ReLU Neural Networks of Polynomial Size for Exact Maximum Flow Computation*

Christoph Hertrich[1][0000−0001−5646−8567] and Leon Sering[2][0000−0003−2953−1115]

[1] London School of Economics and Political Science, UK
c.hertrich@lse.ac.uk
[2] ETH Zurich, Switzerland
sering@math.ethz.ch

Abstract. This paper studies the expressive power of artificial neural networks with rectified linear units. In order to study them as a model of *real-valued* computation, we introduce the concept of *Max-Affine Arithmetic Programs* and show equivalence between them and neural networks concerning natural complexity measures. We then use this result to show that two fundamental combinatorial optimization problems can be solved with polynomial-size neural networks. First, we show that for any undirected graph with n nodes, there is a neural network (with fixed weights and biases) of size $\mathcal{O}(n^3)$ that takes the edge weights as input and computes the value of a minimum spanning tree of the graph. Second, we show that for any directed graph with n nodes and m arcs, there is a neural network of size $\mathcal{O}(m^2n^2)$ that takes the arc capacities as input and computes a maximum flow. Our results imply that these two problems can be solved with strongly polynomial time algorithms that solely uses affine transformations and maxima computations, but no comparison-based branchings.

Keywords: Neural Network Expressivity · Strongly Polynomial Algorithms · Minimum Spanning Tree Problem · Maximum Flow Problem.

1 Introduction

Artificial neural networks (NNs) achieved breakthrough results in various application domains like computer vision, natural language processing, autonomous driving, and many more [40]. Also in the field of combinatorial optimization (CO), promising approaches to utilize NNs for problem solving or improving classical solution methods have been introduced [7]. However, the theoretical understanding of NNs still lags far behind these empirical successes.

All neural networks considered in this paper are *feedforward neural networks with rectified linear unit (ReLU) activations*, one of the most popular models in practice [19]. These NNs are directed, acyclic, computational graphs in which each edge is equipped with a fixed weight and each node with a fixed bias. Each node (*neuron*) computes an affine transformation of the outputs of its

* The full version is available on arXiv: https://arxiv.org/abs/2102.06635.

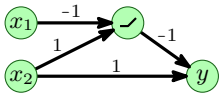

Fig. 1: A small NN with two input neurons $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$, a single ReLU neuron labelled with the shape of the ReLU function, and one output neuron $\boldsymbol{y}$. It computes the function

$$\begin{aligned}\boldsymbol{x} &\mapsto \boldsymbol{y}\\ &= \boldsymbol{x}_2 - \max\{0, \boldsymbol{x}_2 - \boldsymbol{x}_1\}\\ &= -\max\{-\boldsymbol{x}_2, -\boldsymbol{x}_1\}\\ &= \min\{\boldsymbol{x}_1, \boldsymbol{x}_2\}.\end{aligned}$$

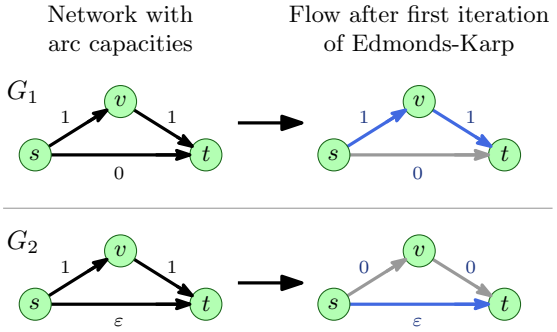

Fig. 2: This example shows that the outcome of one iteration of the Edmonds-Karp algorithm for computing a maximum flow depends discontinuously on the arc capacities. Here, a small adjustment of the capacity of arc st leads to a drastic change of the flow after the first iteration.

predecessors and applies the ReLU activation function $x \mapsto \max\{0, x\}$ on top. The full NN then computes a function mapping real-valued inputs to real-valued outputs. A simple example is given in Figure 1.

The neurons are commonly organized in *layers*. The *depth*, *width*, and *size* of an NN are defined as the number of layers, the maximum number of neurons per layer, and the total number of neurons, respectively. An important theoretical question about these NNs is concerned with their expressivity: which functions can be represented by an NN of a certain depth, width, or size?

Neural network expressivity has been thoroughly investigated from an approximation point of view. For example, so-called *universal approximation theorems* [3, 11, 31] show that every continuous function on a bounded domain can be arbitrarily well approximated with only a single nonlinear layer. However, for a full theoretical understanding of this fundamental machine learning model it is necessary to understand what functions can be *exactly* expressed with different NN architectures. For instance, insights about exact representability have boosted our understanding of the computational complexity of the task to train an NN with respect to both, algorithms [4, 36] and hardness results [9, 18, 20]. It is known that a function can be expressed with a ReLU NN if and only if it is *continuous and piecewise linear* (CPWL) [4]. However, many surprisingly basic questions remain open. For example, it is not known whether two layers of ReLU units (with any width) are sufficient to compute the function $f\colon \mathbb{R}^4 \to \mathbb{R}$, $x \mapsto \max\{0, x_1, x_2, x_3, x_4\}$ [24, 28].

In this paper we explore another fundamental question within the research stream of exact representability: what are families of CPWL functions that can be represented with ReLU NNs of polynomial size? In other words, using NNs as a model of computation operating on *real* numbers (in contrast to Turing ma-

chines or Boolean circuits, which operate on binary encodings), which problems do have polynomial complexity in this model?

Our motivation to study this model stems from a variety of different perspectives, including strongly polynomial time algorithms, arithmetic circuit complexity, parallel computation, and learning theory. We believe that classical combinatorial optimization problems are a natural example to study this model of computation because their algorithmic properties are well understood in each of these areas.

Clearly, if there are polynomial-size NNs to solve a certain problem, and assuming that the weights of these NNs are computable in polynomial time[3], then there exists a strongly polynomial time algorithm for that problem, simply by executing the NN. However, the converse might be false. This is due to the fact that ReLU NNs only allow a very limited set of possible operations, namely affine combinations and maxima computations. In particular, every function computed by such NNs is continuous, making it impossible to realize instructions like a simple **if**-branching based on a comparison of real numbers. In fact, there are related models of computation for which the use of branchings is exponentially powerful [32].

For some CO problems, classical algorithms do not involve comparison-based branchings and, thus, can easily be implemented as an NN. This is, for example, true for many dynamic programs. In these cases, the existence of efficient NNs follows immediately. We refer to Hertrich and Skutella [29] for some examples of this kind. In particular, polynomial-size NNs to compute the length of a shortest path in a network from given arc lengths are possible.

For other problems, like the Minimum Spanning Tree Problem or the Maximum Flow Problem, all classical algorithms use comparison-based branchings. For example, many maximum flow algorithms use them to decide whether an arc is part of the *residual network*. More specifically, in the Edmonds-Karp algorithm a slight perturbation (from 0 to ε) in the capacities can lead to different augmenting path and therefore to a completely different intermediate flow; see Figure 2. Such a discontinuous behavior can never be represented by a ReLU NN.

1.1 Our Main Results

In order to make it easier to think about NNs in an algorithmic way, we introduce the pseudo-code language *Max-Affine Arithmetic Programs* (MAAPs). We show that MAAPs and NNs are equivalent (up to constant factors) concerning three basic complexity measures corresponding to depth, width, and overall size of NNs. Hence, MAAPs serve as a convenient tool for constructing NNs with bounded size and could be useful for further research about NN expressivity beyond the scope of this paper.

We use this result to prove our two main theorems. The first one shows that computing the value of a minimum spanning tree has polynomial complexity on NNs. The proof is based on a result from subtraction-free circuit complexity [17].

[3] In circuit complexity language, one would say "if there is a uniform neural network family to solve a certain problem".

Theorem 1. *For a fixed graph with n vertices, there exists an NN of depth $\mathcal{O}(n \log n)$, width $\mathcal{O}(n^2)$, and size $\mathcal{O}(n^3)$ that correctly maps a vector of edge weights to the value of a minimum spanning tree.*

The second result shows that computing a maximum flow has polynomial complexity on NNs. Since all classical algorithms involve conditional branchings based on the comparison of real numbers, the proof involves the development of a new strongly polynomial maximum flow algorithm which avoids such branchings. While, in terms of standard running times, the algorithm is definitely not competitive with algorithms that exploit comparison-based branchings, it is of independent interest with respect to the structural understanding of flow problems.

Theorem 2. *Let $G = (V, E)$ be a fixed directed graph with $s, t \in V$, $|V| = n$, and $|E| = m$. There exists an NN of depth and size $\mathcal{O}(m^2 n^2)$ and width $\mathcal{O}(1)$ that correctly maps a vector of arc capacities to a vector of flow values in a maximum s-t-flow.*

Let us point out that in case of minimum spanning trees, the NN computes only the objective value, while for maximum flows, the NN computes the actual solution. There is a structural reason for this difference: Due to their continuous nature, ReLU NNs cannot compute a discrete solution vector, like an indicator vector of the optimal spanning tree, because infinitesimal changes of the edge weights would lead to jumps in the output. For the Maximum Flow Problem, however, the optimal flow itself does indeed have a continuous dependence on the arc capacities.

1.2 Discussion of the Results

Before presenting our result in more detail, we discuss the significance and limitations of our results from various perspectives. Due to space constraints, we refer to the full version for a more detailed discussion.

Learning Theory. A standard approach to create a machine learning model usually contains the following two steps. The first step is to fix a particular *hypothesis class*. When using NNs, this means to fix an architecture, that is, the underlying graph of the NN. Then, each possible choice of weights and biases of all affine transformations in the network constitutes one hypothesis in the class. The second step is to run an optimization routine to find a hypothesis in the class that fits given training data as accurately as possible. A core theme in learning theory is to analyse how the choice of the hypothesis class influences different kind of errors made by the machine learning model.

While there exist many attempts to mathematically explain the mysterious success of modern NNs [8], there is still a long way ahead of us. Understanding what CPWL functions are actually contained in the hypothesis classes defined by NNs of a certain size (in particular, polynomial size) is a key insight in this direction. We see our combinatorial, exact perspective as a counterbalance and complement to the usual approximate point of view.

Strongly Polynomial Time Algorithms. As pointed out above, polynomial-size NNs correspond to a subclass of strongly polynomial time algorithms with a very limited set of operations allowed. Given that this subclass stems from one of the most basic machine learning models, our grand vision, to which we contribute with our results, is to understand for different CO problems whether they admit strongly polynomial time algorithms of this type.

Algorithms of this type have not been known before for the two problems considered in this paper. It remains an open question whether such algorithms, and hence, polynomial-size NNs, exist to solve other CO problems for which strongly polynomial time algorithms are known. Can they, for instance, compute the weight of a minimum weight perfect matching in (bipartite) graphs? Can they compute the cost of a minimum cost flow from either node demands or arc costs, while the other of the two quantities is considered to be fixed?

A major open question is also to prove lower bounds on NN sizes. Can we find a family of CPWL functions (corresponding to a CO problem or not) that can be evaluated in strongly polynomial time, but *not* computed by polynomial-size NNs? While proving lower bounds in complexity theory always seems to be a challenging task, we believe that not all hope is lost. For example, in the area of *extended formulations*, it has been shown that there exist problems (in particular, minimum weight perfect matching) which can be solved in strongly polynomial time, but every linear programming formulation to this problem must have exponential size [52]. Possibly, one can show in the same spirit that also polynomial-size NN representations are not achievable.

Boolean Circuits. Even though NNs are naturally a model of real computation, it is worth to have a look at their computational power with respect to Boolean inputs. Interestingly, this makes understanding the computational power of NNs much easier. It is easy to see that ReLU NNs can directly simulate AND-, OR-, and NOT-gates, and thus every Boolean circuit [44]. Hence, in Boolean arithmetics, every problem in P can be solved with polynomial-size NNs.

However, requiring the networks to solve a problem for all possible real-valued inputs seems to be much stronger. Consequently, the class of functions representable with polynomial-size NNs is much less understood than in Boolean arithmetics. Our results suggest that rethinking and forbidding basic algorithmic paradigms (like comparison-based branchings) can help towards improving this understanding.

Arithmetic Circuits. As a circuit model with real-valued computation, ReLU networks are naturally closely related to *arithmetic circuits*. Just like NNs, arithmetic circuits are computational graphs in which each node computes some arithmetic expression (traditionally addition or multiplication) from the outputs of all its predecessors. Arithmetic circuits are well-studied objects in complexity theory [56]. Closer to ReLU NNs, there is a special kind of arithmetic circuits called *tropical circuits* [33]. In contrast to ordinary arithmetic circuits, they contain maximum (or minimum) gates instead of sum gates and sum gates instead of product gates. Thus, they are arithmetic circuits in the max-plus algebra.

A tropical circuit can be simulated by an NN of roughly the same size since NNs can compute maxima and sums. However, neural networks are strictly more powerful than tropical circuits for two reasons: they can realize subtractions (that is, tropical division) by using negative weights and scalar multiplication (tropical exponentiation) with any real number. Thus, lower bounds on the size of tropical circuits do not apply to NNs. A particular example with an exponential gap between NNs and tropical circuits is the computation of the value of a minimum spanning tree. By Jukna and Seiwert [34], no polynomial-size tropical circuit can do this. However, Theorem 1 shows that NNs of cubic size (in the number of nodes of the input graph) are sufficient for this task.

Parallel Computation. Neural networks are naturally a model of parallel computation by performing all operations within one layer at the same time. Without going into detail here, the depth of an NN is related to the running time of a parallel algorithm, its width is related to the required number of processing units, and its size to the total amount of work conducted by the algorithm. One takeaway from this perspective is that, although the result by Arora et al. [4] guarantees that logarithmic depth should be sufficient to compute a maximum flow, this would probably require superpolynomial width and size. The reason is that the Maximum Flow Problem is *P-complete* [22,23], meaning that it probably cannot be efficiently parallelized.

1.3 Further Related Work

Using NNs to solve optimization problems started with so-called *Hopfield networks* in the 1980s [30, 35, 57], which has also been specialized to the Maximum Flow Problem [2, 14, 45]. However, the NNs used in these works are conceptually very different from modern feedforward NNs that are considered in this paper.

In recent years interactions between NNs and CO have regained a lot of attention in the literature [7], for example, for boosting MIP solvers [42] and solving specific CO problems [6, 16, 37, 38, 47, 60]. These approaches usually are of heuristic nature without quality or running time guarantees.

Concerning the expressivity of ReLU neural networks, various trade-offs between depth and width of NNs [4,15,25,27,41,46,51,53,58,59,62] and approaches to count and bound the number of linear regions of a ReLU NN [26,43,50,51,54] have been found. NNs have been studied from a circuit complexity point of view before [5, 49, 55]. However, these works focus on Boolean circuit complexity of NNs with sigmoid or threshold activation functions. We are not aware of previous work investigating the computational power of ReLU NNs as arithmetic circuits operating on the real numbers.

For an introduction to classical minimum spanning tree and maximum flow algorithms, we refer to textbooks [1, 39, 61]. The asymptotically fastest known combinatorial maximum flow algorithm due to Orlin [48] runs in $\mathcal{O}(nm)$ time for n nodes and m arcs. Recently, almost linear, weakly polynomial algorithms based on interior point methods have been developed [10]. However, polynomial-size NNs necessarily correspond to strongly polynomial algorithms.

2 Algorithms and Proof Overview

In this section we provide an intuitive overview of how we prove our results. The detailed proofs are deferred to the full version due to space constraints.

Max-Affine Arithmetic Programs. For the purpose of algorithmic investigations of ReLU NNs, we introduce the pseudo-code language *Max-Affine Arithmetic Programs* (MAAPs). A MAAP operates on real-valued variables. The only operations allowed in a MAAP are computing maxima and affine transformations of variables as well as parallel and sequential **for** loops with a *fixed*[4] number of iterations. In particular, no **if** branchings are allowed. With a MAAP A, we associate three complexity measures $d(A)$, $w(A)$, and $s(A)$, which can easily be calculated from a MAAP's description. The intuition behind these measures is that they correspond (up to constant factors) to the depth, width, and size of an NN computing the same function as the MAAP does. We formalize this intuition by proving the following proposition, which is similar to the transformation of circuits into *straight-line programs* in Boolean or arithmetic circuit complexity.

Proposition 3. *For a function $f\colon \mathbb{R}^n \to \mathbb{R}^m$ the following is true.*

(i) If f can be computed by a MAAP A, then it can also be computed by an NN with depth $d(A)+1$, width $w(A)$, and size $s(A)$.
(ii) If f can be computed by an NN with depth $d+1$, width w, and size s, then it can also be computed by a MAAP A with $d(A) = d$, $w(A) = 2w$, and $s(A) = 4s$.

The proof of the proposition works by providing explicit constructions to convert a MAAP into an NN (part (i)), and vice versa (part (ii)) while taking care that the different complexity measures translate respectively.

The takeaway from this exercise is that for proving that NNs of a certain size can compute certain functions, it is sufficient to develop an algorithm in the form of a MAAP that computes the same function and to bound its complexity measures $d(A)$, $w(A)$, and $s(A)$.

Minimum Spanning Trees. A spanning tree in an undirected graph is a set of edges that is connected, spans all vertices, and does not contain any cycle. For given edge weights, the Minimum Spanning Tree Problem is to find a spanning tree with the least possible total edge weight.

Classical algorithms for the Minimum Spanning Tree Problem, for example Kruskal's or Prim's algorithm, compare the edge weights and use comparison-based branchings to determine the order in which edges are added to the solution. Thus, they cannot be written as a MAAP or implemented as an NN. Instead, Theorem 1 can be shown by translating an arithmetic circuit with additional

[4] In this context, *fixed* means that the number of iterations cannot depend on the specific instance. It can still depend on the size of the instance (e.g., the size of the graph in case of the two CO problems considered in this paper).

Algorithm 1: MST_n: Compute the value of a minimum spanning tree for the complete graph on $n \geq 3$ vertices.

Input: Edge weights $(\boldsymbol{x}_{ij})_{1\leq i<j\leq n}$.

$\boldsymbol{y}_n \leftarrow \min_{i\in[n-1]} \boldsymbol{x}_{in}$
for each $1 \leq i < j \leq n-1$ **do parallel**
  $\boldsymbol{x}'_{ij} \leftarrow \min\{\, \boldsymbol{x}_{ij},\ \boldsymbol{x}_{in} + \boldsymbol{x}_{jn} - \boldsymbol{y}_n \,\}$
return $\boldsymbol{y}_n + \mathrm{MST}_{n-1}\left((\boldsymbol{x}'_{ij})_{1\leq i<j\leq n-1}\right)$

division gates by Fomin et al. [17] to a tropical circuit with additional subtraction gates. We refer to the full version for more details.

While this tropicalization is already sufficient to justify the existence of polynomial-size NNs to compute the value of a minimum spanning tree, to unveil the algorithmic ideas behind this construction, we provide an equivalent, completely combinatorial proof of Theorem 1, making use of MAAPs and Proposition 3.

Without loss of generality, we restrict ourselves to complete graphs. Edges missing in the actual input graph can be represented with large weights such that they will never be included in a minimum spanning tree. For $n = 2$ vertices, the MAAP simply returns the weight of the only edge of the graph. For $n \geq 3$, our MAAP is given in Algorithm 1.

Let us mention that the use of recursions is just a technicality because for each fixed n, the recursion can be unrolled and the MAAP can be stated explicitly. In each step, one node of the graph is deleted and all remaining edge weights are updated in such a way that the objective value of the minimum spanning tree problem in the original graph can be calculated from the objective value in the smaller graph. This idea of removing the vertices one by one can be seen as the translation of the so-called *star-mesh transformation* used by Fomin et al. [17] into the combinatorial world.

We prove Theorem 1 in the full version by, firstly, showing that Algorithm 1 indeed computes the correct objective value, and secondly, bounding its complexity measures $d(A)$, $w(A)$, and $s(A)$ and applying Proposition 3.

Maximum Flows. For a given directed graph with a source node s, a sink node t, and nonnegative capacities on each arc, the Maximum Flow Problem asks to find a flow value for each arc such that no capacity is exceeded, the inflow equals the outflow at each node except for s and t, and the outflow at s (or equivalently the inflow at t) is maximised.

Since classical maximum flow algorithms rely on conditional branchings based on the comparison of real numbers (for instance, to check which arcs are contained in the residual network), we develop a new maximum flow algorithm in the form of a MAAP (see Algorithms 2 and 3), which then translates to an NN of the claimed size by Proposition 3. In the description of the algorithm, we assume without loss of generality that for each arc $e = uv \in E$ also its reverse arc vu

Algorithm 2: Compute a maximum flow for a fixed graph $G = (V, E)$.

Input: Capacities $(\boldsymbol{\nu}_e)_{e \in E}$.

// Initializing:
for each $uv \in \vec{E}$ **do parallel**
  $\boldsymbol{x}_{uv} \leftarrow 0$ // flow; negative value correspond to flow on vu
  $\boldsymbol{c}_{uv} \leftarrow \boldsymbol{\nu}_{uv}$ // residual forward capacities
  $\boldsymbol{c}_{vu} \leftarrow \boldsymbol{\nu}_{vu}$ // residual backward capacities

// Main part:
for $k = 1, \ldots, n-1$ **do**
  for $i = 1, \ldots, m$ **do**
    $(\boldsymbol{y}_e)_{e \in \vec{E}} \leftarrow \texttt{FindAugmentingFlow}_k((\boldsymbol{c}_e)_{e \in E})$
    /* Returns an augmenting flow (respecting the residual capacities) that only uses paths of length exactly k and saturates at least one arc. */
    // Augmenting:
    for each $uv \in \vec{E}$ **do parallel**
      $\boldsymbol{x}_{uv} \leftarrow \boldsymbol{x}_{uv} + \boldsymbol{y}_{uv}$
      $\boldsymbol{c}_{uv} \leftarrow \boldsymbol{c}_{uv} - \boldsymbol{y}_{uv}$
      $\boldsymbol{c}_{vu} \leftarrow \boldsymbol{c}_{vu} + \boldsymbol{y}_{uv}$
return $(\boldsymbol{x}_e)_{e \in \vec{E}}$

is contained in E and let $\vec{E}$ denote a subset of all arcs containing exactly one arc for each pair of antiparallel arcs. To point out the ability of neural networks to parallelize well, we sometimes use parallel loops even though this does not significantly reduce asymptotic complexity measures in our case.

To explain our algorithm, let us start by recalling the key ideas of the classical Edmonds-Karp-Dinic algorithm [12, 13]. The algorithm repeatedly finds a shortest s-t path in the residual graph $G^* = (V, E^*)$, and sends the maximum possible amount of flow on such a path, that is, saturates at least one arc. The algorithm terminates by returning a minimum cut once t cannot be reached from s in the residual graph. The key insight in the analysis is that the distance from s to t in the residual graph is non-decreasing, and strictly increases within at most m such iterations. Thus, the number of iterations can be bounded by $\mathcal{O}(nm)$.

A shortest path can be characterized by *distance labels*. The vector $d \in \mathbb{R}_+^V$ is a distance labelling if $d(s) = 0$ and $d(v) \leq d(u) + 1$ for every residual arc $uv \in E^*$. If there exists an s-t path P such that $d(v) = d(u) + 1$ for every arc in P, then P is a shortest path. Identifying a shortest path is equivalent to finding distance labels and such a path. We note that the preflow-push algorithm [21] explicitly relies on using distance labels and pushing flow on residual arcs uv with $d(v) = d(u) + 1$. However, finding such a labelling requires **if**-branchings as it needs to identify the arcs in E^*, that is, arcs with positive residual capacity.

At a high level, our algorithm is similar, but it avoids knowing the arcs in the residual graph and the length k of the shortest residual s-t path explicitly. Instead, we guess k in each iteration of the main procedure (Algorithm 2),

making sure that we never overestimate the true length. The guess is initialized as $k = 1$ and, in accordance with the Edmonds-Karp-Dinic analysis, we increment k by one in every m iterations. Based on our guess for k, we use a subroutine $\texttt{FindAugmentingFlow}_k$ (Algorithm 3) with the following feature: if the actual shortest path length is exactly k, the subroutine will send flow from s to t on (possibly multiple) paths of length exactly k, saturating at least one arc. If the shortest path is longer than k, nothing happens in the current iteration.

Instead of distance labels, the subroutine computes *fattest path values* $\boldsymbol{a}_{i,v}$ (line 7 to 11) that represent the maximum amount of flow that can be sent from v to t on a path of length exactly i. Such values can be obtained by a simple dynamic program that is easy to implement as a MAAP. Thus, a path $(s = v_k, v_{k-1}, \dots, v_1, v_0 = t)$ of length exactly k is contained in the residual network if and only if $\boldsymbol{a}_{i,v_i} > 0$ for all $i = 1, \dots, k$. Our algorithm makes sure that we only send flow along arcs that are contained in such paths. In particular, the current iteration will send positive flow if and only if $\boldsymbol{a}_{k,s} > 0$. However, we cannot recover the shortest s-t path with capacity $\boldsymbol{a}_{k,s}$. Therefore, in general, flow will not be sent along a single path and the value of the flow output by $\texttt{FindAugmentingFlow}_k$ might be strictly less than $\boldsymbol{a}_{k,s}$.

After computing the $\boldsymbol{a}_{i,v}$ values, $\texttt{FindAugmentingFlow}_k$ greedily pushes flow from s towards t, using a lexicographic selection rule to pick the next arc to push flow on (line 12 to 22). On the high level, this is similar to the preflow-push algorithm, but using the $\boldsymbol{a}_{i,v}$ values that encode the shortest path distance information implicitly. This may leave some nodes with excess flow; a final cleanup phase (line 23 to 29) is needed to send the remaining flow back to the source s.

An example for the $\texttt{FindAugmentingFlow}_k$-subroutine is given in Figure 3. We emphasize again that, although the description of the subroutine in the example in Figure 3 seems to rely heavily on the distance of a node to t, this information is calculated and used only in an implicit way via the precomputed $\boldsymbol{a}_{i,v}$ values. This way, we are able to implement the subroutine without the usage of comparison-based branchings.

The proof of correctness for our algorithm consists of two main steps. The first step is the analysis of thesubroutine. This involves carefully showing that the returned flow indeed satisfies flow conservation, is feasible with respect to the residual capacities, uses only arcs that lie on a s-t-path of length exactly k in the residual network, and most importantly, if such a path exists, it saturates at least one arc. This last property can be shown using the lexicographic selection rule to pick the next arc to push flow on. Note that, in general, the subroutine neither returns a single path (as in the Edmonds-Karp algorithm [13]), nor a blocking flow (as in the Dinic algorithm [12]). The second main step is to show that, nevertheless, the properties of the subroutine are sufficient to ensure that the distance from s to t in the residual network increases at least every m iterations, such that we terminate with a maximum flow after nm iterations.

With the correctness of the whole MAAP at hand, Theorem 2 follows by simply counting the complexity measures $d(A)$, $w(A)$, and $s(A)$, and applying Proposition 3.

Algorithm 3: FindAugmentingFlow$_k$ for a fixed graph $G = (V, E)$ and a fixed length k.

Input: Residual capacities $(\boldsymbol{c}_e)_{e \in E}$.

`// Initializing:`
for each $vw \in \vec{E}$ **do parallel**
  $\boldsymbol{z}_{vw} \leftarrow 0$ `// flow in residual network`
  $\boldsymbol{z}_{wv} \leftarrow 0$
for each $(i, v) \in [k] \times (V \setminus \{ t \})$ **do parallel**
  $\boldsymbol{Y}_v^i \leftarrow 0$ `// excessive flow at` v `in iteration` i `(from` k `to 1)`
  $\boldsymbol{a}_{i,v} \leftarrow 0$ `// initialize fattest path values`

`// Determining the fattest path values:`
for each $v \in N_t^-$ **do parallel**
  $\boldsymbol{a}_{1,v} \leftarrow \boldsymbol{c}_{vt}$
for $i = 2, 3, \ldots, k$ **do**
  for each $v \in V \setminus \{ t \}$ **do parallel**
    $\boldsymbol{a}_{i,v} \leftarrow \max_{w \in N_v^+ \setminus \{ t \}} \min \{ \boldsymbol{a}_{i-1,w}, \boldsymbol{c}_{vw} \}$

`// Pushing flow of value` $\boldsymbol{a}_{k,s}$ `from` s `to` t`:`
$\boldsymbol{Y}_s^k \leftarrow \boldsymbol{a}_{k,s}$ `// excessive flow at` s
for $i = k, k-1, \ldots, 2$ **do**
  for $v \in V \setminus \{ t \}$ *in index order* **do**
    for $w \in N_v^+ \setminus \{ t \}$ *in index order* **do**
      `// Push flow out of` v `and into` w`:`
      $\boldsymbol{f} \leftarrow \min \{ \boldsymbol{Y}_v^i, \boldsymbol{c}_{vw}, \boldsymbol{a}_{i-1,w} - \boldsymbol{Y}_w^{i-1} \}$ `// value we can push over` vw `such that this flow can still arrive at` t
      $\boldsymbol{z}_{vw} \leftarrow \boldsymbol{z}_{vw} + \boldsymbol{f}$
      $\boldsymbol{Y}_v^i \leftarrow \boldsymbol{Y}_v^i - \boldsymbol{f}$
      $\boldsymbol{Y}_w^{i-1} \leftarrow \boldsymbol{Y}_w^{i-1} + \boldsymbol{f}$
for each $v \in N_t^-$ **do parallel**
  `// Push flow out of` v `and into` t`:`
  $\boldsymbol{z}_{vt} \leftarrow \boldsymbol{Y}_v^1$
  $\boldsymbol{Y}_v^1 \leftarrow 0$

`// Clean-up by bounding:`
for $i = 2, 3, \ldots, k-1$ **do**
  for $w \in V \setminus \{ t \}$ *in reverse index order* **do**
    for $v \in N_w^- \setminus \{ t \}$ *in reverse index order* **do**
      $\boldsymbol{b} \leftarrow \min \{ \boldsymbol{Y}_w^i, \boldsymbol{z}_{vw} \}$ `// value we can push backwards along` vw
      $\boldsymbol{z}_{vw} \leftarrow \boldsymbol{z}_{vw} - \boldsymbol{b}$
      $\boldsymbol{Y}_w^i \leftarrow \boldsymbol{Y}_w^i - \boldsymbol{b}$
      $\boldsymbol{Y}_v^{i+1} \leftarrow \boldsymbol{Y}_v^{i+1} + \boldsymbol{b}$

for each $uv \in \vec{E}$ **do parallel**
  $\boldsymbol{y}_{vw} \leftarrow \boldsymbol{z}_{vw} - \boldsymbol{z}_{wv}$
return $(\boldsymbol{y}_e)_{e \in \vec{E}}$

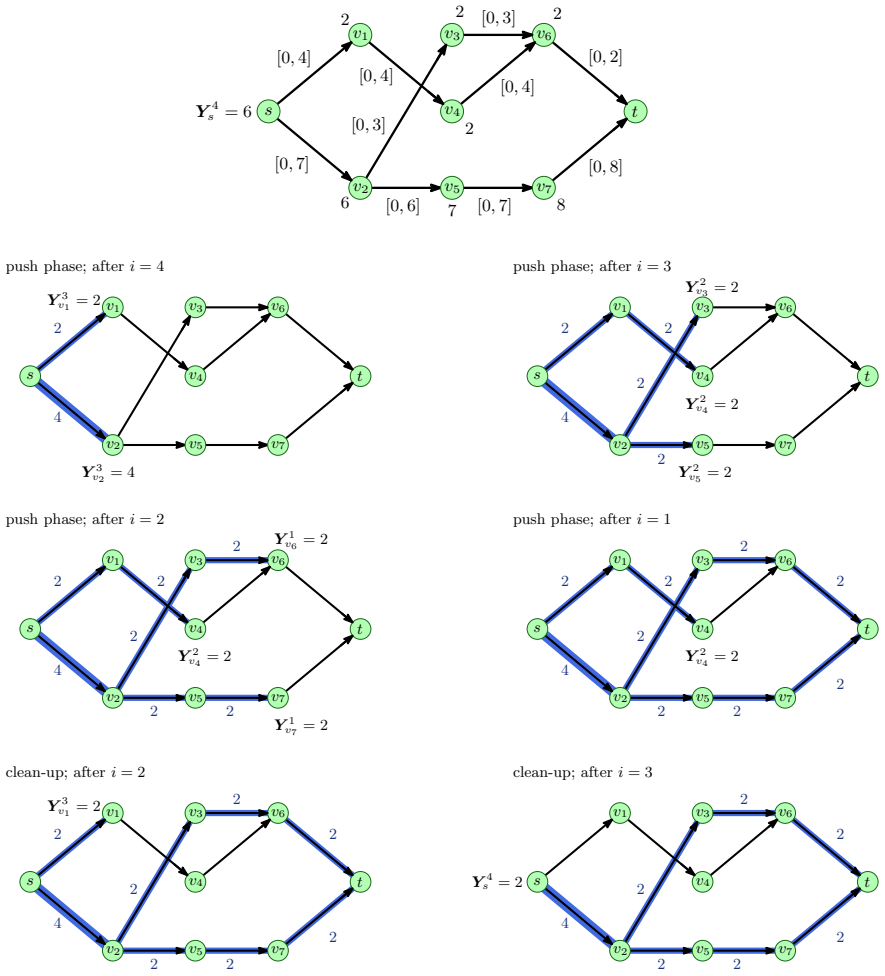

Fig. 3: Example of the `FindAugmentingFlow`$_k$ subroutine for $k = 4$. The edge labels in the top figure are the residual capacity bounds in the current iteration. The first step is to compute the fattest path values $\boldsymbol{a}_{i,v}$, which are depicted as node labels in the top figure. The values $\boldsymbol{Y}_v^i$ always denote the excessive flow of a vertex v with distance i from the sink. All values that are not displayed are zero. At s, we initialize $\boldsymbol{Y}_s^4 = \boldsymbol{a}_{4,s} = 6$. Then, excessive flow is pushed greedily towards the sink, as shown in the four figures in the middle. While doing so, we ensure that at each vertex the arriving flow does not exceed its value $\boldsymbol{a}_{i,v}$. For this reason, flow can get stuck, as it happens at v_4 in this example. Therefore, in a final cleanup phase, depicted in the two bottom figures, we push flow back to the source s. Observe that the result is an s-t-flow that is feasible with respect to the residual capacities, uses only paths of length $k = 4$, and saturates the arc v_6t.

Acknowledgements A large portion of this work was completed while both authors were affiliated with TU Berlin. We thank Max Klimm, Jennifer Manke, Arturo Merino, Martin Skutella, and László Végh for many inspiring and fruitful discussions and valuable comments.

References

1. Ahuja, R.K., Magnanti, T.L., Orlin, J.B.: Network flows: theory, algorithms, and applications. Prentice Hall, Upper Saddle River, New Jersey, USA (1993)
2. Ali, M.M., Kamoun, F.: A neural network approach to the maximum flow problem. In: IEEE Global Telecommunications Conference GLOBECOM'91: Countdown to the New Millennium. Conference Record. pp. 130–134 (1991)
3. Anthony, M., Bartlett, P.L.: Neural network learning: Theoretical foundations. Cambridge University Press (1999)
4. Arora, R., Basu, A., Mianjy, P., Mukherjee, A.: Understanding deep neural networks with rectified linear units. In: International Conference on Learning Representations (2018)
5. Beiu, V., Taylor, J.G.: On the circuit complexity of sigmoid feedforward neural networks. Neural Networks **9**(7), 1155–1171 (1996)
6. Bello, I., Pham, H., Le, Q.V., Norouzi, M., Bengio, S.: Neural combinatorial optimization with reinforcement learning. arXiv:1611.09940 (2016)
7. Bengio, Y., Lodi, A., Prouvost, A.: Machine learning for combinatorial optimization: a methodological tour d'horizon. arXiv:1811.06128 (2018)
8. Berner, J., Grohs, P., Kutyniok, G., Petersen, P.: The modern mathematics of deep learning. arXiv:2105.04026 (2021)
9. Bertschinger, D., Hertrich, C., Jungeblut, P., Miltzow, T., Weber, S.: Training fully connected neural networks is $\exists\mathbb{R}$-complete. arXiv:2204.01368 (2022)
10. Chen, L., Kyng, R., Liu, Y.P., Peng, R., Gutenberg, M.P., Sachdeva, S.: Maximum flow and minimum-cost flow in almost-linear time. arXiv:2203.00671 (2022)
11. Cybenko, G.: Approximation by superpositions of a sigmoidal function. Mathematics of control, signals and systems **2**(4), 303–314 (1989)
12. Dinic, E.A.: Algorithm for solution of a problem of maximum flow in a network with power estimation. Soviet Mathematics Doklady **11**, 1277–1280 (1970)
13. Edmonds, J., Karp, R.M.: Theoretical improvements in algorithmic efficiency for network flow problems. Journal of the ACM **19**(2), 248–264 (1972)
14. Effati, S., Ranjbar, M.: Neural network models for solving the maximum flow problem. Applications and Applied Mathematics **3**(3), 149–162 (2008)
15. Eldan, R., Shamir, O.: The power of depth for feedforward neural networks. In: Conference on Learning Theory. pp. 907–940 (2016)
16. Emami, P., Ranka, S.: Learning permutations with sinkhorn policy gradient. arXiv:1805.07010 (2018)
17. Fomin, S., Grigoriev, D., Koshevoy, G.: Subtraction-free complexity, cluster transformations, and spanning trees. Foundations of Computational Mathematics **16**(1), 1–31 (2016)
18. Froese, V., Hertrich, C., Niedermeier, R.: The computational complexity of relu network training parameterized by data dimensionality. arXiv:2105.08675 (2021)
19. Glorot, X., Bordes, A., Bengio, Y.: Deep sparse rectifier neural networks. In: Proceedings of the fourteenth international conference on artificial intelligence and statistics. pp. 315–323 (2011)

20. Goel, S., Klivans, A.R., Manurangsi, P., Reichman, D.: Tight hardness results for training depth-2 ReLU networks. In: 12th Innovations in Theoretical Computer Science Conference (ITCS '21) (2021)
21. Goldberg, A.V., Tarjan, R.E.: A new approach to the maximum-flow problem. Journal of the ACM (JACM) **35**(4), 921–940 (1988)
22. Goldschlager, L.M., Shaw, R.A., Staples, J.: The maximum flow problem is log space complete for P. Theoretical Computer Science **21**(1), 105–111 (1982)
23. Greenlaw, R., Hoover, H.J., Ruzzo, W.L.: Limits to parallel computation: P-completeness theory. Oxford University Press (1995)
24. Haase, C.A., Hertrich, C., Loho, G.: Lower bounds on the depth of integral ReLU neural networks via lattice polytopes. In: International Conference on Learning Representations (ICLR) (2023)
25. Hanin, B.: Universal function approximation by deep neural nets with bounded width and ReLU activations. Mathematics **7**(10), 992 (2019)
26. Hanin, B., Rolnick, D.: Complexity of linear regions in deep networks. In: International Conference on Machine Learning. pp. 2596–2604 (2019)
27. Hanin, B., Sellke, M.: Approximating continuous functions by ReLU nets of minimal width. arXiv:1710.11278 (2017)
28. Hertrich, C., Basu, A., Di Summa, M., Skutella, M.: Towards lower bounds on the depth of ReLU neural networks. Advances in Neural Information Processing Systems **34**, 3336–3348 (2021)
29. Hertrich, C., Skutella, M.: Provably good solutions to the knapsack problem via neural networks of bounded size. In: Thirty-Fifth AAAI Conference on Artificial Intelligence (to appear) (2021), full version: *arXiv:2005.14105*
30. Hopfield, J.J., Tank, D.W.: "Neural" computation of decisions in optimization problems. Biological Cybernetics **52**(3), 141–152 (1985)
31. Hornik, K.: Approximation capabilities of multilayer feedforward networks. Neural networks **4**(2), 251–257 (1991)
32. Jerrum, M., Snir, M.: Some exact complexity results for straight-line computations over semirings. Journal of the ACM (JACM) **29**(3), 874–897 (1982)
33. Jukna, S.: Lower bounds for tropical circuits and dynamic programs. Theory of Computing Systems **57**(1), 160–194 (2015)
34. Jukna, S., Seiwert, H.: Greedy can beat pure dynamic programming. Information Processing Letters **142**, 90–95 (2019)
35. Kennedy, M.P., Chua, L.O.: Neural networks for nonlinear programming. IEEE Transactions on Circuits and Systems **35**(5), 554–562 (1988)
36. Khalife, S., Basu, A.: Neural networks with linear threshold activations: structure and algorithms. In: International Conference on Integer Programming and Combinatorial Optimization. pp. 347–360. Springer (2022)
37. Khalil, E., Dai, H., Zhang, Y., Dilkina, B., Song, L.: Learning combinatorial optimization algorithms over graphs. Advances in neural information processing systems **30** (2017)
38. Kool, W., van Hoof, H., Welling, M.: Attention, learn to solve routing problems! In: International Conference on Learning Representations (2019)
39. Korte, B., Vygen, J.: Combinatorial Optimization: Theory and Algorithms. Springer, 4th edn. (2008)
40. LeCun, Y., Bengio, Y., Hinton, G.: Deep learning. Nature **521**, 436–444 (2015)
41. Liang, S., Srikant, R.: Why deep neural networks for function approximation? In: International Conference on Learning Representations (2017)
42. Lodi, A., Zarpellon, G.: On learning and branching: a survey. TOP **25**(2), 207–236 (2017)

43. Montufar, G.F., Pascanu, R., Cho, K., Bengio, Y.: On the number of linear regions of deep neural networks. Advances in neural information processing systems **27** (2014)
44. Mukherjee, A., Basu, A.: Lower bounds over boolean inputs for deep neural networks with ReLU gates. arXiv:1711.03073 (2017)
45. Nazemi, A., Omidi, F.: A capable neural network model for solving the maximum flow problem. Journal of Computational and Applied Mathematics **236**(14), 3498–3513 (2012)
46. Nguyen, Q., Mukkamala, M.C., Hein, M.: Neural networks should be wide enough to learn disconnected decision regions. In: International Conference on Machine Learning. pp. 3737–3746 (2018)
47. Nowak, A., Villar, S., Bandeira, A.S., Bruna, J.: Revised Note on Learning Algorithms for Quadratic Assignment with Graph Neural Networks. arXiv:1706.07450 (2017)
48. Orlin, J.B.: Max flows in O(nm) time, or better. In: Proceedings of the Forty-Fifth Annual ACM Symposium on Theory of Computing (STOC '13). pp. 765–774. Association for Computing Machinery (2013)
49. Parberry, I., Garey, M.R., Meyer, A.: Circuit complexity and neural networks. MIT Press (1994)
50. Pascanu, R., Montufar, G., Bengio, Y.: On the number of inference regions of deep feed forward networks with piece-wise linear activations. In: International Conference on Learning Representations (2014)
51. Raghu, M., Poole, B., Kleinberg, J., Ganguli, S., Dickstein, J.S.: On the expressive power of deep neural networks. In: International Conference on Machine Learning. pp. 2847–2854 (2017)
52. Rothvoß, T.: The matching polytope has exponential extension complexity. Journal of the ACM (JACM) **64**(6), 1–19 (2017)
53. Safran, I., Shamir, O.: Depth-width tradeoffs in approximating natural functions with neural networks. In: International Conference on Machine Learning. pp. 2979–2987 (2017)
54. Serra, T., Tjandraatmadja, C., Ramalingam, S.: Bounding and counting linear regions of deep neural networks. In: International Conference on Machine Learning. pp. 4565–4573 (2018)
55. Shawe-Taylor, J.S., Anthony, M.H., Kern, W.: Classes of feedforward neural networks and their circuit complexity. Neural networks **5**(6), 971–977 (1992)
56. Shpilka, A., Yehudayoff, A.: Arithmetic circuits: A survey of recent results and open questions. Now Publishers Inc (2010)
57. Smith, K.A.: Neural networks for combinatorial optimization: A review of more than a decade of research. INFORMS Journal on Computing **11**(1), 15–34 (1999)
58. Telgarsky, M.: Representation benefits of deep feedforward networks. arXiv:1509.08101 (2015)
59. Telgarsky, M.: Benefits of depth in neural networks. In: Conference on Learning Theory. pp. 1517–1539 (2016)
60. Vinyals, O., Fortunato, M., Jaitly, N.: Pointer networks. Advances in neural information processing systems **28** (2015)
61. Williamson, D.P.: Network Flow Algorithms. Cambridge University Press (2019)
62. Yarotsky, D.: Error bounds for approximations with deep relu networks. Neural Networks **94**, 103–114 (2017)