The Simplex Algorithm Is NP-Mighty

YANN DISSER, TU Darmstadt
MARTIN SKUTELLA, TU Berlin

We show that the Simplex Method, the Network Simplex Method—both with Dantzig's original pivot rule—and the Successive Shortest Path Algorithm are *NP-mighty*. That is, each of these algorithms can be used to solve, with polynomial overhead, any problem in NP implicitly during the algorithm's execution. This result casts a more favorable light on these algorithms' exponential worst-case running times. Furthermore, as a consequence of our approach, we obtain several novel hardness results. For example, for a given input to the Simplex Algorithm, deciding whether a given variable ever enters the basis during the algorithm's execution and determining the number of iterations needed are both NP-hard problems. Finally, we close a long-standing open problem in the area of network flows over time by showing that earliest arrival flows are NP-hard to obtain.

CCS Concepts: • **Theory of computation → Discrete optimization**; *Network flows*; Complexity classes; • **Mathematics of computing** → *Combinatorial optimization*;

Additional Key Words and Phrases: Simplex algorithm, network simplex, successive shortest paths, NP-mightiness, earliest arrival flows

ACM Reference format:
Yann Disser and Martin Skutella. 2018. The Simplex Algorithm Is NP-Mighty. *ACM Trans. Algorithms* 15, 1, Article 5 (November 2018), 19 pages.
https://doi.org/10.1145/3280847

1 INTRODUCTION

Understanding the complexity of algorithmic problems is a central challenge in the theory of computing. Traditionally, complexity theory operates from the point of view of the problems we encounter in the world by considering a fixed problem and asking how *nice* an algorithm the problem admits with respect to running time, memory consumption, robustness to uncertainty in the input, determinism, and the like. In this article, we advocate a different perspective by considering a particular algorithm and asking how powerful (or *mighty*) the algorithm is (i.e., what the most difficult problems are that the algorithm can be used to solve "implicitly" during its execution).

Related Literature. A traditional approach to capturing the mightiness of an algorithm is to ask how difficult the exact problem is that the algorithm was designed to solve; thatis, what is

An extended abstract of this paper appeared in the proceedings of SODA 2015; see [7]. The research described in this paper was partially supported by the DFG Priority Programme 1736 "Algorithms for Big Data" (grant SK 58/10-2) and by the Alexander von Humboldt-Foundation.
Authors' addresses: Y. Disser, Department of Mathematics, TU Darmstadt, Dolivostraße 15, 64293 Darmstadt, Germany; email: disser@mathematik.tu-darmstadt.de; M. Skutella, Institute of Mathematics, TU Berlin, Straße des 17. Juni 136, 10623 Berlin, Germany; email: martin.skutella@tu-berlin.de.

1549-6325/2018/11-ART5 $15.00
https://doi.org/10.1145/3280847

the complexity of predicting the algorithm's final outcome. For optimization problems, however, if there are multiple optimum solutions to an instance, predicting which optimum solution a specific algorithm will produce might be more difficult than finding an optimum solution in the first place. If this is the case, the algorithm can be considered to be mightier than the problem it is solving suggests. A prominent example for this phenomenon are search algorithms for problems in the complexity class PLS (for *Polynomial Local Search*), introduced by Johnson, Papadimitriou, and Yannakakis [16]. Many problems in PLS are complete with respect to so-called *tight* reductions, which implies that finding any optimum solution reachable from a specific starting solution via local search is PSPACE-complete [25]. Any local search algorithm for such a problem can thus be considered to be PSPACE-mighty. Goldberg, Papadimitriou, and Savani [13] established similar PSPACE-completeness results for algorithms solving search problems in the complexity class PPAD (for *Polynomial Parity Argument in Directed Graphs* [24]) and, in particular, for the well-known Lemke-Howson algorithm [21] for finding Nash equilibria in bimatrix games.

Preliminary versions of our article appeared in Disser and Skutella [6, 7]. As our main result, we show that the (Network) Simplex Algorithm with Danzig's original pivot rule *implicitly* solves NP-hard problems. Concurrently, Adler, Papadimitriou, and Rubinstein [1] gave an artificial pivot rule for which it is even PSPACE-complete to decide whether a given basis will appear during the algorithm's execution. They show that this is not the case for the Simplex Algorithm with the shadow vertex pivot rule. Note that the corresponding algorithm might still implicitly solve hard problems in our sense. Recently, and inspired by our results, Fearnley and Savani [8] strengthened our results for the Simplex Algorithm with Danzig's pivot rule. They show that the resulting algorithm even *explicitly* solves a PSPACE-complete problem. Note that their result is specifically tailored to the general Simplex Method and does not hold for the Network Simplex Algorithm or the Successive Shortest Path Algorithm. By showing that the latter are already implicitly solving hard problems, we are able to infer interesting consequences (e.g., for earliest arrival flows). Other recent work regarding the complexity of algorithms was conducted by Fearnely and Savani [9], and by Roughgarden and Wang [26].

A Novel Approach. We propose to take the analysis of algorithms beyond understanding the complexity of the exact problem an algorithm is solving and argue, instead, that the mightiness of an algorithm could also be classified by the complexity of the problems that the algorithm can be made to solve *implicitly*. In particular, we do not consider an algorithm as a black box that turns a given input into a well-defined output. Instead, we are interested in the entire process of computation (i.e., the sequence of the algorithm's internal states) that leads to the final output, and we ask how meaningful this process is in terms of valuable information that can be drawn from it. As we show in this article, sometimes very limited information on an algorithm's process of computation can be used to solve problems that are considerably more complex than the problem the algorithm was actually designed for.

We define the mightiness of an algorithm via the complexity of the problems that it can solve *implicitly* in this way, and, in particular, we say that an algorithm is NP-mighty if it implicitly solves all problems in NP (precise definitions are given later). Note that in order to make mightiness a meaningful concept, we need to make sure that mindless exponential algorithms like simple counters do not qualify as being NP-mighty, while algorithms that explicitly solve hard problems do. This goal is achieved by carefully restricting the allowed computational overhead as well as the access to the algorithm's process of computation.

Considered Algorithms. For an algorithm's mightiness to lie beyond the complexity class of the problem it was designed to solve, its running time must be excessive for this complexity class. Most algorithms that are inefficient in this sense would quickly be disregarded as wasteful and not meriting further investigation. Dantzig's Simplex Method [4] is a famous exception to this rule. Empirically, it belongs to the most efficient methods for solving linear programs. However, Klee and Minty [20] showed that the Simplex Algorithm with Dantzig's original pivot rule exhibits exponential worst-case behavior. Similar results are known for many other popular pivot rules (see, e.g., Amenta and Ziegler [2]). On the other hand, by the work of Khachiyan [18, 19] and later Karmarkar [17], it is known that linear programs can be solved in polynomial time. Spielman and Teng [28] developed the concept of smoothed analysis in order to explain the practical efficiency of the Simplex Method despite its poor worst-case behavior.

Minimum-cost flow problems form a class of linear programs featuring a particularly rich combinatorial structure allowing for numerous specialized algorithms. The first such algorithm is Dantzig's Network Simplex Method [5] which is an interpretation of the general Simplex Method applied to this class of problems. In this article, we consider the primal (Network) Simplex Method together with Dantzig's pivot rule, which always selects the nonbasic variable with the most negative reduced cost to enter the basis. We refer to this variant of the (Network) Simplex Method as the *(Network) Simplex Algorithm.*

One of the simplest and most basic algorithms for minimum-cost flow problems is the Successive Shortest Path Algorithm, which iteratively augments flow along paths of minimum cost in the residual network [3, 14]. According to Ford and Fulkerson [10], the underlying theorem stating that such an augmentation step preserves optimality "*may properly be regarded as the central one concerning minimal cost flows.*" Zadeh [31] presented a family of instances forcing the Successive Shortest Path Algorithm and also the Network Simplex Algorithm into exponentially many iterations. On the other hand, Tardos [29] proved that minimum-cost flows can be computed in strongly polynomial time, and Orlin [23] gave a polynomial variant of the Network Simplex Method.

Main Contribution. We argue that the exponential worst-case running time of the (Network) Simplex Algorithm and the Successive Shortest Path Algorithm is, for some instances, due to the computational difficulty of the solution scheme these algorithms implement, rather than being caused by excessive repetition of the same operations or the like. While both algorithms sometimes take longer than necessary to reach their primary objective (namely, to find an optimum solution to a particular linear program), they perform meaningful computations internally that may require them to implicitly solve difficult problems. To make this statement more precise, we introduce a definition of "implicitly solving: that is as minimalistic as possible with regards to the extent in which we are permitted to use the algorithm's internal state. The following definition refers to the *complete configuration* of a Turing machine (i.e., a binary representation of the machine's internal state, contents of its tape, and position of its head).

Definition 1.1. An algorithm given by a Turing machine T *implicitly solves* a decision problem $\mathcal{P}$ if, for a given instance I of $\mathcal{P}$, it is possible to compute in polynomial time an input I' for T and a bit b in the complete configuration of T, such that I is a yes-instance if and only if b flips at some point during the execution of T for input I'.

An algorithm that implicitly solves a particular NP-hard decision problem, implicitly solves all problems in NP. We call such algorithms *NP-mighty.*

Definition 1.2. An algorithm is *NP-mighty* if it implicitly solves every decision problem in NP.

Note that every algorithm that *explicitly* solves an NP-hard decision problem, by definition, also implicitly solves this problem (assuming, without loss of generality, that a single bit indicates if the Turing machine has reached an accepting state) and thus is NP-mighty. Also note that our definitions are tailored to the class NP but could be generalized to other complexity classes. In that context, our notion of implicitly solving a decision problem should be referred to more precisely as "NP-implicit."

The preceding definitions turn out to be sufficient for our purposes. We remark, however, that slightly more general versions of Definition 1.1, involving constantly many bits or broader/free access to the algorithm's output, seem reasonable as well. In this context, access to the exact number of iterations needed by the algorithm also seems reasonable as it may provide valuable information. In fact, our results below still hold if the number of iterations is all we may use of an algorithm's behavior. Most importantly, our definitions have been formulated with some care in an attempt to distinguish exponential-time algorithms that implement sophisticated solution schemes from those that instead "waste time" on less meaningful operations. We discuss this critical point in some further detail.

Constructions of exponential time worst-case instances for algorithms usually rely on gadgets that somehow force an algorithm to count (i.e., to enumerate over exponentially many configurations). Such counting behavior by itself cannot be considered meaningful, and, consequently, an algorithm should certainly exhibit more elaborate behavior to qualify as being NP-mighty. As an example, consider the *simple counting algorithm* (Turing machine) that counts from a given positive number down to zero; that is, the Turing machine iteratively reduces the binary number on its tape by one until it reaches zero. To show that this algorithm is *not* NP-mighty, we need to assume that P≠NP, as otherwise the polynomial-time transformation of inputs can already solve NP-hard problems. Since, for sufficiently large inputs, every state of the simple counting algorithm is reached, and since every bit on its tape flips at some point, our definitions are meaningful in the following sense.

PROPOSITION 1.3. *Unless $P = NP$, the simple counting algorithm is not* NP*-mighty while every algorithm that solves an* NP*-hard problem is* NP*-mighty.*

Our main result explains the exponential worst-case running time of the following algorithms with their computational power.

THEOREM 1.4. *The Simplex Algorithm, the Network Simplex Algorithm (both with Dantzig's pivot rule), and the Successive Shortest Path Algorithm are* NP*-mighty.*

We prove this theorem by showing that the algorithms implicitly solve the NP-complete PARTITION problem (cf. Garey and Johnson [12]). To this end, we show how to turn a given instance of PARTITION in polynomial time into a minimum-cost flow network with a distinguished arc e, such that the Network Simplex Algorithm (or the Successive Shortest Path Algorithm) augments flow along arc e in one of its iterations if and only if the PARTITION instance has a solution. Under the mild assumption that in an implementation of the Network Simplex Algorithm or the Successive Shortest Path Algorithm fixed bits are used to store the flow variables of arcs, this implies that these algorithms implicitly solve PARTITION in terms of Definition 1.1.

A central part of our network construction is a recursively defined family of counting gadgets on which these minimum-cost flow algorithms take exponentially many iterations. These counting gadgets are, in some sense, simpler than Zadeh's 40-year-old "bad networks" [31] and thus interesting in their own right. By slightly perturbing the costs of the arcs according to the values of a given PARTITION instance, we manage to force the considered minimum-cost flow algorithms into enumerating all possible solutions. In contrast to counters, we show that the internal states of these algorithms reflect whether or not they encountered a valid PARTITION solution (in the sense of Definition 1.1).

Further Results. We mention interesting consequences of our main results just discussed (proofs in Section 5). We first state complexity results that follow from our proof of Theorem 1.4.

Corollary 1.5. *Determining the number of iterations needed by the Simplex Algorithm, the Network Simplex Algorithm, and the Successive Shortest Path Algorithm for a given input is* NP*-hard.*

Corollary 1.6. *Deciding for a given linear program whether a given variable ever enters the basis during the execution of the Simplex Algorithm is* NP*-hard.*

Another interesting implication is for parametric flows and parametric linear programming.

Corollary 1.7. *Determining whether a parametric minimum-cost flow uses a given arc (i.e., assigns positive flow value for any parameter value) is* NP*-hard. In particular, determining whether the solution to a parametric linear program uses a given variable is* NP*-hard. Also, determining the number of different basic solutions over all parameter values is* NP*-hard.*

We also obtain the following complexity result on 2-dimensional projections of polyhedra.

Corollary 1.8. *Given a d-dimensional polytope P defined by a system of linear inequalities, determining the number of vertices of P's projection onto a given 2-dimensional subspace is* NP*-hard.*

We finally mention a result for a long-standing open problem in the area of network flows over time (see, e.g., Skutella [27] for an introduction to this area). The goal in *earliest arrival flows* is to find an s-t-flow over time that simultaneously maximizes the amount of flow that has reached the sink node t at any point in time [11]. It is known since the early 1970s that the Successive Shortest Path Algorithm can be used to obtain such an earliest arrival flow [22, 30]. All known encodings of earliest arrival flows, however, suffer from exponential worst-case size, and, ever since, it has been an open problem whether there is a polynomial encoding which can be found in polynomial time. The following corollary implies that, in a certain sense, earliest arrival flows are NP-hard to obtain.

Corollary 1.9. *Determining the average arrival time of flow in an earliest arrival flow is* NP*-hard.*

Note that an s-t-flow over time is an earliest arrival flow if and only if it minimizes the average arrival time of flow [15].

Outline. After establishing some minimal notation in Section 2, we proceed to proving Theorem 1.4 for the Successive Shortest Path Algorithm in Section 3. In Section 4, we adapt the construction for the Network Simplex Algorithm. Explanations and proofs of the above-mentioned corollaries are given in Section 5. Finally, Section 6 highlights interesting open problems for future research.

2 PRELIMINARIES

In the following sections, we show that the Successive Shortest Path Algorithm and the Network Simplex Algorithm implicitly solve the classical Partition problem. An instance of Partition is given by a vector of positive numbers $\vec{a} = (a_1, \dots, a_n) \in \mathbb{Q}^n$ and the problem is to decide whether there is a subset $I \subseteq \{1, \dots, n\}$ with $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$. This problem is well-known to be NP-complete (cf. [12]). Throughout this article, we consider an arbitrary fixed instance $\vec{a}$ of Partition. Without loss of generality, we assume $A := \sum_{i=1}^{n} a_i < 1/12$ and that all values a_i, $i \in \{1, \dots, n\}$, are multiples of ε for some fixed $\varepsilon > 0$ (polynomially representable and depending on the instance).

Let $\vec{v} = (v_1, \dots, v_n) \in \mathbb{Q}^n$ and $k \in \mathbb{N}$, with $k_j \in \{0, 1\}$, $j \in \mathbb{Z}_{\geq 0}$, being the jth bit in the binary representation of k (i.e., $k_j := \lfloor k/2^j \rfloor \bmod 2$). We define $\vec{v}^{[k]}_{i_1, i_2} := \sum_{j=i_1+1}^{i_2} (-1)^{k_{j-1}} v_j$, $\vec{v}^{[k]}_{i} := \vec{v}^{[k]}_{0,i}$, and $\vec{v}^{[k]}_{i,i} = 0$. The following characterization will be useful later.

Proposition 2.1. *The* Partition *instance $\vec{a}$ admits a solution if and only if there is a $k \in \{0, \dots, 2^n - 1\}$ for which $\vec{a}^{[k]}_n = 0$.*

Throughout this article, we construct instances of the *minimum cost (maximum) flow problem* that we use as input to the Successive Shortest Path Algorithm and the Network Simplex

Algorithm. In this context, a *network* $N = (G, u, s, t)$ consists of a directed graph $G = (V, E)$ together with arc capacities $u : E \to \mathbb{R}_{\geq 0} \cup \{\infty\}$, as well as a source $s \in V$ and a sink $t \in V$. In the following, we denote $\delta^-(v) := (V \times \{v\}) \cap E$ and $\delta^+(v) := (\{v\} \times V) \cap E$ for all $v \in V$. A *flow* in a network is a function $f : E \to \mathbb{R}_{\geq 0}$ that obeys capacities (i.e., $f(e) \leq u(e)$ for all $e \in E$) and conserves flow (i.e., $\sum_{e \in \delta^-(v)} f(e) = \sum_{e \in \delta^+(v)} f(e)$ for all $v \in V \setminus \{s, t\}$). The *residual network* $N_f = (G_f, u_f, s, t)$ of N with respect to f is defined over the residual (multi)graph $G_f = (V, E \cup \bar{E})$, where $\bar{E} := \{(v, v') \in V \times V : (v', v) \in E\}$ is the set of reverse arcs (for simplicity of notation, we assume that G_f is a simple graph, and, in general E and $\bar{E}$ can no longer be expressed as sets of tuples). The costs of arcs $e \in \bar{E}$ in the residual network N_f are given by $c(e) := -c(\bar{e})$, where $\bar{e}$ denotes the reverse arc of e. The capacities u_f of the residual network are given by

$$u_f(e) := \begin{cases} u(e) - f(e) & \text{if} e \in E, \\ f(\bar{e}) & \text{if} e \in \bar{E}. \end{cases}$$

A *maximum flow* is a flow f that maximizes the flow *value* $|f| := \sum_{e \in \delta^+(s)} f(e) - \sum_{e \in \delta^-(s)} f(e)$. In the *minimum cost (maximum) flow problem*, we are given a network $N = (G, u, s, t)$ and need to find a maximum flow f that minimizes the cost $\sum_{e \in E} c(e) \cdot f(e)$ with respect to given arc costs $c : E \to \mathbb{R}$.

In the *parametric minimum cost flow* problem, we are given a network N with arc costs c and need to determine a *parametric flow* $f : E \times \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$, such that, for all $\lambda \in \mathbb{R}_{\geq 0}$, we have that $f_\lambda(e) := f(e, \lambda)$ defines a minimum cost flow among all flows of value λ, if such flows exist.

Finally, a *flow over time* on a network with *transit times* $\tau : E \to \mathbb{R}_{\geq 0}$ is a function $f : E \times \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$, such that the following hold:

- $f_e(\theta) := f(e, \theta)$ is Lebesgue-integrable for every fixed $e \in E$.
- $f_e(\theta) \leq u(e)$ for all $e \in E$ and $\theta \in \mathbb{R}_{\geq 0}$.
- $\mathrm{ex}_f(v, \theta) := \sum_{e \in \delta^-(v)} \int_0^{\theta - \tau(e)} f_e(\xi)\, d\xi - \sum_{e \in \delta^+(v)} \int_0^{\theta} f_e(\xi)\, d\xi = 0$ for for every $\theta \in \mathbb{R}_{\geq 0}$ and every $v \in V \setminus \{s, t\}$.

An *earliest arrival flow* is a flow over time f that simultaneously maximizes $\mathrm{ex}_f(t, \theta)$ for all $\theta \in \mathbb{R}_{\geq 0}$. For more details regarding flows over time, we refer to Skutella [27].

3 SUCCESSIVE SHORTEST PATH ALGORITHM

Consider a network N with a source node s, a sink node t, and non-negative arc costs. The Successive Shortest Path Algorithm starts with the zero-flow and iteratively augments flow along a minimum-cost s-t-path in the current residual network, until a maximum s-t-flow has been found. Note that the residual network is a subnetwork of N's bidirected network, where the cost of a backward arc is the negative of the cost of the corresponding forward arc.

3.1 A Counting Gadget for the Successive Shortest Path Algorithm

In this section, we construct a family of networks for which the Successive Shortest Path Algorithm takes an exponential number of iterations. Assume we have a network N_{i-1} with source s_{i-1} and sink t_{i-1} which requires 2^{i-1} iterations that each augment one unit of flow. We can obtain a new network N_i with only two additional nodes s_i, t_i for which the Successive Shortest Path Algorithm takes 2^i iterations. To do this, we add two arcs (s_i, s_{i-1}), (t_{i-1}, t_i) with capacity 2^{i-1} and cost 0, and two arcs (s_i, t_{i-1}), (s_{i-1}, t_i) with capacity 2^{i-1} and very high cost. The idea is that, in the first 2^{i-1} iterations, one unit of flow is routed along the arcs of cost 0 and through N_{i-1}. After 2^{i-1} iterations, both the arcs (s_i, s_{i-1}), (t_{i-1}, t_i) and the subnetwork N_{i-1} are completely saturated and the Successive Shortest Path Algorithm starts to use the expensive arcs (s_i, t_{i-1}), (s_{i-1}, t_i). Each of

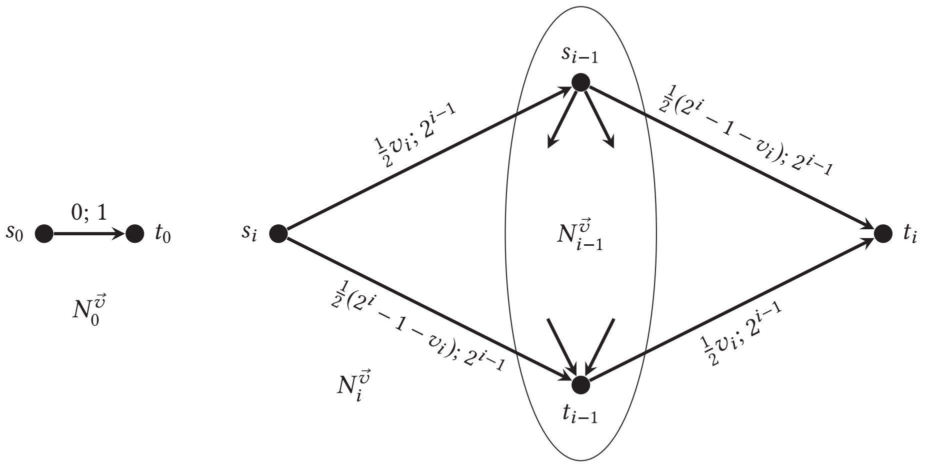
s_0
t_0
$0; 1$
$N_0^{\vec{v}}$
s_i
t_i
s_{i-1}
t_{i-1}
$N_{i-1}^{\vec{v}}$
$N_i^{\vec{v}}$
$\frac{1}{2}v_i; 2^{i-1}$
$\frac{1}{2}(2^i - 1 - v_i); 2^{i-1}$
$\frac{1}{2}(2^i - 1 - v_i); 2^{i-1}$
$\frac{1}{2}v_i; 2^{i-1}$

Fig. 1. Recursive definition of the counting gadget $N_i^{\vec{v}}$ for the Successive Shortest Path Algorithm and $\vec{v} \in \{\vec{a}, -\vec{a}\}$. Arcs are labeled by their cost and capacity in this order. The cost of the shortest s_i-t_i-path in iteration $j = 0, \dots, 2^i - 1$ is $j + \vec{v}_i^{[j]}$.

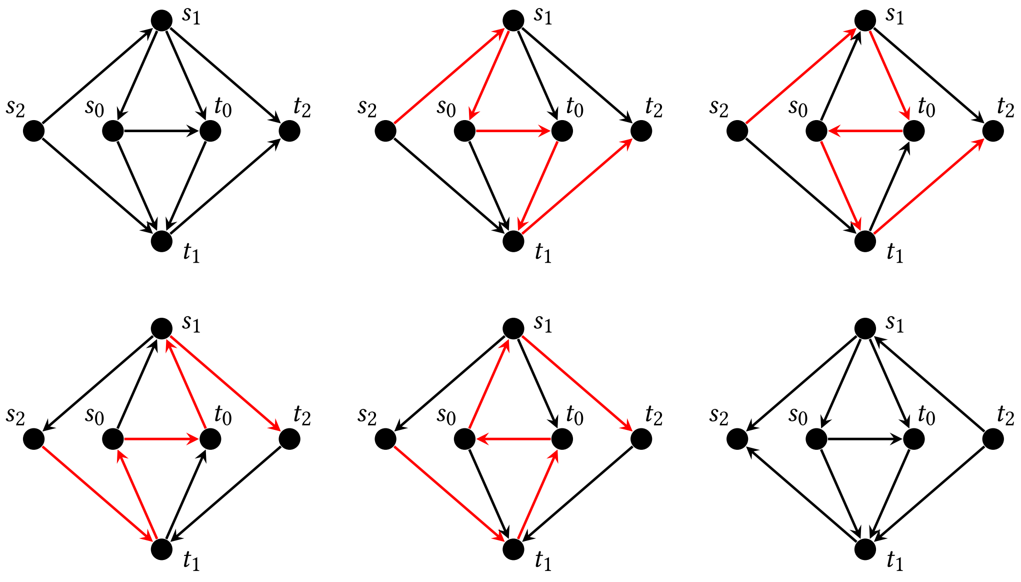

Fig. 2. Illustration of the iterations performed by the Successive Shortest Path Algorithm on the counting gadget $N_2^{\vec{a}}$. The shortest path in each iteration is marked in red, and arcs are oriented in the direction in which they are used next. Note that, after $2^i = 4$ iterations, the configuration is the same as in the beginning if we switch the roles of s_2 and t_2.

the next 2^{i-1} iteration adds one unit of flow along the expensive arcs and removes one unit of flow from the subnetwork N_{i-1}.

We tune the cost of the expensive arcs to $2^{i-1} - \frac{1}{2}$, which turns out to be just expensive enough (see Figure 1, with $v_i = 0$). This leads to a particularly nice progression of the costs of shortest paths, where the shortest path in iteration $j = 0, 1, \dots, 2^i - 1$ simply has cost j (Figure 2).

Our goal is to use this counting gadget to iterate over all candidate solutions for a Partition instance $\vec{v}$ (we later use the gadget for $\vec{v} \in \{\vec{a}, -\vec{a}\}$, where $\vec{a}$ is the fixed partition instance of Section 2). Motivated by Proposition 2.1, we perturb the costs of the arcs in such a way that the shortest path in iteration j has cost $j + \vec{v}_i^{[j]}$. We achieve this by adding $\frac{1}{2}v_i$ to the cheap arcs (s_i, s_{i-1}), (t_{i-1}, t_i) and subtracting $\frac{1}{2}v_i$ from the expensive arcs (s_i, t_{i-1}), (s_{i-1}, t_i). If the value of v_i is small enough, this modification does not affect the overall behavior of the gadget. The first 2^{i-1} iterations now have an additional cost of v_i while the next 2^{i-1} iterations have an additional cost of $-v_i$, which leads to the desired cost when the modification is applied recursively.

Figure 1 shows the recursive construction of our counting gadget $N_n^{\vec{v}}$ that encodes the Partition instance $\vec{v}$. The following lemma formally establishes the crucial properties of the construction.

Lemma 3.1. *For $\vec{v} \in \{\vec{a}, -\vec{a}\}$ and $i = 1, \dots, n$, the Successive Shortest Path Algorithm applied to network $N_i^{\vec{v}}$ with source s_i and sink t_i needs 2^i iterations to find a maximum s_i-t_i-flow of minimum cost. In each iteration $j = 0, 1, \dots, 2^i - 1$, the algorithm augments one unit of flow along a path of cost $j + \vec{v}_i^{[j]}$ in the residual network.*

Proof. We prove the lemma by induction on i, together with the additional property that, after 2^i iterations, none of the arcs in $N_{i-1}^{\vec{v}}$ carries any flow, while the arcs in $N_i^{\vec{v}} \setminus N_{i-1}^{\vec{v}}$ are fully saturated. First consider the network $N_0^{\vec{v}}$. In each iteration where $N_0^{\vec{v}}$ does not carry flow, one unit of flow can be routed from s_0 to t_0. Conversely, when $N_0^{\vec{v}}$ is saturated, one unit of flow can be routed from t_0 to s_0. In either case, the associated cost is 0. With this in mind, it is clear that on $N_1^{\vec{v}}$ the Successive Shortest Path Algorithm terminates after two iterations. In the first, one unit of flow is sent along the path s_1, s_0, t_0, t_1 of cost $v_1 = \vec{v}_1^{[0]}$. In the second iteration, one unit of flow is sent along the path s_1, t_0, s_0, t_1 of cost $-v_1 = \vec{v}_1^{[1]}$. Afterward, the arc (s_0, t_0) does not carry any flow, while all other arcs are fully saturated.

Now assume the claim holds for $N_{i-1}^{\vec{v}}$ and consider network $N_i^{\vec{v}}$, $i > 1$. Observe that every path using either of the arcs (s_i, t_{i-1}) or (s_{i-1}, t_i) has a cost of more than $2^{i-1} - 3/4$. To see this, note that the cost of these arcs is bounded individually by $\frac{1}{2}(2^i - 1 - v_i) > 2^{i-1} - 3/4$, since $|v_i| < A < 1/4$. On the other hand, it can be seen inductively that the shortest t_{i-1}-s_{i-1}-path in the bidirected network associated with $N_{i-1}^{\vec{v}}$ has cost at least $-2^{i-1} + 1 - A > -2^{i-1} + 3/4$. Hence, using both (s_i, t_{i-1}) and (s_{i-1}, t_i) in addition to a path from t_{i-1} to s_{i-1} incurs cost at least $2^{i-1} - 3/4$. By induction, in every iteration $j < 2^{i-1}$, the Successive Shortest Path Algorithm thus does not use the arcs (s_i, t_{i-1}) or (s_{i-1}, t_i) but instead augments one unit of flow along the arcs (s_i, s_{i-1}), (t_{i-1}, t_i) and along an s_{i-1}-t_{i-1}-path of cost $j + \vec{v}_{i-1}^{[j]} < 2^{i-1} - 3/4$ through the subnetwork $N_{i-1}^{\vec{v}}$. The total cost of this s_i-t_i-path is $v_i + (j + \vec{v}_{i-1}^{[j]}) = j + \vec{v}_i^{[j]}$, since $j < 2^{i-1}$.

After 2^{i-1} iterations, the arcs (s_i, s_{i-1}) and (t_{i-1}, t_i) are both fully saturated, as well as (by induction) the arcs in $N_{i-1}^{\vec{v}} \setminus N_{i-2}^{\vec{v}}$, while all other arcs are without flow. Consider the residual network of $N_{i-1}^{\vec{v}}$ at this point. If we increase the costs of the four residual arcs in $N_{i-1}^{\vec{v}} \setminus N_{i-2}^{\vec{v}}$ by $\frac{1}{2}(2^{i-1} - 1)$ and switch the roles of s_{i-1} and t_{i-1}, we obtain back the original subnetwork $N_{i-1}^{\vec{v}}$. The shift of the residual costs effectively makes every t_{i-1}-s_{i-1}-path more expensive by $2^{i-1} - 1$, but does not otherwise affect the behavior of the network. We can thus use induction again to infer that, in every iteration $j = 2^{i-1}, \dots, 2^i - 1$, the Successive Shortest Path Algorithm augments one unit of flow along a path via $s_i, t_{i-1}, N_{i-1}^{\vec{v}}, s_{i-1}, t_i$. Accounting for the shift in cost by $2^{i-1} - 1$, we obtain that this path has a total cost of

$$(2^i - 1 - v_i) + \left(j - 2^{i-1} + \vec{v}_{i-1}^{[j - 2^{i-1}]}\right) - (2^{i-1} - 1) = j + \vec{v}_i^{[j]},$$

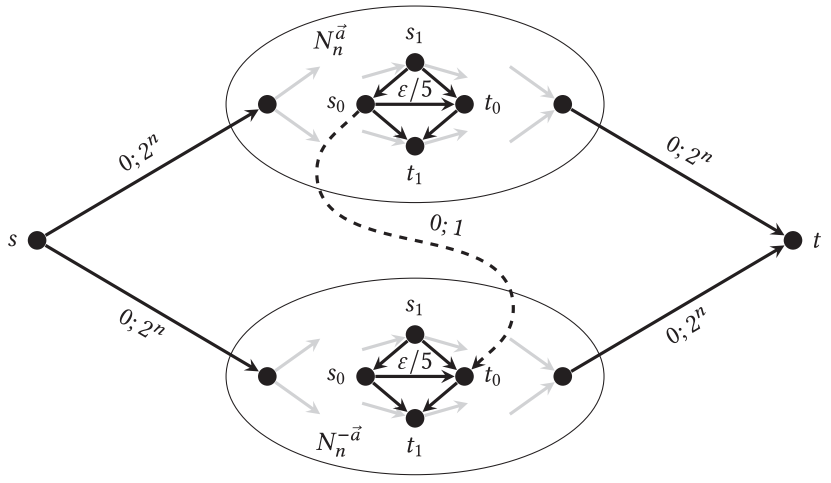

Fig. 3. Illustration of network $G_{\text{ssp}}^{\vec{a}}$. The subnetworks $N_n^{\vec{a}}$ and $N_n^{-\vec{a}}$ are advanced independently by the Successive Shortest Path Algorithm without using arc e, unless the Partition instance $\vec{a}$ has a solution.

where we used $\vec{v}_{i-1}^{[j-2^{i-1}]} = \vec{v}_{i-1}^{[j]}$ and $\vec{v}_{i-1}^{[j]} - v_i = \vec{v}_i^{[j]}$ for $j \in [2^{i-1}, 2^i)$. After 2^i iterations the arcs in $N_i^{\vec{v}} \setminus N_{i-1}^{\vec{v}}$ are fully saturated and all other arcs carry no flow. □

3.2 The Successive Shortest Path Algorithm Implicitly Solves Partition

We use the counting gadget of Section 3.1 to prove Theorem 1.4 for the Successive Shortest Path Algorithm. Let $G_{\text{ssp}}^{\vec{a}}$ be the network consisting of the two gadgets $N_n^{\vec{a}}$, $N_n^{-\vec{a}}$, connected to a new source node s and a new sink t (Figure 3). In both gadgets, we add the arcs (s, s_n) and (t_n, t) with capacity 2^n and cost 0. We introduce an additional arc e (dashed in the figure) of capacity 1 and cost 0 from node s_0 of gadget $N_n^{\vec{a}}$ to node t_0 of gadget $N_n^{-\vec{a}}$. Finally, we increase the costs of the arcs (s_0, t_0) in both gadgets from 0 to $\frac{1}{5}\varepsilon$. Recall that $\varepsilon > 0$ is related to $\vec{a}$ by the fact that all a_i's are multiples of ε (i.e., a cost smaller than ε is insignificant compared to all other costs).

Lemma 3.2. *The Successive Shortest Path Algorithm on network $G_{\text{ssp}}^{\vec{a}}$ augments flow along arc e if and only if the* Partition *instance $\vec{a}$ has a solution.*

Proof. First observe that our slight modification of the cost of arc (s_0, t_0) in both gadgets $N_n^{\vec{a}}$ and $N_n^{-\vec{a}}$ does not affect the behavior of the Successive Shortest Path Algorithm. This is because the cost of any path in G is perturbed by at most $\frac{2}{5}\varepsilon$, and hence the shortest path remains the same in every iteration. The only purpose of the modification is tie-breaking.

Consider the behavior of the Successive Shortest Path Algorithm on the network $G_{\text{ssp}}^{\vec{a}}$ with arc e removed. In each iteration, the shortest s-t-path goes via one of the two gadgets. By Lemma 3.1, each gadget can be in one of $2^n + 1$ states, and we number these states increasingly from 0 to 2^n by the order of their appearance during the execution of the Successive Shortest Path Algorithm. The shortest s-t-path through either gadget in state $j = 0, \ldots, 2^n - 1$ has a cost in the range $[j - A, j + A]$, and hence it is cheaper to use a gadget in state j than the other gadget in state $j + 1$. This means that after every two iterations, both gadgets are in the same state.

Now consider the network $G_{\text{ssp}}^{\vec{a}}$ with arc e put back. We show that, as before, if the two gadgets are in the same state before iteration $2j$, $j = 0, \ldots, 2^n - 1$, then they are again in the same state two iterations later. More importantly, arc e is used in iterations $2j$ and $2j+1$ if and only if $\vec{a}_n^{[j]} = 0$. This proves the lemma since, by Proposition 2.1, $\vec{a}_n^{[j]} = 0$ for some $j < 2^n$ if and only if the Partition instance $\vec{a}$ has a solution.

To prove our claim, assume that both gadgets are in the same state before iteration $2j$. Let P^+ be the shortest s-t-path that does not use any arc of $N_n^{-\vec{a}}$, P^- be the shortest s-t-path that does not use any arc of $N_n^{\vec{a}}$, and P be the shortest s-t-path using arc e. Note that one of these paths is the overall shortest s-t-path. We distinguish two cases, depending on whether the arc (s_0, t_0) currently carries flow 0 or 1 in both gadgets.

If (s_0, t_0) carries flow 0, then P^+, P^- use arc (s_0, t_0) in forward direction. Therefore, by Lemma 3.1, the cost of P^+ is $j + \vec{a}_n^{[j]} + \frac{1}{5}\varepsilon$, while the cost of P^- is $j - \vec{a}_n^{[j]} + \frac{1}{5}\varepsilon$. On the other hand, path P follows P^+ to node s_0 of $N_n^{\vec{a}}$, then uses arc e, and finally follows P^- to t. The cost of this path is exactly j. If $\vec{a}_n^{[j]} \neq 0$, then one of P^+, P^- is cheaper than P, and the next two iterations augment flow along paths P^+ and P^-. Otherwise, if $\vec{a}_n^{[j]} = 0$, then P is the shortest path, followed in the next iteration by the path from s to node t_0 of $N_n^{-\vec{a}}$ along P^-, along arc e in the backward direction to node s_0 of $N_n^{\vec{a}}$, and finally to t along P^+, for a total cost of $j + \frac{2}{5}\varepsilon$.

If (s_0, t_0) carries flow 1, then P^+, P^- use arc (s_0, t_0) in a backward direction. By Lemma 3.1, the cost of P^+ is $j + \vec{a}_n^{[j]} - \frac{1}{5}\varepsilon$, while the cost of P^- is $j - \vec{a}_n^{[j]} - \frac{1}{5}\varepsilon$. On the other hand, path P follows P^+ to node s_0 of $N_n^{\vec{a}}$, then uses arc e, and finally follows P^- to t. The cost of this path is $j - \frac{2}{5}\varepsilon$. If $\vec{a}_n^{[j]} \neq 0$, then one of P^+, P^- is cheaper than P, and the next two iterations augment flow along paths P^+ and P^-. Otherwise, if $\vec{a}_n^{[j]} = 0$, then P is the shortest path, followed in the next iteration by the path from s to node t_0 of $N_n^{-\vec{a}}$ along P^-, along arc e in backwards direction to node s_0 of $N_n^{\vec{a}}$, and finally to t along P^+, for a total cost of j. □

We assume that a single bit of the complete configuration of the Turing machine corresponding to the Successive Shortest Path Algorithm can be used to distinguish whether arc e carries a flow of 0 or a flow of 1 during the execution of the algorithm and that the identity of this bit can be determined in polynomial time. Under this natural assumption, we get the following result, which implies Theorem 1.4 for the Successive Shortest Path Algorithm.

Corollary 3.3. *The Successive Shortest Path Algorithm solves* Partition *implicitly.*

4 SIMPLEX ALGORITHM AND NETWORK SIMPLEX ALGORITHM

In this section, we adapt our construction for the Simplex Algorithm and, in particular, for its interpretation for the minimum-cost flow problem, the Network Simplex Algorithm. In this specialized version of the Simplex Algorithm, a basic feasible solution is specified by a spanning tree T such that the flow value on each arc of the network not contained in T is either zero or equal to its capacity. We refer to this tree simply as the basis or the spanning tree. The reduced cost of a residual non-tree arc e equals the cost of sending one unit of flow in the direction of e around the unique cycle obtained by adding e to T. For a pair of nodes, the unique path connecting these nodes in the spanning tree T is referred to as the *tree-path* between the two nodes. Note that while we set up the initial basis and flow manually in the constructions of the following sections, determining the initial feasible flow algorithmically via the algorithm of Edmonds and Karp, ignoring arc costs, yields the same result. Our construction ensures that all intermediate solutions of the Network Simplex Algorithm are nondegenerate. Moreover, in every iteration there is a unique non-tree arc of minimum reduced cost which is used as a pivot element.

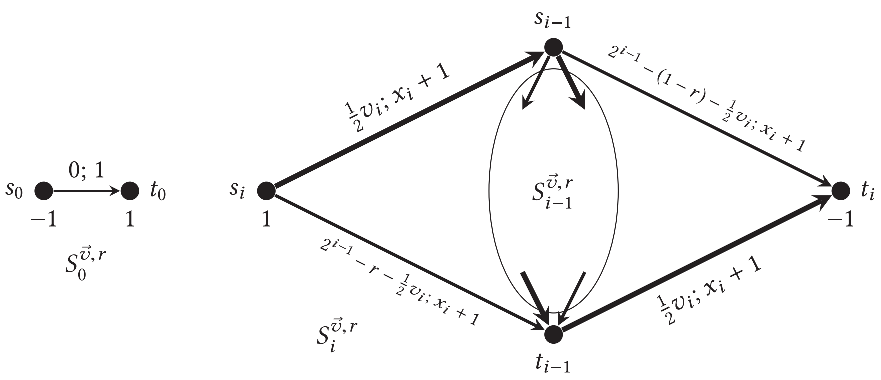
s_0
t_0
$0; 1$
-1
1
$S_0^{\vec{v},r}$
s_i
1
s_{i-1}
t_{i-1}
t_i
-1
$\frac{1}{2}v_i; x_i + 1$
$2^{i-1} - (1-r) - \frac{1}{2}v_i; x_i + 1$
$2^{i-1} - r - \frac{1}{2}v_i; x_i + 1$
$\frac{1}{2}v_i; x_i + 1$
$S_{i-1}^{\vec{v},r}$
$S_i^{\vec{v},r}$

Fig. 4. Recursive definition of the counting gadget $S_i^{\vec{v},r}$ for the Network Simplex Algorithm, $\vec{v} \in \{\vec{a}, -\vec{a}\}$, and a parameter $r \in (2A, 1-2A)$, $r \neq 1/2$. The capacities of the arcs of $S_i^{\vec{a},r} \setminus S_{i-1}^{\vec{a},r}$ are $x_i + 1 = 3 \cdot 2^{i-1}$. If we guarantee that there always exists a tree-path from t_i to s_i with sufficiently negative cost outside of the gadget, the cost of iteration $3k$, $k = 0, \ldots, 2^i - 1$, within the gadget is $k + \vec{v}_i^{[k]}$. Bold arcs are in the initial basis and carry a flow of at least 1 throughout the execution.

4.1 A Counting Gadget for the Network Simplex Algorithm

We design a counting gadget for the Network Simplex Algorithm (Figure 4), similar to the gadget $N_i^{\vec{v}}$ of Section 3.1 for the Successive Shortest Path Algorithm. Since the Network Simplex Algorithm augments flow along cycles obtained by adding one arc to the current spanning tree, we assume that the tree always contains an external tree-path from the sink of the gadget to its source with a very low (negative) cost. This assumption will be justified in Section 4.2, when we embed the counting gadget into a larger network.

The main challenge when adapting the gadget $N_i^{\vec{v}}$ is that the spanning trees in consecutive iterations of the Network Simplex Algorithm differ in one arc only, since in each iteration a single arc may enter the basis. However, successive shortest paths in $N_i^{\vec{v}}$ differ by exactly two tree-arcs between consecutive iterations. We obtain a new gadget $S_i^{\vec{v}}$ from $N_i^{\vec{v}}$ by modifying arc capacities in such a way that we get two intermediate iterations between every consecutive pair of successive shortest paths in $N_i^{\vec{v}}$. These iterations serve as a transition between the two paths and their corresponding spanning trees. Recall that in $N_i^{\vec{v}}$ the capacities of the arcs of $N_i^{\vec{v}} \setminus N_{i-1}^{\vec{v}}$ are exactly the same as the capacity of the subnetwork $N_{i-1}^{\vec{v}}$. In $S_i^{\vec{v}}$, we increase the capacity of the additional arcs by one unit relative to the capacity of $S_{i-1}^{\vec{v}}$. The resulting capacities of the arcs in $S_i^{\vec{v}} \setminus S_{i-1}^{\vec{v}}$ are x_i (for the moment), where $x_i = 2x_{i-1} + 1$ and $x_1 = 2$ (i.e., $x_i = 3 \cdot 2^{i-1} - 1$).

Similar to before, after $2x_{i-1}$ iterations, the subnetwork $S_{i-1}^{\vec{v}}$ is saturated. In contrast, however, at this point the arcs (s_i, s_{i-1}), (t_{i-1}, t_i) are not saturated yet. Instead, in the next two iterations, the arcs (s_i, t_{i-1}), (s_{i-1}, t_i) enter the basis and one unit of flow gets sent via the paths s_i, s_{i-1}, t_i and s_i, t_{i-1}, t_i, which saturates the arcs (s_i, s_{i-1}), (t_{i-1}, t_i) and eliminates them from the basis. Afterward, in the next $2x_{i-1}$ iterations, flow is sent via (s_i, t_{i-1}), (s_{i-1}, t_i) and through $S_{i-1}^{\vec{v}}$ as before (see Figure 5 for an example execution of the Network Simplex Algorithm on $S_2^{\vec{v}}$).

For the construction to work, we need that, in every nonintermediate iteration, arc (s_0, t_0) not only enters the basis but, more importantly, is also the unique arc to leave the basis. In other words, we want to ensure that no other arc becomes tight in these iterations. For this purpose, we add an initial flow of 1 along the paths $s_i, s_{i-1}, \ldots, s_0$ and $t_0, t_1, \ldots, t_i$ by adding supply 1 to s_i, t_0 and demand 1 to s_0, t_i and increasing the capacities of the affected arcs by 1 (Figure 6). Note that the path used by the initial flow has lowest cost for a flow of 1. The arcs of the two paths are the only

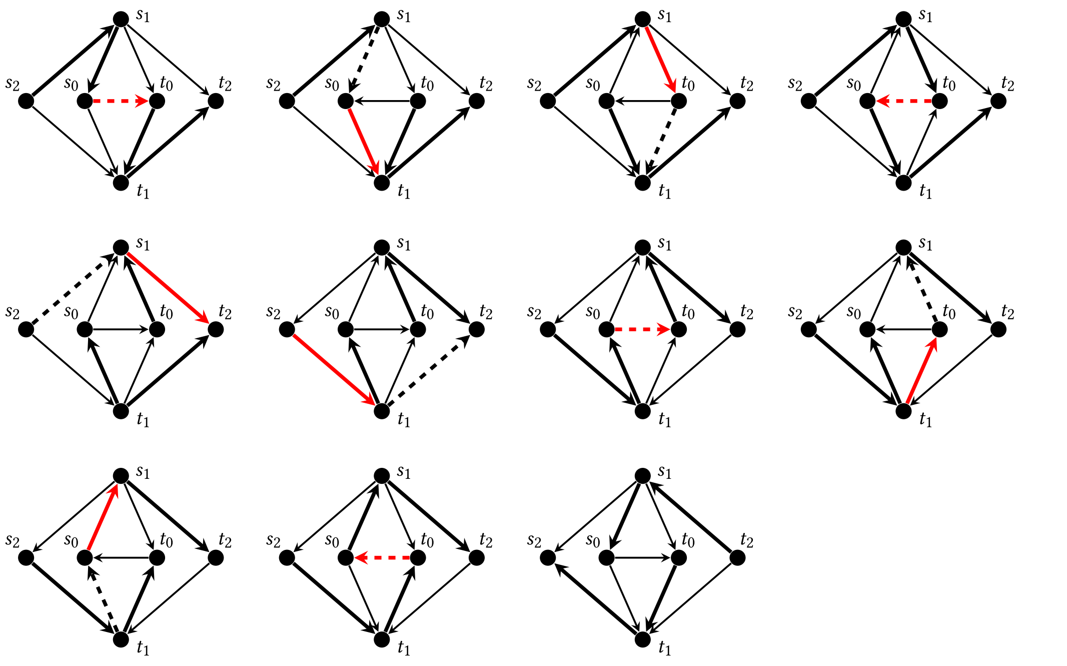

Fig. 5. Illustration of the iterations performed by the Network Simplex Algorithm on the counting gadget $S_2^{\vec{a},r}$ for $r < 1/2$. The external tree-path from t_2 to s_2 is not shown. Bold arcs are in the basis before each iteration, the red arc enters the basis, and the dashed arc exits the basis. Arcs are oriented in the direction in which they are used next. Note that after $2x_2 = 3 \cdot 2^2 - 2 = 10$ iterations, the configuration is the same as in the beginning if we switch the roles of s_2 and t_2.

arcs from the gadget that are contained in the initial spanning tree. We also increase the capacities of the arcs (s_i, t_{i-1}), (s_{i-1}, t_i) by one to ensure that these arcs are never saturated.

Finally, we also make sure that, in every iteration, the arc entering the basis is unique. To achieve this, we introduce a parameter $r \in (2A, 1-2A)$, $r \neq 1/2$ and replace the costs of $2^{i-1} - \frac{1}{2} - \frac{1}{2}v_i$ of the arcs (s_i, t_{i-1}), (s_{i-1}, t_i) by new costs $2^{i-1} - r - \frac{1}{2}v_i$ and $2^{i-1} - (1-r) - \frac{1}{2}v_i$, respectively.

We later use the final gadget $S_n^{\vec{v},r}$ as part of a larger network G by connecting the nodes s_n, t_n to nodes in $G \setminus S_n^{\vec{v},r}$. The following lemma establishes the crucial properties of the gadget used in such a way as a part of a larger network G.

Lemma 4.1. *Let $S_i^{\vec{v},r}$, $\vec{v} \in \{\vec{a}, -\vec{a}\}$, be part of a larger network G and assume that, before every iteration of the Network Simplex Algorithm on G where flow is routed through $S_i^{\vec{v},r}$ there is a tree-path from t_i to s_i in the residual network of G that has cost smaller than -2^{i+1} and capacity greater than 1. Then, there are exactly $2x_i = 3 \cdot 2^i - 2$ iterations in which one unit of flow is routed from s_i to t_i along arcs of $S_i^{\vec{v},r}$. Moreover:*

(1) *In iteration $j = 3k$, $k = 0, \dots, 2^i - 1$, arc (s_0, t_0) enters the basis carrying flow $k \bmod 2$ and immediately exits the basis again carrying flow $(k+1) \bmod 2$. The cost incurred by arcs of $S_i^{\vec{v},r}$ is $k + \vec{v}_i^{[k]}$.*

(2) *In iterations $j = 3k+1, 3k+2$, $k = 0, \dots, 2^i - 2$, for some $0 \le i' \le i$, the cost incurred by arcs of $S_i^{\vec{v},r}$ is $k + r + \vec{v}_{i',i}^{[k]}$ and $k + (1-r) + \vec{v}_{i',i}^{[k]}$ in order of increasing cost. One of the arcs $(s_{i'}, s_{i'-1})$, $(s_{i'-1}, t_{i'})$ and one of the arcs $(s_{i'}, t_{i'-1})$, $(t_{i'-1}, t_{i'})$ each enter and leave the basis in these iterations.*

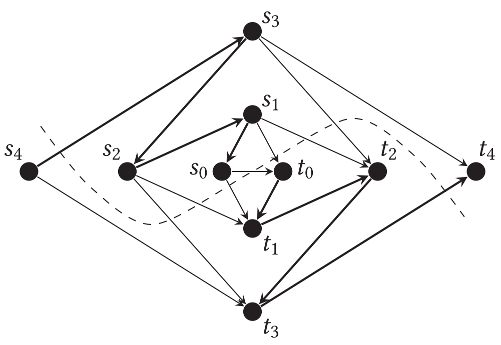

Fig. 6. Illustration of the initial flow in the network $S_4^{\vec{v},r}$. Thick arcs initially carry a single unit of flow. The indicated, directed s_4-t_4-cut (dashed) has capacity $2x_4 + 2$, which corresponds to a residual capacity of $2x_4$ when taking into account the initial flow.

Proof. First observe that throughout the execution of the Network Simplex Algorithm on G, one unit of flow must always be routed along both of the paths $s_i, s_{i-1}, \dots, s_0$ and $t_0, t_1, \dots, t_i$. This is because there is an initial flow of one along these paths, all of $s_0, \dots, s_{n-1}$ have in-degree 1, and all of $t_0, \dots, t_{n-1}$ have out-degree 1, which means that the flow cannot be rerouted.

We prove the lemma by induction on $i > 0$, together with the additional property, that, after $2x_i$ iterations, the arcs in $S_{i-1}^{\vec{v},r}$ carry their initial flow values, while the arcs in $S_i^{\vec{v},r} \setminus S_{i-1}^{\vec{v},r}$ all carry x_i additional units of flow (which implies that (s_i, s_{i-1}) and (t_{i-1}, t_i) are saturated). Also, the configuration of the basis is identical to the initial configuration, except that the membership in the basis of arcs in $S_i^{\vec{v},r} \setminus S_{i-1}^{\vec{v},r}$ is inverted. In the following, we assume that $r \in (2A, 1/2)$, which means that the arc (s_{i-1}, t_i) is always selected to enter the basis before the arc (s_i, t_{i-1}) : the case where $r \in (1/2, 1 - 2A)$ is analogous. In each iteration j, let P_j denote the tree-path outside of $S_i^{\vec{v},r}$ from t_i to s_i of cost $c_j < -2^{i+1}$ and capacity greater than 1.

For $i = 1$, the Network Simplex Algorithm performs the following four iterations involving $S_1^{\vec{v},r}$ (see Figure 2 for an illustration embedded in $S_2^{\vec{v},r}$). In the first iteration, (s_0, t_0) enters the basis and one unit of flow is routed along the cycle s_1, s_0, t_0, t_1, P_0 of cost $v_1 + c_0 = \vec{v}_1^{[0]} + c_0$. This saturates arc (s_0, t_0), which is the unique arc to become tight (since P_0 has capacity greater than 1) and thus exits the basis again. In the second iteration, (s_0, t_1) enters the basis and one unit of flow is routed along the cycle s_1, s_0, t_1, P_1 of cost $r + c_1 = r + \vec{v}_{1,1}^{[0]} + c_1$, thus saturating (together with the initial flow of 1) arc (s_0, s_1) of capacity $x_1 + 1 = 3$. Since P_1 has capacity greater than 1, this is the only arc to become tight and it thus exits the basis. In the third iteration, (s_1, t_0) enters the basis and one unit of flow is routed along the cycle s_1, t_0, t_1, P_2 of cost $(1 - r) + c_2 = (1 - r) + \vec{v}_{1,1}^{[0]} + c_2$. Similar to before, (t_0, t_1) is the only arc to become tight and thus exits the basis. In the fourth and final iteration, (s_0, t_0) enters the basis and one unit of flow is routed along the cycle s_1, t_0, s_0, t_1, P_3 of cost $1 - v_1 + c_3 = \vec{v}_1^{[1]} + c_3$, which causes (s_0, t_0) to become empty and leave the basis. Thus, after four iterations, arc (s_0, t_0) in $S_0^{\vec{v},r}$ carries its initial flow of value 0, while the arcs in $S_1^{\vec{v},r} \setminus S_0^{\vec{v},r}$ all carry $2 = x_1$ additional units of flow. Also, the arcs $(s_0, t_1), (s_1, t_0)$ replaced the arcs $(s_1, s_0), (t_0, t_1)$ in the basis.

To see, for $i > 0$, that $S_i^{\vec{v},r}$ is saturated after $2x_i$ units of flow have been routed from s_i to t_i, consider the directed s_i-t_i-cut induced by $\{s_i, t_{i-1}, t_{i-2}, \dots, t_0\}$ in the initial residual graph $S_i^{\vec{v},r}$. This cut consists of the arcs (s_i, s_{i-1}), (t_{i-1}, t_i). The capacity of the cut is exactly $2x_i + 2$ and the initial flow over the cut is 2 (Figure 6).

Now assume our claim holds for $S_{i-1}^{\vec{v},r}$ and $S_{i-1}^{\vec{v},1-r}$ and consider $S_i^{\vec{v},r}$. Consider the first $2x_{i-1}$ iterations $j = 0, \ldots, 2x_{i-1} - 1$ and set $k := \lfloor j/3 \rfloor < 2^{i-1}$. It can be seen inductively that the shortest path from t_{i-1} to s_{i-1} in the bidirected network associated with $S_{i-1}^{\vec{v},r}$ has cost at least $-2^{i-1} + 1 - A > -2^{i-1} + 1 - r$. Hence, every path from s_i to t_i using either or both of the arcs (s_i, t_{i-1}) or (s_{i-1}, t_i) has cost greater than $2^{i-1} - (1-r) - A > 2^{i-1} - 1 + A$. By induction, we can thus infer that none of these arcs enters the basis in iterations $j < 2x_{i-1}$, and instead an arc of $S_{i-1}^{\vec{v},r}$ enters (and exits) the basis and one unit of flow gets routed from s_i to t_i via the arcs $(s_i, s_{i-1}), (t_{i-1}, t_i)$. We may use induction here since, before iteration j, the path $t_{i-1}, t_i, P_j, s_i, s_{i-1}$ has cost $v_i + c_j < v_i - 2^{i+1} < -2^i$ and its capacity is greater than 1, since both $(s_i, s_{i-1}), (t_{i-1}, t_i)$ have capacity $x_i + 1 = 2x_{i-1} + 2$, leaving one unit of spare capacity even after a flow of $2x_{i-1}$ has been routed along them in addition to the initial unit of flow. The additional cost contributed by arcs $(s_i, s_{i-1}), (t_{i-1}, t_i)$ is v_i, which is in accordance with our claim since $\vec{v}_{\ell,i-1}^{[k]} + v_i = \vec{v}_{\ell,i}^{[k]}$ for all $\ell \in \{0, \ldots, i-1\}$ and $k \in \{0, \ldots, 2^{i-1} - 1\}$.

Because $S_{i-1}^{\vec{v},r}$ is fully saturated after $2x_{i-1}$ iterations, in the next iteration $j = 2x_{i-1} = 3 \cdot 2^{i-1} - 2$, $k := \lfloor j/3 \rfloor = 2^{i-1} - 1$, arc (s_{i-1}, t_i) is added to the basis and one unit of flow is sent along the path s_i, s_{i-1}, t_i, thus saturating the capacity $x_i + 1 = 2x_{i-1} + 2$ of arc (s_i, s_{i-1}) and incurring a cost of $2^{i-1} - (1-r) = k + r + \vec{v}_{i,i}^{[k]}$. Note that this cost is higher than the cost of each of the previous iterations. The saturated arc has to exit the basis since, by assumption, P_j has capacity greater than 1. Similarly, in the following iteration $j = 2x_{i-1} + 1 = 3 \cdot 2^{i-1} - 1$, $k := \lfloor j/3 \rfloor = 2^{i-1} - 1$, the cost is $2^{i-1} - r = k + (1-r) + \vec{v}_{i,i}^{[k]}$ and arc (t_{i-1}, t_i) is replaced by (s_i, t_{i-1}) in the basis.

By induction, at this point (s_{i-2}, t_{i-1}) and (s_{i-1}, t_{i-2}) are in the basis, the arcs of $S_{i-1}^{\vec{v},r} \setminus S_{i-2}^{\vec{v},r}$ carry a flow of x_{i-1} in addition to their initial flow, and $S_{i-2}^{\vec{v},r}$ is back to its initial configuration. To be able to apply induction on the residual network of $S_{i-1}^{\vec{v},r}$, we shift the costs of the arcs at s_{i-1} by $-(2^{i-2} - r)$ and the costs of the arcs at t_{i-1} by $-(2^{i-2} - (1-r))$ in the residual network of $S_{i-1}^{\vec{v},r}$. Since we shift costs uniformly across cuts, this only affects the costs of paths but not the structural behavior of the gadget. Specifically, the costs of all paths from t_{i-1} to s_{i-1} in the residual network are increased by exactly $2^{i-1} - 1$. If we switch the roles of s_{i-1} and t_{i-1}, say $\tilde{s}_{i-1} := t_{i-1}$ and $\tilde{t}_{i-1} := s_{i-1}$, we obtain the residual network of $S_{i-1}^{\vec{v},1-r}$ with its initial flow. This allows us to use induction again for the next $2x_{i-1}$ iterations.

To apply the induction hypothesis, we need the tree-path from $\tilde{t}_{i-1} = s_{i-1}$ to $\tilde{s}_{i-1} = t_{i-1}$ to maintain cost smaller than -2^i and capacity greater than 1. This is fulfilled since P_j has cost smaller than -2^{i+1}, which is sufficient even with the additional cost of $2^i - 1 - v_i$ incurred by arcs $(s_i, \tilde{s}_{i-1}), (\tilde{t}_{i-1}, t_i)$. The residual capacity of $(t_i, \tilde{t}_{i-1})$ and $(\tilde{s}_{i-1}, s_i)$ is $x_i > 2x_{i-1}$ and thus sufficient as well. By induction for $S_{i-1}^{\vec{v},1-r}$, we may thus conclude that in iterations $j = 2x_{i-1} + 2, \ldots, 2x_i - 1$, $k := \lfloor j/3 \rfloor \geq 2^{i-1}$, one unit of flow is routed via $(s_i, t_{i-1}), S_{i-1}^{\vec{v},r}, (s_{i-1}, t_i)$. The cost of $(s_i, \tilde{s}_{i-1})$ and $(\tilde{t}_{i-1}, t_i)$ together is $2^i - 1 - v_i$. The cost of iteration $j' = j - 2x_{i-1} - 2$, $k' := \lfloor j'/3 \rfloor = k - 2^{i-1}$, in $S_{i-1}^{\vec{v},1-r}$ is $k' + y + \vec{v}_{\ell,i-1}^{[k']}$, for $y \in \{0, r, (1-r)\}$ and $\ell \in \{0, \ldots, i-1\}$ chosen according to the different cases of the lemma. Accounting for the shift by $2^{i-1} - 1$ of the cost compared with the residual network of $S_{i-1}^{\vec{v},r}$, the incurred total cost in $S_{i-1}^{\vec{v},r}$ is

$$\begin{aligned}(2^i - 1 - v_i) + \left(k' + y + \vec{v}_{\ell,i-1}^{[k']}\right) - (2^{i-1} - 1)\\ = 2^{i-1} + k' + y - v_i + \vec{v}_{\ell,i-1}^{[k']} = k + y + \vec{v}_{\ell,i}^{[k]},\end{aligned}$$

where we used $-v_i + \vec{v}_{\ell,i-1}^{[k']} = \vec{v}_{\ell,i}^{[k'+2^{i-1}]}$ since $k' < 2^{i-1}$. This concludes the proof. □

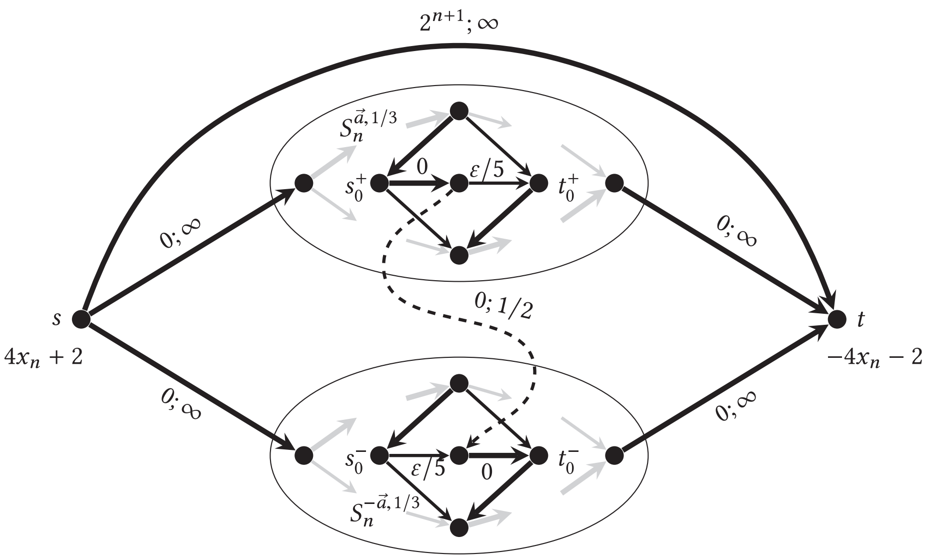

Fig. 7. Illustration of network $G_{\text{ns}}^{\vec{a}}$. The subnetworks $S_n^{\vec{a},1/3}$ and $S_n^{-\vec{a},1/3}$ are advanced independently by the Network Simplex Algorithm without using the dashed arc e, unless the Partition instance $\vec{a}$ has a solution. Bold arcs are in the initial basis and carry a flow of at least 1 throughout the execution of the algorithm.

4.2 The Network Simplex Algorithm Implicitly Solves Partition

We construct a network $G_{\text{ns}}^{\vec{a}}$ similar to the network $G_{\text{ssp}}^{\vec{a}}$ of Section 3.2. Without loss of generality, we assume that $a_1 = 0$. The network $G_{\text{ns}}^{\vec{a}}$ consists of the two gadgets $S_n^{\vec{a},1/3}$, $S_n^{-\vec{a},1/3}$, connected to a new source node s and a new sink t (Figure 7). Let s_i^+, t_i^+ denote the nodes of $S_n^{\vec{a},1/3}$ and s_i^-, t_i^- denote the nodes of $S_n^{-\vec{a},1/3}$. We introduce arcs (s, s_n^+), (s, s_n^-), (t_n^+, t), (t_n^-, t), each with capacity ∞ and cost 0. The supply 1 of s_n^+ and s_n^- is moved to s and the initial flow on arcs (s, s_n^+) and (s, s_n^-) is set to 1. Similarly, the demand 1 of t_n^+ and t_n^- is moved to t and the initial flow on arcs (t_n^+, t) and (t_n^-, t) is set to 1. Finally, we add an infinite capacity arc (s, t) of cost 2^{n+1}, increase the supply of s and the demand of t by $4x_n$, and set the initial flow on (s, t) to $4x_n$.

In addition, we add two nodes c^+, c^- and replace the arc (s_0^+, t_0^+) by two arcs (s_0^+, c^+), (c^+, t_0^+) of capacity 2 and cost 0 (for the moment), and analogously for the arc (s_0^-, t_0^-) and c^-. Finally, we move the demand of 1 from s_0^+ to c^+ and the supply of 1 from t_0^- to c^-. The arcs (s_0^+, c^+) and (c^-, t_0^-) carry an initial flow of 1 and are part of the initial basis. Observe that these modifications do not change the behavior of the gadgets. In addition to the properties of Lemma 4.1, we have that whenever the arc (s_0, t_0) previously carried a flow of 1, now (c^+, t_0^+) or (s_0^-, c^-) is in the basis, and, whenever (s_0, t_0) previously did not carry flow, now (s_0^+, c^+) or (c^-, t_0^-) is in the basis.

We increase the costs of the arcs (c^+, t_0^+) and (s_0^-, c^-) from 0 to $\frac{1}{5}\varepsilon$, again without affecting the behavior of the gadgets (note that we can perturb all costs in $S_n^{-\vec{a},1/3}$ further to ensure that every pivot step is unique). Finally, we add one more arc $e = (c^+, c^-)$ with cost 0 and capacity $\frac{1}{2}$.

Lemma 4.2. *Arc e enters the basis in some iteration of the Network Simplex Algorithm on network $G_{\text{ns}}^{\vec{a}}$ if and only if the* Partition *instance $\vec{a}$ has a solution.*

Proof. First observe that $\vec{a}_n^{[2k]} = \vec{a}_n^{[2k+1]}$ for $k \in 0, \ldots, 2^{n-1}$ since, by assumption, $a_1 = 0$.

Similar to the proof of Lemma 3.2, in isolation, each of the two gadgets can be in one of $2x_n$ states (Lemma 4.1), which we label by the number of iterations needed to reach each state. Assuming that both gadgets are in state $12k$ after some number of iterations, we show that both gadgets will reach state $12k + 12$ together as well. In addition, we show that, in the iterations in-between, arc e enters the basis if and only if $\vec{a}_n^{[4k]} = 0$ and thus $\vec{a}_n^{[4k+1]} = 0$, or $\vec{a}_n^{[4k+2]} = 0$ and thus $\vec{a}_n^{[4k+3]} = 0$. Consider the situation where both gadgets are in state $12k$. Note that in this state the arcs in $S_1^{\vec{v},1/3}$ and $S_1^{-\vec{v},1/3}$ are back in their original configuration.

Let $P^\pm$ denote the tree-path from $t_1^\pm$ to $s_1^\pm$, and let $P^{\pm\mp}$ denote the tree-path from $t_1^\mp$ to $s_1^\pm$. We refer to these paths as the *outer* paths. Observe that, since the gadgets are in the same state, the costs of the outer paths differ by at most $A < 1/4$. In the next iterations, flow is sent along a cycle containing one of the outer paths, and we analyze only the part of each cycle without the outer path. Let $P_0^\pm, P_1^\pm, P_2^\pm, P_3^\pm$ be the four successive shortest paths within the gadget $S_1^{\pm\vec{a},1/3}$. The costs of these paths are $\frac{1}{5}\varepsilon$, 1/3, 2/3, $1 - \frac{1}{5}\varepsilon$, respectively. Note that, since $A < 1/6$, the costs of the paths stay in the same relative order within each gadget throughout the algorithm.

If $\vec{a}_n^{[4k]} < 0$, then P^+ is the cheapest of the outer paths by a margin of more than $\varepsilon/2$. Thus, in the first iteration, (c^+, t_0^+) replaces (s_0^+, c^+) in the basis closing the path P_0^+. In the next five iterations, the paths P_0^-, P_1^+, P_1^-, P_2^+, P_2^- are closed in this order. The final two iterations are P_3^+, P_3^-, similar to the first two iterations, as $\vec{a}_n^{[4k+1]} = \vec{a}_n^{[4k]} < 0$. At this point, 8 iterations have passed and both gadgets are in state $12k + 6$.

If $\vec{a}_n^{[4k]} > 0$, then P^- is the cheapest of the outer paths by a margin of more than $\varepsilon/2$. Thus, the first iteration closes the path P_0^-. The next five iterations are via P_0^+, P_1^-, P_1^+, P_2^-, P_2^+, in this order. The final two iterations are P_3^-, P_3^+, similar to the first two iterations, as $\vec{a}_n^{[4k+1]} = \vec{a}_n^{[4k]} > 0$. At this point, 8 iterations have passed and both gadgets are in state $12k + 6$.

If $\vec{a}_n^{[4k]} = 0$, then all four outer paths have the same cost. The first iteration is via the path $s_1^+, s_0^+, c^+, c^-, t_0^-, t_1^-$; that is, arc e enters and leaves the basis for a cost of 0 and an additional flow of 1/2. The next two iterations are via $P_1^\pm$, each for a cost of $\frac{1}{5}\varepsilon$ and an additional flow of 1/2. The fourth iteration is via the path $s_1^-, s_0^-, c^-, c^+, t_0^+, t_1^+$ (i.e., arc e enters and leaves the basis again for a cost of $\frac{2}{5}\varepsilon$ and an additional flow of 1/2). The next iterations are as before: via P_1^+, P_1^-, P_2^-, P_2^+, in this order. The final four iterations are similar to the first four iterations, again twice using e, as $\vec{a}_n^{[4k+1]} = \vec{a}_n^{[4k]} = 0$. At this point, 12 iterations have passed and both gadgets are in state $12k + 6$.

The next four iterations (two for each gadget) do not involve the subnetworks $S_1^{\vec{a},1/3}$ and $S_1^{-\vec{a},1/3}$, and do thus not use e. The iterations going from state $12k + 6$ to state $12k + 12$ are analogous to the above if we exchange the roles of $s_1^\pm$ and $t_1^\pm$. This concludes the proof. □

Again, we assume that a single bit of the complete configuration of the Turing machine corresponding to the Simplex Algorithm can be used to detect whether a variable is in the basis and that the identity of this bit can be determined in polynomial time. Under this natural assumption, we get the following result, which implies Theorem 1.4 for the Network Simplex Algorithm and thus the Simplex Algorithm.

Corollary 4.3. *The Network Simplex Algorithm implicitly solves* Partition.

5 CONSEQUENCES

We now give proofs for the Corollaries of Section 1.

Corollary 1.5 *Determining the number of iterations needed by the Simplex Algorithm, the Network Simplex Algorithm, and the Successive Shortest Path Algorithm for a given input is* NP*-hard.*

Proof. We first show that determining the number of iterations needed by the Successive Shortest Path Algorithm for a given minimum-cost flow instance is NP-hard. We replace the arc e in

$G^{\vec{a}}_{\text{ssp}}$ of Section 3 by two parallel arcs, each with a capacity of 1/2 and slightly perturbed costs. This way, every execution of the Successive Shortest Path Algorithm that previously did not use arc e is unaffected, while executions using e require additional iterations. Thus, by Lemma 3.2, the Successive Shortest Path Algorithm on network $G^{\vec{a}}_{\text{ssp}}$ takes more than 2^{n+1} iterations if and only if the Partition instance $\vec{a}$ has a solution.

The proof for the Network Simplex Algorithm (and thus the Simplex Algorithm) follows from the proof of Lemma 4.2, observing that the Network Simplex Algorithm takes more than $4x_n$ iterations for network $G^{\vec{a}}_{\text{ns}}$ if and only if the Partition instance $\vec{a}$ has a solution. □

Corollary 1.6 *Deciding for a given linear program whether a given variable ever enters the basis during the execution of the Simplex Algorithm is* NP*-hard.*

Proof. The proof is immediate via Lemma 4.2 and the fact that Partition is NP-hard. □

Corollary 1.7 *Determining whether a parametric minimum-cost flow uses a given arc (i.e., assigns positive flow value for any parameter value) is* NP*-hard. In particular, determining whether the solution to a parametric linear program uses a given variable is* NP*-hard. Also, determining the number of different basic solutions over all parameter values is* NP*-hard.*

Proof. This follows from the fact that the Successive Shortest Path Algorithm solves a parametric minimum-cost flow problem, together with Lemma 3.2 and Corollary 1.5. □

Corollary 1.8 *Given a d-dimensional polytope P defined by a system of linear inequalities, determining the number of vertices of P's projection onto a given 2-dimensional subspace is* NP*-hard.*

Proof. Let P be the polytope of all feasible s-t-flows in network $G^{\vec{a}}_{\text{ssp}}$ of Section 3.2. Consider the 2-dimensional subspace S defined by flow value and cost of a flow. Let P' be the projection of P onto S. The lower envelope of P' is the parametric minimum-cost flow curve for $G^{\vec{a}}_{\text{ssp}}$, while the upper envelope is the parametric maximum-cost flow curve for $G^{\vec{a}}_{\text{ssp}}$.

The s-t-paths of maximum cost in $G^{\vec{a}}_{\text{ssp}}$ are the four paths via s_n, s_{n-1}, t_n or via s_n, t_{n-1}, t_n in both of the gadgets. Each of these paths has cost $2^{n-1} - \frac{1}{2}$, and the total capacity of all paths together is 2^{n+1} which is equal to the maximum flow value from s to t. Therefore, the upper envelope of P' consists of a single edge.

The number of edges on the lower envelope of P' is equal to the number of different costs among all successive shortest paths in $G^{\vec{a}}_{\text{ssp}}$. If we slightly perturb the costs of the two arcs in $G^{\vec{a}}_{\text{ssp}}$ with cost $\frac{1}{5}\varepsilon$, we can ensure that each successive shortest path has a unique cost. The claim then follows by Corollary 1.5. □

Corollary 1.9 *Determining the average arrival time of flow in an earliest arrival flow is NP-hard.*

Proof. Consider network $G^{\vec{a}}_{\text{ssp}}$ introduced in Section 3.2 and scale all arc costs by a sufficiently large integer to make them integral. Moreover, let $\xi := 1/2^{n+2}$ and change the cost of arc e in $G^{\vec{a}}_{\text{ssp}}$ from 0 to ξ. Notice that this modification does not change the sequence of paths chosen by the Successive Shortest Path Algorithm. Denote the resulting network by $G^{\vec{a}}_{\xi}$. Jarvis and Ratliff [15] proved that an earliest arrival flow has minimum average arrival time (and vice versa). We therefore consider in the following the earliest arrival flow on $G^{\vec{a}}_{\xi}$ that can be obtained from the paths found by the Successive Shortest Path Algorithm (see, e.g., Skutella [27] for details).

As argued in Section 3, the Successive Shortest Path Algorithm takes 2^{n+1} iterations on network $G^{\vec{a}}_{\xi}$. In each iteration $i = 0, \ldots, 2^{n+1} - 1$, it augments one unit of flow along some path P_i of cost $c(P_i)$ in the residual network. Notice that $c(P_i)$ is integral unless it contains arc e. In the latter case, $c(P_i) = z \pm \xi$ for some $z \in \mathbb{Z}_{\geq 0}$. Let $k := |\{i \in \{0, \ldots, 2^{n+1} - 1\} : P_i \text{ contains } e\}|$ such that $0 \leq k \leq 2^{n+1}$. In particular, $k = 0$ if and only if the Partition instance $\vec{a}$ has a solution.

An earliest arrival flow with integral time horizon $T > c(P_{2^{n+1}}) \geq c(P_i)$ sends flow at rate 1 into path P_i from time 0 up to time $T - c(P_i)$, for $i = 0, \ldots, 2^{n+1} - 1$. In particular, the total flow sent along P_i over time is $T - c(P_i)$. This flow arrives at the sink at rate 1 between time $c(P_i)$ and time T; its average arrival time is $\frac{1}{2}(T + c(P_i))$. Thus, the overall average arrival time of flow at the sink is

$$\frac{1}{F} \sum_{i=0}^{2^{n+1}-1} \Big(T - c(P_i)\Big) \tfrac{1}{2} \Big(T + c(P_i)\Big) = \frac{1}{2F} \sum_{i=0}^{2^{n+1}-1} \Big(T^2 - c(P_i)^2\Big) \ , \tag{1}$$

where F is the total amount of flow sent into the sink)i.e., the value of a maximum s-t-flow over time). Since T is integral, it follows from Equation (1) that $2F$ times the average arrival time is of the form $\alpha \pm \beta\xi - k\xi^2$ with $\alpha, \beta \in \mathbb{Z}_{\geq 0}$. Since $0 \leq k \leq 2^{n+1}$ and $\xi = 1/2^{n+2}$ divides α, this value is a multiple of ξ if and only if $k = 0$; that is, if and only if the Partition instance $\vec{a}$ has a solution.

Since the maximum value F of an s-t-flow over time can be computed in polynomial time [10], we can decide Partition by observing the minimum average arrival time of a maximum s-t-flow over time in $G_{\xi}^{\vec{a}}$. □

6 CONCLUSION

We have introduced the concept of *NP-mightiness* as a novel means of classifying the computational power of algorithms. Furthermore, we have given a justification for the exponential worst-case behavior of Successive Shortest Path Algorithm and the (Network) Simplex Method (with Dantzig's pivot rule): These algorithms can implicitly solve any problem in NP.

We hope that our approach will turn out to be useful in developing a better understanding of other algorithms that suffer from poor worst-case behavior. In particular, we believe that our results can be carried over to the Simplex Method with other pivot rules. Furthermore, even polynomial-time algorithms with a superoptimal worst-case running time are an interesting subject. Such algorithms might implicitly solve problems that are presumably more difficult than the problem they were designed for. In order to achieve meaningful results in this context, our definition of "implicitly solving" (Definition 1.1) would need to be modified by further restricting the running time of the transformation of instances.

Note that the decision problems underlying Corollaries 1.5 through 1.9 do not seem to lie in NP. For Corollary 1.6, Fearnley and Savani [8] have already shown PSPACE-hardness, and similar results may hold for the other problems. Determining the exact complexity of these problems remains an open question. Similarly, it would be interesting to investigate whether the Successive Shortest Path Algorithm is PSPACE-mighty.

REFERENCES

[1] I. Adler, C. H. Papadimitriou, and A. Rubinstein. 2014. On simplex pivoting rules and complexity theory. In *Proceedings of the 17th International Conference on Integer Programming and Combinatorial Optimization (IPCO)*. Springer, 13–24.

[2] N. Amenta and G. M. Ziegler. 1996. Deformed products and maximal shadows of polytopes. In *Proceedings of the AMS-IMS-SIAM Joint Summer Research Conference in the Mathematical Sciences*, Bernard Chazelle, Jacob E. Goodman and Richard Pollack (Eds.). American Mathematical Society, 57–90.

[3] R. G. Busacker and P. J. Gowen. 1960. *A Procedure for Determining a Family of Minimum-Cost Network Flow Patterns.* Technical Paper ORO-TP-15. Operations Research Office, The Johns Hopkins University, Bethesda, Maryland.

[4] G. B. Dantzig. 1951. Maximization of a linear function of variables subject to linear inequalities. In *Activity Analysis of Production and Allocation – Proceedings of a Conference*, Tj. C. Koopmans (Ed.). Wiley, 339–347.

[5] G. B. Dantzig. 1962. *Linear Programming and Extensions.* Princeton University Press.

[6] Y. Disser and M. Skutella. 2013. The simplex algorithm is NP-mighty. *CoRR* abs/1311.5935 (2013).

[7] Y. Disser and M. Skutella. 2015. The simplex algorithm is NP-mighty. In *Proceedings of the 26th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, 858–872.

[8] J. Fearnley and R. Savani. 2015. The complexity of the simplex method. In *Proceedings of the 47th ACM Symposium on Theory of Computing (STOC)*. ACM, 201–208.

[9] J. Fearnley and R. Savani. 2016. The complexity of all-switches strategy improvement. In *Proceedings of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, 130–139.

[10] L. R. Ford and D. R. Fulkerson. 1962. *Flows in Networks*. Princeton University Press.

[11] D. Gale. 1959. Transient flows in networks. *Michigan Mathematical Journal* 6 (1959), 59–63.

[12] M. R. Garey and D. S. Johnson. 1979. *Computers and Intractability, A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company.

[13] P. W. Goldberg, C. H. Papdimitriou, and R. Savani. 2013. The complexity of the homotopy method, equilibrium selection, and lemke-howson solutions. *ACM Transactions on Economics and Computation* 1, 2 (2013), 1–25.

[14] M. Iri. 1960. A new method of solving transportation-network problems. *Journal of the Operations Research Society of Japan* 3 (1960), 27–87.

[15] J. J. Jarvis and H. D. Ratliff. 1982. Some equivalent objectives for dynamic network flow problems. *Management Science* 28 (1982), 106–108.

[16] D. S. Johnson, C. H. Papadimitriou, and M. Yannakakis. 1988. How easy is local search? *Journal of Computer andSystem Science* 37 (1988), 79–100.

[17] N. Karmarkar. 1984. A new polynomial-time algorithm for linear programming. *Combinatorica* 4 (1984), 373–395.

[18] L. G. Khachiyan. 1979. A polynomial algorithm in linear programming. *Soviet Mathematics Doklady* 20 (1979), 191–194.

[19] L. G. Khachiyan. 1980. Polynomial algorithms in linear programming. *U.S.S.R. Computational Mathematics and Mathematical Physics* 20 (1980), 53–72.

[20] V. Klee and G. J. Minty. 1972. How good is the simplex algorithm? In *Inequalities III*, O. Shisha (Ed.). Academic Press, 159–175.

[21] C. E. Lemke and J. T. Howson. 1964. Equilibrium points of bimatrix games. *Journal of the Society for Industrial and Applied Mathematics* 12, 2 (1964), 413—423.

[22] E. Minieka. 1973. Maximal, lexicographic, and dynamic network flows. *Operations Research* 21 (1973), 517–527.

[23] J. B. Orlin. 1997. A polynomial time primal network simplex algorithm for minimum cost flows. *Mathematical Programming* 78 (1997), 109–129.

[24] C. H. Papadimitriou. 1994. On the complexity of the parity argument and other inefficient proofs of existence. *Journal of Computer and System Science* 48 (1994), 498–532.

[25] C. H. Papadimitriou, A. A. Schäffer, and M. Yannakakis. 1990. On the complexity of local search. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing (STOC)*. ACM, 438–445.

[26] T. Roughgarden and J. R. Wang. 2016. The complexity of the k-means method. In *Proceedings of the 24th Annual European Symposium on Algorithms (ESA'16)*. 78(14). Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[27] M. Skutella. 2009. An introduction to network flows over time. In *Research Trends in Combinatorial Optimization*, W. Cook, L. Lovász, and J. Vygen (Eds.). Springer, 451–482.

[28] D. A. Spielman and S.-H. Teng. 2004. Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. *Journal of the ACM* 51 (2004), 385–463.

[29] É. Tardos. 1985. A strongly polynomial minimum cost circulation algorithm. *Combinatorica* 5 (1985), 247–255.

[30] W. L. Wilkinson. 1971. An algorithm for universal maximal dynamic flows in a network. *Operations Research* 19 (1971), 1602–1612.

[31] N. Zadeh. 1973. A bad network problem for the simplex method and other minimum cost flow algorithms. *Mathematical Programming* 5 (1973), 255–266.

Received September 2017; revised May 2018; accepted August 2018