Computing Nash Equilibria for Scheduling on Restricted Parallel Links*

Martin Gairing† *Thomas Lücking*† *Marios Mavronicolas*‡ *Burkhard Monien*†

(May 11, 2007)

*A preliminary version of this work appeared in the *Proceedings of the 36th Annual ACM Symposium on Theory of Computing,* pp. 613–622, June 2004. This work has been partially supported by the IST Program of the European Union under contract numbers IST-1999-14186 (ALCOM-FT), IST-2001-33116 (FLAGS), 001907 (DELIS) and 015964 (AEOLUS). Parts of the work were done while the last author was visiting the Department of Computer Science, University of Cyprus.

†Faculty of Computer Science, Electrical Engineering and Mathematics, University of Paderborn, Fürstenallee 11, 33102 Paderborn, Germany. Email: {gairing,luck,bm}@uni-paderborn.de

‡Department of Computer Science, University of Cyprus, P. O. Box 20537, Nicosia CY-1678, Cyprus. Currently visiting the Faculty of Computer Science, Electrical Engineering and Mathematics, University of Paderborn. Email: mavronic@ucy.ac.cy

Abstract

We consider the problem of routing n *users* on m parallel *links* under the restriction that each user may only be routed on a link from a certain set of *allowed links* for the user. So, this problem is equivalent to the correspondingly restricted scheduling problem of assigning n *jobs* to m parallel *machines*. In a *Nash equilibrium*, no user may improve its own *Individual Cost* (*latency*) by unilaterally switching to another link from its set of allowed links.

For *identical links*, we present, as our main result, a polynomial time algorithm to compute from any given assignment a Nash equilibrium with non-increased *makespan*. The algorithm gradually transforms the assignment by pushing the unsplittable user *traffics* through a *flow network*, which is constructed from the users and the links. The algorithm uses ideas from *blocking flows*.

Furthermore, we use techniques simular to those in the generic PREFLOWPUSH algorithm to approximate in polynomial time a schedule with optimum makespan. This results to an improved approximation factor of $2 - \frac{1}{w_1}$ for *identical links,* where w_1 is the largest user traffic, and to an approximation factor of 2 for *related links*.

1 Introduction

1.1 Motivation and Framework

The concept of *Nash equilibrium* [22, 23] has become an important mathematical tool for analyzing the behavior of selfish users in non-cooperative systems. The celebrated result of Nash [22, 23] guarantees the existence of a Nash equilibrium in *mixed* strategies for every (finite) *strategic game* and many algorithms have been devised to compute one (see [20] for an overview). However, the complexity of computing a Nash equilibrium for a strategic game has only recently started to attract systematic study, after the problem was advocated as one of the most important open problems in Theoretical Computer Science [24].

Some recent breakthrough work has established the intractability of computing a Nash equilibrium for a strategic game with at least four players [4]. Cheng and Deng [3] soon thereafter showed that the case of two players is already intractable. The limit of the class of strategic games whose Nash equilibria can be computed efficiently is still far from being well understood. In this work, we extend the current limit by employing some new algorithmic techniques. These techniques come from identifying some natural connections between the problems of computing a Nash equilibrium and computing a maximum *network flow* (cf., [1]).

Our starting point is the well known KP model for *selfish routing*, introduced by Koutsoupias and Papadimitriou [18]. Here, n non-cooperative *users* wish to route their unsplittable *traffics* $w_1, \ldots, w_n$ through a simple network from *source* to *sink*, joined by m parallel *links* with *capacities* $c_1, \ldots, c_m$. In the case of *identical links*, all links have equal capacity; link capacities may vary arbitrarily in the case of *related links*. Each user chooses a link as its *strategy* and wishes to minimize its *latency*. An assignment where no user has an incentive to unilaterally change its strategy is called a *(pure) Nash equilibrium* [22, 23]. There is also a global objective function called *Social Cost,* which is the *makespan* (the maximum latency); however, users do not adhere to it. The KP model reflects an extension of the *job scheduling* problem on *related machines* [15] to cope with selfishness; it is also a special case of *weighted congestion games* [21], equipped with Social Cost.

In contrast to general strategic games, there is always a (*pure*) Nash equilibrium for the KP model [8, Theorem 1], and there are known polynomial time algorithms to compute one [7, 8]. This is no more known to be the case for the most general case of *unrelated links,* where there is, instead of traffics and capacities, an arbitrary *cost* (or *processing time*) for each user on each link. This extension of the KP model reflects the job scheduling problem on *unrelated machines* [15]. Computing Nash equilibria for this extension appears to be intractable in the current state-of-the-art.

In this work, we consider a middle ground between the KP model with related links and its most general extension with unrelated links, which we call the model of *restricted parallel links*. While the new model still adopts related links, the critical additional assumption is that each user i is only allowed to ship its traffic through a subset S_i of the links; these are called the *allowed links* for user i. The

cost of a user on a link is either its incurred latency if the link is allowed (like in the KP model), or infinite otherwise. Clearly, introducing allowed links into the KP model can change the set of its Nash equilibria; so, alternative algorithms are needed for computing Nash equilibria in the new model. Our work provides the *first* such algorithms.

The model of restricted parallel links reflects a special case of the job scheduling problem on unrelated machines, which we call *restricted related machines*. In some parts of our study, we shall restrict to the case of *restricted identical parallel links*, which naturally corresponds to *restricted identical machines*.

1.2 Contribution and Significance

Our approach for computing Nash equilibria for restricted parallel links is modular. We first compute an assignment which approximates well the optimum Social Cost. For the case of identical links, we then transform this assignment into a Nash equilibrium with non-increased Social Cost. As a bonus, the obtained Nash equilibrium approximates well the Social Cost.

1.2.1 Approximation of Optimum Social Cost

Computing an assignment that approximates the optimum Social Cost on restricted parallel links can be cast as a special case of the *single-source unsplittable flow* problem introduced in the seminal work of Kleinberg [16]. In this special case, the flow network is a 2-*layered bipartite graph*. All previous approaches [6, 16, 17] to the unsplittable flow problem relied on first computing a splittable flow and then using rounding techniques to compute an unsplittable flow from the splittable flow. Such techniques have yielded an approximation factor of 2.

In this work, we pursue a simpler approach to compute an unsplittable flow directly for the special case of the flow network we consider. The key idea is to use techniques from the PREFLOW-PUSH algorithm [13, 14] in the setting of unsplittable flow. However, such techniques need to be adapted so that they still allow for pushing traffics but not for splitting them.

Our approach yields approximation algorithms for optimum Social Cost for both cases of identical links and related links, which are purely combinatorial and much simpler than all previous ones. More specifically, we obtain:

- For the case of identical links, our approximation algorithm has approximation factor $2 - \frac{1}{w_1}$, where w_1 is the largest traffic. The achieved factor is the *first* to break the barrier of 2 [19].
- For the case of related links, our approximation algorithm has approximation factor 2.

For identical links, the running time of the approximation algorithm is $\mathcal{O}(mS \log W)$, where S is the total number of allowed links and W is the total traffic of the users; for related links, the running time slightly increases to $\mathcal{O}(mS \log(mW))$.

1.2.2 Nashification

For the case of identical links, we present, as our main result, an algorithm NASHIFY for the transformation of an arbitrary assignment into a Nash equilibrium. The algorithm is an instance of a more general technique known as *Nashification* [7, 11], which achieves the transformation without increasing Social Cost.

In contrast to previous Nashification algorithms for the KP model [7, 11], we rely here again on using techniques from network flows. More specifically, we first design an algorithm UNSPLITTABLE-BLOCKING-FLOW to compute a *blocking flow* [5] in the unsplittable setting. Intuitively, an (unsplittable) flow is *blocking* if each directed path from source to sink contains a *saturated* edge. This algorithm extends ideas from the splittable setting to the unsplittable setting; the key is that flow is pushed but not split. The blocking flow algorithm is then extensively used within the algorithm NASHIFY.

The running time of the algorithm NASHIFY is $\mathcal{O}(nmS(\log W + m^2))$.

1.2.3 Summary

Putting together the two pieces yields the *first* polynomial time algorithm to compute a pure Nash equilibrium that approximates well the optimum Social Cost for the case of of restricted identical parallel links. The approximation factor is $2 - \frac{1}{w_1}$, while the total running time is $\mathcal{O}(nmS(\log W + m^2))$.

The modularity of our approach allows for modular improvements to the approximation factor of optimum Social Cost for a Nash equilibrium. Any improvement to the approximation factor of optimum Social Cost will do when combined with algorithm NASHIFY.

Computing a Nash equilibrium for the case of related links of the model remains an important open problem. So remains of course the problem of computing a Nash equilibrium for the case of unrelated links, but we feel that this problem may be much harder.

Our work is the *first* to apply the generic PREFLOW-PUSH algorithm [13, 14] to the setting of unsplittable flows. We believe that such application may have further potential.

1.3 Related Work and Comparison

The model of restricted parallel links has been also studied in an independent work by Awerbuch *et al.* [2]. That work focused mostly on the case of identical links, but also considered the case of unrelated links. The work of Awerbuch *et al.* [2] provided bounds on the *Price of Anarchy* [18], but examined neither the problem of approximating optimum Social Cost nor the problem of computing a Nash equilibrium, which are the subject of our work. Additional bounds on the Price of Anarchy for restricted parallel links have been shown by Gairing *et al.* [10].

Computing an assignment with optimum Social Cost for job scheduling on unrelated machines was first considered by Horowitz and Sahni [15]; they presented an exponential time, dynamic programming

algorithm. They also presented a FPTAS to approximate the optimum assignment for a *constant* number of machines. For the general case, Lenstra *et al.* [19] proved that an approximation factor less than $\frac{3}{2}$ is *not* possible unless $\mathcal{P} = \mathcal{NP}$. This holds even if all processing times are taken from $\{1, 2, \infty\}$; in contrast, an optimum assignment can be computed in polynomial time if all processing times are taken from $\{1, 2\}$. Lenstra *et al.* [19] also presented a polynomial time algorithm with approximation factor 2; this algorithm computes an optimum fractional solution and then uses rounding. Recently, Shchepin and Vakhania [25] introduced a new rounding technique yielding an improved approximation factor of $2 - \frac{1}{m}$.

The first constant-factor, polynomial time approximation algorithms for the (single-source) unsplittable flow problem were already obtained by Kleinberg [16]. Further such algorithms based on rounding techniques were presented by Kolliopoulos and Stein [17]; one achieved approximation factor 3 for the general case, while another achieved approximation factor $2 - \frac{1}{C}$ for the case where all processing times are taken from $\{p, Cp, \infty\}$, for any $C > 1$ with $\frac{1}{C} \geq p > 0$. Dinitz *et al.* [6] showed how to turn a splittable flow into an unsplittable flow in polynomial time; this yielded an approximation factor of 2.

1.4 Organization

Restricted parallel links are introduced in Section 2. In Section 3, we introduce a framework for unsplittable network flows. Section 4 presents the approximation algorithms for an assignment with optimum Social Cost. The unsplittable blocking flow algorithm is presented in Section 5; in turn, this algorithm is used within the Nashification algorithm in Section 6. We conclude, in Section 7, with a discussion and some open problems.

2 Restricted Parallel Links

Throughout, denote for each integer $k \geq 1$, $[k] = \{1, 2, \ldots, k\}$. We consider a network consisting of a set $\mathcal{L} = [m]$ of m *parallel links* from a *source* node to a *sink* node. Each *user* from a set $\mathcal{U} = [n]$ wishes to route a particular amount of *unsplittable* traffic along a (non-fixed) link from source to sink. Denote by w_i the (integer) *traffic* of user $i \in \mathcal{U}$. Assume, without loss of generality, that $w_1 \geq \ldots \geq w_n$, and denote $W = \sum_{i \in \mathcal{U}} w_i$ (the *total traffic*). The *traffic vector* $\mathbf{w} = (w_1, \ldots, w_n)$ is the tuple of all user traffics.

Denote by $c_j > 0$ the (integer) *capacity* of link $j \in \mathcal{L}$. The *capacity vector* $\mathbf{c} = (c_1, \ldots, c_m)$ is the tuple of all link capacities. In the case of *identical links,* all link capacities are equal to 1. Link capacities may vary arbitrarily in the case of *related links*. An *instance* is denoted as a tuple $\langle \mathbf{w}, \mathbf{c} \rangle$. In the case of identical links, we denote an instance by $\langle \mathbf{w}, m \rangle$.

Associated with each user $i \in \mathcal{U}$ is a (non-empty) *strategy set* $S_i \subseteq \mathcal{L}$, which is the set of *allowed links* for user i. Denote $S = \sum_{i \in \mathcal{U}} |S_i|$ the total size of strategy sets. A *strategy* for a user $i \in \mathcal{U}$ is some

specific link $s_i \in S_i$. An *assignment* $\mathbf{s} = (s_1, \ldots, s_n)$ is a tuple of strategies, one for each user. For a set $\mathcal{B} \subseteq \mathcal{L}$ of links and an assignment $\mathbf{s}$, denote as $\mathbf{s}(\mathcal{B})$ the restriction of $\mathbf{s}$ to links in $\mathcal{B}$; so, $\mathbf{s} = \mathbf{s}(\mathcal{L})$. By abuse of notation, a *partial assignment* $\mathbf{s} = (s_1, \ldots, s_n)$ is a tuple of strategies where s_i might be nil (i.e., left unspecified) for some users $i \in \mathcal{U}$. In an *empty assignment* $\mathbf{s}$, $s_i = \mathsf{nil}$ for all users $i \in \mathcal{U}$.

The *latency* for traffic w through link j is $\frac{w}{c_j}$. For the assignment $\mathbf{s}$, the *load* $\delta_j(\mathbf{s})$ on link j is the sum of traffics of all users assigned to link j; thus, $\delta_j(\mathbf{s}) = \sum_{k \in \mathcal{U} | s_k = j} w_k$. The *Individual Cost* $\mathsf{IC}_i(\mathbf{s})$ of user $i \in \mathcal{U}$ in the assignment $\mathbf{s}$ is the latency of the link it chooses; that is,

$$\mathsf{IC}_i(\mathbf{s}) = \frac{\delta_{s_i}(\mathbf{s})}{c_{s_i}}.$$

Associated with an instance $\langle \mathbf{w}, \mathbf{c} \rangle$ and an assignment $\mathbf{s}$ is the *Social Cost* [18, Section 2], denoted $\mathsf{SC}(\mathbf{w}, \mathbf{c}, \mathbf{s})$, which is the maximum, over all links, latency for the load on the link; so,

$$\mathsf{SC}(\mathbf{w}, \mathbf{c}, \mathbf{s}) \quad = \quad \max_{j \in \mathcal{L}} \frac{\delta_j(\mathbf{s})}{c_j},$$

or equivalently

$$\mathsf{SC}(\mathbf{w}, \mathbf{c}, \mathbf{s}) \quad = \quad \max_{i \in \mathcal{U}} \mathsf{IC}_i(\mathbf{s}).$$

The *optimum* [18, Section 2] associated with an instance $\langle \mathbf{w}, \mathbf{c} \rangle$ is the least possible, over all assignments, of the maximum, over all links, latency for the load on the link; so,

$$\mathsf{OPT}(\mathbf{w}, \mathbf{c}) \quad = \quad \min_{\mathbf{s} \in S_1 \times \ldots \times S_n} \mathsf{SC}(\mathbf{w}, \mathbf{c}, \mathbf{s}) \ .$$

A user $i \in \mathcal{U}$ is *satisfied* in assignment $\mathbf{s}$ if for all links $j \in S_i$,

$$\mathsf{IC}_i(\mathbf{s}) \quad \leq \quad \frac{\delta_j(\mathbf{s}) + w_i}{c_j};$$

so, a satisfied user has no incentive to unilaterally change its strategy. An *unsatisfied* user is one that is not satisfied. The assignment $\mathbf{s}$ is a *Nash equilibrium* [22, 23] if all users are satisfied.

3 Unsplittable Network Flows

We outline a framework for unsplittable network flows, which will be used later for both approximation and Nashification algorithms.

All algorithms are controlled by two parameters w and Δ. Intuitively, w will be used to further restrict the assignment of users to links; Δ will be determined by binary search, and it will be integer for the case of identical links and rational for the case of related links.

Given some $w > 0$, we consider a *flow network* $G_{\mathbf{s}}(w)$ representing a partial assignment $\mathbf{s}$, called the *residual network*:

Definition 3.1 (Residual Network) *Let $w > 0$ and $\mathbf{s}$ be a partial assignment. The* residual network *$G_{\mathbf{s}}(w)$ is a directed bipartite graph $G_{\mathbf{s}}(w) = (V, E_{\mathbf{s}}(w))$ with $V = \mathcal{L} \cup \mathcal{U}$ and $E_{\mathbf{s}}(w) = E^1_{\mathbf{s}}(w) \cup E^2_{\mathbf{s}}(w)$, where:*

$$\begin{aligned} E^1_{\mathbf{s}}(w) &= \{(j,i) \mid j \in \mathcal{L}, i \in \mathcal{U}, s_i = j, w_i \leq w \cdot c_j\}, \text{ and} \\ E^2_{\mathbf{s}}(w) &= \{(i,j) \mid j \in \mathcal{L}, i \in \mathcal{U}, j \in S_i \setminus \{s_i\}, w_i \leq w \cdot c_j\}. \end{aligned}$$

The partite sets of the (bipartite) residual network $G_{\mathbf{s}}(w)$ are $\mathcal{L}$ and $\mathcal{U}$, respectively. Observe that V depends on neither w nor $\mathbf{s}$; in contrast, $E_{\mathbf{s}}(w)$ depends on both the partial assignment $\mathbf{s}$ and the parameter w.

For our approximation algorithm for identical links, w will be chosen large enough so that the constraint $w_i \leq w \cdot c_j$ is always fulfilled; thus, it will not impose any restriction on the set of edges $E_{\mathbf{s}}(w)$. This will not be the case for our approximation algorithm for related links.

We use the parameters Δ and w and the (partial) assignment $\mathbf{s}$ to partition the set of links $\mathcal{L}$ into three sets $\mathcal{L}^-(\mathbf{s}, \Delta, w)$, $\mathcal{L}^0(\mathbf{s}, \Delta, w)$ and $\mathcal{L}^+(\mathbf{s}, \Delta, w)$ as follows:

Definition 3.2 (Partition of Links) *Let $\Delta, w > 0$ and $\mathbf{s}$ be a partial assignment. Partition the set of links $\mathcal{L}$ into three subsets:*

$$\begin{aligned} \mathcal{L}^-(\mathbf{s}, \Delta, w) &= \{j \in \mathcal{L} : \delta_j(\mathbf{s}) \leq \Delta \cdot c_j\} \\ \mathcal{L}^0(\mathbf{s}, \Delta, w) &= \{j \in \mathcal{L} : \Delta \cdot c_j < \delta_j(\mathbf{s}) \leq (\Delta + w) \cdot c_j\} \\ \mathcal{L}^+(\mathbf{s}, \Delta, w) &= \{j \in \mathcal{L} : \delta_j(\mathbf{s}) > (\Delta + w) \cdot c_j\} \end{aligned}$$

Intuitively, nodes in $\mathcal{L}^+(\mathbf{s}, \Delta, w)$ and $\mathcal{L}^-(\mathbf{s}, \Delta, w)$ will be interpreted as *source* and *sink* nodes, respectively. Since the parameter Δ and the assignment $\mathbf{s}$ will vary, both $\mathcal{L}^+(\mathbf{s}, \Delta, w)$ and $\mathcal{L}^-(\mathbf{s}, \Delta, w)$ will be varying as well; thus, there will be no fixed source or sink nodes. We note that Definition 3.2 applies to any set of links $\mathcal{B} \subseteq \mathcal{L}$ as well.

Our algorithms will further use a *height function* $h : V \to \mathbb{N}$, which depends on the parameters w and Δ. Given w and Δ and a partial assignment $\mathbf{s}$, the height function h has the *height property*: $h(u) = 0$ for all nodes $u \in \mathcal{L}^-(\mathbf{s}, \Delta, w)$, and $h(u) \leq h(v) + 1$ for every edge $(u, v) \in E_{\mathbf{s}}(w)$.

4 Approximation of Optimum Social Cost

In Section 4.1, we adapt the generic PREFLOW-PUSH algorithm [13, 14] to unsplittable flows and derive the algorithm UNSPLITTABLE-PREFLOW-PUSH. In Section 4.2, we will apply algorithm UNSPLITTABLE-PREFLOW-PUSH to compute an assignment that approximates optimum Social Cost.

4.1 The Algorithm UNSPLITTABLE-PREFLOW-PUSH

4.1.1 Preliminaries

Fix the parameters Δ and w and a partial assignment $\mathbf{s}$. Define the *excess flow* $e(u)$ into node $u \in V$ as follows.

$$e(u) \quad = \quad \begin{cases} 1\,, & \text{if either} \quad (1)\ u \in \mathcal{L}^+(\mathbf{s}, \Delta, w), \\ & \qquad\ \text{or} \quad (2)\ u \in U \text{ and } s_u = \mathsf{nil}, \\ 0\,, & \text{otherwise.} \end{cases}$$

A node $u \in V$ with $e(u) = 1$ will be called *overflowing*.

The algorithm UNSPLITTABLE-PREFLOW-PUSH will use the two basic operations PUSH and LIFT, depicted in Figure 1.

PUSH(u, v)

Precondition:

$e(u) = 1$, $h(u) \leq 2m$ and $h(u) = h(v) + 1$ with $(u, v) \in E_{\mathbf{s}}(w)$.

Effect:

1: $E_{\mathbf{s}}(w) \leftarrow (E_{\mathbf{s}}(w) \setminus \{(u, v)\}) \cup \{(v, u)\}$;

2: update $\mathbf{s}$, $e(u)$, $e(v)$.

LIFT(u)

Precondition:

$e(u) = 1$, $h(u) \leq 2m$ and $h(u) \leq h(v)$ for all edges $(u, v) \in E_{\mathbf{s}}(w)$.

Effect:

1: $h(u) \leftarrow \begin{cases} 1 + \min\{h(v) \mid (u, v) \in E_{\mathbf{s}}(w)\} & \text{, if there exists an edge } (u, v) \in E_{\mathbf{s}}(w) \\ 2m + 1 & \text{, else.} \end{cases}$

Figure 1: The operations PUSH and LIFT in precondition-effect style

- PUSH(u, v) can be applied on edge $(u, v) \in E_{\mathbf{s}}(w)$ if u is overflowing, $h(u) \leq 2m$ and $h(u) = h(v) + 1$. The operation PUSH(u, v) replaces the edge (u, v) with the edge (v, u). In other words, the direction of the edge (u, v) is reversed; so each PUSH operation is *saturating*. Also, the assignment $\mathbf{s}$ and the excess flows $e(u)$ and $e(v)$ are updated.

  If $u \in \mathcal{L}$, then PUSH(u, v) has the effect that user $v \in \mathcal{U}$ gets unassigned from link u; afterwards, $s_v = \mathsf{nil}$ (v is not assigned to any link). On the other hand, if $u \in \mathcal{U}$, then PUSH(u, v) assigns the unassigned user u to link v.

  Observe that PUSH(u, v) preserves the height property.

- LIFT(u) can be applied on node $u \in V$ if u is overflowing, $h(u) \leq 2m$ and $h(u) \leq h(v)$ for all edges $(u, v) \in E_{\mathbf{s}}(w)$. We consider two different cases:
  - If there exists an edge $(u, v) \in E_{\mathbf{s}}(w)$, then LIFT($u$) increases the height of u to $1 + \min\{h(v) \mid (u, v) \in E_{\mathbf{s}}(w)\}$. Observe that this is the maximum value for $h(u)$ that preserves the height property.
  - If there does not exists an edge $(u, v) \in E_{\mathbf{s}}(w)$, then u will always stay overflowing. In this case, LIFT(u) increases the height of u to $2m + 1$. This assures that u will not participate in any further PUSH or LIFT operation.

  In either case LIFT(u) preserves the height property.

4.1.2 The Algorithm

UNSPLITTABLE-PREFLOW-PUSH(Δ, w)

Input: positive rational numbers Δ, w

Output: partial assignment $\mathbf{s}$

1: set $\mathbf{s}$ to the empty assignment;
2: construct $G_{\mathbf{s}}(w)$;
3: **for** each node $u \in V$ **do**
4: $\quad h(u) \leftarrow 0$;
5: $\quad$ compute $e(u)$; ▷ $e(u) = 1$ if $u \in \mathcal{U}$ and $e(u) = 0$ if $u \in \mathcal{L}$
6: **end for**
7: **while** some PUSH or LIFT operation is applicable **do**
8: $\quad$ perform any applicable PUSH or LIFT operation; ▷ PUSH modifies $\mathbf{s}$
9: **end while**
10: **return** $\mathbf{s}$;

Figure 2: The algorithm UNSPLITTABLE-PREFLOW-PUSH

The algorithm UNSPLITTABLE-PREFLOW-PUSH first initializes $\mathbf{s}$ to be the empty assignment and constructs the residual network $G_{\mathbf{s}}(w)$. Then, all values of the height function are set to 0 and the excess flows $e(u)$ for each node $u \in V$ are computed. Since $\mathbf{s}$ is the empty assignment, it is sufficient to set $e(u) = 1$ for all users $u \in \mathcal{U}$ and $e(u) = 0$ for all links $u \in \mathcal{L}$. Afterwards, the algorithm keeps executing PUSH and LIFT operations as long as there are still such applicable operations. Observe, that a PUSH modifies the current partial assignment $\mathbf{s}$. When there is no further applicable PUSH or LIFT operation, the current partial assignment $\mathbf{s}$ is returned. From the preconditions for the operations PUSH and LIFT and the height property, we immediately observe:

Observation 4.1 *Let* $\mathbf{s}$ *be the partial assignment computed by* UNSPLITTABLE-PREFLOW-PUSH(Δ, w). *Then one of the following conditions holds:*

(1) $e(u) = 0$ *for all nodes* $u \in V$.

(2) $h(u) > 2m$ *for all nodes* $u \in V$ *with* $e(u) = 1$.

We prove:

Lemma 4.1 UNSPLITTABLE-PREFLOW-PUSH *executes at most* $\mathcal{O}(m(n+m))$ LIFT *and* $\mathcal{O}(mS)$ PUSH *operations.*

Proof: Consider first any node $u \in V$. We will analyze the number of executed operations LIFT(u). Initially, $h(u) = 0$. By the precondition for LIFT(u), $h(u) \leq 2m$, while LIFT(u) increases the height of u by at least 1. It follows that LIFT(u) is executed at most $2m+1$ times. Since there are $n+m$ nodes in the graph $G_{\mathbf{s}}(w)$, this implies that there are executed at most $\mathcal{O}(m(n+m))$ LIFT operations.

Consider now any edge $(u, v) \in E_{\mathbf{s}}(w)$. We will analyze the total number of executed operations PUSH(u, v) and PUSH(v, u). When PUSH(u, v) is executed, $h(u) = h(v) + 1$, (u, v) is deleted from $E_{\mathbf{s}}(w)$ and (v, u) is added. Before PUSH(v, u) can be executed, $h(v)$ must increase by at least 2. It follows that between any two PUSH(u, v) and PUSH(v, u) operations, there is either a LIFT(u) or a LIFT(v) operation. Thus, the number of PUSH operations per edge is $\mathcal{O}(m)$. Since $G_{\mathbf{s}}(w)$ has at most S edges, it follows that $\mathcal{O}(mS)$ PUSH operations are executed. ■

We now briefly present a simple implementation of UNSPLITTABLE-PREFLOW-PUSH yielding a running time of $\mathcal{O}(mS)$. We maintain a set $\mathcal{Q}$ of nodes $u \in V$ where either LIFT(u) is applicable or there exists an edge $(u, v) \in E_{\mathbf{s}}(w)$ such that PUSH(u, v) is applicable. By the height property, $\mathcal{Q} = \{u \in V \mid e(u) = 1 \text{ and } h(u) \leq 2m\}$. Updating $\mathcal{Q}$ takes time $\mathcal{O}(1)$ per PUSH or LIFT operation.

Each node $u \in V$ has also an ordered list of its incident (incoming or outgoing) edges in $G_{\mathbf{s}}(w)$; u has also a *current edge* $\{u, v\}$, which is a candidate for the next PUSH operation out of u. Initially, the current edge is the first edge in the edge list of node u.

The running time of UNSPLITTABLE-PREFLOW-PUSH is dominated by the time needed for the **while**-loop. We implement this loop as follows: As long as $\mathcal{Q}$ is not empty, we take a node u from $\mathcal{Q}$ and apply PUSH(u, v) to the current edge $\{u, v\}$ if this operation is applicable. If not, we replace the current edge by the next edge in the edge list of node u; or if $\{u, v\}$ was the last edge on this list we make the first edge on the list the current one and apply LIFT(u). LIFT(u) is applicable since for each edge $(u, v) \in E_{\mathbf{s}}(w)$, $h(u) \leq h(v)$. This is because $h(u)$ has not changed since $\{u, v\}$ was the current edge, $h(v)$ never decreases, and $h(v) > h(u)$ if the edge (u, v) was added to $E_{\mathbf{s}}(w)$ by a PUSH(v, u) operation after $\{u, v\}$ was the current edge. For a detailed presentation of this implementation, we refer to Goldberg and Tarjan [14, Section 4].

We have seen in the proof of Lemma 4.1 that each node $u \in V$ can execute at most $\mathcal{O}(m)$ LIFT(u) operations. LIFT(u) takes time $\mathcal{O}(\deg(u))$, where $\deg(u)$ is the number of edges on the edge list of u. Without checking the precondition, a PUSH operation takes time $\mathcal{O}(1)$. Thus, the total time spent in LIFT(u) and PUSH$(u, \cdot)$ operations is $\mathcal{O}(m \deg(u))$ plus $\mathcal{O}(1)$ per PUSH out of node u. Summing up over all nodes and using Lemma 4.1, we obtain:

Lemma 4.2 UNSPLITTABLE-PREFLOW-PUSH *runs in* $\mathcal{O}(mS)$ *time.*

We remark that the analysis of UNSPLITTABLE-PREFLOW-PUSH is much simpler than the one of its splittable counterpart, the generic PREFLOW-PUSH algorithm [13, 14]. This is due to the fact that each PUSH operation is now saturating.

4.2 Applications

We now show how to use the algorithm UNSPLITTABLE-PREFLOW-PUSH to approximate an assignment with optimum Social Cost. Identical links and related links are considered in Sections 4.2.1 and 4.2.2, respectively.

4.2.1 Identical Links

For a (partial) assignment $\mathbf{s}$, consider the residual network $G_{\mathbf{s}}(w_1) = (V, E_{\mathbf{s}}(w_1))$. Since w_1 is the largest user traffic, the constraint $w_i \leq w_1$ does not impose any restriction on $E_{\mathbf{s}}(w_1)$.

The parameter Δ will always be chosen as an integer. Given a (partial) assignment $\mathbf{s}$ and an integer Δ, partition the set of links $\mathcal{L}$ according to Definition 3.2. As Δ and w_1 are integer, and $c_j = 1$ for all links $j \in \mathcal{L}$, it follows that

$$\begin{aligned} \mathcal{L}^-(\mathbf{s}, \Delta, w_1) &= \{j \in \mathcal{L} : \delta_j(\mathbf{s}) \leq \Delta\} \\ \mathcal{L}^0(\mathbf{s}, \Delta, w_1) &= \{j \in \mathcal{L} : \Delta + 1 \leq \delta_j(\mathbf{s}) \leq \Delta + w_1\} \\ \mathcal{L}^+(\mathbf{s}, \Delta, w_1) &= \{j \in \mathcal{L} : \delta_j(\mathbf{s}) \geq \Delta + w_1 + 1\} \ . \end{aligned}$$

We start with an informal description of the approximation algorithm. We will run the algorithm UNSPLITTABLE-PREFLOW-PUSH with argument (Δ, w_1). The first parameter Δ will be determined by binary search, while the second parameter is fixed to w_1. The intention is to find a Δ which is a lower bound on $\mathsf{OPT}(\mathbf{w}, m)$, and use it to compute an assignment $\mathbf{s}$ with $\mathsf{SC}(\mathbf{w}, m, \mathbf{s}) \leq \Delta + w_1$. We prove:

Lemma 4.3 *If* UNSPLITTABLE-PREFLOW-PUSH(Δ, w_1) *returns an assignment where* $e(u) = 1$ *for some node* $u \in V$*, then* $\mathsf{OPT}(\mathbf{w}, m) > \Delta + 1$.

Proof: Let $\mathbf{s}$ be the (partial) assignment returned by UNSPLITTABLE-PREFLOW-PUSH(Δ, w_1). Since $G_{\mathbf{s}}(w_1)$ is bipartite, the maximum length of a path between any two nodes $u, v \in V$ is at most $2m$.

Fix now a node $u \in V$ with $e(u) = 1$. By Observation 4.1, we have $h(u) > 2m$. Assume, by way of contradiction, that there is a path from node $u \in V$ to some link $v \in \mathcal{L}^-(\mathbf{s}, \Delta, w_1)$. By the height condition, $h(v) = 0$ and for each edge (i, j) on this path, $h(i) \leq h(j) + 1$. This implies $h(u) \leq 2m$, a contradiction to $h(u) > 2m$. Thus, there is no path from node $u \in V$ to any link $v \in \mathcal{L}^-(\mathbf{s}, \Delta, w_1)$. Let $\mathcal{B} \subseteq \mathcal{L}$ be the set of links that are still reachable from node u. Note that no user assigned to a link in $\mathcal{B}$ can be assigned to a link outside $\mathcal{B}$ (since otherwise, there would be a path to this link). Note also that $\delta_j(\mathbf{s}) \geq \Delta + 1$ for all links $j \in \mathcal{B}$. Now u is either an unassigned user that can only be assigned to a link in $\mathcal{B}$, or $u \in \mathcal{B}$ and $\delta_u(\mathbf{s}) > \Delta + w_1$. In either case, this implies $\mathsf{OPT}(\mathbf{w}, m) > \Delta + 1$. This completes the proof of the lemma. ■

We now use UNSPLITTABLE-PREFLOW-PUSH to approximate an assignment with optimum Social Cost. We do this by a sequence of calls to UNSPLITTABLE-PREFLOW-PUSH(Δ, w_1) with appropriate integers Δ. By binary search on $\Delta \in [0, W], \Delta \in \mathbb{N}$, we find a pair of integers $(\Delta - 1, \Delta)$ with the following properties:

- UNSPLITTABLE-PREFLOW-PUSH(Δ, w_1) returns an assignment $\mathbf{s}$ with $e(u) = 0$ for all nodes $u \in V$, that is, all users are assigned to some link and $\delta_j(\mathbf{s}) \leq \Delta + w_1$ for all links $j \in \mathcal{L}$. So, $\mathsf{SC}(\mathbf{w}, m, \mathbf{s}) \leq \Delta + w_1$.
- UNSPLITTABLE-PREFLOW-PUSH$(\Delta - 1, w_1)$ returns a (partial) assignment $\mathbf{t}$ where $e(u) = 1$ for some node $u \in V$. It follows from Lemma 4.3 that $\mathsf{OPT}(\mathbf{w}, m) > \Delta$ and thus $\mathsf{OPT}(\mathbf{w}, m) \geq \Delta + 1$.

We have

$$\begin{aligned}
\frac{\mathsf{SC}(\mathbf{w}, m, \mathbf{s})}{\mathsf{OPT}(\mathbf{w}, m)} &\leq \frac{\Delta + w_1}{\mathsf{OPT}(\mathbf{w}, m)} && (\text{since } \mathsf{SC}(\mathbf{w}, m, \mathbf{s}) \leq \Delta + w_1) \\
&\leq \frac{\mathsf{OPT}(\mathbf{w}, m) - 1 + w_1}{\mathsf{OPT}(\mathbf{w}, m)} && (\text{since } \mathsf{OPT}(\mathbf{w}, m) \geq \Delta + 1) \\
&\leq 2 - \frac{1}{w_1} && (\text{since } w_1 \leq \mathsf{OPT}(\mathbf{w}, m)).
\end{aligned}$$

Lemma 4.2 shows that UNSPLITTABLE-PREFLOW-PUSH runs in $\mathcal{O}(mS)$ time. The binary search on $\Delta \in [0, W]$ contributes a factor of $\log W$. In conclusion, we have:

Theorem 4.4 *Consider the model of restricted parallel links, for the case of identical links. Then, there is an algorithm that computes an assignment with Social Cost within a factor of* $2 - \frac{1}{w_1}$ *from optimum in time* $\mathcal{O}(mS \log W)$.

4.2.2 Related Links

In this section we set $w = \Delta$. Given a (partial) assignment $\mathbf{s}$ and a rational number Δ, consider the residual network $G_{\mathbf{s}}(\Delta) = (V, E_{\mathbf{s}}(\Delta))$. Moreover, partition the set of links $\mathcal{L}$ according to Definition

3.2. Here:

$$\begin{aligned}\mathcal{L}^-(\mathbf{s},\Delta,\Delta) &= \{j \in \mathcal{L} : \delta_j(\mathbf{s}) \leq \Delta \cdot c_j\}\\ \mathcal{L}^0(\mathbf{s},\Delta,\Delta) &= \{j \in \mathcal{L} : \Delta \cdot c_j < \delta_j(\mathbf{s}) \leq 2 \cdot \Delta \cdot c_j\}\\ \mathcal{L}^+(\mathbf{s},\Delta,\Delta) &= \{j \in \mathcal{L} : \delta_j(\mathbf{s}) > 2 \cdot \Delta \cdot c_j\}\ .\end{aligned}$$

Recall, that $E_{\mathbf{s}}(\Delta)$ consists only of edges $\{i,j\}$ with $i \in U$ and $j \in L$, such that $w_i \leq \Delta \cdot c_j$. This restriction is necessary to ensure that a link from $\mathcal{L}^+(\mathbf{s},\Delta,\Delta)$ is not transfered to $\mathcal{L}^-(\mathbf{s},\Delta,\Delta)$ by reassigning a single user.

Similar to Lemma 4.3, we prove:

Lemma 4.5 *If* UNSPLITTABLE-PREFLOW-PUSH(Δ,Δ) *returns an assignment where* $e(u) = 1$ *for some node* $u \in V$*, then* $\mathsf{OPT}(\mathbf{w},\mathbf{c}) > \Delta$.

Proof: Let $\mathbf{s}$ be the (partial) assignment computed by UNSPLITTABLE-PREFLOW-PUSH(Δ,Δ). Fix a node $u \in V$ with $e(u) = 1$. By Observation 4.1, we have $h(u) > 2m$. With the same argument as in Lemma 4.3, it follows that there is no path from node $u \in V$ to some link $v \in \mathcal{L}^-(\mathbf{s},\Delta,\Delta)$ in $G_{\mathbf{s}}(\Delta)$.

Let $\mathcal{B} \subseteq \mathcal{L}$ be the set of links that are still reachable from node u. Since $G_{\mathbf{s}}(\Delta)$ consists only of edges $\{i,j\}$ with $i \in \mathcal{U}, j \in \mathcal{L}$ and $w_i \leq \Delta \cdot c_j$, it is now possible that some user $i \in \mathcal{U}$ which is assigned to some link in $\mathcal{B}$ has a link $p \in \mathcal{L} \setminus \mathcal{B}$ in its strategy set S_i. However, on link p user i would cause latency $\frac{w_i}{c_p} > \Delta$. On the other hand, if we do not move users from links in $\mathcal{B}$ to links outside $\mathcal{B}$, then $\mathsf{IC}_i(\mathbf{s}) > \Delta$ for all users $i \in \mathcal{U}$ with $s_i \in \mathcal{B}$. It follows that $\mathsf{OPT}(\mathbf{w},\mathbf{c}) > \Delta$. This completes the proof of the lemma. ■

Again we make a sequence of calls to UNSPLITTABLE-PREFLOW-PUSH(Δ,Δ) with appropriate rational numbers Δ. We choose Δ from the set of rational numbers $\{\frac{1}{c_1},\ldots,\frac{W}{c_1}\} \cup \ldots \cup \{\frac{1}{c_m},\ldots,\frac{W}{c_m}\}$. These are at most mW rational numbers and they include all possible values of link latencies.

By binary search, we find a pair of rational numbers (Δ',Δ), where Δ' is the largest value with $\Delta' < \Delta$ in our sample space, with the following properties:

- UNSPLITTABLE-PREFLOW-PUSH(Δ,Δ) returns an assignment $\mathbf{s}$ where $e(u) = 0$ for all nodes $u \in V$, that is, all users are assigned to some link and $\mathsf{SC}(\mathbf{w},\mathbf{c},\mathbf{s}) \leq 2\Delta$.
- UNSPLITTABLE-PREFLOW-PUSH(Δ',Δ') returns a (partial) assignment where $e(u) = 1$ for some node $u \in V$. By Lemma 4.5 this implies that $\mathsf{OPT}(\mathbf{w},\mathbf{c}) > \Delta'$ and thus $\mathsf{OPT}(\mathbf{w},\mathbf{c}) \geq \Delta$.

It follows that

$$\frac{\mathsf{SC}(\mathbf{w},\mathbf{c},\mathbf{s})}{\mathsf{OPT}(\mathbf{w},\mathbf{c})} \leq \frac{2\Delta}{\Delta} = 2\,.$$

One call to UNSPLITTABLE-PREFLOW-PUSH takes $\mathcal{O}(mS)$ time (Lemma 4.2). The binary search on $\Delta \in \{\frac{j}{c_i} : 1 \leq j \leq W, 1 \leq i \leq m\}$ can be implemented to run in $\mathcal{O}(\log(mW))$ time. So, we have:

Theorem 4.6 *Consider the model of restricted parallel links for the case of related links. Then, there exists an algorithm that computes an assignment with Social Cost within a factor of* 2 *from optimum in time* $\mathcal{O}(mS\log(mW))$.

5 The Algorithm UNSPLITTABLE-BLOCKING-FLOW

In this section, we introduce a blocking flow algorithm, called UNSPLITTABLE-BLOCKING-FLOW, which will be extensively used by our Nashification algorithm is Section 6. For the rest of the paper, we consider the case of identical links.

To control our blocking flow algorithm we use two integer parameters Δ and w. Here, w will refer to a certain traffic size, and Δ will be determined by binary search.

For an assignment $\mathbf{s}$ and a traffic size w, consider the residual network $G_{\mathbf{s}}(w) = (V, E_{\mathbf{s}}(w))$. Given a set of links $\mathcal{L}$, an assignment $\mathbf{s}$, an integer Δ and a traffic size w, we partition the set of links $\mathcal{L}$ according to Definition 3.2. Since w and Δ are integer,

$$\begin{aligned}\mathcal{L}^-(\mathbf{s},\Delta,w) &= \{j\in\mathcal{L} : \delta_j(\mathbf{s}) \le \Delta\}\\ \mathcal{L}^0(\mathbf{s},\Delta,w) &= \{j\in\mathcal{L} : \Delta+1 \le \delta_j(\mathbf{s}) \le \Delta+w\}\\ \mathcal{L}^+(\mathbf{s},\Delta,w) &= \{j\in\mathcal{L} : \delta_j(\mathbf{s}) \ge \Delta+w+1\}\ .\end{aligned}$$

Roughly speaking, algorithm UNSPLITTABLE-BLOCKING-FLOW$(\mathcal{L},\mathbf{s},\Delta,w)$ shifts users so that the latencies of links from $\mathcal{L}^-(\mathbf{s},\Delta,w)$ are never decreased, the latencies of links from $\mathcal{L}^+(\mathbf{s},\Delta,w)$ are never increased, and links from $\mathcal{L}^0(\mathbf{s},\Delta,w)$ remain in $\mathcal{L}^0(\mathbf{s},\Delta,w)$. Our algorithm is controlled by a height function $h : V \to \mathbb{N}_0$ with $h(j) = \text{dist}_{G_{\mathbf{s}}(w)}(j, \mathcal{L}^-(\mathbf{s},\Delta,w))$ for all $j \in V$. We call an edge (u,v) *admissible*, if $h(u) = h(v)+1$. In an *admissible path*, all edges are admissible. For each node $j \in V$ with $0 < h(j) < \infty$, define $\mathsf{Suc}(j)$ to be the set of successors of node j; this is the set of nodes to which j has an admissible edge, so that

$$\mathsf{Suc}(j) = \{i \in V : (j,i) \in E_{\mathbf{s}}(w) \text{ and } h(j) = h(i)+1\}\,.$$

Note that $\mathsf{Suc}(j)$ also defines the set of admissible edges leaving j. Let $\mathsf{suc}(j)$ be the first node in a list implementation of the set $\mathsf{Suc}(j)$. We proceed to define:

Definition 5.1 *A link* $j \in \mathcal{L}$ *with* $0 < h(j) < \infty$ *is* helpful *if* $\delta_j(\mathbf{s}) \ge \Delta + 1 + w_{\mathsf{suc}(j)}$.

Observe, that a link $j \in \mathcal{L}^+(\mathbf{s},\Delta,w)$ with $0 < h(j) < \infty$ is always helpful. However, there might also be helpful links in $\mathcal{L}^0(\mathbf{s},\Delta,w)$.

Definition 5.2 *A* helpful path *is a sequence* $v_0,\ldots,v_r$ *(with* $r \ge 2$ *and even), where* $v_{2i} \in \mathcal{L}$ *for all* $0 \le i \le r/2$ *and* $v_{2i+1} \in \mathcal{U}$ *for all* $0 \le i < r/2$ *such that*

(1) v_0 *is a helpful link of minimum height,*

(2) $(v_i, v_{i+1}) \in E_{\mathbf{s}}(w)$ *and* $h(v_i) = h(v_{i+1}) + 1$ *for all* $0 \leq i \leq r-1$*,*

(3) $\Delta + 1 \leq \delta_{v_{2i}}(\mathbf{s}) + w_{\mathsf{suc}(v_{2i-2})} - w_{\mathsf{suc}(v_{2i})} \leq \Delta + w$ *for all* $0 < i < r/2$*,*

(4) $\delta_{v_r}(\mathbf{s}) + w_{\mathsf{suc}(v_{r-2})} \leq \Delta + w$*.*

We prove:

Lemma 5.1 *Let* v_0 *be a helpful link of minimum height. Then, for some even* $r \geq 2$ *the sequence* $v_0, \ldots, v_r$ *where* $\mathsf{suc}(v_i) = v_{i+1}$ *for all* $0 \leq i \leq r-1$ *is a helpful path.*

Proof: We will show that the sequence $v_0, \ldots, v_r$ satisfies the conditions from Definition 5.2. Condition (1) is immediate. Condition (2) follows from the definition of suc. It remains to show conditions (3) and (4).

Note that a link $j \in \mathcal{L}^+(\mathbf{s}, \Delta, w)$ is helpful if $h(j) < \infty$. Thus, since v_0 is a helpful link of minimum height, then all links $v_2, v_4, \ldots, v_{r-2}$ belong to $\mathcal{L}^0(\mathbf{s}, \Delta, w)$, and link v_r belongs to $\mathcal{L}^0(\mathbf{s}, \Delta, w) \cup \mathcal{L}^-(\mathbf{s}, \Delta, w)$. Therefore,

$$\delta_{v_{2i}}(\mathbf{s}) \leq \Delta + w,$$

for all $0 < i \leq r/2$. Furthermore, none of these links is helpful, which implies that

$$\delta_{v_{2i}}(\mathbf{s}) < \Delta + 1 + w_{\mathsf{suc}(v_{2i})},$$

for all $0 < i \leq r/2$. There are two cases to consider now. If $\delta_{v_{2i}}(\mathbf{s}) + w_{\mathsf{suc}(v_{2i-2})} \leq \Delta + w$, then $r = 2i$ and condition (4) holds. On the other hand, since $w \geq w_{\mathsf{suc}(v_{2i})}$, the fact that $\delta_{v_{2i}}(\mathbf{s}) + w_{\mathsf{suc}(v_{2i-2})} \geq \Delta + w + 1$ implies

$$\delta_{v_{2i}}(\mathbf{s}) + w_{\mathsf{suc}(v_{2i-2})} - w_{\mathsf{suc}(v_{2i})} \quad \geq \quad \Delta + 1,$$

proving the lower bound in (3). To prove the upper bound, recall that $\delta_{v_{2i}}(\mathbf{s}) < \Delta + 1 + w_{\mathsf{suc}(v_{2i})}$ for all $0 < i \leq r/2$ (since v_{2i} is not helpful). Since $w_{\mathsf{suc}(v_{2i-2})} \leq w$, this implies

$$\delta_{v_{2i}}(\mathbf{s}) + w_{\mathsf{suc}(v_{2i-2})} - w_{\mathsf{suc}(v_{2i})} \quad \leq \quad \Delta + w,$$

proving the upper bound in (3). This completes the proof. ∎

We are now ready to present the algorithm UNSPLITTABLE-BLOCKING-FLOW. The algorithm is depicted in Figure 3. Initially, the height function h is computed as the distance in $G_{\mathbf{s}}(w)$ of each node to the set of links $\mathcal{L}^-(\mathbf{s}, \Delta, w)$. Then, the algorithm proceeds in phases. In each phase first the minimum height $d = \min\{h(v) \mid v \in \mathcal{L}^+(\mathbf{s}, \Delta, w)\}$ over all (helpful) links from $\mathcal{L}^+(\mathbf{s}, \Delta, w)$ is computed. Inside each phase, we do not update the height function, but we successively choose a helpful link v of minimum height and we push users along the helpful path induced by v and adjust

UNSPLITTABLE-BLOCKING-FLOW($\mathcal{L}, \mathbf{s}, \Delta, w$)

Input: set of links $\mathcal{L}$
assignment $\mathbf{s}$
positive integers Δ, w

Output: assignment $\mathbf{t}$

compute h;
while $\mathcal{L}^-(\mathbf{s}, \Delta, w) \neq \emptyset$ **and** $\exists v \in \mathcal{L}^+(\mathbf{s}, \Delta, w) : h(v) < \infty$ **do**
  $d \leftarrow \min\{h(v) \mid v \in \mathcal{L}^+(\mathbf{s}, \Delta, w)\}$;
  while $\exists$ admissible path from $v \in \mathcal{L}^+(\mathbf{s}, \Delta, w)$ with $h(v) = d$ to $\mathcal{L}^-(\mathbf{s}, \Delta, w)$ **do**
    choose helpful link v of minimum height;
    push users along helpful path defined by v;
    update $\mathbf{s}, G_{\mathbf{s}}(w)$;
  end while
  recompute h;
end while
return $\mathbf{s}$;

Figure 3: UNSPLITTABLE-BLOCKING-FLOW

the assignment accordingly. In order to update $G_{\mathbf{s}}(w)$ we have to change the direction of two arcs for each user push. The phase ends when there is no further admissible path from a link $v \in \mathcal{L}^+(\mathbf{s}, \Delta, w)$ with $h(v) = d$ to some link in $\mathcal{L}^-(\mathbf{s}, \Delta, w)$. Before the new phase starts, we recompute h, and we check whether we need to start a new phase or not. UNSPLITTABLE-BLOCKING-FLOW stops when either $\mathcal{L}^-(\mathbf{s}, \Delta, w) = \emptyset$ or $h(v) = \infty$ for all $v \in \mathcal{L}^+(\mathbf{s}, \Delta, w)$. Denote $\mathbf{t}$ the assignment computed by UNSPLITTABLE-BLOCKING-FLOW($\mathcal{L}, \mathbf{s}, \Delta, w$).

We prove:

Lemma 5.2 *For* $\mathbf{t}$*, the following conditions hold:*

(1) $j \in \mathcal{L}^-(\mathbf{s}, \Delta, w) \Rightarrow \delta_j(\mathbf{t}) \geq \delta_j(\mathbf{s})$*,*
(2) $j \in \mathcal{L}^0(\mathbf{s}, \Delta, w) \Rightarrow \Delta + 1 \leq \delta_j(\mathbf{t}) \leq \Delta + w$*,*
(3) $j \in \mathcal{L}^+(\mathbf{s}, \Delta, w) \Rightarrow \delta_j(\mathbf{t}) \leq \delta_j(\mathbf{s})$*.*

Proof: UNSPLITTABLE-BLOCKING-FLOW only pushes users along a helpful path that is defined by a helpful link v of minimum height $h(v)$. The properties of a helpful path (Definition 5.2) ensure that we never add a link to $\mathcal{L}^-(\mathbf{s}, \Delta, w)$ or $\mathcal{L}^+(\mathbf{s}, \Delta, w)$, which implies (2). Since a link $j \in \mathcal{L}^+(\mathbf{s}, \Delta, w)$ with $0 < h(j) < \infty$ is always helpful, such a link j can only be the first link in a helpful path. It follows that j does not receive load, which implies (3). By the definition of a helpful path, a link $j \in \mathcal{L}^-(\mathbf{s}, \Delta, w)$ can only be the last link on a helpful path. But this link can only receive load, which implies (1). ■

The following corollary is an immediate consequence of Lemma 5.2 and the fact that UNSPLITTABLE-BLOCKING-FLOW$(\mathcal{L}, \mathbf{s}, \Delta, w)$ does not change the assignment $\mathbf{s}$ if either $\mathcal{L}^+(\mathbf{s}, \Delta, w) = \emptyset$ or $\mathcal{L}^-(\mathbf{s}, \Delta, w) = \emptyset$.

Corollary 5.3 *For* $\mathbf{t}$*,* $\max_{j \in \mathcal{L}} \delta_j(\mathbf{t}) \leq \max_{j \in \mathcal{L}} \delta_j(\mathbf{s})$ *and* $\min_{j \in \mathcal{L}} \delta_j(\mathbf{t}) \geq \min_{j \in \mathcal{L}} \delta_j(\mathbf{s})$*.*

We continue to prove:

Lemma 5.4 *For* $\mathbf{t}$*, one of the following conditions holds:*

(1) $\mathcal{L}^-(\mathbf{t}, \Delta, w) = \emptyset$*.*
(2) $\mathcal{L}^+(\mathbf{t}, \Delta, w) = \emptyset$*.*
(3) there exists some set of links $\mathcal{B} \subset \mathcal{L}$ *such that:*

(a) $\delta_j(\mathbf{t}) \geq \Delta + 1$ *for all* $j \in \mathcal{B}$*,*
(b) $\delta_j(\mathbf{t}) \leq \Delta + w$ *for all* $j \in \mathcal{L} \setminus \mathcal{B}$*,*
(c) $s_i \in \mathcal{B} \Rightarrow S_i \subseteq \mathcal{B}$ *for all* $i \in \mathcal{U}$ *with* $w_i \leq w$*.*

Proof: If either (1) or (2) holds, then the algorithm terminates. So assume that $\mathcal{L}^-(\mathbf{t}, \Delta, w) \neq \emptyset$ and $\mathcal{L}^+(\mathbf{t}, \Delta, w) \neq \emptyset$, and that the algorithm terminates. It follows that $h(v) = \infty$ for all $v \in \mathcal{L}^+(\mathbf{t}, \Delta, w)$ which implies that in $G_{\mathbf{t}}(w)$ there is no path from a link in $\mathcal{L}^+(\mathbf{t}, \Delta, w)$ to a link in $\mathcal{L}^-(\mathbf{t}, \Delta, w)$. Define $\mathcal{B}$ to be the set of links that are reachable from $\mathcal{L}^+(\mathbf{t}, \Delta, w)$. Since $\mathcal{L}^-(\mathbf{t}, \Delta, w)$ is not reachable from a link in $\mathcal{L}^+(\mathbf{t}, \Delta, w)$, (3a) holds. All links in $\mathcal{L}^+(\mathbf{t}, \Delta, w)$ are in $\mathcal{B}$, therefore (3b) holds. By the definition of $\mathcal{B}$, if a user i is assigned to a link $s_i \in \mathcal{B}$, then it can not be assigned to a link in $\mathcal{L} \setminus \mathcal{B}$ which implies (3c). The proof is now complete. ■

We are now ready to prove:

Theorem 5.5 UNSPLITTABLE-BLOCKING-FLOW *runs in* $\mathcal{O}(mS)$ *time.*

Proof: We consider a *phase* of algorithm UNSPLITTABLE-BLOCKING-FLOW to be a single pass through the outer **while**-loop. In a phase we first compute the minimum height $d \leftarrow \min\{h(v) \mid v \in \mathcal{L}^+(\mathbf{s}, \Delta, w)\}$ of a link in $\mathcal{L}^+(\mathbf{s}, \Delta, w)$. Let G_d denote the subgraph of $G_{\mathbf{s}}(w)$ defined by the admissible edges connecting nodes of height at most d. As defined earlier, let $\mathsf{Suc}(v)$ denote the set of successors of v in G_d for any node v in G_d. Our algorithm will manipulate the graph G_d by deleting edges, i.e. by deleting elements from $\mathsf{Suc}(v)$ for some nodes $v \in V$.

In the algorithm, we successively choose a helpful link v of minimum height in G_d, and we push users along the helpful path induced by v. Pushing a user along an edge results in changing the direction of this edge in $G_{\mathbf{s}}(w)$ and deleting it from G_d. This can make other nodes helpful. We see later, how the problem of finding the next helpful link of minimum height is solved. We push users along such helpful paths until no further helpful path in G_d from a link $v \in \mathcal{L}^+(\mathbf{s}, \Delta, w)$ with $h(v) = d$ exists.

If no further such path exists, then we reached a blocking flow. Thus, after updating the height function, the minimum height d of a link in $\mathcal{L}^+(\mathbf{s}, \Delta, w)$ increased (see Ahuja et al. [1]). Since $G_{\mathbf{s}}(w)$ is a bipartite graph, d has to increase by at least 2 Furthermore, the maximum height of any link v with admissible path to a link in $\mathcal{L}^-(\mathbf{s}, \Delta, w)$ is at most $2m$. This implies that the number of phases is a most m.

Note that S is an upper bound on the number of edges in $G_{\mathbf{s}}(w)$. So, the computation of the height function h and the construction of the graph G_d can both be done in $\mathcal{O}(S)$ time using *breadth-first search*. Recall that $\mathsf{suc}(v)$ is the first element in the list implementation of $\mathsf{Suc}(v)$. Every node keeps track of its position in these lists. We always choose the first entry $\mathsf{suc}(u)$ from $\mathsf{Suc}(u)$ to define a helpful path. As we shall see, this is sufficient for our algorithm and it reduces the running time.

During the run of the algorithm, the edge $(u, \mathsf{suc}(u))$ may be deleted. In this case, $\mathsf{suc}(u)$ is also deleted from $\mathsf{Suc}(u)$ and from the list implementation of $\mathsf{Suc}(u)$. If $\mathsf{Suc}(u)$ is not empty after this deletion, then the first element in its list implementation is defined and $\mathsf{suc}(u)$ is newly defined, without mentioning this explicitly in the algorithm, to be this element.

Inside a phase we always push users along a path that is induced by a helpful link v of minimum height $h(v)$. There are two ways in which a link v' with $h(v') \leq h(v)$ which was not helpful before can become helpful. Either the load $\delta_{v'}$ or the first successor $\mathsf{suc}(v') \in \mathsf{Suc}(v')$ changed. Consider a link v' on the path induced by v. Link v' can become helpful since both the load $\delta_{v'}$ and $\mathsf{suc}(v')$ have changed. Furthermore, it is possible that $\mathsf{Suc}(v')$ became empty. In this case v' is no longer on an admissible path to a link in $\mathcal{L}^-(\mathbf{s}, \Delta, w)$ and therefore v' and all edges (u, v') entering v' are deleted from G_d. So, v' is extracted from $\mathsf{Suc}(u)$. This again can make u helpful, since $\mathsf{suc}(u)$ may have changed. It is also possible that $\mathsf{Suc}(u)$ became empty in which case we proceed recursively.

We now show how to find the next helpful link of minimum height efficiently using a *leveled stack* K. Intuitively, K holds all nodes that could be helpful or could make other nodes helpful. The nodes are always ordered by their height such that on top of K there is a node with minimal height. A *level* on K is a maximal set of successive entries of constant height. We have two basic operations for our leveled stack:

- The operation READ extracts the node on top of K. READ takes $\mathcal{O}(1)$ time.
- The operation WRITE inserts a node to the level in K according to the height of the node, provided that it is not already on the stack. By WRITE we will add a node either to the same level as the top node of K or to the level below. By storing a pointer with each node on K that refers to the first node in the level below, all executed WRITE operations can be executed in $\mathcal{O}(1)$ time.

At the beginning of each phase every link is potentially helpful. Thus, we initialize K with all nodes from $u \in G_d$. We then do the following until K becomes empty:

- We take node u to be the one returned by executing READ on K.
- If $\mathsf{Suc}(u)$ is not empty and if u is helpful then we PUSH$(u, \mathsf{suc}(u))$, we delete the edge $(u, \mathsf{suc}(u))$ from G_d, we WRITE(u) to K (since u may still be helpful) and we WRITE$(\mathsf{suc}(u))$ to K (since $\mathsf{suc}(u)$ may now be helpful).
- If $\mathsf{Suc}(u)$ is empty and $u \notin \mathcal{L}^-(\mathbf{s}, \Delta, w)$, then u is no longer on an admissible path to a link in $\mathcal{L}^-(\mathbf{s}, \Delta, w)$. In this case for all $(v, u) \in G_d$ we delete (v, u) from G_d (since (v, u) is no longer admissible) and we execute the operation WRITE(v) to K.

When K becomes empty, the phase ends.

In going down the stack, all edges are deleted which are incident to the node examined and which are no longer on an admissible path to a link in $\mathcal{L}^-(\mathbf{s}, \Delta, w)$. So, if a link u is recognized to be helpful, then it defines a helpful path as described in Lemma 5.1. This also implies that if a node $v \in \mathcal{L}^+(\mathbf{s}, \Delta, w)$ is not helpful when read from the stack, then $\mathsf{Suc}(v) = \emptyset$ and there is no longer an admissible path to a link from $\mathcal{L}^-(\mathbf{s}, \Delta, w)$.

We now analyze the time needed for one phase of the algorithm. The processing of a node u takes constant time if $\mathsf{Suc}(u) \neq \emptyset$. If $\mathsf{Suc}(u) = \emptyset$, then all nodes v with $(v, u) \in G_d$ have to be inserted into the leveled stack and the edges (v, u) have to be deleted from G_d. Note that together with each WRITE operation, an edge is deleted from G_d. Since the number of edges in G_d is at most S, the total number of WRITE operations is $\mathcal{O}(S)$.

Initially, there are $n + m$ nodes on K. There are two cases where a node v, no longer on K, may become helpful:

- The node v is on a helpful path because in this case $\mathsf{suc}(v)$ is recomputed. But note that in this case, the edge on the helpful path leading to v is deleted from G_d. So, the number of insertions of this type is $\mathcal{O}(S)$.

- The node v is connected to a node u with $\mathsf{Suc}(u) = \emptyset$. We have already argued that the number of insertions of this type is $\mathcal{O}(S)$.

Since each stack operation can be executed in constant time, it follows that one phase of UNSPLITTABLE-BLOCKING-FLOW takes time $\mathcal{O}(S)$. The claim follows, since there are at most m phases. ■

6 Nashification

In this section, we present a Nashification algorithm, called NASHIFY, that computes a pure Nash equilibrium from any given assignment in the case of identical links.

The algorithm first finds an assignment satisfying all users with traffic w_1 by recursively applying UNSPLITTABLE-BLOCKING-FLOW from Section 5. This algorithm, called RECURSIVEUBF, is given in Section 6.1. We then *fix* the assignment of all users with traffic w_1 and proceed with the next smaller traffic while making sure that all fixed users stay satisfied. To make sure that all fixed users stay satisfied, we introduce lower and upper bounds on the load of the links, such that the load of each link is always in its bounds, the lower bound only increases and the upper bound only decreases. This is done until all users are satisfied. In order to achieve this, NASHIFY makes extensive use of algorithm UNSPLITTABLE-BLOCKING-FLOW. A detailed description of algorithm NASHIFY is in Section 6.2. In the following, we denote $w = w_i$ for some user $i \in \mathcal{U}$.

6.1 RECURSIVEUBF

We first turn our attention to RECURSIVEUBF($\mathcal{B}, \mathbf{s}(\mathcal{B}), [l, u], w$), which is depicted in Figure 4. If $l \leq \delta_j(\mathbf{s}) \leq u + w$ for all links $j \in \mathcal{B}$ prior to a call to RECURSIVEUBF($\mathcal{B}, \mathbf{s}(\mathcal{B}), [l, u], w$), then it computes an assignment, where no user with traffic w which is assigned to some link in $\mathcal{B}$ can improve by moving to some other link in $\mathcal{B}$. By a series of calls to UNSPLITTABLE-BLOCKING-FLOW($\mathcal{B}, \mathbf{s}(\mathcal{B}), \Delta, w$) we compute an assignment $\mathbf{t}(\mathcal{B})$ where $\mathcal{B}^-(\mathbf{t}(\mathcal{B}), \Delta, w)$ and $\mathcal{B}^+(\mathbf{t}(\mathcal{B}), \Delta, w)$ are either both empty or both non-empty. Parameter Δ is chosen by binary search $\Delta \in [l, u], \Delta \in \mathbb{N}$, as follows:

- If UNSPLITTABLE-BLOCKING-FLOW($\mathcal{B}, \mathbf{s}(\mathcal{B}), \Delta, w$) returns an assignment $\mathbf{t}(\mathcal{B})$ where $\mathcal{B}^-(\mathbf{t}(\mathcal{B}), \Delta, w) = \emptyset$ and $\mathcal{B}^+(\mathbf{t}(\mathcal{B}), \Delta, w) \neq \emptyset$, then we increase Δ.
- Otherwise, if UNSPLITTABLE-BLOCKING-FLOW($\mathcal{B}, \mathbf{s}(\mathcal{B}), \Delta, w$) returns an assignment $\mathbf{t}(\mathcal{B})$ where $\mathcal{B}^-(\mathbf{t}(\mathcal{B}), \Delta, w) \neq \emptyset$ and $\mathcal{B}^+(\mathbf{t}(\mathcal{B}), \Delta, w) = \emptyset$, then we decrease Δ.

If after the binary search, $\mathcal{B}^-(\mathbf{t}(\mathcal{B}), \Delta, w) = \emptyset$ and $\mathcal{B}^+(\mathbf{t}(\mathcal{B}), \Delta, w) = \emptyset$, then we have computed an assignment $\mathbf{t}(\mathcal{B})$ where all users with traffic at least w are satisfied. If neither $\mathcal{B}^-(\mathbf{t}(\mathcal{B}), \Delta, w) = \emptyset$ nor $\mathcal{B}^+(\mathbf{t}(\mathcal{B}), \Delta, w) = \emptyset$ it follows that condition (3) from Lemma 5.4 holds. Define $\mathcal{B}'$ as the set of links still reachable from $\mathcal{B}^+(\mathbf{t}(\mathcal{B}), \Delta, w)$ (Lemma 5.4) and let $\overline{\mathcal{B}'}$ be the complement of $\mathcal{B}'$ in $\mathcal{B}$. In this case

RECURSIVEUBF($\mathcal{B}, \mathbf{s}(\mathcal{B}), [l, u], w$)

Input: A set of links $\mathcal{B}$, an assignment $\mathbf{s}(\mathcal{B})$, an interval $[l, u]$ and a traffic size w.

Output: An assignment $\mathbf{t}(\mathcal{B})$.

1: $\Delta \leftarrow \lfloor (l+u)/2 \rfloor$;
2: **if** $\Delta = u$ **then**
3: **return** $\mathbf{s}(\mathcal{B})$
4: **end if**
5: $\mathbf{t}(\mathcal{B}) \leftarrow$ UNSPLITTABLE-BLOCKING-FLOW$(\mathcal{B}, \mathbf{s}(\mathcal{B}), \Delta, w)$;
6: **if** $\mathcal{B}^-(\mathbf{t}(\mathcal{B}), \Delta, w) = \emptyset$ **and** $\mathcal{B}^+(\mathbf{t}(\mathcal{B}), \Delta, w) \neq \emptyset$ **then**
7: $\mathbf{t}(\mathcal{B}) \leftarrow$ RECURSIVEUBF$(\mathcal{B}, \mathbf{t}(\mathcal{B}), [\Delta + 1, u], w)$;
8: **else if** $\mathcal{B}^-(\mathbf{t}(\mathcal{B}), \Delta, w) \neq \emptyset$ **and** $\mathcal{B}^+(\mathbf{t}(\mathcal{B}), \Delta, w) = \emptyset$ **then**
9: $\mathbf{t}(\mathcal{B}) \leftarrow$ RECURSIVEUBF$(\mathcal{B}, \mathbf{t}(\mathcal{B}), [l, \Delta], w)$;
10: **else if** $\mathcal{B}^-(\mathbf{t}(\mathcal{B}), \Delta, w) \neq \emptyset$ **and** $\mathcal{B}^+(\mathbf{t}(\mathcal{B}), \Delta, w) \neq \emptyset$ **then**
11: split $\mathcal{B}$ (according to Lemma 5.4 (3)) into sets $\mathcal{B}'$ and $\overline{\mathcal{B}'}$;
12: $\mathbf{t}(\mathcal{B}') \leftarrow$ RECURSIVEUBF$(\mathcal{B}', \mathbf{t}(\mathcal{B}'), [\Delta + 1, u], w)$;
13: $\mathbf{t}(\overline{\mathcal{B}'}) \leftarrow$ RECURSIVEUBF$(\overline{\mathcal{B}'}, \mathbf{t}(\overline{\mathcal{B}'}), [l, \Delta], w)$;
14: $\mathbf{t}(\mathcal{B}) \leftarrow \mathbf{t}(\mathcal{B}') \cup \mathbf{t}(\overline{\mathcal{B}'})$;
15: **end if**
16: **return** $\mathbf{t}(\mathcal{B})$;

Figure 4: RECURSIVEUBF

we split our instance into two parts. One part with all links in $\mathcal{B}'$ and all users that are currently assigned to a link in $\mathcal{B}'$, the other part holds the complement. Whenever $\mathcal{B}$ is split into $\mathcal{B}'$ and $\overline{\mathcal{B}'}$, condition (3) from Lemma 5.4 implies that no user v with $w_v \leq w$, assigned to a link in $\mathcal{B}'$, has a link from $\overline{\mathcal{B}'}$ in its strategy set.

We recursively proceed with the binary search on Δ in both parts of the instance. For the part that corresponds to $\mathcal{B}'$, we increase Δ, while in the other part we decrease Δ. The recursive splitting of $\mathcal{B}$ (line 11) defines a partition of the links into sets $\mathcal{B}_1, \ldots, \mathcal{B}_p$. At the end, all parts $\mathcal{B}_1, \ldots, \mathcal{B}_p$ are put together to form $\mathbf{t}(\mathcal{B})$.

For each $\mathcal{B}_k, k \in [p]$, define a lower bound $\mathsf{Low}(\mathcal{B}_k)$ on the load of all links from $\mathcal{B}_k$ as the last value for Δ after the binary search on Δ in $\mathcal{B}_k$. This implies:

Lemma 6.1 *If $l \leq \delta_j(\mathbf{s}) \leq u + w$ for all $j \in \mathcal{B}$, then* RECURSIVEUBF$(\mathcal{B}, \mathbf{s}(\mathcal{B}), [l, u], w)$ *returns a reassignment $\mathbf{t}(\mathcal{B})$ of users in $\mathcal{B}$, a partition of $\mathcal{B}$ into p sets $\mathcal{B}_1, \ldots, \mathcal{B}_p$ for some p, and (implicit) numbers $\mathsf{Low}(\mathcal{B}_k)$ for $k \in [p]$, such that:*

(1) $u \geq \mathsf{Low}(\mathcal{B}_1) > \ldots > \mathsf{Low}(\mathcal{B}_p) \geq l$ *for all $k \in [p]$.*

(2) $\mathsf{Low}(\mathcal{B}_k) \leq \delta_j(\mathbf{t}) \leq \mathsf{Low}(\mathcal{B}_k) + w$ *for all* $j \in \mathcal{B}_k$ *and for all* $k \in [p]$.

(3) No user u *with* $w_u \leq w$ *assigned to a link in* $\mathcal{B}_k$ *has a link from* $\mathcal{B}_\ell$ *in its strategy set, if* $\ell > k$.

By the postconditions of Lemma 6.1, it follows that in the assignment computed by RECURSIVEUBF$(\mathcal{B}, \mathbf{s}(\mathcal{B}), [l, u], w)$, no user with traffic w which is assigned to a link in $\mathcal{B}$ can improve by moving to some other link in $\mathcal{B}$. In order to preserve this property, we have to ensure that in further computations the lower bounds only increase and the upper bounds only decrease. We denote the upper bound by $\mathsf{Up}(\mathcal{B}_k)$ for all links from $\mathcal{B}_k$, and in coincidence with (2) we set $\mathsf{Up}(\mathcal{B}_k) = \mathsf{Low}(\mathcal{B}_k) + w$.

6.2 NASHIFY

We are ready to present algorithm NASHIFY that converts any given assignment $\mathbf{s}$ into a pure Nash equilibrium $\mathbf{t}$ with non-increased Social Cost. Let $\widetilde{w}_1 > \ldots > \widetilde{w}_r$ be all different user traffics from $w_1, \ldots, w_n$. The idea is to compute a sequence of assignments $\mathbf{t}_0, \ldots, \mathbf{t}_r$ such that $\mathbf{t}_0 = \mathbf{s}$, and such that for all assignments $\mathbf{t}_i$ with $1 \leq i \leq r$, all users $j \in \mathcal{U}$ with $w_j \geq \widetilde{w}_i$ are satisfied. We call the computation of $\mathbf{t}_i$ from $\mathbf{t}_{i-1}$ *stage* i. The aim in stage i is to compute an assignment $\mathbf{t}_i$ from $\mathbf{t}_{i-1}$ such that in $\mathbf{t}_i$ all users $u \in \mathcal{U}$ with $w_u \geq \widetilde{w}_i$ are satisfied.

NASHIFY(s)

▷ stage 1:

1: $\mathbf{t}_1 \leftarrow$RECURSIVEUBF$(\mathcal{L}, \mathbf{s}(\mathcal{L}), [0, \max_j \delta_j(\mathbf{s})], \widetilde{w}_1)$

▷ stages 2,. . . , r:

2: **for** $i \leftarrow 2$ to r **do**
3: **while** there are sets of active links **do**
4: execute SWEEP over the active links;
5: **end while**

▷ $\mathbf{t}_i$ is the current assignment:

6: **end for**

Figure 5: NASHIFY

Figure 5 shows the high-level structure of our Nashification algorithm. It first uses the procedure RECURSIVEUBF to compute an assignment $\mathbf{t}_1$, where all users with traffic $\widetilde{w}_1$ are satisfied. Afterwards we iteratively satisfy users with traffic $\widetilde{w}_2, \ldots, \widetilde{w}_r$ making sure that users with larger traffic remain satisfied (lines 2-6). We do this by executing SWEEP over the sets of active links. In the following, we define what we mean by sets of active links, and we describe how a SWEEP over these sets of active links is executed.

Lemma 6.1 implies that after stage 1, all users with traffic $\widetilde{w}_1$ are satisfied. Furthermore, the links are partitioned into p_1 sets $\mathcal{B}_1, \ldots, \mathcal{B}_{p_1}$ with $\mathsf{Up}(\mathcal{B}_k) = \mathsf{Low}(\mathcal{B}_k) + \widetilde{w}_1$ for all $k \in [p_1]$, and no user

$u \in \mathcal{U}$ with $w_u \leq \widetilde{w}_1$, which is assigned to a link from $\mathcal{B}_k$ can be assigned to a link from $\mathcal{B}_\ell$ when $k < \ell$.

We now describe stage $i > 1$ (lines 2-6 in Figure 5). During stage i, the lower bound on the load of a link only increases and the upper bound only decreases. This implies that fixed users (that is, all users $j \in \mathcal{U}$ with $w_j > \widetilde{w}_i$) remain satisfied. At the beginning of stage i, we have an assignment $\mathbf{t}_{i-1}$, where the links are partitioned into p_{i-1} sets $\mathcal{B}_1, \ldots, \mathcal{B}_{p_{i-1}}$ with $\mathsf{Up}(\mathcal{B}_k) = \mathsf{Low}(\mathcal{B}_k) + \widetilde{w}_{i-1}$, for all $k \in [p_{i-1}]$, and no user u, which is assigned to a link from $\mathcal{B}_k$ can be assigned to a link from $\mathcal{B}_\ell$ when $k < \ell$.

During each stage i, we always maintain an assignment where the links are partitioned into q sets $\mathcal{C}_1, \ldots, \mathcal{C}_q$ for some q. They are ordered such that $\mathsf{Up}(\mathcal{C}_k) > \mathsf{Up}(\mathcal{C}_{k+1})$ and $\mathsf{Low}(\mathcal{C}_k) \geq \mathsf{Low}(\mathcal{C}_{k+1})$ for all k with $1 \leq k < q$.

At the beginning of a SWEEP, we have three classes of sets (see Figure 6):

- Some sets of links $\mathcal{C}_k, 1 \leq k < x$, have not been considered yet and fulfill $\mathsf{Up}(\mathcal{C}_k) - \mathsf{Low}(\mathcal{C}_k) = \widetilde{w}_{i-1}$.
- Moreover, some sets of links $\mathcal{C}_k, q \geq k > y$, have been *done in stage* i already and fulfill $\mathsf{Up}(\mathcal{C}_k) - \mathsf{Low}(\mathcal{C}_k) = \widetilde{w}_i$.
- Finally, we have sets $\mathcal{C}_x, \ldots, \mathcal{C}_y$ of *active* links, with $\widetilde{w}_i < \mathsf{Up}(\mathcal{C}_k) - \mathsf{Low}(\mathcal{C}_k) \leq \widetilde{w}_{i-1}$ and $\mathsf{Low}(\mathcal{C}_k) = \mathsf{Low}(\mathcal{C}_y)$ for all $k \in [x, y]$.

Initially, $\mathcal{C}_j = \mathcal{B}_j$ for all $1 \leq j \leq p_{i-1}$, the links from $\mathcal{C}_{p_{i-1}}$ are active, and the remaining links have not been considered. During a SWEEP, the number of partitions q may change. We prove in Lemma 6.2 that at the beginning of each SWEEP, the *sweep property* introduced below holds:

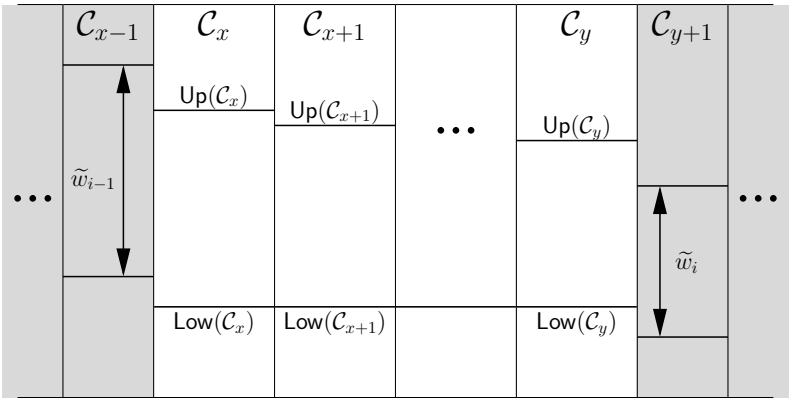

Figure 6: Sets of active links in stage i at the beginning of a sweep

Definition 6.1 (Sweep Property during stage i)

(1) There is a partition of the links into q sets $\mathcal{C}_1, \ldots, \mathcal{C}_q$ for some q with $\mathsf{Low}(\mathcal{C}_1) \geq \ldots \geq \mathsf{Low}(\mathcal{C}_q)$ and $\mathsf{Up}(\mathcal{C}_1) > \ldots > \mathsf{Up}(\mathcal{C}_q)$.

(2) If link $j \in \mathcal{C}_k$, then $\mathsf{Low}(\mathcal{C}_k) \leq \delta_j \leq \mathsf{Up}(\mathcal{C}_k)$.

(3) No user u with $w_u \leq \widetilde{w}_i$ which is assigned to a link in $\mathcal{C}_k$ has a link from $\mathcal{C}_\ell$ in its strategy set S_u, if $\ell > k$.

(4) There are integers x and y with $1 \leq x \leq y \leq q$, such that:

(a) $\mathsf{Up}(\mathcal{C}_k) - \mathsf{Low}(\mathcal{C}_k) = \widetilde{w}_{i-1}$ for $1 \leq k < x$,

(b) $\mathsf{Up}(\mathcal{C}_k) - \mathsf{Low}(\mathcal{C}_k) = \widetilde{w}_i$ for $y < k \leq q$, and

(c) $\widetilde{w}_i < \mathsf{Up}(\mathcal{C}_k) - \mathsf{Low}(\mathcal{C}_k) \leq \widetilde{w}_{i-1}$ and $\mathsf{Low}(\mathcal{C}_k) = \mathsf{Low}(\mathcal{C}_y)$ for all $x \leq k \leq y$.

We use the definition of Sweep Property to define active links.

Definition 6.2 *Let x, y be as in Definition 6.1. Then, a link j with $j \in \mathcal{C}_k$, $x \leq k \leq y$, is called* active.

A SWEEP is shown in Figure 7 and works on active links as follows: At the beginning of SWEEP, the sweep property holds. The aim of SWEEP is to process links in $\mathcal{C}_y$ such that they do not have to be considered again in this stage, or to make all links in $\mathcal{C}_{x-1}$ active by increasing the lower bound of all active links to $\mathsf{Low}(\mathcal{C}_{x-1})$. In order to preserve the structure of our assignment, we choose $\Delta = \min\{\mathsf{Up}(\mathcal{C}_y) - \widetilde{w}_i, \mathsf{Low}(\mathcal{C}_{x-1})\}$. We insert all sets into a list List such that $\mathsf{List} = [\mathcal{C}_x, \ldots, \mathcal{C}_y]$. Then, as long as there are at least two sets in List, we do the following: We extract the first element, say $\mathcal{D}_1$, of List and apply UNSPLITTABLE-BLOCKING-FLOW to the sub-instance defined by the set $\mathcal{D}_1$. UNSPLITTABLE-BLOCKING-FLOW$(\mathcal{D}_1, \mathbf{s}(\mathcal{D}_1), \Delta, \widetilde{w}_i)$ returns an assignment $\mathbf{t}(\mathcal{D}_1)$, where one of the following conditions hold:

(1) $\mathcal{D}_1^+(\mathbf{t}(\mathcal{D}_1), \Delta, \widetilde{w}_i) = \emptyset$: In this case, all links in $\mathcal{D}_1$ have load at most $\Delta + \widetilde{w}_i$, and Corollary 5.3 implies that this property is preserved. Let $\mathcal{D}_2$ be the next element in List. Before the call, $\mathsf{Up}(\mathcal{D}_1) > \mathsf{Up}(\mathcal{D}_2) > \Delta + \widetilde{w}_i$ was true. After the call, the loads of all links in $\mathcal{D}_1$ are bounded by $\Delta + \widetilde{w}_i$. So, by setting $\mathsf{Up}(\mathcal{D}_1) \leftarrow \mathsf{Up}(\mathcal{D}_2)$, we get a new upper bound on the loads of the links in $\mathcal{D}_1$, and we fulfill the requirement that upper bounds can be only decreased. $\mathcal{D}_1$ and $\mathcal{D}_2$ are merged, and the union of both sets is inserted into List. This way, the number of sets in the list is decreased by 1.

(2) $\mathcal{D}_1^-(\mathbf{t}(\mathcal{D}_1), \Delta, \widetilde{w}_i) = \emptyset$ and $\mathcal{D}_1^+(\mathbf{t}(\mathcal{D}_1), \Delta, \widetilde{w}_i) \neq \emptyset$: In this case, all links in $\mathcal{D}_1$ have load at least Δ, and Corollary 5.3 implies that this property is preserved. Thus, we are allowed to set $\mathsf{Low}(\mathcal{D}_1) \leftarrow \Delta$. We are done with $\mathcal{D}_1$ during this execution of SWEEP.

Require: $\mathsf{List} = [\mathcal{C}_x, \ldots, \mathcal{C}_y]$ is a list of the sets of *active* links

$\Delta \leftarrow \min\{\mathsf{Up}(\mathcal{C}_y) - \widetilde{w}_i, \mathsf{Low}(\mathcal{C}_{x-1})\}$;
while $|\mathsf{List}| \geq 2$ **do**
  $\mathcal{D}_1 \leftarrow \mathsf{ExtractFirst}(\mathsf{List})$;
  $\mathbf{t}(\mathcal{D}_1) \leftarrow$ UNSPLITTABLE-BLOCKING-FLOW$(\mathcal{D}_1, \mathbf{s}(\mathcal{D}_1), \Delta, \widetilde{w}_i)$;
  if $\mathcal{D}_1^+(\mathbf{t}(\mathcal{D}_1), \Delta, \widetilde{w}_i) = \emptyset$ **then**
    $\mathcal{D}_2 \leftarrow \mathsf{ExtractFirst}(\mathsf{List})$;
    $\mathsf{Up}(\mathcal{D}_1) \leftarrow \mathsf{Up}(\mathcal{D}_2)$;
    $\mathcal{D}_1 \leftarrow \mathcal{D}_1 \cup \mathcal{D}_2$; $\mathsf{Insert}(\mathcal{D}_1, \mathsf{List})$;
  else if $\mathcal{D}_1^-(\mathbf{t}(\mathcal{D}_1), \Delta, \widetilde{w}_i) = \emptyset$ **and** $\mathcal{D}_1^+(\mathbf{t}(\mathcal{D}_1), \Delta, \widetilde{w}_i) \neq \emptyset$ **then**
    $\mathsf{Low}(\mathcal{D}_1) \leftarrow \Delta$ and output: "links in $\mathcal{D}_1$ are *done in this sweep*";
  else if $\mathcal{D}_1^-(\mathbf{t}(\mathcal{D}_1), \Delta, \widetilde{w}_i) \neq \emptyset$ **and** $\mathcal{D}_1^+(\mathbf{t}(\mathcal{D}_1), \Delta, \widetilde{w}_i) \neq \emptyset$ **then**
    split $\mathcal{D}_1$ (according to Lemma 5.4 (3)) into sets $\mathcal{D}_1'$ and $\overline{\mathcal{D}_1'}$;
    $\mathsf{Low}(\mathcal{D}_1') \leftarrow \Delta$ and output: "links in $\mathcal{D}_1'$ are *done in this sweep*";
    $\mathcal{D}_2 \leftarrow \mathsf{ExtractFirst}(\mathsf{List})$;
    $\mathsf{Up}(\overline{\mathcal{D}_1'}) \leftarrow \mathsf{Up}(\mathcal{D}_2)$;
    $\mathcal{D}_1 \leftarrow \overline{\mathcal{D}_1'} \cup \mathcal{D}_2$; $\mathsf{Insert}(\mathcal{D}_1, \mathsf{List})$;
  end if
end while

▷ Different handling of last set

$\mathcal{D}_1 \leftarrow \mathsf{ExtractFirst}(\mathsf{List})$;
if $\Delta = \mathsf{Up}(\mathcal{D}_1) - \widetilde{w}_i$ **then**
  RECURSIVEUBF$(\mathcal{D}_1, \mathbf{s}(\mathcal{D}_1), [\mathsf{Low}(\mathcal{D}_1), \Delta], \widetilde{w}_i)$ and output: "links in $\mathcal{D}_1$ are *done in stage* i";
else
  $\mathbf{t}(\mathcal{D}_1) \leftarrow$ UNSPLITTABLE-BLOCKING-FLOW$(\mathcal{D}_1, \mathbf{s}(\mathcal{D}_1), \Delta, \widetilde{w}_i)$;
  if $\mathcal{D}_1^-(\mathbf{t}(\mathcal{D}_1), \Delta, \widetilde{w}_i) = \emptyset$ **then**
    $\mathsf{Low}(\mathcal{D}_1) \leftarrow \Delta$ and output: "links in $\mathcal{D}_1$ are *done in this sweep*";
  else if $\mathcal{D}_1^-(\mathbf{t}(\mathcal{D}_1), \Delta, \widetilde{w}_i) \neq \emptyset$ **and** $\mathcal{D}_1^+(\mathbf{t}(\mathcal{D}_1), \Delta, \widetilde{w}_i) = \emptyset$ **then**
    $\mathsf{Up}(\mathcal{D}_1) \leftarrow \Delta + \widetilde{w}_i$;
    RECURSIVEUBF$(\mathcal{D}_1, \mathbf{t}(\mathcal{D}_1), [\mathsf{Low}(\mathcal{D}_1), \Delta], \widetilde{w}_i)$ and output: "links in $\mathcal{D}_1$ are *done in stage* i";
  else if $\mathcal{D}_1^-(\mathbf{t}(\mathcal{D}_1), \Delta, \widetilde{w}_i) \neq \emptyset$ **and** $\mathcal{D}_1^+(\mathbf{t}(\mathcal{D}_1), \Delta, \widetilde{w}_i) \neq \emptyset$ **then**
    split $\mathcal{D}_1$ (according to Lemma 5.4 (3)) into sets $\mathcal{D}_1'$ and $\overline{\mathcal{D}_1'}$;
    $\mathsf{Low}(\mathcal{D}_1') \leftarrow \Delta$ and output: "links in $\mathcal{D}_1'$ are *done in this sweep*";
    $\mathsf{Up}(\overline{\mathcal{D}_1'}) \leftarrow \Delta + \widetilde{w}_i$;
    RECURSIVEUBF$(\overline{\mathcal{D}_1'}, \mathbf{t}(\overline{\mathcal{D}_1'}), [\mathsf{Low}(\overline{\mathcal{D}_1'}), \Delta], \widetilde{w}_i)$ and output: "links in $\overline{\mathcal{D}_1'}$ are *done in stage* i";
  end if
end if

Figure 7: SWEEP over the sets of active links

(3) $\underline{\mathcal{D}_1^-(\mathbf{t}(\mathcal{D}_1), \Delta, \widetilde{w}_i) \neq \emptyset \text{ and } \mathcal{D}_1^+(\mathbf{t}(\mathcal{D}_1), \Delta, \widetilde{w}_i) \neq \emptyset}$: In this case, we split $\mathcal{D}_1$ according to condition (3) from Lemma 5.4 into sets $\mathcal{D}_1'$ and $\overline{\mathcal{D}_1'}$. Condition (3c) implies, that no user that is assigned to a link in $\mathcal{D}_1'$ can be assigned to a link in $\overline{\mathcal{D}_1'}$. Since the load on each link in $\mathcal{D}_1'$ is at least Δ, we can set $\mathsf{Low}(\mathcal{D}_1') \leftarrow \Delta$. The load of each link in $\overline{\mathcal{D}_1'}$ is at most $\Delta + \widetilde{w}_i$. Thus, since the upper bound of the next element, say $\mathcal{D}_2$, in List is $\mathsf{Up}(\mathcal{D}_2) > \Delta + \widetilde{w}_i$, we again can extract $\mathcal{D}_2$ from List, set $\mathsf{Up}(\overline{\mathcal{D}_1'}) \leftarrow \mathsf{Up}(\mathcal{D}_2)$, merge $\overline{\mathcal{D}_1'}$ and $\mathcal{D}_2$, and insert it in List. We are done with $\mathcal{D}_1'$ during this execution of SWEEP.

So, in each case, the number of sets in List is decreased by 1. Now, we consider the case that there is only one set, say $\mathcal{D}_1$, in List. This case has to be handled differently.

If $\Delta = \mathsf{Up}(\mathcal{D}_1) - \widetilde{w}_i$, then we apply RECURSIVEUBF to the sub-instance defined by $\mathcal{D}_1$ in the interval $[\mathsf{Low}(\mathcal{D}_1), \Delta]$ with traffic size $\widetilde{w}_i$. Otherwise, when $\Delta = \mathsf{Low}(\mathcal{C}_{x-1})$, we apply UNSPLITTABLE-BLOCKING-FLOW to the sub-instance defined by the set $\mathcal{D}_1$. UNSPLITTABLE-BLOCKING-FLOW$(\mathcal{D}_1, \mathbf{s}(\mathcal{D}_1), \Delta, \widetilde{w}_i)$ returns an assignment $\mathbf{t}(\mathcal{D}_1)$ where one of the following conditions holds:

(1) $\underline{\mathcal{D}_1^-(\mathbf{t}(\mathcal{D}_1), \Delta, \widetilde{w}_i) = \emptyset}$: Here, we set $\mathsf{Low}(\mathcal{D}_1) \leftarrow \Delta$.

(2) $\underline{\mathcal{D}_1^-(\mathbf{t}(\mathcal{D}_1), \Delta, \widetilde{w}_i) \neq \emptyset \text{ and } \mathcal{D}_1^+(\mathbf{t}(\mathcal{D}_1), \Delta, \widetilde{w}_i) = \emptyset}$: In this case, we set $\mathsf{Up}(\mathcal{D}_1) \leftarrow \Delta + \widetilde{w}_i$ and apply RECURSIVEUBF to the sub-instance defined by $\mathcal{D}_1$ in the interval $[\mathsf{Low}(\mathcal{D}_1), \Delta]$ with traffic size $\widetilde{w}_i$.

(3) $\underline{\mathcal{D}_1^-(\mathbf{t}(\mathcal{D}_1), \Delta, \widetilde{w}_i) \neq \emptyset \text{ and } \mathcal{D}_1^+(\mathbf{t}(\mathcal{D}_1), \Delta, \widetilde{w}_i) \neq \emptyset}$: Here, we split $\mathcal{D}_1$ according to condition (3) Lemma Lemma 5.4 into sets $\mathcal{D}_1'$ and $\overline{\mathcal{D}_1'}$. For $\mathcal{D}_1'$ we set $\mathsf{Low}(\mathcal{D}_1') \leftarrow \Delta$ and for $\overline{\mathcal{D}_1'}$ we set $\mathsf{Up}(\overline{\mathcal{D}_1'}) \leftarrow \Delta + \widetilde{w}_i$ and we apply RECURSIVEUBF to the sub-instance defined by $\overline{\mathcal{D}_1'}$ in the interval $[\mathsf{Low}(\overline{\mathcal{D}_1'}), \Delta]$ with traffic size $\widetilde{w}_i$.

After each sweep, by renumbering the partitions, we get a new assignment that again has the same structure as in Definition 6.1. This completes the description of SWEEP. We prove:

Lemma 6.2 *The sweep property holds at the beginning of each execution of* SWEEP. *Moreover, in each execution, either a non-empty set of links is added to the set of active links, or some non-empty set of links is stage-finalized.*

Proof: In order to prove that the sweep property holds at the beginning of each execution of SWEEP, we first show that properties (1), (2) and (3) are maintained in each execution. We then show that property (4) holds after SWEEP is executed.

(1) In each execution of SWEEP, the upper bound of a set $\mathcal{D}_1$ of links is only decreased when it is joined with the next set $\mathcal{D}_2$ in List. Thus, the property that the upper bounds are strictly decreasing in the sequence of sets is maintained. Moreover, if the lower bound of a set of links is increased to Δ, then it is not considered again. Thus, all sets of active links which are not considered again have lower bound Δ, proving that the lower bounds are decreasing in the sequence of sets.

(2) Both algorithms UNSPLITTABLE-BLOCKING-FLOW an RECURSIVEUBF maintain property (2) due to Lemma 6.1 and Corollary 5.3. Moreover, we only decrease an upper bound if $\mathcal{D}_1^+(\mathbf{t}(\mathcal{D}_1), \Delta, \widetilde{w}_i) = \emptyset$, and we increase a lower bound only if $\mathcal{D}_1^-(\mathbf{t}(\mathcal{D}_1), \Delta, \widetilde{w}_i) = \emptyset$. This also maintains (2).

(3) If we split a set $\mathcal{D}_1$ into sets $\mathcal{D}'_1$ and $\overline{\mathcal{D}'_1}$, then still no user on a link in $\mathcal{D}'_1 \cup \overline{\mathcal{D}'_1}$ has a link in a set in List in its strategy set. Moreover, by Lemma 5.4, also no user on a link in $\mathcal{D}'_1$ has a link in $\overline{\mathcal{D}'_1}$ in its strategy set. Thus, property (3) is maintained.

During each SWEEP, we always have a partition of the links into sets such that for some t the sets $\mathcal{C}_t, \ldots, \mathcal{C}_y$ are still in List and $\mathsf{Low}(\mathcal{C}_k) = \mathsf{Low}(\mathcal{C}_y)$ holds for all $t \leq k \leq y$. In general, List contains one further set, say $\mathcal{D}_1$, with $\mathsf{Low}(\mathcal{D}_1) = \mathsf{Low}(\mathcal{C}_y)$ which was obtained by processing $\mathcal{C}_x, \ldots, \mathcal{C}_{t-1}$. For each link from $\mathcal{C}_x, \ldots, \mathcal{C}_{t-1}$ either the lower bound increased to Δ or it is included in $\mathcal{D}_1$. Links with lower bound Δ are not processed further in this sweep.

After the **while** loop, List contains exactly one set of links $\mathcal{D}_1$. For all links not included in $\mathcal{D}_1$, the lower bound was increased to Δ. We increase the lower bound of some of the links in $\mathcal{D}_1$ to Δ, and process the remaining links such that they do not have to be considered again in this stage.

After each execution of SWEEP, we either increased the lower bound of all sets of active links to $\mathsf{Low}(\mathcal{C}_{x-1})$ (which makes $\mathcal{C}_{x-1}$ active), or we processed some links from $\mathcal{C}_y$ by applying RECURSIVEUBF to $\mathcal{C}_y$ such that these links do not have to be considered again in this stage. This defines the new values of x, y for property (2) in the natural way, and it proves the second statement. ■

We continue to prove:

Lemma 6.3 *After stage i, every user u with traffic $w_u \geq \widetilde{w}_i$ is satisfied.*

Proof: Stage i ends when all links are processed such that they do not have to be considered in this stage. After stage i we have an assignment $\mathbf{t}_i$, where the links are partitioned into p_i sets $\mathcal{B}_1, \ldots, \mathcal{B}_{p_i}$ with $\mathsf{Low}(\mathcal{B}_1) > \ldots > \mathsf{Low}(\mathcal{B}_{p_i})$ and $\mathsf{Up}(\mathcal{B}_k) = \mathsf{Low}(\mathcal{B}_k) + \widetilde{w}_i$, for all $k \in [p_i]$. Therefore, no user u with traffic $w_u = \widetilde{w}_i$ that is assigned to a link from $\mathcal{B}_k$ can improve by moving to a link from $\mathcal{B}_\ell$, if $\ell \leq k$. Furthermore, no user u, that is assigned to a link from $\mathcal{B}_k$, can be assigned to a link from $\mathcal{B}_\ell$ when $k < \ell$. Thus, all users with traffic $\widetilde{w}_i$ are satisfied. Since we only increased the lower bounds and decreased the upper bounds on the load of the links, all users u with traffic $w_u > \widetilde{w}_i$ remain satisfied during stage i. ■

We are now ready to prove:

Theorem 6.4 *Consider the model of restricted parallel links for the case of identical links. Given any assignment* $\mathbf{s}$*,* NASHIFY$(\mathbf{s})$ *computes a Nash equilibrium with non-increased Social Cost in time $\mathcal{O}(rmS(\log W + m^2))$, where r is the number of distinct traffic sizes.*

Proof: Lemma 6.3 implies that after stage r *all* users are satisfied and therefore the resulting assignment $\mathbf{t}_r$ is a Nash equilibrium. In no step of the algorithm, the overall maximum latency is increased, so NASHIFY$(\mathbf{s})$ computes a Nash equilibrium with non-increased Social Cost.

It remains to show that the running time is $\mathcal{O}(rmS(\log W + m^2))$. By Theorem 5.5, one call to UNSPLITTABLE-BLOCKING-FLOW takes $\mathcal{O}(mS)$ time. Since in each SWEEP a set of links becomes active or some links are processed such that they do not have to be considered again, we have at most $\mathcal{O}(m)$ executions of SWEEP per stage. Not counting the possible binary search (by a call to RECURSIVEUBF) at the end of each SWEEP, we have at most $\mathcal{O}(m)$ calls to UNSPLITTABLE-BLOCKING-FLOW per each SWEEP. Since in a stage the binary search is done on disjoint subsets of the users and links, the total time for all the binary searches in a stage is $\mathcal{O}(mS \log W)$. Since there are r stages, the total time is $\mathcal{O}(rmS(\log W + m^2))$, as claimed. ■

7 Epilogue

We have further expanded the class of strategic games whose Nash equilibria can be efficiently computed. The class includes now an extension of the selfish routing game from [18] that accommodates restrictions on the pattern of sharing links. To establish tractability, we have imported techniques from network flows into a Nashification algorithm. We believe that such techniques may be the key to settling other instances of the problem that are not yet known to be tractable.

As a by-product, we have improved the current record for the approximation factor of optimum makespan for restricted identical machines. Even more so, we have both improved the approximation factor and simplified the techniques for computing single-source unsplittable flows in the special case of two-layered bipartite graphs.

Subsequent to this paper, Gairing *et al.* [12] provided a combinatorial 2-approximation algorithm for optimum makespan for the general case of unrelated links. For their fast and simple approximation algorithm, the procedure UNSPLITTABLE-BLOCKING-FLOW from this paper is an essential element.

Perhaps, the most interesting extension of our work is to extend our Nashification algorithm to the case of related links, or even to unrelated links. Another important open questions is to improve the approximation factor for optimum makespan (or proving that no further improvement is possible). Perhaps some powerful techniques from network flows will be handy in this endeavor.

References

[1] R. K. Ahuja, T. L. Magnanti and J. B. Orlin, *Network Flows: Theory, Algorithms and Applications,* Prentice Hall, 1993.

[2] B. Awerbuch, Y. Azar, Y. Richter and D. Tsur, "Tradeoffs in Worst-Case Equilibria," *Theoretical Computer Science*, Vol. 361, Nos. 2-3, pp. 200–209, September 2006.

[3] X. Chen and X. Deng. "Settling the Complexity of Two-Player Nash Equilibrium," *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science*, pp. 261–272, October 2006.

[4] C. Daskalakis, P. W. Goldberg and C. H. Papadimitriou, "The Complexity of Computing a Nash Equilibrium," *Proceedings of the 38th Annual ACM Symposium on Theory of Computing*, pp. 71–78, May 2006.

[5] E. A. Dinits, "Algorithm for Solution of a Problem of Maximum Flow in a Network with Power Estimation," *Soviet Mathematics Doklady,* Vol. 11, pp. 1277–1280, 1970.

[6] Y. Dinitz, N. Garg and M. X. Goemans, "On the Single-Source Unsplittable Flow Problem," *Combinatorica,* Vol. 19, No. 1, pp. 17–41, January 1999.

[7] R. Feldmann, M. Gairing, T. Lücking, B. Monien and M. Rode, "Nashification and the Coordination Ratio for a Selfish Routing Game," *Proceedings of the 30th International Colloquium on Automata, Languages, and Programming,* pp. 514–526, Vol. 2719, Lecture Notes in Computer Science, Springer-Verlag, June/July 2003.

[8] D. Fotakis, S. Kontogiannis, E. Koutsoupias, M. Mavronicolas and P. Spirakis, "The Structure and Complexity of Nash Equilibria for a Selfish Routing Game," *Proceedings of the 29th International Colloquium on Automata, Languages, and Programming,* pp. 123–134, Vol. 2380, Lecture Notes in Computer Science, Springer-Verlag, July 2002.

[9] L. R. Ford and D. R. Fulkerson, "A Simple Algorithm for Finding Maximal Network Flows and an Application to the Hitchcock Problem," Canadian Journal of Mathematics, Vol. 9, pp. 210–218, 1957.

[10] M. Gairing, T. Lücking, M. Mavronicolas and B. Monien. "The Price of Anarchy for Restricted Parallel Links," *Parallel Processing Letters*, Vol. 16, No. 1, pp. 117–131, March 2006.

[11] M. Gairing, T. Lücking, M. Mavronicolas, B. Monien and P. Spirakis, "Structure and Complexity of Extreme Nash Equilibria," *Theoretical Computer Science*, Vol. 343, Nos. 1–2, pp. 133–157, October 2005.

[12] M. Gairing, B. Monien and A. Woclaw, "A Faster Combinatorial Approximation Algorithm for Scheduling Unrelated Parallel Machines," *Proceedings of the 32nd International Colloquium on Automata, Languages, and Programming,* pp. 828–839, Vol. 3580, Lecture Notes in Computer Science, Springer-Verlag, July 2005. Also, accepted to *Theoretical Computer Science*.

[13] A. V. Goldberg, *Efficient Graph Algorithms for Sequential and Parallel Computers,* Ph.D. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1987.

[14] A. V. Goldberg and R. E. Tarjan, "A New Approach to the Maximum Flow Problem," *Journal of the ACM,* Vol. 35, No. 4, pp. 921–940, October 1988.

[15] E. Horowitz and S. Sahni, "Exact and Approximate Algorithms for Scheduling Nonidentical Processors," *Journal of the ACM*, Vol. 23, No. 2, pp. 317–327, April 1976.

[16] J. Kleinberg, "Single-Source Unsplittable Flow," *Proceedings of the 37th Annual IEEE Symposium on Foundations of Computer Science,* pp. 68–77, October 1996.

[17] S. G. Kolliopoulos and C. Stein, "Approximation Algorithms for Single-Source Unsplittable Flow," *SIAM Journal on Computing,* Vol. 31, No. 3, pp. 919–946, June 2002.

[18] E. Koutsoupias and C. Papadimitriou, "Worst-Case Equilibria," *Proceedings of the 16th International Symposium on Theoretical Aspects of Computer Science*, pp. 404–413, Vol. 1563, Lecture Notes in Computer Science, Springer-Verlag, March 1999.

[19] J. K. Lenstra, D. B. Shmoys and É. Tardos, "Approximation Algorithms for Scheduling Unrelated Parallel Machines," *Mathematical Programming,* Vol. 46, No. 3, pp. 259–271, April 1990.

[20] R. D. McKelvey and A. McLennan, "Computation of Equilibria in Finite Games," Chapter 2 in *Handbook of Computational Economics*, H. Amman, D. Kendrick and J. Rust, eds., pp. 87–142, Vol. 1, 1996.

[21] I. Milchtaich, "Congestion Games with Player-Specific Payoff Functions," *Games and Economic Behavior,* Vol. 13, No. 1 pp. 111–124, March 1996.

[22] J. F. Nash, "Equilibrium Points in n-Person Games," *Proceedings of the National Academy of Sciences of the United States of America,* Vol. 36, pp. 48–49, 1950.

[23] J. F. Nash, "Non-Cooperative Games," *Annals of Mathematics,* Vol. 54, No. 2, pp. 286–295, 1951.

[24] C. H. Papadimitriou, "Algorithms, Games and the Internet," *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing,* pp. 749–753, July 2001.

[25] E. V. Shchepin and N. Vakhania, ”An Optimal Rounding Gives a Better Approximation for Scheduling Unrelated Machines,” *Operations Research Letters*, Vol. 33, No. 2, pp. 127–133, March 2005.