A compositional approach to the max-flow problem

Kfoury, Assaf

Computer Science Department, Boston University


http://hdl.handle.net/2144/20811

Boston University
A Compositional Approach to the Max-Flow Problem

Assaf Kfoury*
Boston University

Abstract

Although written as a friendly rejoinder to two negative reviews of a 10-page extended abstract, entitled “A Compositional Approach to Network Algorithms,” itself based on a report by the same title [3], this report is intended to be a gentler and more informal addendum to its precursor.

1 Introduction

An earlier report, entitled “A Compositional Approach to Network Algorithms” [3] (or CANA for short henceforth), proposed a new approach for the design and analysis of network algorithms in general. For illustrative purposes, the last section in CANA considered the classical max-flow and min-flow problems restricted to a special case that allowed for favorable comparison with alternative approaches.

The special case in CANA was that of a network \( \mathcal{N} \) whose underlying graph is planar, directed, and with multiple sources and sinks. Every arc \( a \) in \( \mathcal{N} \) was assigned a lower-bound \( c(a) \) and an upper-bound \( \bar{c}(a) \), with the standard requirement that every feasible flow \( f \) in \( \mathcal{N} \) must satisfy \( 0 \leq c(a) \leq f(a) \leq \bar{c}(a) \).

I submitted a 10-page abstract based on CANA to STOC 2014. Three anonymous reviewers were assigned to CANA, to which I will refer by A, B, and C. Two reviewers seemed to doubt the correctness of the last result in the abstract (reviewer A), or the methodology underlying it and its novelty (reviewer B), and only in relation to the max-flow problem. STOC 2014 did not accept my abstract.

The concluding statement of reviewer A was: “Hence, the idea presented in the paper seems to be not working.” Reviewer B stated: “Essentially, the author is suggesting to solve max-flow by divide and conquer/dynamic programming over a decomposition of the graph. The subproblems, however, only seem to consider solutions in which the edges incident to a piece in the decomposition are saturated in one direction. This cannot lead to optimal solutions in which fractional flow assignments are needed,” and concluded flat out: “Since the application does not result in novel results, the paper should not be accepted.” In fact, CANA does not use dynamic programming,1 does not disallow fractional flow assignments, and does not use an approach already tried before (to the best of a literature search). Moreover, neither reviewer A nor reviewer B makes a mention of the fact that CANA also produces a min-flow value, simultaneously at no extra cost, with a max-flow value.2

The present report extracts material from CANA only to the extent needed to respond to the problematic comments of reviewers A and B. The presentation here thus considers only the max-flow problem and ignores the simultaneous min-flow problem – and also ignores other aspects of CANA that are unrelated to max-flow and min-flow, and, in particular, its benefits for algorithm implementation:

- it supports distributed design that alternative approaches do not (‘distributed’ means ‘parallel’ and more [3]),
- it achieves the preceding by defining an appropriate typing theory for networks,

---

*Partially supported by NSF awards CCF-0820138 and CNS-1135722.
1Certainly not ‘dynamic programming’ in the usual sense, as defined in Chapt. 3 in [1] or Chapt. 15 in [2], for example.
2The standard way of reducing a min-flow problem to a max-flow problem consists in introducing, for each arc with non-zero lower bound, a new (source,sink) pair (Problem 6.18 in [1]). If we join all new sources together and all new sinks together, this reduction destroys planarity, and if not, it introduces as many new sources and sinks as there are arcs in it. This problem does not arise in CANA. The three reviewers’ reports – A, B, and C – are included in full in Appendix A.
which were the motivation for developing CANA’s methodology in the first place. Achieving an asymptotic linear-time complexity was an incidental result. Contrary to my intention or expectation, reviewers A and B limited their comments to the latter (the complexity result), not the former, which they also evidently misunderstood (in two unrelated ways). Reviewer C focused on the former (the type-based framework for design and analysis), but also thought that, given the centrality of types and typings in this work, STOC is perhaps the wrong forum for it, and concluded: “If previous work along this line is any indication […], the work is more suitable for […] programming language conferences.”³

In brief, one goal of this report is to counter reviewers A and B, by way of a guide with extended simple examples and enough evidence that:

1. the complexity bound for max-flow in directed planar graphs, as stated in CANA [3], is correct, and
2. the methodology to obtain that bound is also correct, with no need to restrict flow assignments in any way.

Another goal is to provide a gentle introduction to parts in CANA [3] dealing with max-flow.

Sections 2 and 3 here are shorter versions of Sections 2 and 3 in CANA [3], with additional informal comments. The rest of the report re-organizes and amplifies several parts from CANA to handle the special case mentioned in point (1), when all lower bounds are also zero. Missing proofs and formal definitions are in [3]. I tried to omit as much of the formal details as possible that are incidental to the goals stated above.

## 2 Flow Networks and Their Typings

We take flow networks in their simplest form, as capacitated finite directed graphs. We repeat standard notions [1], but now adapted to our context. A flow network \( \mathcal{N} \) is a pair \( \mathcal{N} = (\mathcal{N}, A) \), where \( \mathcal{N} \) is a finite set of nodes and \( A \) a finite set of directed arcs, with each arc connecting two distinct nodes (no self-loops and no multiple arcs in the same direction connecting the same two nodes). We write \( \mathbb{R} \) and \( \mathbb{R}_+ \) for the sets of reals and non-negative reals. Such a flow network \( \mathcal{N} \) is supplied with a capacity function \( c: A \to \mathbb{R}_+ \), such that \( c(a) > 0 \) for every \( a \in A \). We write tail\((a)\) and head\((a)\) for the two ends of arc \( a \in A \). The set \( A \) of arcs is the disjoint union of three sets, i.e., \( A = A_\# \cup A_{\text{in}} \cup A_{\text{out}} \) where:

\[
A_\# := \{ a \in A | \text{head}(a) \in \mathcal{N} \text{ and } \text{tail}(a) \in \mathcal{N} \}\] (the internal arcs of \( \mathcal{N} \)),

\[
A_{\text{in}} := \{ a \in A | \text{head}(a) \in \mathcal{N} \text{ and } \text{tail}(a) \notin \mathcal{N} \}\] (the input arcs of \( \mathcal{N} \)),

\[
A_{\text{out}} := \{ a \in A | \text{head}(a) \notin \mathcal{N} \text{ and } \text{tail}(a) \in \mathcal{N} \}\] (the output arcs of \( \mathcal{N} \)).

A flow is a function \( f: A \to \mathbb{R}_+ \) which, if feasible, satisfies “flow conservation” at every node and “capacity constraints” at every arc, both defined as in the standard formulation [1].

We call a bounded closed interval \([r, r']\) of real numbers (possibly negative) a type. A typing is a partial map \( T \) (possibly total) that assigns types to subsets of the input and output arcs. Formally, \( T \) is of the following form, where \( A_{\text{in,out}} = A_{\text{in}} \cup A_{\text{out}} \) and \( \mathcal{P}(\ ) \) is the power-set operator, \( \mathcal{P}(A_{\text{in,out}}) = \{ A | A \subseteq A_{\text{in,out}} \} \):

\[
T : \mathcal{P}(A_{\text{in,out}}) \to \mathcal{T}(\mathbb{R}) \quad \text{where} \quad \mathcal{T}(\mathbb{R}) := \{ [r, r'] \mid r, r' \in \mathbb{R} \text{ and } r \leq r' \},
\]

i.e., \( \mathcal{T}(\mathbb{R}) \) is the set of bounded closed intervals. As a function, \( T \) is not totally arbitrary and satisfies conditions that make it a network typing; in particular, it will always be that \( T(\emptyset) = [0, 0] = \{0\} = T(A_{\text{in,out}}) \), the latter condition expressing the fact that the total amount entering a network must equal the total amount exiting it.

---

³A judgment call on the part of reviewer C, which I accept, but which is also questionable: Is a programming language conference really a better venue for work on the design and analysis of network algorithms? There was no combinatorial optimization to speak of, in earlier articles that used the typing theory presented in CANA [3].

⁴The notation “\( A_{\text{in,out}} \)” is ambiguous, because it does not distinguish between input arcs and output arcs. We use it nonetheless for succinctness. The context will always make clear which members of \( A_{\text{in,out}} \) are input arcs and which are output arcs.
An input/output assignment (or IO assignment) is a function \( f : A_{\text{in, out}} \to \mathbb{R}_+ \). For a flow \( f : A \to \mathbb{R}_+ \), or an IO assignment \( f : A_{\text{in, out}} \to \mathbb{R}_+ \), we say \( f \) satisfies the typing \( T \) iff, for every \( A \in \mathcal{P}(A_{\text{in, out}}) \) such that \( T(A) \) is defined and \( T(A) = [r_1, r_2] \), we have:

\[
r_1 \leq \sum f(A \cap A_{\text{in}}) - \sum f(A \cap A_{\text{out}}) \leq r_2
\]

where \( \sum f(X) \) means \( \sum \{ f(x) \mid x \in X \} \). In words, this says that the “sum of the values assigned by \( f \) to input arcs” minus the “sum of the values assigned by \( f \) to output arcs” is within the interval \([r_1, r_2]\).

3 Principal typings

We say a typing \( T \) is sound for network \( \mathcal{N} \) if:

- Every IO assignment \( f : A_{\text{in, out}} \to \mathbb{R}_+ \) satisfying \( T \) is extendable to a feasible flow \( f' : A \to \mathbb{R}_+ \) in \( \mathcal{N} \).

A sound typing is one that is generally more conservative than required to prevent system’s malfunction: It filters out all unsafe IO assignments, i.e., not extendable to feasible flows, and perhaps a few more that are safe.

For our application here (max-flow problem), not only do we want to assemble networks for their safe operation, we want to operate them to the limit of their safety guarantees. We therefore say a typing \( T \) is complete for network \( \mathcal{N} \) if:

- Every feasible flow \( f : A \to \mathbb{R}_+ \) in \( \mathcal{N} \) satisfies \( T \).

Let \( |A_{\text{in}}| = p \geq 1 \) and \( |A_{\text{out}}| = q \geq 1 \), and assume a fixed ordering of the arcs in \( A_{\text{in, out}} \). An IO assignment \( f : A_{\text{in, out}} \to \mathbb{R}_+ \) specifies a point, namely \( \{ f(a) \mid a \in A_{\text{in, out}} \} \), in the vector space \( \mathbb{R}^{p+q} \), and the collection of all IO assignments satisfying a typing \( T \) form a compact convex polyhedral set (or polytope) in the first orthant \((\mathbb{R}_+)^{p+q}\), which we denote \( \text{Poly}(T) \).

One complication when dealing with typings as polytopes arises from alternative representations (convex hulls vs. intersections of halfspaces). We choose to represent them by intersecting halfspaces, with some (not all) redundancies in the defining linear inequalities eliminated. We thus say the typing \( T \) is tight if, for every \( A \subseteq A_{\text{in, out}} \) for which \( T(A) \) is defined and every \( r \in T(A) \), there is an IO assignment \( f \in \text{Poly}(T) \) such that:

\[
r = \sum f(A \cap A_{\text{in}}) - \sum f(A \cap A_{\text{out}}).
\]

Informally, \( T \) is tight if no defined \( T(A) \) contains redundant information.

Another kind of redundancy occurs when an interval/type \( T(A) \) is defined for some \( A = B \cup B' \subseteq A_{\text{in, out}} \) with \( B \neq \emptyset \neq B' \) even though there is no connection between \( B \) and \( B' \). We eliminate this kind of redundancy via what we call “locally total” typings. We need a preliminary notion. A network \( \mathcal{M} = (\mathcal{N}, \mathbb{B}) \) is a subnetwork of network \( \mathcal{N} = (\mathcal{N}, \mathbb{A}) \) if \( \mathbb{M} \subseteq \mathcal{N} \) and \( \mathbb{B} \subseteq \mathbb{A} \) such that:

\[
\mathbb{B}_# = \{ a \in \mathbb{A} \mid \text{head}(a) \in \mathbb{M} \& \text{tail}(a) \in \mathbb{M} \},
\]

\[
\mathbb{B}_{\text{in}} = \{ a \in \mathbb{A} \mid \text{head}(a) \in \mathbb{M} \& \text{tail}(a) \notin \mathbb{M} \},
\]

\[
\mathbb{B}_{\text{out}} = \{ a \in \mathbb{A} \mid \text{head}(a) \notin \mathbb{M} \& \text{tail}(a) \in \mathbb{M} \}.
\]

We also say \( \mathcal{M} \) is the subnetwork of \( \mathcal{N} \) induced by \( \mathbb{M} \). The subnetwork \( \mathcal{M} \) is a component of \( \mathcal{N} \) if \( \mathcal{M} \) is connected and \( \mathbb{B}_# \subseteq \mathbb{A}_# \), \( \mathbb{B}_{\text{in}} \subseteq \mathbb{A}_{\text{in}} \), and \( \mathbb{B}_{\text{out}} \subseteq \mathbb{A}_{\text{out}} \), i.e., \( \mathcal{M} \) is a maximal connected subnetwork of \( \mathcal{N} \). If network \( \mathcal{N} \) contains two distinct component \( \mathcal{M} \) and \( \mathcal{M}' \), there is no communication between \( \mathcal{M} \) and \( \mathcal{M}' \), and the typings of the latter two can be computed independently of each other. We say a typing \( T \) for \( \mathcal{N} \) is locally total if, for all components \( \mathcal{M} = (\mathcal{N}, \mathbb{B}) \) and \( \mathcal{M}' = (\mathcal{N}', \mathbb{B}') \) of \( \mathcal{N} \), and all \( B \in \mathbb{B}_{\text{in, out}} \) and \( B' \in \mathbb{B}'_{\text{in, out}} \):

- The interval/type \( T(B) \) is defined.
- If \( \mathcal{M}' \neq \mathcal{M} \) and \( B \neq \emptyset \neq B' \), the interval/type \( T(B \cup B') \) is not defined.
Whereas “tight” and “locally total” can be viewed (and are in fact) properties of a typing $T$, independent of any network $\mathcal{N}$ for which $T$ is a typing, “sound” and “complete” are properties of $T$ relative to a particular $\mathcal{N}$. If $\mathcal{N}$ has only one component (itself), a locally-total typing for $\mathcal{N}$ is a total function on $\mathcal{P}(\mathcal{A}_{\text{in,out}})$. We can prove:

**Theorem 1** (Uniqueness of Locally Total, Tight, Sound and Complete Typings). For all networks $\mathcal{N}$, there is a unique typing $T$ which is locally total, tight, sound and complete – henceforth called the principal typing of $\mathcal{N}$.

The principal typing of $\mathcal{N}$ is a characterization of all IO assignments extendable to feasible flows in $\mathcal{N}$. In the context of this report, there are two important corollaries of Theorem 1, stated next.

**Corollary 2.** Let $\mathcal{N}$ be a connected network, whose set of input arcs is $\mathcal{A}_{\text{in}}$ and set of output arcs is $\mathcal{A}_{\text{out}}$. If $T$ is the principal typing of $\mathcal{N}$, then the value of a max-flow in $\mathcal{N}$ is the upper limit of the interval/type $T(\mathcal{A}_{\text{in}})$ or, equivalently, the negation of the lower limit of $T(\mathcal{A}_{\text{out}})$.

We say two networks $\mathcal{N}$ and $\mathcal{N}'$ are similar if they have the same number $p \geq 1$ of input arcs and same number $q \geq 1$ of output arcs. To avoid incidental technicalities in the next corollary, we assume that input/output arcs come as ordered lists, say $\{a_1, \ldots, a_p, a_{p+1}, \ldots, a_{p+q}\}$ (for $\mathcal{N}$) and $\{a'_1, \ldots, a'_p, a'_{p+1}, \ldots, a'_{p+q}\}$ (for $\mathcal{N}'$).

**Corollary 3.** Two similar networks $\mathcal{N}$ and $\mathcal{N}'$ are equivalent iff their principal typings are equal, modulo the renaming $a_i \mapsto a'_i$ for every $1 \leq i \leq p+q$.

For the proof of Theorem 1, we can compute a principal typing typing $T$ for network $\mathcal{N}$ via linear-programming as follows: For every $A \subseteq \mathcal{A}_{\text{in,out}}$, we specify an objective $\theta_A$ to be minimized and maximized:

$$\theta_A := \sum (A \cap \mathcal{A}_{\text{in}}) - \sum (A \cap \mathcal{A}_{\text{out}}),$$

corresponding to the two limits of the type $T(A)$, relative to the collection $\mathcal{C}$ of flow-preservation equations (one for each node, $n = |\mathcal{N}|$ of them) and capacity-constraint inequalities (one for each arc, $m = |\mathcal{A}|$ of them). Following this approach in the proof of Theorem 1, it is straightforward to show that the resulting typing $T$ is complete for $\mathcal{N}$. More difficult, and perhaps unexpected, is that the resulting $T$ is also sound for $\mathcal{N}$ (see definition of soundness at the beginning of this section).

Although the existence of principal typings is important for other purposes [3], if we use linear programming to prove Theorem 1, in order to compute a principal typing and then extract a max-flow value from it, we do not get much of an advantage over alternative approaches. We have known for a long time how to compute a max-flow value (or a min-flow value or other linearly-expressible measures of a network $\mathcal{N}$) using linear programming [1]. But the resulting run-time complexity, even with the best-performing linear-programming procedure, is often worse than that of max-flow algorithms that work directly on the underlying graph of $\mathcal{N}$.

In the rest of this report, by way of extended simple examples, we spell out the facts that make it possible to compute the principal typing of a network, with and without invoking a pre-defined linear-programming procedure, in Sections 5 and 7, respectively – and without messing up the asymptotic run-time complexity. There is no dynamic programming anywhere, nor are there restrictions on flow assignments and arc capacities.

### 4 Simple Examples

Consider a directed 3-regular planar graph $\mathcal{N}_0$, with a single source (one input arc) and a single sink (one output arc), where every arc $a$ is assigned an upper-bound capacity $c(a) > 0$. A very simple example of such a network $\mathcal{N}_0$, with 11 internal arcs, is shown in Figure 1. For simplicity, the arc capacities in $\mathcal{N}_0$ are integral and relatively small in magnitude, shown in framed boxes, but these are not requirements for our approach to work.

Our proposed algorithm for computing the max-flow value proceeds by disassembling and reassembling $\mathcal{N}_0$. It disassembles $\mathcal{N}_0$ down to its one-node subnetworks, computes their principal typings, and then reassembles $\mathcal{N}_0$ in stages – thus obtaining larger and larger subnetworks – in order to guide the process of combining the one-node principal typings and produce the principal typing for the whole of $\mathcal{N}_0$ at the end.
Figure 1: A simple network $N_0$.

Figure 2: Bad reassembling strategy (or binding schedule) for $N_0$: Subnetworks' external dimensions depend on number $n$ of nodes. Read left-to-right, top-to-bottom; shaded areas are subnetworks reassembled at each stage.

Figure 3: Good reassembling strategy (or binding schedule) for $N_0$: Subnetworks’ external dimensions are independent of number $n$ of nodes. Read left-to-right, top-to-bottom; shaded areas are subnetworks so far reassembled.
This process is started conveniently from a node on the outer face (we here choose the source of $N_0$, the node incident to its input arc), though it does not have to, and is depicted in Figure 2 and again, differently, in Figure 3. Note that, even though $N_0$ has one input arc and one output arc, intermediate subnetworks in the process of reassembling $N_0$ will generally have several input arcs and several output arcs.

Both ways of reassembling $N_0$, in Figure 2 and in Figure 3, produce the principal typing of $N_0$. There is however a crucial difference that affects the algorithm’s run-time complexity (and its implementation too):

- The former (Figure 2) allows the external dimension to grow with $n$, while the latter (Figure 3) keeps a uniform bound (here 4) on the external dimension of all intermediate subnetworks,

where $n$ (here $n = 8$) is the number of nodes in the network and external dimension $= \text{(number of input arcs)} + \text{(number of output arcs)}$.

What we call the binding schedule in the full report [3] is the order in which one-node subnetworks are reassembled. Figure 2 and Figure 3 show two different binding schedules – call them $\sigma_1$ and $\sigma_2$, respectively. And what we call the index of a binding schedule $\sigma$ is the least upper bound on the external dimension of subnetworks reassembled according to $\sigma$. Thus, in this example, $\text{index}(\sigma_1) = n - 1 = 7$ and $\text{index}(\sigma_2) = 4$.

More generally, even though the algorithm is non-deterministic, we show in [3] that it can greedily choose its next step so that the resulting binding schedule $\sigma$ is optimal, i.e., $\sigma$ has the smallest possible index($\sigma$). In words, the algorithm can proceed in such a way that the least upper bound on the external dimension of intermediate subnetworks is minimized and independent of the number $n$ of nodes. The crucial fact is this:

- There are natural graph topologies for which an optimal binding schedule $\sigma$ is such that $\text{index}(\sigma)$ does not depend on the number $n$ of nodes in the graph.

One such topology is that of 3-regular $k$-outerplanar networks (considered in [3]) for any $k \geq 1$. $N_0$ in Figure 1 is 3-regular 1-outerplanar. But there are other topologies that can be also handled in just the same way, such as, for $k \geq 1$, the class of all networks that have a so-called tree decomposition of width $k$ (not covered in [3]); every network in this class has an optimal binding schedule $\sigma$ such that $\text{index}(\sigma)$ depends on $k$ but not on $n$.

![Figure 4: Another good reassembling strategy (or binding schedule) for $N_0$: Subnetworks' external dimensions are independent of number $n$ of nodes. Read left-to-right, top-to-bottom; shaded areas are subnetworks reassembled.](image)

Optimal binding schedules are not uniquely defined for the same network. Another optimal schedule for the same network $N_0$, call it $\sigma_3$, is shown in Figure 4. This one is obtained by applying Algorithm 4 in Appendix C of the earlier report [3], which binds all cross arcs first (the top row of Figure 4) before handling peeling arcs (the second and third rows of Figure 4). As in [3], we distinguish between peeling arcs (drawn in boldface) and cross arcs (drawn in thin face).

The asymptotic run-time complexity is the same for all optimal binding schedules. However, the constants
hidden by the “big $O$” notation may differ from one optimal schedule to another. In Section 7, we explicitly compute the hidden constants of the time complexity, as a function of the number of performed arithmetical operations, something we did not do in [3]. It turns out that the hidden constants corresponding to schedule $\sigma_3$ are smaller than those corresponding to $\sigma_2$, as we later explain.

5 Computing Principal Typings (Method 1)

To compute the principal typing of a one-node subnetwork, with 3 incident arcs whose lower-bound capacities = 0, is straightforward.\(^5\) For example, the one-node subnetwork whose node is the middle upper node in Figure 1 is shown in Figure 5, where we named the input arcs $x_4^+$ and $x_6^+$ and the output arc $x_7^-$. 

Our naming convention is to call “$x_4^+$” the “input half” of arc $x_4$ and “$x_4^-$” the “output half” of arc $x_4$, and similarly for all the other arcs. Later, when we splice together $x_4^+$ and $x_7^-$, we get back the original $x_4$.

Its principal typing $U_1: \mathcal{P}(\{x_4^+, x_6^+, x_7^\}) \rightarrow \mathcal{I}(\mathbb{R})$ is specified by the type assignments $U_1(\emptyset) = U_1(\{x_4^+, x_6^+, x_7^\}) = [0, 0] = \{0\}$ in addition to:

$$
\begin{align*}
  x_4^+ & \colon [0, 10] \quad x_6^+ & \colon [0, 5] \quad -x_7^- & \colon [-10, 0] \\
  x_4^+ + x_6^+ & \colon [0, 10] \quad x_4^+ - x_7^- & \colon [-5, 0] \quad x_6^+ - x_7^- & \colon [-10, 0]
\end{align*}
$$

where the notation “$x_4^+ - x_7^- : [-5, 0]$” means $U_1(\{x_4^+, x_7^-\}) = [-5, 0]$ and similarly for the other type assignments. The minus preceding “$x_7^-$” indicates that $x_7^-$ is an output arc. The type assignment $x_4^+ - x_7^- : [-5, 0]$ is obtained by observing that:

$$
\begin{align*}
  0 & = \text{max flow that can enter } x_4^+ \text{ when flow exiting } x_7^- \text{ is minimized (which is here 0),} \\
  -5 & = \text{max flow that can exit } x_7^- \text{ when flow entering } x_4^+ \text{ is minimized (which is here 0),}
\end{align*}
$$

and similarly for the other type assignments. (Entering flow is positive, exiting flow negative.) Another way of understanding the type assignment $x_4^+ - x_7^- : [-5, 0]$ is that, for every IO assignment $f : \{x_4^+, x_6^+, x_7^-\} \rightarrow \mathbb{R}_+$ in the subnetwork, it is the case that: “$-5 \leq f(x_4^+) - f(x_7^-) \leq 0$” is a necessary condition for $f$ to be lifted to a feasible flow.

We omit the details of how the typings $U_2$ and $U_3$ (below) are computed, because the typing $U_4$, which requires the prior determination of $U_2$ and $U_3$, is more complicated and can be used as a guide to obtain the simpler $U_2$ and $U_3$.

The typing $U_2: \mathcal{P}(\{x_4^+, x_5^+, x_7, x_8^\}) \rightarrow \mathcal{I}(\mathbb{R})$ is the principal typing of the two-node subnetwork shown in Figure 6, which consists of the middle upper node and the middle lower node of the network in Figure 1. We use the same naming conventions for the arcs as in $U_1$ above. $U_2$ is specified by the type assignments $U_2(\emptyset) = U_2(\{x_4^+, x_5^+, x_7, x_8^\}) = [0, 0] = \{0\}$, in addition to:

$$
\begin{align*}
  x_4^+ & \colon [0, 10] \quad x_5^+ & \colon [0, 8] \\
  -x_7^- & \colon [-10, 0] \quad -x_8^- & \colon [-8, 0] \\
  x_4^+ + x_5^+ & \colon [0, 18] \quad x_4^+ - x_7^- & \colon [-5, 0] \\
  x_4^+ - x_8^- & \colon [-8, 10] \quad x_5^+ - x_7^- & \colon [-10, 8] \quad x_5^+ - x_8^- & \colon [0, 5] \\
  x_4^+ + x_5^+ - x_7^- & \colon [0, 8] \quad x_4^+ + x_5^+ - x_8^- & \colon [0, 10] \quad x_4^+ - x_7^- - x_8^- & \colon [-8, 0] \quad x_5^+ - x_7^- - x_8^- & \colon [-10, 0]
\end{align*}
$$

\(^5\) When there are lower-bounds > 0, and more than 3 incident arcs, the computation is a little more involved (Lemma 15 in [3]).
The typing $U_3 : \mathcal{P}(\{a_1, x_4, x_5\}) \rightarrow \mathcal{I}(\mathbb{R})$ is the principal typing of the three-node subnetwork shown in Figure 7, which consists of the three leftmost nodes of the network in Figure 1. $U_3$ is specified by the type assignments $U_3(\varnothing) = U_3(\{a_1, x_4^-, x_5^-\}) = [0, 0] = \{0\}$, in addition to:

$$a_1 : [0, 15] \quad -x_4^- : [-10, 0] \quad -x_5^- : [-8, 0]$$
$$a_1 - x_4 : [0, 8] \quad a_1 - x_5^+ : [0, 10] \quad -x_4^- - x_5^- : [-15, 0]$$

We next combine the principal typings $U_2$ and $U_3$ to obtain the principal typing $U_4 : \mathcal{P}(\{a_1, x_7, x_8^-\}) \rightarrow \mathcal{I}(\mathbb{R})$ of the five-node subnetwork shown in Figure 8 without re-visiting the latter’s internal details. We can compute $U_4$ in one of two different ways:

**Method 1:** In this section, using a pre-defined LP procedure.

**Method 2:** In Section 7, without using a pre-defined LP procedure.

But note carefully in this section:

- *If we can reassemble the network according to a binding schedule $\sigma$ such that $\text{index}(\sigma) = k$ is independent of $n$ (the total number of nodes in the network), then, even though we use a pre-defined LP procedure, we are able to keep the overall run-time asymptotically linear in $n$.\)

This is so because, at every stage of the algorithm, we apply the LP procedure to optimize objective functions relative to a set $\mathcal{C}$ of constraints written in terms of only the input/output arcs of two subnetworks — say, $M_1$ and $M_2$ — each of external dimension $\leq k$, and excluding their internal arcs; $\mathcal{C}$ is the set of constraints induced by the principal typings of $M_1$ and $M_2$, together with equality constraints (one for each arc binding).

This is in sharp contrast to the standard approach of setting up a linear-programming problem to compute the max-flow value where optimization is relative to a large collection of flow-preservation equations (one for every node in the entire network, $n$ of them) and capacity-constraint inequalities (one for every arc in the entire network, $m$ of them) using $m$ variables/arc names. (See, for example, Chapters 15, 16, and 17 in [1].) In this standard approach, the resulting asymptotic complexity is, at best, a non-linear polynomial in $m$ and $n$ (if the LP procedure uses the ellipsoid method or the interior-point method or one of their variants which, although generally weakly polynomial, can be made strongly polynomial but still non-linear for max-flow problems).

Nonetheless, there is still a drawback in using a pre-defined LP procedure in this section. This is so because a pre-defined LP procedure is normally set up to handle linear-programming problems in general and thus uses arithmetical operations far in excess of the optimization we need for our particular situation. As a result, it is also difficult to make explicit the constants that are hidden in the “big $O$” notation of the asymptotic run-time. Making explicit the hidden constants will not be a problem with **Method 2** in Section 7.

The preceding comments are illustrated when we apply **Method 1** to compute the principal typing $U_4$ of the subnetwork in Figure 8. For this, we first collect the linear constraints induced by $U_2$ and $U_3$, namely:

**Induced by $U_2$:**

1. $0 \leq x_4^- \leq 10$
2. $0 \leq x_5^- \leq 8$
3. $0 \leq x_7^- \leq 10$
4. $0 \leq x_8^- \leq 8$
5. $0 \leq x_4^+ + x_5^- \leq 18$
6. $-5 \leq x_4^- - x_7^- \leq 0$
7. $-8 \leq x_4^- - x_8^- \leq 10$
8. $-10 \leq x_5^- - x_7^- \leq 8$
9. $0 \leq x_7^- + x_8^- \leq 18$
10. $0 \leq x_4^+ + x_5^- - x_7^- \leq 8$
11. $0 \leq x_4^+ + x_5^- - x_8^- \leq 10$
12. $0 \leq -x_4^+ + x_7^- + x_8^- \leq 8$
13. $0 \leq -x_4^+ + x_7^- + x_8^- \leq 10$
14. $x_4^+ + x_5^- - x_7^- - x_8^- = 0$

**Induced by $U_3$:**

1. $0 \leq a_1 \leq 15$
2. $0 \leq x_4^- \leq 10$
3. $0 \leq x_5^- \leq 8$
4. $0 \leq a_1 - x_4^- \leq 8$
5. $0 \leq a_1 - x_5^- \leq 10$
6. $0 \leq x_5^- + x_5^- \leq 15$
7. $a_1 - x_4^- - x_5^- = 0$

---

$^6$**Method 1** is not used in [3], precisely because we wanted to avoid the use of a pre-defined LP procedure. We include it here because it is easier to understand than **Method 2** in Section 7.
If \( C \) is the set of the preceding constraints, together with the equality constraints \( \{ x_4^+ = x_4^-, x_5^+ = x_5^- \} \), we use our pre-defined LP procedure to compute the types/interval which \( U_4 \) assigns to sets \( A \in \mathcal{P}(\{ a_1, x_7, x_8^\prime \}) \). For example, if \( A = \{ a_1, x_7^- \} \) and \( U_4(A) = [r_1, r_2] \), then \( r_1 \) and \( r_2 \) are the minimum and maximum possible values, respectively, of the objective function \( \theta_A = a_1 \cdot x_7^- \) relative to the constraint set \( C \).

Proceeding inductively, if \( U_2 \) and \( U_3 \) are principal for their respective subnetworks, so will the typing \( U_4 \) be principal for the subnetwork in Figure 8 (this can be proved directly or, more simply, by invoking Theorem 1). In this case, we have to apply the LP procedure to constraints involving 7 variables \( \{ a_1, x_4^+, x_4^-, x_5^+, x_5^-, x_7, x_8^\prime \} \), which is the total number of input/output arcs in the two subnetworks that are to be spliced together. Using Method 1, the type assignments made by principal typing \( U_4 \) are the following, in addition to \( U_4(\emptyset) = U_4(\{ a_1, x_7, x_8^\prime \}) = [0, 0] = \{ 0 \} \):

\[
\begin{align*}
a_1 & : [0, 15] & -x_7^- & : [-10, 0] & -x_8^- & : [-8, 0] \\
a_1 - x_7^- & : [0, 8] & a_1 - x_8^- & : [0, 10] & -x_7^- - x_8^- & : [-15, 0]
\end{align*}
\]

Because \( U_4 \) is principal for the subnetwork in Figure 8, we now have a necessary and sufficient condition for an IO assignment \( f : \{ a_1, x_7, x_8^\prime \} \to \mathbb{R}_+ \) to be extendable to a feasible flow in the subnetwork:

- \( f : \{ a_1, x_7, x_8^\prime \} \to \mathbb{R}_+ \) is extendable to a feasible flow iff \( f \) satisfies the typing \( U_4 \).

In the same way, we continue the reassembling of the entire network \( N_0 \) from its one-node subnetworks, and use the process to guide the combining of its subnetworks’ principal typings, in order to finally obtain the principal typing \( T_0 : \mathcal{P}(\{ a_1, a_2 \}) \to \mathcal{I}(\mathbb{R}) \) of \( N_0 \). In addition to \( T_0(\emptyset) = T_0(\{ a_1, a_2 \}) = [0, 0] \), the typing \( T_0 \) makes the assignments “\( a_1 : [0, 13] \)” and “\( -a_2 : [-13, 0] \)”, from which we can finally extract the max-flow value (here, 13).

Figure 9 shows the underlying \( n \)-node graph of a network \( N_\infty \) generalizing network \( N_0 \) in Figure 1. We omit the directions of internal arcs in \( N_\infty \) because they do not affect the execution of the algorithm (in Method 1).

In the same way we handled the network \( N_0 \), we can disassemble \( N_\infty \) down to its one-node subnetworks, compute their principal typings, and then from the latter, reassemble the entire network in order to compute the principal typing \( T_\infty : \mathcal{P}(\{ a_1, a_2 \}) \to \mathcal{I}(\mathbb{R}) \) of \( N_\infty \).

The cost of computing the principal typings of the one-node subnetworks of \( N_\infty \) is \( O(n) \). If we choose a binding schedule \( \sigma \) such that \( \text{index}(\sigma) = k \) is independent of \( n \), then each stage in the process of reassembling \( N_\infty \) has a cost \( \leq f(k) \), for some function \( f \), which is the cost of executing our pre-defined LP procedure on an LP instance with at most \( 2^k \) constraints over at most \( k \) variables. The total cost of computing the principal typing \( T_\infty \) is therefore \( O(n \cdot f(k)) \). We do better in Section 7.

6 From Arbitrary Networks to 3-Regular Networks

The algorithm in the last section of the earlier report [3] depends on resolving two questions:

(A) How to transform a given network \( N' \) into an equivalent 3-regular network \( N'' \), and

(B) How to compute an optimal binding schedule \( \sigma \) for \( N'' \).
In [3], we answered question (A) in general and answered question (B) in the particular case of planar graphs. The planar embedding of a network has outerplanarity $k \geq 1$ if it has $k$ layers of nodes, i.e., after iteratively removing the nodes (and incident arcs) on the outer face at most $k$ times, we obtain the empty network. A planar network is of outerplanarity $k$ if it has a planar embedding (not necessarily unique) of outerplanarity $k$.

**Lemma 4 (From Arbitrary Networks to 3-Regular Networks).** Let $\mathcal{N} = (\mathcal{N}, \mathcal{A})$ be a flow network, not necessarily planar. In time $O(n)$, we can transform $\mathcal{N}$ into a similar network $\mathcal{N}' = (\mathcal{N}', \mathcal{A}')$ such that:

1. There are no two-node cycles in $\mathcal{N}'$.
2. The degree of every node in $\mathcal{N}'$ is 3.
3. Every typing $T : P(\mathcal{A}_{in, out}) \rightarrow I(\mathbb{R})$ is principal for $\mathcal{N}$ iff $T$ is principal for $\mathcal{N}'$.
4. $|\mathcal{N}'| \leq 2m$ and $|\mathcal{A}'| \leq 3m$, where $m = |\mathcal{A}|$.
5. If $\mathcal{N}$ is given in a $k$-outerplanar embedding, $\mathcal{N}'$ is returned in a $k'$-outerplanar embedding with $k' \leq 2k$.

Lemma 4 here is Lemma 6 in [3], where the proof is included in Appendix C. Parts 1-4 are straightforward, only part 5 is a little complicated to prove. One consequence of part 5 is that, if $\mathcal{N}$ is planar, then so is $\mathcal{N}'$, and the outerplanarity of the latter increases by a factor of at most 2.

An example of the transformation in the proof of Lemma 4 is shown in Figure 10: A node $\nu$ of degree $= 5$ in $\mathcal{N}$ is transformed into a 5-node cycle, where every node has degree $= 3$. In this example, there are two entering arcs $\{1, 5\}$ and three exiting arcs $\{2, 3, 4\}$, so that the maximum amount that can transit through node $\nu$, from inputs to outputs, is $K = \min \{ e(1) + e(5), e(2) + e(3) + e(4) \}$.

An example of how an entire network is transformed is shown in Figure 11, from a 1-outerplanar $\mathcal{N}_1$ to an equivalent 2-outerplanar $\mathcal{N}_2'$, which is then redrawn on a rectangular grid. The redrawing on a rectangular grid is not strictly necessary, but makes it easy to define an optimal binding schedule and to reason about it.\(^7\) Again, as in [3] and in Section 4, we distinguish between peeling arcs (in boldface) and cross arcs (in thin face).

**Computing optimal binding schedules.** Much of Appendix C in the earlier report [3] is devoted to the question of how to reassemble planar networks according to a schedule $\sigma$ that makes $\text{index}(\sigma)$ independent of the number $n$ of nodes. This is what Algorithm 4 in Appendix C does, but before running the algorithm on a 3-regular planar network $\mathcal{N}''$ (itself the result of transforming a given planar network $\mathcal{N}'$), $\mathcal{N}''$ is pre-processed further by adding redundant arcs so that $\mathcal{N}'$ has the shape of concentric undirected cycles, where each cycle consists of all the peeling arcs of the same level. A redundant arc is a new arc $a'$ joining the middle of two existing arcs, $a''$ and $a'''$, such $c(a') = 0$.

The result of this pre-processing, which can be carried out in linear time without destroying the planarity of $\mathcal{N}'$ and without increasing its outerplanarity, is the conclusion of Lemma 29 in [3]. Although understanding and

\(^7\)If there is a need for it, redrawing a 3-regular planar graph on a rectangular grid can always be done in linear time (measured by the number of times a node is visited and its position in the plane is changed) [4].
reasoning about the reassembling of a planar network in general is far easier if it is in the shape of concentric (undirected) cycles, we dispense with this pre-processing in this report, as our running examples are simple enough to understand without it.

If we directly apply Algorithm 4 in Appendix C of report [3] to network $N'_1$ of Figure 11, we essentially obtain the binding schedule shown in Figure 12. (The difference here is that we also skipped what we called the second iteration in Algorithm 4, which does not affect the optimality of the final binding schedule returned by the algorithm.)
There are other approaches to reassembling networks, also producing optimal binding schedules that are generally different from the one returned by Algorithm 4. The approach in [3] starts by binding as many of the cross arcs as possible so that all resulting subnetworks have \( \leq 2 \) nodes (middle top row in Figure 12). It then follows by binding as many peeling arcs as needed to make every node, other than some (if any) of the source nodes and sink nodes, part of a 2-node subnetwork (right top row in Figure 12). It then continues by binding all remaining arcs (second, third, and fourth rows in Figure 12) until the network is fully reassembled. This process is carried out in time \( O(n') \), where \( n' \) is the number of nodes in the transformed network \( N' \) according to Lemma 4 and where the hidden constants do not depend on the outerplanarity \( k' \). The parameter \( k' \) appears in the hidden constants of the asymptotic complexity only when we compute principal typings of subnetworks.

Another more complicated example, network \( N_2 \), is shown in Figure 13. This network was considered in Appendix C of [3] already, with the difference here being that we have limited the network to one input arc and one output arc. The transformation of \( N_2 \) into an equivalent 3-regular network \( N'_2 \) does not increase the outerplanarity, which is 3 in this case. Figures 8 and 9 in Appendix C of [3] show how Algorithm 4 determines an optimal binding schedule \( \sigma'_2 \) for network \( N'_2 \) such that \( \text{index}(\sigma'_2) = 6 \).
7 Computing Principal Typings (Method 2)

In a first stage, we compute the principal typings of all one-node subnetworks. In a second stage, we reassemble the former to produce subnetworks each with \( \leq 2 \) nodes and with external dimension \( \leq 4 \) (see the rightmost partially reassembled networks in the top row of Figures 4 and 12, and in the bottom row of Figure 13).

In a third stage of the reassembling, the trunk part is the subnetwork whose size increases every time it is merged with an additional subnetwork of size \( \leq 2 \) and external dimension \( \leq 4 \). The reassembling proceeds in a way such that the external dimension of the trunk is independent of the number \( n \) of nodes: In Figure 4, the external dimension of the trunk is \( \leq 3 \); in Figure 12, the external dimension of the trunk is \( \leq 4 \); and in Figures 8 and 9 in the earlier report [3] (corresponding to the network in Figure 13 where we kept only one input arc and one output arc), the external dimension of the trunk is \( \leq 6 \). We thus need to concern ourselves with computing, in succession:

- the principal typings of \( n \) one-node subnetworks, then
- from the preceding, the principal typings of at most \( \lceil n/2 \rceil \) two-node subnetworks, and then
- from the preceding, the principal typing of a trunk which is updated at most \( \lceil n/2 \rceil \) times.

We state the crucial lemma on which Method 2 is based. The negation of an interval/type \([r, s]\) where \( r \leq s \) is the interval/type \([-s, -r]\), which we also denote \(-[r, s]\).

**Lemma 5.** Let \( T : \mathcal{P}(A_{in, out}) \to \mathcal{I}(\mathbb{R}) \) be the principal typing of a network \( N \) whose set of input/output arcs is \( A_{in, out} \). We then have, for every two-part partition of \( A_{in, out} \), say \( A \cup B = A_{in, out} \):

1. If both \( T(A) \) and \( T(B) \) are defined, then \( T(A) = -T(B) \). In particular, if \( A = \emptyset \) and \( B = A_{in, out} \), then \( T(\emptyset) = T(A_{in, out}) = [0, 0] \).
2. If \( T(A) \) is defined and \( T(B) \) is undefined, then \( T(A) = [-\max \theta_B, -\min \theta_B] = [-\min \theta_B, \max \theta_B] \), where the objective function \( \theta_B := \sum (B \cap A_{in}) - \sum (B \cap A_{out}) \) is minimized and maximized, respectively, w.r.t. the set of linear constraints induced by \( T \).

The two parts of Lemma 5 here are Lemmas 12 and 13 in [3]. In estimating the cost, we count addition, subtraction, and comparison of two numbers. In contrast to standard implementations of LP procedures, we can carry out the optimization without involving any multiplication, division, or any other arithmetical operation.

**One-node subnetworks.** Consider typing \( U_1 \) in Section 5. It makes the following type assignments:

\[
\begin{align*}
    x_4^+ & : 0, \min \{e(x_4), e(x_7)\} \quad & = [0, 10] \quad i.e. \ U_1(\{x_4^+\}) = [0, 10] \\
    x_6^+ & : 0, \min \{e(x_6), e(x_7)\} \quad & = [0, 5] \quad i.e. \ U_1(\{x_6^+\}) = [0, 5] \\
    -x_7^- & : -\min \{e(x_7), e(x_4) + e(x_6)\}, 0 \quad & = [-10, 0] \quad i.e. \ U_1(\{x_7^-\}) = [-10, 0]
\end{align*}
\]

We have framed the arithmetical operations used so far. There is no need for any other operation, because:

1. \( U_1(\emptyset) = U_1(\{x_4^+, x_6^+, x_7^-\}) = [0, 0] = \{0\} \), and
2. By Lemma 5,
   \[
   U_1(\{x_4^+, x_6^+\}) = -U_1(\{x_7^-\}) = [0, 10], \quad U_1(\{x_4^+, x_7^-\}) = -U_1(\{x_6^+\}) = [-5, 0],
   U_1(\{x_6^+, x_7^-\}) = -U_1(\{x_4^+\}) = [-10, 0].
   \]

We have carried out the computation for the principal typing of the subnetwork in Figure 5, but the number of additions (one) and comparisons (three) is the same for all one-node subnetworks. For a \( n \)-node network:

---

The wording of the two lemmas in [3] is slightly different, but equivalent.
• The cost of computing the principal typings of one-node subnetworks, each with two input arcs and one output arc (or one input arc and two output arcs), is $4 \cdot n$ operations (addition and comparison only).

In computing the principal typings of subsequent subnetworks, we frame all already-used arithmetical operations in the determination of the principal typings of one-node subnetworks: These arithmetical operations need not be carried out again, and their results can be stored in a pre-processing phase.

**Two-node subnetworks.** In a 3-regular directed network, we ignore two-node subnetworks of the form shown in Figure 14, where the directions of the 4 arcs $\{y_1, y_2, y_3, y_4\}$ are omitted. Regardless of the directions of these arcs, either no flow transits through the subnetwork (when $y_1$ and $y_2$ are directed both outward or both inward), or one of the two arcs $\{y_3, y_4\}$ can be deleted (when $y_1$ is directed inward and $y_2$ outward, or vice-versa).

Hence, the only relevant cases of a two-node subnetwork are when the external dimension is 4. The four possible configurations that we need to consider are shown in Figure 15. An example of Configuration A (leftmost in Figure 15) is the subnetwork shown in Figure 6, whose principal typing $U_2$ can be computed as shown in Figure 16. By Lemma 5, the 7 type assignments shown in Figure 16 fully specify the typing $U_2$. We thus need 4 unframed additions, 1 unframed subtraction, and 5 unframed comparisons to determine $U_2$. 

![Figure 15: Four relevant configurations of two-node subnetworks.](image)

![Figure 16: Computing the type assignments of principal typing $U_2$. Framed operations are carried out previously.](image)
Similarly, the principal typing of a two-node subnetwork of the form Configuration B or Configuration C can be determined using 3 unframed additions and 6 unframed comparisons. The principal typing of a two-node subnetwork of the form Configuration D can be determined using 9 unframed comparisons. In all cases,

- The cost of computing the principal typing of a relevant two-node subnetwork of external dimension 4 is at most 10 unframed arithmetical operations (addition, subtraction, comparison).
- The cost of computing the principal typings of relevant two-node subnetworks of external dimension 4 in a n-node network is at most $5 \cdot n$ unframed arithmetical operations (addition, subtraction, comparison).

**Merging the trunk with subnetworks of external dimension $\leq 4$.** The initial trunk is a one-node or two-node subnetwork of external dimension 3 (e.g., in the case of networks $N_0$ in Figure 4, $N_\infty$ in Figure 9, and $N_1'$ in Figure 13) or external dimension 4 (e.g., in the case of $N_1'$ in Figure 12).

Suppose the entire network is reassembled according to a binding schedule $\sigma$ such that $\text{index}(\sigma) = k \geq 2$. At an intermediate step of the algorithm, the external dimension of the trunk may be as large as $k$ without invoking a pre-defined LP procedure this time. A very simple example of merging a trunk (that in Figure 7) with a two-node subnetwork (that in Figure 6) is carried out next, to illustrate the general methodology according to Method 2, thus obtaining typing $\sigma$ once more, but without invoking a pre-defined LP procedure this time.

We can set $U_4(\emptyset) := [0,0]$ and $U_4(\{a_1,x_7,x_8\}) := [0,0]$ right off. As for the non-trivial subsets of $\{a_1,x_7,x_8\}$, we simultaneously compute a type and its negation for each pair of disjoint subsets $(A,B)$ such that $A \cap B = \emptyset$.

For illustrative purposes, we do this only for the pair $A = \{a_1,x_7\}$ and $B = \{x_8\}$, with the other pairs being handled in the same way. The computation of a type for $\{a_1,x_7\}$ can be schematically represented by the tree in Figure 17, where the types are not yet inserted; the computation proceeds according to the instructions of Algorithm 3 (called CompPT) and its subroutine Algorithm 2 (called BindPT) in [3].

For later reference, call a tree such as in Figure 17 an execution tree to compute a tight type (specifically here, the type of the subset $\{a_1,x_7\}$ in the subnetwork with input/output arcs $\{a_1,x_7,x_8\}$). Along the leftmost path, from the leftmost leaf-node moving upward to the root node:

1. We first compute the initial type of $\{a_1,x_7\}$, i.e., before we do any arc binding.
2. We then narrow the interval/type of $\{a_1,x_7\}$ as a result of binding $\{x_4^-,x_8^+\}$, but not yet $\{x_5^-,x_5^+\}$. This requires knowing the initial type of $\{x_5^-,x_5^+\}$, before any arc binding. The set $\{x_5^-,x_5^+,x_8\}$ is what we call the complement of $\{a_1,x_7\}$ after binding $\{x_4^-,x_8^+\}$, but not yet $\{x_5^-,x_5^+\}$.
3. We then narrow the type of $\{a_1,x_7\}$ a second time, as a result of binding $\{x_5^-,x_5^+\}$. This requires knowing the type of $\{x_5\}$ after binding $\{x_4^-,x_8^+\}$ but not yet $\{x_5^-,x_5^+\}$. The set $\{x_5\}$ is the complement of $\{a_1,x_7\}$ after binding both $\{x_4^-,x_8^+\}$ and $\{x_5^-,x_5^+\}$.

After binding $\{x_4^-,x_8^+\}$ and $\{x_5^-,x_5^+\}$;

After binding $\{x_4^-,x_8^+\}$:

Initial type of $\{a_1,x_7\}$:

![Figure 17: Execution tree to compute the tight type of $\{a_1,x_7\}$ before inserting the types.](image-url)
The initial type $U_4(\{a_1, x_7\})$, i.e., at the leftmost leaf-node in Figure 17, is computed by setting:

$$U_4(\{a_1, x_7\}) := U_3(\{a_1\}) + U_2(\{x_7\}) = [0, 15] + [-10, 0] = [-10, 15],$$

where we can add the types $U_3(\{a_1\})$ and $U_2(\{x_7\})$ to obtain the type $U_4(\{a_1, x_7\})$, because there is not yet any communication between $U_2$ and $U_3$. The type $U_4(\{a_1, x_7\}) = [-10, 15]$ is shown as “$a_1 \cdot x_7 \in [-10, 15]$” at the leftmost leaf-node in Figure 18, following our notational conventions.

To determine the type of $\{a_1, x_7\}$ after we bind $\{x_4, x_5\}$, we invoke Lemma 5, which requires that we first compute the type of the complement set $\{x_5, x_5^+, x_8\}$ prior to any binding, which is:

$$U_4(\{x_5, x_5^+, x_8\}) := U_3(\{x_5\}) + U_2(\{x_5^+, x_8\}) = [-8, 0] + [0, 5] = [-8, 5],$$

which is at the second leaf-node from the left in Figure 18 and where we again exploit the fact that there is no communication between $U_2$ and $U_3$ yet. The type of $\{a_1, x_7\}$ after binding $\{x_4, x_5\}$, but not yet $\{x_5, x_8\}$, is narrowed as follows:

$$U_4(\{a_1, x_7\}) := U_4(\{a_1, x_7\}) \cap U_4(\{x_5, x_5^+, x_8\}) = [-10, 15] \cap [-8, 5] = [-5, 8],$$

where in accordance with Lemma 5, we have to take the negation of the interval/type $U_4(\{x_5, x_5^+, x_8\})$ before we intersect it with the interval/type $U_4(\{a_1, x_7\})$. This first updated type $U_4(\{a_1, x_7\})$ is shown at the parent node of the leftmost leaf-node in Figure 18.

In the same fashion, we compute the types of all the other nodes in the execution tree. When we insert all the types, we obtain the tree in Figure 18. The last updated type $U_4(\{a_1, x_7\})$ is at the root node in Figure 18.

In this example, the merging of the trunk part (the subnetwork in Figure 7) whose principal typing is $U_3$ with the 2-node subnetwork (in Figure 6) whose principal typing is $U_2$, involves two bindings: $\{x_4, x_5\}$ and $\{x_5, x_5^+, x_8\}$. The execution trees in this case, one for every non-trivial pair of complements, i.e., every pair $(A, B)$ such that $A \neq \emptyset \neq B$ and $A \cup B = \{a_1, x_7, x_8\}$, are each a full binary tree of height 2. There are 3 non-trivial pairs $(A, B)$ of $\{a_1, x_7, x_8\}$, and therefore 3 execution trees are needed to fully specify $U_4$, the principal typing of the trunk in Figure 8.

In general, the merging of a trunk $\mathcal{M}$ of external dimension $d \geq 2$, which is always with a subnetwork $\mathcal{M}'$ of size $\leq 2$ and external dimension $d' = 3$ (if $\mathcal{M}'$ has one node) or external dimension $d' = 4$ (if $\mathcal{M}'$ has two nodes), involves $\leq 3$ bindings:

- If the merging involves 1 binding, we need to carry out $(2^{d+d'-3} - 1)$ execution trees, each of height $= 1$. 
- If the merging involves 2 bindings, we need to carry out $(2^{d+d'-5} - 1)$ execution trees, each of height $= 2$. 
- If the merging involves 3 bindings, we need to carry out $(2^{d+d'-7} - 1)$ execution trees, each of height $= 3$.

These numbers, in the three preceding cases, are justified as follows. The merging of $\mathcal{M}$ and $\mathcal{M}'$ produces a larger trunk, which we denote $\mathcal{M} \oplus \mathcal{M}'$ as in the earlier report [3], of external dimension:

$$d + d' - 2 \quad \text{or} \quad d + d' - 4 \quad \text{or} \quad d + d' - 6,$$

respectively. Hence, the number of non-trivial subsets of input/output arcs in $\mathcal{M} \oplus \mathcal{M}'$ is:

$$2^{d+d'-2} - 2 \quad \text{or} \quad 2^{d+d'-4} - 2 \quad \text{or} \quad 2^{d+d'-6} - 2,$$

respectively, and the number of non-trivial pairs of input/output complements in $\mathcal{M} \oplus \mathcal{M}'$ is:

$$2^{d+d'-3} - 1 \quad \text{or} \quad 2^{d+d'-5} - 1 \quad \text{or} \quad 2^{d+d'-7} - 1,$$

respectively. An arithmetical operation in an execution tree is an addition (at leaf-nodes) or a comparison (at non-leaf nodes). Depending on whether $\mathcal{M} \oplus \mathcal{M}'$ involves one or two or three bindings, we thus need at most:
The upper bound $3 \cdot (2^k - 1)$ is not tight, for two reasons. First, it assumes that the external dimension of the trunk can be as large as $k$ at every stage of the algorithm, whereas this external dimension in fact starts at 3 or 4, reaches $k$ at intermediate stages, and gradually decreases down to 2 at the last stage (when there is one input arc and one output arc in the entire network). And, second, the upper bound “$3 \cdot (2^k - 1)$” is for the case involving only one binding when updating the trunk from $M$ to $M@M'$, whereas “most” (not quantified here) updatings of the trunk involve two or three bindings.

The correctness of Method 2 is established in full generality by Lemmas 17, 19, and 20 in [3]. We point out several of its salient features:

- The only numbers that occur during execution are arc capacities and numbers obtained by adding and subtracting together arc capacities.

- There are no restrictions on the numbers themselves.

- If arc capacities are integers (resp. rational numbers, resp. real numbers), then the two limits of every interval/type in a principal typing are integers (resp. rational numbers, resp. real numbers).
8 Concluding Remarks

Both Method 1 in Section 5 and Method 2 in Section 7 can directly deal with two extensions, with minor adjustments and no increase in the run-time complexity (measured by counting arithmetical operations):

(a) the presence of multiple sources and multiple sinks, and/or
(b) the presence of non-zero lower-bound capacities, in addition to upper-bound capacities.

The number of multiple sources and multiple sinks in extension (a) becomes part of index(σ), when the network is reassembled according to a binding schedule σ. The lower bounds in extension (b) are directly included in all the steps of the algorithm involving arithmetical operations. For networks with both extensions, (a) and (b), Method 2 is used in the earlier report [3], but not Method 1. Although easier to present and understand, Method 1 relies on the use of a pre-defined LP procedure, which we deliberately want to avoid in the implementation of our algorithms.

Originally designed to support a framework for system modeling and system analysis in the large, our compositional approach turned out to be flexible enough and adaptable to other situations of design and analysis, as illustrated in this report and its precursor [3]. As long as an appropriate composable (or syntax-directed) notion of a principal typing can be defined, which formally encodes desirable invariant properties at the boundary of a component of which it is a typing, our compositional approach can be extended to handle more complicated flow networks that are constrained by other measures simultaneously – in addition to lower bounds and upper bounds on arc capacities. Some of these extensions are listed in Section 6 (“Future Work”) of the earlier report [3], which are the object of current work, starting with the case of multi-commodity flow networks.

References


A Appendix: Three Anonymous Reviews

Below are the three anonymous reviews – A, B, and C – of the earlier report [3], which can be downloaded from here and here, to match against what A, B, and C say. The entire report was submitted to STOC 2014; the first 10 pages were the extended abstract, the remaining pages were optional.

Review A: The paper presents an idea of how to compute maximum flow in networks using divide-and-conquer approach. Consider any s-t separating set of edges E in the graph, then the idea is to compute all possible flows from s to E then all possible flow from E to t. Then combine then to get the maximum flow from s to t. If it was possible to find a set E of constant size then one could hope to get an algorithm running in $O(n)$ time. However, there is a problems with this approach. The projection of the flow into constant number of edges does not need to have a constant description. Hence, the idea presented in the paper seems to be not working.
Review B: The paper presents a typing theory (as known in PL theory) to network flow problems and applies the presented framework to maximum flow with multiple sources and multiple sinks. The claimed general theory is really only presented as applied to flow problems, and here I believe there are several problems. Essentially, the author is suggesting to solve max flow by divide & conquer(dynamic programming over a decomposition of the graph). The subproblems, however, only seem to consider solutions in which the edges incident to a piece in the decomposition are saturated in one direction. This cannot lead to optimal solutions in which fractional flow assignments are needed. The application to maximum flow in $k$-outerplanar graphs results in an algorithm that is worse than already-known algorithms. The author’s algorithm is $O(n)$ where the constant depends exponentially on $k$ (and the dependence on the number of sources is not explicit). The algorithm due to Johnson & Venkatesan (FOCS ’83) or Borradaile & Harutyunyan (IWOCA ’13) result in $O(kn)$-time algorithms. By enumerating over the source-sink pairs according to, e.g., Borradaile, Klein, Mozes, Nussbaum, Wilff-Nilsen (FOCS ’11), this gives an $O(t^2kn)$ time algorithm where $t$ is the number of sources and sinks. The author should note that a linear-time algorithm for unit-capacity max-flow in planar graphs was given by Brandeis & Wagner predates that of Eisenstat & Klein.

Since the paper’s proposed framework is not fully fleshed out and since the application does not result in novel results, the paper should not be accepted.

Review C: The main part of the paper is introducing typing theory for network, defining principal typings for flows and showing how networks can be assembled and disassembled under the defn. The motivation for disassembling and assembling is to provide a compositional approach to analyze networks incrementally and handle components that are supplied over time. (The principle typings can be computed efficiently, e.g. via linear programming though the paper does it differently, namely in a divide-and-conquer manner.) This paper assumes all components are given initially. Therefore, it can be disassembled into the smallest units (single node), compute their principal typing and finally combine to produce the typing for the entire network. This section of the paper really feels like a simple divide-and-conquer approach. It is unclear to me how novel the paper is up till here for people who are familiar with typing theory. The last section of the paper is an application to the multi-src multi-sink max/min flow problem in planar networks with capacities. Its central claim is that the principle typing of a planar graph takes linear time. As a result the flow problem can be solved in linear time as well. The best previous running time had an extra polylog factor.

Unfortunately, the paper has failed to interest me. I’m not sure if typing theory itself would have a broad appeal to the STOC audience, although the application to multi-commodity flow in capacitated planar graphs is interesting. If the previous work along this line is any indication e.g. [4,5,6,32,33] the work is more suitable for specialized conferences such as programming language conferences.