.

Steps 2 and 3 are performed as one: we search over (i, j') for a violating pair $\mathbf{f}_i + \mathbf{g}_j > \mathbf{C}_{ij}$. For fastest convergence in terms of number of steps, one would intuitively choose the pair with maximal violation $\mathbf{f}_i + \mathbf{g}_j - \mathbf{C}_{ij}$. However, as this is the most time-intensive step, usually other schemes are used, such as block-search: one chooses the pair with maximal violation in only subset of the edges of size, say, $\sqrt{mn}$. If no violation is found, then optimality has been reached. We note that this is the only step of the algorithm that lends itself to efficient parallelization, although we did not pursue this.

Step 1 requires a search (e.g. DFS or BFS) over the forest $\mathcal{G}(\mathbf{P})$ and Step 4 requires finding the unique cycle in a graph. Therefore, the question of how one should store $\mathcal{G}(\mathbf{P})$ arises. There are simple and general ways one may represent a graph on a computer: adjacency matrices and adjacency lists. Both of these has trade-offs. For example, adjacency matrices require a lot of memory and finding neighbours of a given vertex is slow, but checking whether an edge exists in the graph and adding & removing edges is instantaneous. Adjacency lists have the advantage of quickly finding neighbouring vertices, but adding, deleting edges and checking whether an edge is present can be slow (see [20] for more details). For our implementation we chose to use adjacency sets. To each vertex there corresponds a set (implemented using hashmaps) containing the neighbouring vertices. This representation uses $\mathcal{O}(m+n)$ memory and iterating over neighbours is fast, a property crucial for fast search and

cycle finding.

Upon implementation however, we realised that the cost of finding cycles by simply searching the adjacency sets and maintaining those hash sets is prohibitive for the performance of the algorithm. The currently available fastest open source discrete optimal transport solver is [5], a C++ implementation based on the graph library LEMON [9, 12]. This is the code that POT [10] and OptimalTransport.jl [21] uses. We now briefly describe the optimized data structure that allows for the superior performance of [5]. For simplicity, let us forget about the special structure of the transport problem, and suppose that we simply have an (undirected) tree $\mathcal{G} = (V, E)$ that we want to store on a computer in a way that provides a fast way to:

1. Remove and add a new edge that maintains the tree structure
2. Find the unique path between two nodes
3. Iterate over a given subtree.

The last point is useful in Step 1 but we won't go into details for the sake of brevity. All of this can be achieved by maintaining 5 arrays of length $|V|$, that we call `parent, inorder, reverse_inorder, last_successor, number_of_successors.` Let us write $1, \dots, n = |V|$ for the vertices of $\mathcal{G}$. First one arbitrarily chooses a root for the tree, which we take to be 1 for simplicity. Then, `parent[i]` is the parent of node i where each edge is taken to point towards the root and `parent[1]=0. inorder[i]` gives the vertex that follows node i in the preorder traversal of the tree, with the last vertex pointing back to the root. `reverse_inorder[i]` is the inverse of `inorder,` it gives the node coming before i in the traversal. `last_successor[i]` gives the node in the subtree rooted at i that comes last in the preorder traversal. Finally, `number_of_succesors[i]` is equal to the size of the subtree rooted at i.

Miraculously, maintaining the tree structure (property 1 above) can be done in an efficient (albeit painful) way, see [2, 12] for the intricate details. Property 2 is trivial: given nodes $i, j \in V$, take steps towards the root one-by-one, always advancing the node with fewer successors, as provided by `number_of_successors.` Property 3 also follows, given a vertex $i \in V$ one takes $j =$`inorder[last_successor[i]]` and iterates $i =$`last_successor[i]` until they hit j.

Due to its complexity, we are still working on the implementation of the above data structure, also referred to as the XTI (Extended Threaded Index) method. Our code currently works for small problems, but there are some bugs not ironed out in the logic of Property 1.

3. Entropically Regularized Optimal Transport. While the theoretical importance of the optimal transport is certainly important, and the network simplex algorithm accelerates the computation of an optimal transport plan, we often wish to compute the optimal transport cost between empirical distributions supported on two data sets; as noted above, [18] proved that if nm then the runtime is cubic in the number of data points; worse, the above described algorithm is not immediately parallelizable and thus, even for mid-sized data sets, let alone large data sets, the run time is prohibitive. As a solution, [8] proposed the entropically regularized optimal transport cost.

In order to describe entropic regularization, we must first define the entropy. For a general distribution λ on $\mathbb{R}$, the entropy is $h(\lambda) = -\mathbb{E}_{X\sim\lambda}[\log X]$. Thus for $\mathbf{a}$ and $\mathbf{P}$, discrete, the definition reduces to

$$h(\mathbf{a}) = -\sum_{i=1}^{m} \log(\mathbf{a}_i)\mathbf{a}_i \qquad\qquad h(\mathbf{P}) = -\sum_{i,j} \log(\mathbf{P}_{ij})\mathbf{P}_{ij} \tag{3.1}$$

where expressions of the form $0 \log 0$ are taken to be zero. It is a classical fact of information theory [7], that if $\mathbf{P}$ is a coupling between $\mathbf{a}$ and $\mathbf{b}$, then

$$h(\mathbf{P}) \leq h(\mathbf{a}) + h(\mathbf{b}) \tag{3.2}$$

with equality if and only if $\mathbf{P}$ is the independent coupling $\mathbf{a} \otimes \mathbf{b}$. The entropically regularized optimal transport plan is defined as the solution to

$$\mathbf{P}_\eta = \operatorname{argmin} \langle \mathbf{C}, \mathbf{P} \rangle - \frac{1}{\eta} h(\mathbf{P}) \qquad \text{such that } \mathbf{P} \in \mathcal{U}(\mathbf{a}, \mathbf{b}) \tag{3.3}$$

Because h is strongly concave, the problem (3.3) has a unique solution. Moreover, intuition from the optimization literature would suggest that (3.3) can be solved more quickly than (2.1) due to the strong convexity of the objective; such intuition would be correct.

Let the relative entropy between two couplings be defined as

$$KL(\mathbf{P}||\mathbf{Q}) = \sum_{i,j} \mathbf{P}_{ij} \log \left(\frac{\mathbf{P}_{ij}}{\mathbf{Q}_{ij}} \right) \tag{3.4}$$

where, again, we declare $0 \log 0 = 0$. Then some easy algebra (detailed in [16, §4.1]) yields an equivalent way of viewing the solution to (3.3):

$$\mathbf{P}_\eta = \mathbf{Proj}_{\mathcal{U}(\mathbf{a},\mathbf{b})} \mathbf{K} = \operatorname{argmin}_{\mathbf{P} \in \mathcal{U}(\mathbf{a},\mathbf{b})} KL(\mathbf{P}||\mathbf{K}) \tag{3.5}$$

where $\mathbf{K}_{ij} = \exp(\eta \mathbf{C}_{ij})$. In fact, first order conditions on the Lagrangian tell us [16] that there are two nonnegative vectors $\mathbf{u}, \mathbf{v}$ such that $\mathbf{P}_{\eta_{ij}} = \mathbf{u}_i \mathbf{K}_{ij} \mathbf{v}_j$. Thus we wish to solve the coupled system of linear equations:

$$\mathbf{u} \mathbin{.*} (\mathbf{K}\mathbf{v}) = \mathbf{a} \qquad \mathbf{v} \mathbin{.*} (\mathbf{K}^t \mathbf{u}) = \mathbf{b} \tag{3.6}$$

where **.*** denotes element-wise multiplication. Note that (3.6) is the classical matrix scaling problem and can be solved via Sinkhorn iterations, as in [1, 8]:

$$\mathbf{u}^{(k+1)} = \mathbf{a} \mathbin{./} \mathbf{K}\mathbf{v}^{(k)} \qquad \mathbf{v}^{(k+1)} = \mathbf{b} \mathbin{./} \mathbf{K}^t \mathbf{u}^{(k+1)} \tag{3.7}$$

where **./** denotes element-wise division. We initialize the iterations with a positive vector. there are two problems with the above algorithm: the first is one of numerical instability and can be easily fixed by using logarithms, as suggested in [1]. The second problem is more serious: the resulting $\operatorname{diag}(\mathbf{u}) \mathbf{K} \operatorname{diag}(\mathbf{v})$ may not be a coupling of $\mathbf{a}$ and $\mathbf{b}$. As a result, we follow [1, Algorithm 2] and introduce a rounding step at the end:

Algorithm 3.1 Round ([1, Algorithm 2])

Data: $\mathbf{P}, \mathbf{a}, \mathbf{b}$
Result: Projection of $\mathbf{P}$ onto $\mathcal{U}(\mathbf{a}, \mathbf{b})$
for $i \in [m]$ **do**
  $x_i \leftarrow \mathrm{a}_i / \mathsf{rowsum}(\mathbf{P})_i \wedge 1$
end
for $j \in [n]$ **do**
  $y_j \leftarrow \mathrm{b}_j / \mathsf{colsum}(\mathbf{P})_j \wedge 1$
end
$F \leftarrow \operatorname{diag}(x) \mathbf{P} \operatorname{diag}(y)$
$\mathsf{err}_{row} \leftarrow$ a - rowsum(F)
$\mathrm{err}_{col} \leftarrow$ b - colsum(F)
Return $\mathrm{F} + \mathrm{err}_{row} \mathsf{err}_{col}^t / ||\mathsf{err}_{row}||_1$

Now, all that remains is to present the Sinkhorn algorithm formally:

Algorithm 3.2 Sinkhorn ([1, Algorithm 3])

Data: $\mathbf{K}, \mathbf{a}, \mathbf{b}$, number of iterations N
Result: Projection of $\mathbf{K}$ onto $\mathcal{U}(\mathbf{a}, \mathbf{b})$
$K^0 \leftarrow$K / norm(K)
$\mathrm{x}^0 \leftarrow 0, \mathrm{y}^0 \leftarrow 0$
for $k \in [N]$ **do**
  $x_i \leftarrow \log \mathbf{a}_i / \mathsf{rowsum}(K^{k-1})_i$ for $1 \le i \le n$
  $y_j \leftarrow \log \mathbf{b}_j / \mathsf{colsum}(K^{k-1})$ for $1 \le j \le n$
  $x^k \leftarrow \mathrm{x}^{k-1} + x$
  $y^k \leftarrow \mathrm{y}^{k-1} + y$
  $K^k \leftarrow \mathrm{diag}(\exp(x^k))\mathbf{K}\,\mathrm{diag}(\exp(y^k))$
end
Return K^N

On the theory side, we have [1, Theorem 1]:

THEOREM 3.1. *Combining the Sinkhorn algorithm with the rounding step yields a transport plan $\mathbf{P} \in \mathcal{U}(\mathbf{a}, \mathbf{b})$ that is an ϵ approximator for the true optimal transport plan in $O(n^2 \log n \epsilon^{-3})$ time, where $\mathbf{a}, \mathbf{b} \in \mathbb{R}^n$ are distributions each supported on n points.*

Recall that the network simplex optimization of the standard optimal transport plan is $O(n^3 \log n)$ and so the Sinkhorn algorithm presents a major theoretical improvement.

Before turning to the implementation side, we discuss one final algorithm which is a slight modification of Sinkhorn. Presented as [1, Algorithm 4], the authors propose a greedy version of Sinkhorn, where only the "worst" row and column of the transport plan are updated at each step. This reduces the updates from n^2 modifications per iteration to n modifications per iteration, resulting in a faster algorithm. In order to select which row or column is to be updated, the authors recommend using the divergence function ρ, where

$$\rho(a, b) = b - a + a \log \frac{a}{b} \tag{3.8}$$

Thus, we are left with the following algorithm:

Algorithm 3.3 Greenkhorn ([1, Algorithm 4])

Data: $\mathbf{K}, \mathbf{a}, \mathbf{b}$, number of iterations N
Result: Projection of $\mathbf{K}$ onto $\mathcal{U}(\mathbf{a}, \mathbf{b})$
$K^0 \leftarrow$K / norm(K)
$\mathrm{x}^0 \leftarrow 0, \mathrm{y}^0 \leftarrow 0$
for $k \in [N]$ **do**
  $I \leftarrow \operatorname{argmax}_i \rho(\mathbf{a}_i, \mathsf{rowsum}(K^{k-1})_i)$
  $J \leftarrow \operatorname{argmax}_j \rho(\mathbf{b}_j, \mathsf{colsum}(K^{k-1})_j)$
  if $\rho(\mathbf{a}_I, \mathsf{rowsum}(K^{k-1})_I) > \rho(\mathbf{b}_J, \mathsf{colsum}(K^{k-1})_J)$ **then**
    $x_I \leftarrow \mathrm{x}_I + \log \mathbf{a}_I / \mathsf{rowsum}(K^{k-1})_I$
  end
  else
    $y_J \leftarrow \mathrm{y}_J + \log \mathbf{b}_J / \mathsf{colsum}(K^{k-1})_J$
  end
  $K^k \leftarrow \mathrm{diag}(\exp(x^k))\mathbf{K}\,\mathrm{diag}(\exp(y^k))$
end
Return K^N

It is the content of [1, Theorem 3] that greenkhorn enjoys a similar convergence guarantee as sinkhorn. The major advantage of greenkhorn is that, because we are making many fewer updates per iteration, the algorithm is faster to run. As written, the algorithm can be implemented with only $O(n)$ computations per iteration; unfortunately, in many real world examples, the algorithm is numerically unstable unless the matrix K^k is renormalized to have ℓ^1 norm 1 at each step, a computation that is $O(n^2)$. Even so, greenkhorn runs more quickly than sinkhorn when n is large. A key advantage of the entropically regularized algorithms is that they are very amenable to parallelization with GPU support; due to time constraints, we opted to optimize a high performance implementation instead. We are now ready to discuss the implementation of the three above described algorithms.

4. Performance Analysis. In this section, we discuss our implementations of the above discussed algorithms. We begin with the network simplex optimizer used to find exact optimal transport plans before moving on to consider the entropic regularization.

4.1. Network Simplex Implementation. In all experiments below, we take μ, ν to be uniform distributions on m, n points respectively, with μ and ν supported on three dimensional i.i.d. Gaussian random variables. In what follows we always take $n = m$.

Recall that in Step 3 of the algorithm we have to search for an entering variable. Instead of checking every pair $(i, j') \in [n] \times [m]$, we apply block search: we only check pairs in a subset (block) of all pairs of size $\alpha(nm)$. To inform our choice of α we performed the experiment depicted in Figure 4.2. We took $n = m = 30$ and measured the average time to compute the optimal transport map on a random dataset, with $\alpha \in (0, 1)$ varying over a grid. Based on the results we proceed with $\alpha = 0.1$.

We now compare our implementation of Network Simplex specialized to the transport problem to the fastest open source optimal transport solver. In Figure 4.1 we see a comparison of running times on log-nanosecond scale, while 4.3 depicts the memory allocation of the two algorithms throughout the experiment on log-gigabyte scale. From 4.1 we see that our implementation is substantially slower than best possible.

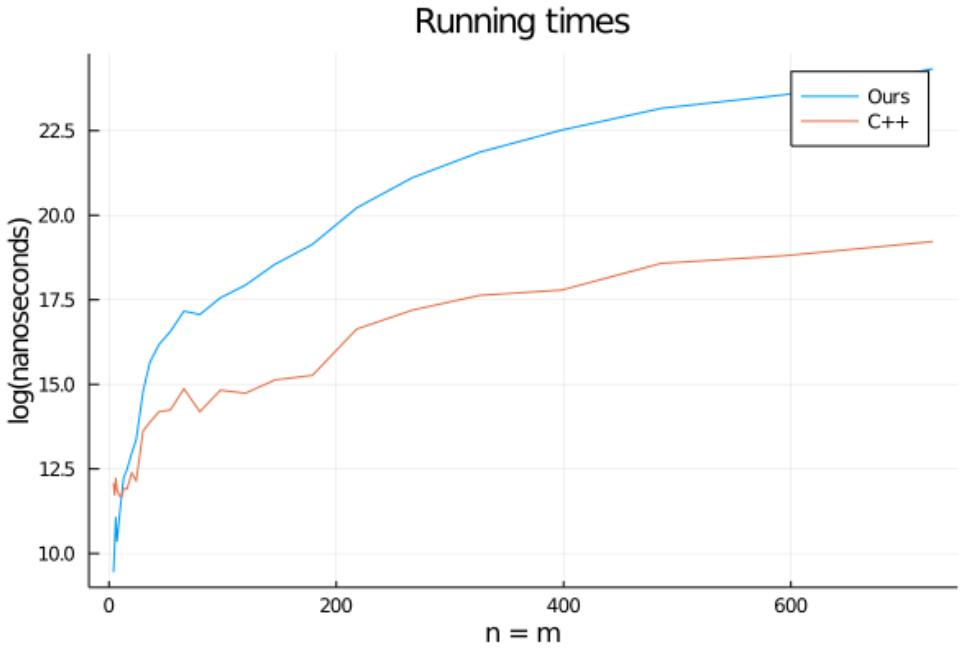

FIG. 4.1. *Comparison of running times between our implementation of the Network Simplex for the transportation problem, and the fastest open source implementation [5] (C++ code called by OptimalTransport.jl).*

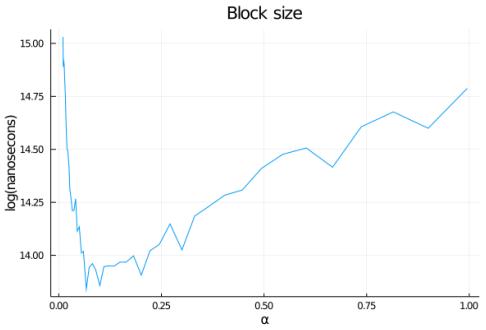

FIG. 4.2. *Comparison of different choices of block-size in Step 3.*

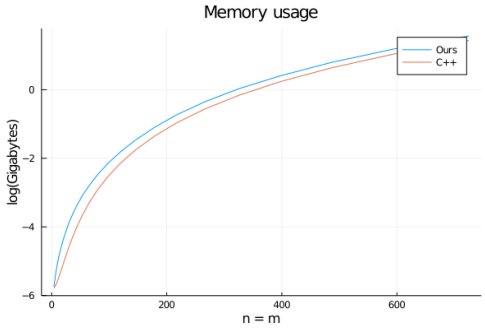

FIG. 4.3. *Comparison of memory usage with [5].*

As we elaborated in Section 2.3, this is due to our sub-optimal representation of the underlying tree structure which substantially slows Step 4. However, from Figure 4.3 we see that our implementation is memory efficient, achieving the same memory usage as [5] up to a small constant factor.

4.2. Entropic Regularization. We compare our sinkhorn and greenkhorn implementations to the sinkhorn implementation of [21], using the latter as a baseline. In order to produce a consistent benchmark, we let μ, ν be uniform distributions on n points, where n varies throughout the discussion. The n points are in $\mathbb{R}^3$, drawn

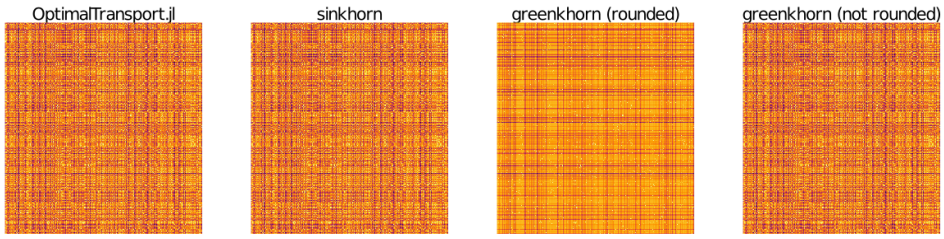

FIG. 4.4. *Transport Plans found by the* sinkhorn *implementations of [21] and us, as well as rounded and unrounded plans found by* greenkhorn*. The figures are heatmaps of the resulting matrices on a log scale. Darker colors denote entries that are smaller. In this example,* $n = 500$.

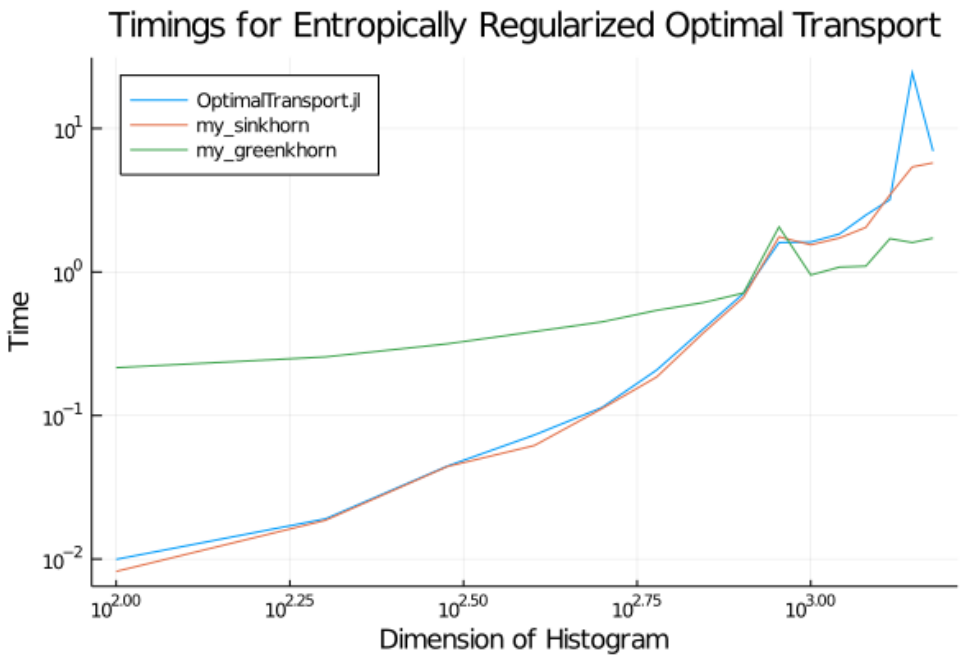

FIG. 4.5. *Timing for* n *varying from 100 to 1500 under the model described above of the baseline* sinkhorn *from [21], our implementation of* sinkhorn *and our implementation of* greenkhorn*, plotted on a log-log scale.*

independently from a standard multivariate Gaussian distribution.

As a first example, we plot what the transport plans actually look like in the above model for $n = 500$ (Figure 4.4). The error between each of the transport plans and the baseline is acceptably small. In our implementation, we exhibit the effect of the rounding step described above. Note that a major difference between transport plans found exactly and those found with the entropic regularization is the lack of sparsity in the latter.

In Figure 4.5, we compare the runtime of two different implementations of sinkhorn (that of [21] and ours) and our implementation of greenkhorn for n between 100 and 1500, $\epsilon = .05$, all plotted on a log scale. We note that as predicted, for large n (more

than about 10^3), the greenkhorn implementation is faster.

Note that our implementation of sinkhorn is essentially identical in timing to the benchmark of [21]. While algorithm 3.2, described above is useful for theoretical analysis, as conducted in [1], the formulation of (3.7) is more practical for high performance implementation. In particular, we do in-place multiplications and divisions for both $u^{(k+1)}$ and $v^{(k+1)}$ in (3.7); this is the bottleneck and our implementation is likely close to optimal, as demonstrated by how close in time our implementation is to the optimum. While it may appear that our implementation is better, the small overhead in the benchmark comes from the fact that [21] implements an early stopping rule that checks if the error falls below some tolerance; in order to compare our implementation to that of [21], we set the tolerance to 10^{-40} to stop it from cutting off early. Note that modifying our implementation to account for early stopping is trivial, but is not relevant to the performance analysis. Due to the in-place multiplication, our implementation of sinkhorn has very little memory allocation, which is part of what gives us the high performance.

Regarding the greenkhorn implementation, we manage to achieve high performance in two ways, both of which are associated to not redundantly computing row and column sums. Our implementation allows the practitioner to choose between renormalizing the coupling at each time step or not. In the former case, we have an $O(n^2)$ update step which certainly increases runtime. In either case, however, we do not need to recalculate the row- and column-sums of the current iterate from scratch, computations which are $O(n^2)$. Instead, we modify the row- and column-sums in-place with an $O(n)$ operation. To do this, we note that if we are modifying, for example, row I by a scaling factor, then only the I^{th} row-sum is changed, scaled by a factor of I. For the j^{th} column-sum, then, we may simply subtract off the old A_{Ij}^{k-1} and add the new A_{Ij}^{k} for each $1 \leq j \leq n$; a similar computation can be done when we update columns instead of rows. Thus, without renormalizing the A^k, we are able to significantly accelerate the iteration step of greenkhorn, as suggested in [1]. In the event that we *are* renormalizing the iterates, which involves an $O(n^2)$ operation, we are still able to do a similar update to the row- and column-sums in the accelerated way, which was not suggested by [1]. Unfortunately, the code supplement of [1] is implemented in MATLAB and so we are unable to directly compare our greenkhorn to theirs, although we suspect that, given the accelerated iteration step, our implementation would be marginally faster.

Finally, to compare the sinkhorn and greenkhorn algorithms, we note that both are attempting to project a known matrix $K = \exp.\left(-\frac{C}{\epsilon}\right)$ onto a feasible set $\mathcal{U}(\mathbf{a}, \mathbf{b})$. Thus, we compare in Figure 4.6 how sinkhorn and greenkhorn compare in how quickly they get close to the feasible set, with distance measured by

$$\text{distance}(\mathbf{P}, \mathcal{U}(\mathbf{a}, \mathbf{b})) = ||\text{rowsum}(\mathbf{P}) - a||_1 + ||\text{colsum}(\mathbf{P}) - \mathbf{b}||_1 \tag{4.1}$$

As suggested by the empirical results of [1], greenkhorn performs much better than sinkhorn and thus our high performance implementation of the former is likely more practical than that of the latter, especially for large n.

5. Conclusion. In this note, we discussed the important problem of optimal transport from an algorithmic perspective. Due to the slow runtime of the network simplex on large histograms, we then reviewed the problem of optimal transport with entropic regularization, which allows for significant acceleration. We then implemented the network simplex, accelerated by a certain data structure known as an XTI as well as a high performance implementation of both sinkhorn and greenkhorn.

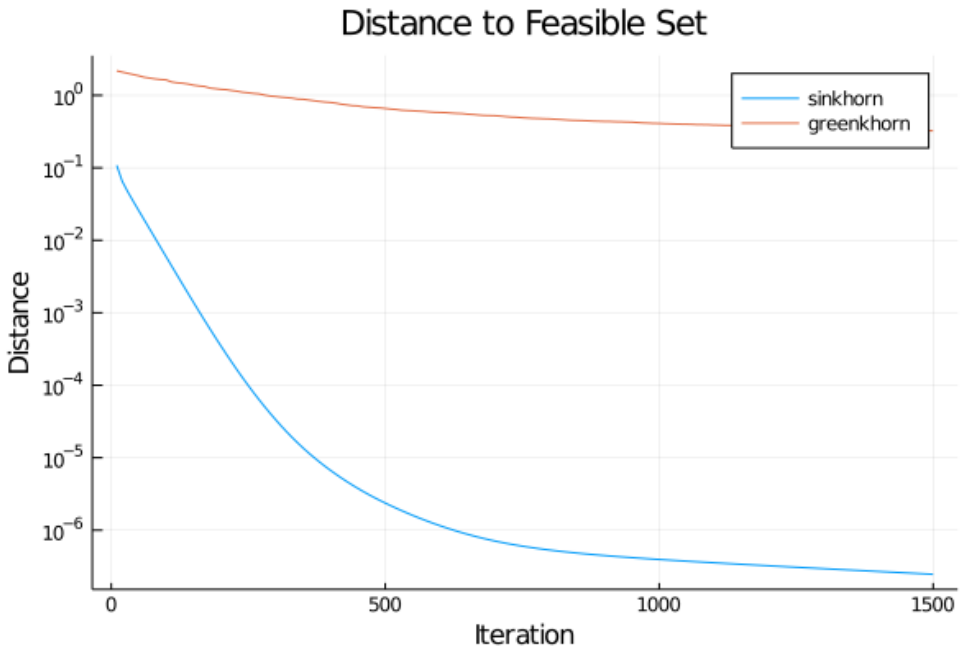

FIG. 4.6. *Distance from the k^{th} iterate of* sinkhorn *and* greenkhorn *to the feasible set $\mathcal{U}(\mathbf{a}, \mathbf{b})$plotted with the distance on the log scale in the standard comparison model described above with $n = 500$ and $\epsilon = .05$.*

We then conducted a thorough performance analysis, noting where the bottleneck is and discussing the steps taken to accelerate the code.

REFERENCES

[1] J. ALTSCHULER, J. NILES-WEED, AND P. RIGOLLET, *Near-linear time approximation algorithms for optimal transport via sinkhorn iteration*, Advances in neural information processing systems, 30 (2017), pp. 1964–1974.

[2] R. BARR, F. GLOVER, AND D. KLINGMAN, *Enhancements of spanning tree labelling procedures for network optimization*, INFOR: Information Systems and Operational Research, 17 (1979), pp. 16–34.

[3] D. P. BERTSEKAS, *Network optimization: continuous and discrete models*, Athena Scientific Belmont, MA, 1998.

[4] D. BERTSIMAS AND J. N. TSITSIKLIS, *Introduction to linear optimization*, vol. 6, Athena Scientific Belmont, MA, 1997.

[5] N. BONNEEL, M. VAN DE PANNE, S. PARIS, AND W. HEIDRICH, *Fast network simplex for optimal transport*, 2018.

[6] R. A. BRUALDI, *Combinatorial matrix classes*, vol. 13, Cambridge University Press, 2006.

[7] T. M. COVER, *Elements of information theory*, John Wiley & Sons, 1999.

[8] M. CUTURI, *Sinkhorn distances: Lightspeed computation of optimal transport*, in Advances in neural information processing systems, 2013, pp. 2292–2300.

[9] B. DEZSŐ, A. JÜTTNER, AND P. KOVÁCS, *Lemon–an open source c++ graph template library*, Electronic Notes in Theoretical Computer Science, 264 (2011), pp. 23–45.

[10] R. FLAMARY AND N. COURTY, *Pot python optimal transport library*, 2017, https://pythonot.github.io/.

[11] L. V. KANTOROVICH, *On the translocation of masses*, Journal of Mathematical Sciences, 133 (2006), pp. 1381–1382.

[12] Z. KIRÁLY AND P. KOVÁCS, *Efficient implementations of minimum-cost flow algorithms*, arXiv

preprint arXiv:1207.6381, (2012).
[13] B. Lévy and E. L. Schwindt, *Notions of optimal transport theory and how to implement them on a computer*, Computers & Graphics, 72 (2018), pp. 135–148.
[14] G. J. Lieberman and F. S. Hillier, *Introduction to operations research*, McGraw-Hill, 2005.
[15] G. Monge, *Mémoire sur la théorie des déblais et des remblais*, Histoire de l'Académie Royale des Sciences de Paris, (1781).
[16] G. Peyré, M. Cuturi, et al., *Computational optimal transport: With applications to data science*, Foundations and Trends® in Machine Learning, 11 (2019), pp. 355–607.
[17] F. Santambrogio, *Optimal transport for applied mathematicians*, Birkäuser, NY, 55 (2015), p. 94.
[18] R. E. Tarjan, *Dynamic trees as search trees via euler tours, applied to the network simplex algorithm*, Mathematical Programming, 78 (1997), pp. 169–177.
[19] C. Villani, *Optimal transport: old and new*, vol. 338, Springer Science & Business Media, 2008.
[20] Wikipedia contributors, *Graph (abstract data type) — Wikipedia, the free encyclopedia*, 2020, https://en.wikipedia.org/w/index.php?title=Graph_(abstract_data_type)&oldid=993216221. [Online; accessed 12-December-2020].
[21] S. Zhang, *Optimaltransport.jl*, Mar. 2020, https://github.com/zsteve/OptimalTransport.jl.