Formalizing Network Flow Algorithms: A Refinement Approach in Isabelle/HOL

DOI:
10.1007/s10817-017-9442-4

Document Version
Accepted author manuscript

Noname manuscript No.
(will be inserted by the editor)

Formalizing Network Flow Algorithms
A Refinement Approach in Isabelle/HOL

Peter Lammich · S. Reza Sefidgar

Received: date / Accepted: date

Abstract We present a formalization of classical algorithms for computing the maximum flow in a network: The Edmonds-Karp algorithm and the push-relabel algorithm. We prove correctness and time complexity of these algorithms. Our formal proof closely follows a standard textbook proof, and is accessible even without being an expert in Isabelle/HOL — the interactive theorem prover used for the formalization.

Using stepwise refinement techniques, we instantiate the generic Ford-Fulkerson algorithm to Edmonds-Karp algorithm, and the generic push-relabel algorithm of Goldberg and Tarjan to both the relabel-to-front and the FIFO push-relabel algorithm. Further refinement then yields verified efficient implementations of the algorithms, which compare well to unverified reference implementations.

Keywords Maximum Flow Problem · Edmonds-Karp Algorithm · Push-Relabel Algorithm · Formal Verification · Isabelle/HOL · Stepwise Refinement

1 Introduction

Computing the maximum flow in a network is an important problem in graph theory. Many other problems, like maximum bipartite matching, edge disjoint paths, circulation demand, as well as various scheduling and resource allocating problems can be reduced to it. The generic Ford-Fulkerson algorithm [14] describes a class of algorithms to solve the maximum flow problem. The basic idea is to repeatedly search for *augmenting paths* along which more flow can be moved from the source to the sink. An important instance is the Edmonds-Karp algorithm [12], which was one of the first algorithms to solve the maximum flow problem in polynomial time ($O(VE^2)$) for the general case of networks with real-valued capacities. Dinitz [11]

Peter Lammich
Institut für Informatik, Technische Universität München, Germany
E-mail: lammich@in.tum.de

S. Reza Sefidgar
Institute of Information Security, Department of Computer Science, ETH Zurich, Switzerland
E-mail: reza.sefidgar@inf.ethz.ch

generalized the idea of augmenting paths to blocking flows, obtaining an $O(V^2E)$ algorithm. Karzanov [22] was the first to propose an $O(V^3)$ algorithm based on the idea of *preflows*. Based on Karzanov's ideas, Goldberg and Tarjan proposed the generic push-relabel algorithm [16]. Implementations of the push-relabel algorithm are among the most efficient maximum flow algorithms. While the generic algorithm has a time complexity of $O(V^2E)$, specific variants of the algorithm achieve even lower complexities down to $O(V^2\sqrt{E})$.

In this paper, we present a formal verification of the Edmonds-Karp algorithm and of the generic push-relabel algorithm. We prove correctness of the algorithms as well as the time complexity bounds of $O(VE^2)$ and $O(V^2E)$, respectively. The formalization is conducted in the Isabelle/HOL proof assistant [39]. Stepwise refinement techniques [46, 1, 2] allow us to elegantly structure our verification to separate the abstract algorithmic ideas from the implementation details: After proving the min-cut-max-flow theorem, it is straightforward to verify a generic version of the Ford-Fulkerson algorithm, which we then refine to Edmonds-Karp algorithm, and finally to an efficient implementation. Similarly, the generic push-relabel algorithm is refined to both the relabel-to-front algorithm [10] and the FIFO push-relabel algorithm [16], which, in turn, are refined to efficient implementations. The efficiency of our verified implementations compares well to unverified reference implementations of the algorithms.

The abstract parts of our verification closely follow the textbook presentation of Cormen et al. [10]. Using the Isar [45] proof language, we were able to produce proofs that are accessible even to non-Isabelle experts.

This paper is an extended version of our conference paper [29], which only covered the Edmonds-Karp algorithm. While there exists another formalization of the Ford-Fulkerson algorithm in Mizar [34][1], we are, to the best of our knowledge, the first that verify a polynomial time maximum flow algorithm, prove the polynomial complexity bound, or provide a verified executable implementation. Moreover, this paper is a case study on elegantly formalizing algorithms.

In the rest of this paper, we give a short informal description of the Ford-Fulkerson algorithm (§2) and present our formalization of it (§3). After introducing the Isabelle Refinement Framework [33, 24] (§4), we report on the instantiation of the Ford-Fulkerson algorithm to the Edmonds-Karp algorithm, and on its complexity proof (§5). Next, we describe the correctness and complexity proof of the generic push-relabel algorithm (§6), and its instantiations to the relabel-to-front and FIFO push-relabel algorithms (§7). Finally, we report on the refinement of all three algorithms to efficient implementations (§8), and their comparison to unverified reference implementations (§9).

The complete formalization is available as a set of three entries in the Archive of Formal Proofs [30, 31, 32].

2 The Ford-Fulkerson Algorithm

In this section, we give a short introduction to the Ford-Fulkerson algorithm, closely following the presentation by Cormen et al. [10].

[1] Section 10.1 provides a detailed discussion

A (flow) network is a directed graph over a finite set of nodes V and edges E, where each edge $(u, v) \in E$ is labeled by a positive real-valued capacity $c(u, v) > 0$. Moreover, there are two distinct nodes $s, t \in V$, which are called *source* and *sink*.

A *flow* f on a network is a labeling of the edges with real values satisfying the following constraints: 1) *Capacity constraint*: the flow on each edge is a non-negative value smaller or equal to the edge's capacity; 2) *Conservation constraint*: For all nodes except s and t, the sum of incoming flows equals the sum of outgoing flows. The value $|f|$ of a flow f is defined to be the sum over the outgoing flows of s minus the sum over the incoming flows of s. Given a network G, the maximum flow problem is to find a flow with the maximum value among all flows of the network.

To simplify reasoning about the maximum flow problem, we assume that our network satisfies some additional constraints: 1) the source only has outgoing edges while the sink only has incoming edges; 2) if the network contains an edge (u, v) then there is no *parallel edge* (v, u) in the reverse direction[2]; and 3) every node of the network must be on a path from s to t. Any network can be transformed in linear time to fulfill these properties, preserving the maximum flow. However, we did not formalize such a transformation.

An important result is the relation between flows and cuts in a network. A *cut* is a partitioning of the nodes into two sets, such that one set contains the source and the other set contains the sink. The capacity of a cut is the sum of the capacities of all edges going from the source's side to the sink's side of the cut. It is easy to see that the value of any flow cannot exceed the capacity of any cut, as all flow from the source must ultimately reach the sink, and thus go through the edges of the cut. The min-cut-max-flow theorem tightens this bound and states that the value of the maximum flow is equal to the capacity of the minimum cut.

The Ford-Fulkerson algorithm is a corollary of this theorem. It is based on a greedy approach: Starting from a zero flow, the value of the flow is iteratively increased until a maximum flow is reached. In order to increase the overall flow value, it may be necessary to redirect some flow, i. e. to decrease the flow passed through specific edges. For this purpose the Ford-Fulkerson algorithm defines the residual graph, which has edges in the same and opposite direction as the network edges. Each edge is labeled by the amount of flow that can be effectively passed along this edge, by either increasing or decreasing the flow on a network edge. Formally, the residual graph G_f of a flow f is the graph induced by the edges with positive labels according to the following labeling function c_f:

$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & \text{if } (u, v) \in E \\ f(v, u) & \text{if } (v, u) \in E \\ 0 & \text{otherwise} \end{cases}$$

In each iteration, the algorithm tries to find an *augmenting path*, i. e. a simple path from s to t in the residual graph. It then pushes as much flow as possible along this path, increasing the current flow. Formally, for an augmenting path p, one first defines the *residual capacity* c_p as the minimum value over all edges of p:

$$c_f(p) = \min\{c_f(u, v) : (u, v) \text{ is on } p\}$$

[2] With $u = v$, this also implies that there are no self loops.

An augmenting path then yields a residual flow f_p, which is the flow that can be passed along this path:

$$f_p(u,v) = \begin{cases} c_f(p) & \text{if } (u,v) \text{ is on } p \\ 0 & \text{otherwise} \end{cases}$$

Finally, to actually push the flow induced by an augmenting path, we define the *augment* function $f{\uparrow}f'$, which augments a flow f in the network by any *augmenting flow* f', i. e. any flow in the residual graph:

$$(f{\uparrow}f')(u,v) = \begin{cases} f(u,v) + f'(u,v) - f'(v,u) & \text{if } (u,v) \in E \\ 0 & \text{otherwise} \end{cases}$$

Note that, for any edge in the network, the augmenting flow in the same direction is added to the flow, while the augmenting flow in the opposite direction is subtracted. This matches the intuition of passing flow in the indicated direction, by either increasing or decreasing the flow of an edge in the network.

The correctness of the Ford-Fulkerson algorithm follows from the min-cut-max-flow theorem, which states that the following three propositions are equivalent:

1. f is a maximum flow in a network G.
2. there is no augmenting path in the residual graph G_f.
3. there is a cut C in G such that the capacity of C is equal to the value of f.

The Ford-Fulkerson algorithm is generic in that it does not specify how to find an augmenting path in the residual graph. There are several possible implementations with different execution times. The generic algorithm is only guaranteed to terminate for networks with rational capacities, while it may infinitely converge against non-maximal flows in the case of irrational edge capacities [14, 47]. When always choosing a *shortest*[3] augmenting path, the number of iterations is bounded by $O(VE)$, even for the general case of real-valued capacities. Note that we write V and E instead of $|V|$ and $|E|$ for the number of nodes and edges if the intended meaning is clear from the context. A shortest path can be found by breadth first search (BFS) in time $O(E)$, yielding the Edmonds-Karp algorithm [12] with a time complexity of $O(VE^2)$.

3 Formalizing the Ford-Fulkerson Algorithm

In this section, we provide a brief overview of our formalization of the Ford-Fulkerson algorithm. In order to develop theory in the context of a fixed graph or network, we use Isabelle's concept of *locales* [3], which allows us to define named contexts that fix some parameters and assumptions. For example, the graph theory is developed in the locale `Graph`, which fixes the edge labeling function `c`, and defines the set of edges and nodes based on `c`:

```
locale Graph = fixes c :: edge ⇒ capacity begin
  definition E ≡ {(u, v). c (u, v) ≠ 0}
  definition V ≡ {u. ∃v. (u, v) ∈ E ∨ (v, u) ∈ E}
  [...]
```

[3] i. e. a path with a minimum number of edges.

We use Isabelle/HOL's non-standard set-comprehension syntax $\{(x_1, \ldots, x_n).\ P\}$ for the set of all tuples $(x_1, \ldots, x_n)$ satisfying P. The locale also provides basic concepts like (simple, shortest) paths and lemmas to reason about them.

Networks extend graphs by adding the source and sink nodes, as well as the network assumptions:

```
locale Network = Graph + fixes s t :: node
  assumes no_incoming_s: ∀u. (u, s) ∉ E
  [...]
```

Most theorems presented in this paper are in the context of the `Network` locale.

3.1 Presentation of Proofs

Informal proofs focus on the relevant thoughts by leaving out technical details and obvious steps. In contrast, a formal proof has to precisely specify each step as the application of some inference rules. Although modern proof assistants provide high-level tactics to summarize some of these steps, formal proofs tend to be significantly more verbose than informal proofs. Moreover, many formal proofs are conducted in *applicative* style, where a proof is merely a program that instructs the proof assistant how to transform the theorem until it is proved. Those proofs tend to be inaccessible without a deep knowledge of the used proof assistant, often requiring a replay of the proof in the proof assistant. In our proofs, we mainly used the *declarative* proof language Isar [45], which allows to write formal proofs that resemble standard mathematical textbook proofs, and are accessible, to a certain extent, even for those not familiar with Isabelle/HOL.

As an example, consider the proof that for a flow f and a residual flow f', the augmented flow $f{\uparrow}f'$ is again a valid flow. In particular, one has to show that the augmented flow satisfies the capacity constraint. Cormen et al. give the following proof, which we display literally here, only replacing the references to "Equation 26.4" by "definition of $\uparrow$":

For the capacity constraint, first observe that if $(u, v) \in E$, then $c_f(v, u) = f(u, v)$. Therefore, we have $f'(v, u) \leq c_f(v, u) = f(u, v)$, and hence

$$\begin{aligned}
(f \uparrow f')(u, v) &= f(u, v) + f'(u, v) - f'(v, u) && \text{(definition of } \uparrow) \\
&\geq f(u, v) + f'(u, v) - f(u, v) && \text{(because } f'(v, u) \leq f(u, v)) \\
&= f'(u, v) \\
&\geq 0.
\end{aligned}$$

In addition,

$$\begin{aligned}
(f \uparrow f')(u, v) &= f(u, v) + f'(u, v) - f'(v, u) && \text{(definition of } \uparrow) \\
&\leq f(u, v) + f'(u, v) && \text{(because flows are nonnegative)} \\
&\leq f(u, v) + c_f(u, v) && \text{(capacity constraint)} \\
&= f(u, v) + c(u, v) - f(u, v) && \text{(definition of } c_f) \\
&= c(u, v).
\end{aligned}$$

In the following we present the corresponding formal proof in Isar:

```
lemma augment_flow_presv_cap:
  shows 0 ≤ (f↑f')(u,v) ∧ (f↑f')(u,v) ≤ c(u,v)
proof (cases (u,v)∈E; rule conjI)
  assume [simp]: (u,v)∈E
  hence f(u,v) = cf(v,u)
    using no_parallel_edge by (auto simp: residualGraph_def)
  also have cf(v,u) ≥ f'(v,u) using f'.capacity_const by auto
  finally have f'(v,u) ≤ f(u,v) .

  have (f↑f')(u,v) = f(u,v) + f'(u,v) - f'(v,u)
    by (auto simp: augment_def)
  also have ... ≥ f(u,v) + f'(u,v) - f(u,v)
    using ‹f'(v,u) ≤ f(u,v)› by auto
  also have ... = f'(u,v) by auto
  also have ... ≥ 0 using f'.capacity_const by auto
  finally show (f↑f')(u,v) ≥ 0 .

  have (f↑f')(u,v) = f(u,v) + f'(u,v) - f'(v,u)
    by (auto simp: augment_def)
  also have ... ≤ f(u,v) + f'(u,v) using f'.capacity_const by auto
  also have ... ≤ f(u,v) + cf(u,v) using f'.capacity_const by auto
  also have ... = f(u,v) + c(u,v) - f(u,v)
    by (auto simp: residualGraph_def)
  also have ... = c(u,v) by auto
  finally show (f↑f')(u, v) ≤ c(u, v) .
qed (auto simp: augment_def cap_positive)
```

The structure of the Isar proof is exactly the same as that of the textbook proof, except that we had to also consider the case $(u, v) \notin E$, which is not mentioned in the informal proof at all, and easily discharged in our formal proof by the *auto*-tactic after the **qed**. We also use exactly the same justifications as the original proof, except that we had to use the fact that there are no parallel edges to show $c_f(v, u) = f(u, v)$, which is not mentioned in the original proof.

3.2 Presentation of Algorithms

In textbooks, it is common to present algorithms in pseudocode, which captures the essential ideas, but leaves open implementation details. As a formal equivalent to pseudocode, we use the monadic programming language provided by the Isabelle Refinement Framework [33, 24]. For example, we define the Ford-Fulkerson algorithm as follows:

```
definition ford_fulkerson_method ≡ do {
  let f = (λ(u,v). 0);

  (f,brk) ← while (λ(f,brk). ¬brk)
    (λ(f,brk). do {
      p ← selectp p. is_augmenting_path f p;
      case p of
        None ⇒ return (f,True)
      | Some p ⇒ return (augment c f p, False)
    })
    (f,False);
  return f
}
```

The code looks quite similar to pseudocode that one would expect in a textbook, but actually is a rigorous formal specification of the algorithm, using nondeterminism to leave open the implementation details (cf. Section 4). Note that we had to use the available combinators of the Isabelle Refinement Framework, which made the code slightly more verbose than we would have liked. In particular we had to explicitly thread a `brk`-flag through the state of the while loop, emulating a `break` statement. We leave it to future work to extend the Isabelle Refinement Framework by more advanced combinators and syntactic sugar to allow for a more concise presentation of abstract algorithms.

Finally, using the Ford-Fulkerson theorem and the verification condition generator of the Isabelle Refinement Framework, it is straightforward to prove (partial) correctness of the Ford-Fulkerson algorithm, which is stated in Isabelle/HOL by the following theorem:

```
theorem (in Network) ford_fulkerson_method ≤ (spec f. isMaxFlow f)
```

4 Refinement in Isabelle/HOL

After having stated and proved correct an algorithm on the abstract level, the next step is to provide an (efficient) implementation. In our case, we first specialize the Ford-Fulkerson algorithm to use shortest augmenting paths, then implement the search for shortest augmenting paths by BFS, and finally use efficient data structures to represent the abstract objects modified by the algorithm.

A natural way to achieve this formally is *stepwise refinement* [46], and in particular *refinement calculus* [1,2], which allows us to systematically transform an abstract algorithm into a more concrete one, preserving its correctness.

In Isabelle/HOL, stepwise refinement is supported by the Isabelle Refinement Framework [33,24]. It features a refinement calculus for programs phrased in a nondeterminism monad. The monad's type is a set of possible results plus an additional value that indicates a failure:

```
datatype α nres = res α set | fail
```

The operation `return x` of the monad describes the single result `x`, and the operation `bind m f` nondeterministically picks a result from `m` and executes `f` on it. The bind operation fails iff either `m = fail`, or `f` may fail for a result in `m`.

We define the *refinement ordering* on α `nres` by lifting the subset ordering with `fail` being the greatest element. Intuitively, $m \leq m'$ means that m is a refinement of m', i. e. all possible results of m are also possible results of m'. Note that the refinement ordering is a complete lattice, and bind is monotonic. Thus, we can define recursion using a fixed point construction [23]. Moreover, we can use the standard Isabelle/HOL constructs for if, let and case distinctions, yielding a fully fledged programming language, shallowly embedded into Isabelle/HOL's logic. For simpler usability, we define standard loop constructs (while, foreach), a syntax for postcondition specifications, and use a Haskell-like do-notation:

```
spec P ≡ spec x. P x ≡ res {x. P x}
do {x ← m; f x} ≡ bind m f
do {m; m'} ≡ bind m (λ_. m')
```

Correctness of a program `m` with precondition `P` and postcondition `Q` is expressed as $P \Longrightarrow m \leq$ `spec r. Q r` (or, eta contracted, just `spec Q`), which means that, if

`P` holds, `m` does not fail and all possible results of `m` satisfy `Q`. Note that we provide different recursion constructs for partial and total correctness: A nonterminating total correct recursion yields `fail`, which satisfies no specification, even if joined with results from other possible runs. On the other hand, a nonterminating partial correct recursion yields `res {}`, which refines any specification and disappears when joined with other results. For a more detailed explanation of partial and total correctness in the Isabelle Refinement Framework, we refer the reader to [33, §2.3].

The Isabelle Refinement Framework also supports data refinement. The representation of results can be changed according to a *refinement relation*, which relates concrete with abstract results: Given a relation `R`, $\Downarrow$`R m` is the set of concrete results that are related to an abstract result in `m` by `R`. If `m = fail`, then also $\Downarrow$`R m = fail`.

In a typical program development, one first designs an initial version m_0 of the algorithm and its specification `P,Q`, and shows $P \Longrightarrow m_0 \leq$ `spec Q`. Then, one iteratively provides refined versions m_i of the algorithm, proving $m_i \leq \Downarrow R_i\ m_{i-1}$. Using transitivity and composability of refinement, one gets $P \Longrightarrow m_i \leq \Downarrow R_i \ldots R_1$ `spec Q`, showing the correctness of the refined algorithm. If no data refinement is performed, R_i is set to the identity relation, in which case $\Downarrow R_i$ becomes the identity function.

Various tools, including a verification condition generator, assist the user in conducting the refinement proofs by breaking them down to statements that do not contain monad constructs any more. In many cases, these verification conditions precisely reflect the core idea of the proof.

Monotonicity of the standard combinators also allows for modular refinement: Replacing a part of a program by a refined version results in a program that refines the original program. This gives us a natural formal model for statements like "we implement shortest path finding by BFS", or "we use arrays to represent the edge labeling".

5 The Edmonds-Karp Algorithm

Specializing the Ford-Fulkerson algorithm to the Edmonds-Karp algorithm is straightforward, as finding a shortest augmenting path is a refinement of finding any augmenting path.

Considerably more effort is required to show that the resulting algorithm terminates within $O(VE)$ iterations. The idea of the proof is as follows: Edges in the opposite direction to an edge on a shortest path cannot lie on a shortest path itself. On every augmentation, at least one edge of the residual graph that lies on a shortest augmenting path is flipped. Thus, either the length of the shortest path increases, or the number of edges that lie on some shortest path decreases. As the length of a shortest path is at most V, there are no more than $O(VE)$ iterations.

Note that Cormen et al. present the same idea a bit differently: They define an edge of the residual graph being *critical* if it lies on a shortest path such that it will be flipped by augmentation. Then, they establish an upper bound of how often an edge can get critical during the algorithm. Our presentation is more suited for a formal proof, as we can directly construct a measure function from it, i. e. a function from flows to natural numbers, which decreases on every iteration and is bounded by $O(VE)$.

Formalizing the above intuitive argument was trickier than it seemed on first glance: While it is easy to prove that, in a *fixed graph*, an edge and its opposite cannot

both lie on shortest paths, generalizing the argument to a graph transformation which may add multiple flipped edges and removes at least one original edge requires some generalization of the statement. Note that a straightforward induction on the length of the augmenting path or on the number of flipped edges fails, as, after flipping the first edge, the path no longer exists.

Having defined the measure function and shown that it decreases on augmentation, it is straightforward to refine the partial correct while loop to a total correct one. We have also instrumented the loop to count its iterations, and asserted an upper bound of $2VE + V = O(VE)$ after the loop.

6 The Generic Push-Relabel Algorithm

We formalize the generic push relabel algorithm of Goldberg and Tarjan [16], closely following the textbook presentation of Cormen et al. [10].

Push-relabel algorithms are based on *preflows*, a generalization of flows by relaxing the conservation constraint to allow more incoming flow than outgoing flow. For a node $u \in V$, the difference between incoming and outgoing flow is called *excess* and denoted by $x_f(u)$. For a preflow, the excess of any node but the source is not negative. A non-sink node with positive excess is called *active*. A preflow without any active nodes is actually a flow.

The push-relabel algorithm maintains a preflow f and a height labeling $l : V \to \mathbb{N}$ of the nodes, with the invariant that for every edge (u, v) of the residual graph we have that $l(u) \leq l(v) + 1$. Moreover, the height of the source is fixed to $|V|$, and the height of the sink is fixed to 0.

The above invariant does not admit augmenting paths: By generalizing the invariant to paths, we get that for every path p from u to v in the residual graph, we have $l(u) \leq l(v) + |p|$. In particular, the length of a path from the source to the sink must be greater than or equal to $|V|$. However, as an augmenting path is simple by definition, its length must be less than $|V|$.

Upon termination of the algorithm, there will be no active nodes. Thus, the preflow is actually a flow, and as there are no augmenting paths, the min-cut-max-flow theorem implies that the flow is maximal.

The push relabel algorithm initializes the heights of all nodes but the source to zero. Moreover, the outgoing edges of the source are set to carry maximum flow, all other edges get zero flow. Then, the algorithm repeats the following operations as long as they apply:

push(u, v) for an edge (u, v) of the residual graph with $l(u) = l(v) + 1$ and u active: move as much flow as possible from u to v. The maximum possible flow is constrained by the residual capacity of the edge, and the excess flow available at u.

relabel(u) for an active node u from which no push operation is possible: increase the height of u to the minimum value such that a push from u becomes possible.

Obviously, if none of the operations apply, all nodes are inactive and the resulting preflow is a maximum flow. The algorithm is generic in the sense that it leaves open the order in which the operations are applied to the different nodes and edges.

6.1 Formalization of the Generic Algorithm

For the complexity reasoning, we will count the different types of operations. Thus, it is convenient to formalize the algorithm as a labeled transition system over flows and heights, the labels indicating the performed operation:

```
inductive_set pr_lts :: ((preflow×labeling) × op_ty × (preflow×labeling)) set
where
  push_pre f l e ⟹ ((f,l),PUSH,(push_eff f e,l)) ∈ pr_lts
| relabel_pre f l u ⟹ ((f,l),RELABEL,(f,relabel_eff f l u)) ∈ pr_lts
```

Here, the **inductive_set** construction defines the least relation that can be derived by the specified rules. A push operation can be implemented in time $O(1)$, (only the flow on a single edge has to be modified) and a relabel operation can be implemented in time $O(V)$. (the new height depends on the heights of the successors in the residual graph) We define `cost PUSH ≡ 1 | cost RELABEL ≡ card(V)` and prove the following theorem:

```
theorem (in Network) generic_preflow_push_OV2E_and_correct:
  assumes ((f₀, l₀), p, (f, l)) ∈ pr_lts*
  shows (∑x←p. cost x) ≤ 26 * (card V)² * card E
    and (f,l)∉Domain pr_lts ⟶ isMaxFlow f
```

where (f_0, l_0) is the initial flow and height labeling, and `pr_lts`* denotes the reflexive transitive closure of `pr_lts`.[4] Intuitively, the cost of any (partial) run `p` from the initial state to `(f,l)` is bounded by $26V^2E = O(V^2E)$, and if the run is complete, i. e. no more operations apply (`(f,l)∉Domain pr_lts`), then the resulting flow is maximal.

For the proof of this theorem, we follow the presentation of Cormen et al. [10]. The correctness proof shows that the algorithm maintains a valid labeling, and its formalization poses no further difficulties. It consists of less than 500 lines, and took us only a few days to complete.

The complexity argument is trickier. Instead of defining a measure function, Cormen counts the different types of operations separately, using quite involved arguments. We have slightly modified Cormen's proofs, trading precision of the constant factors for simplicity of the formalization.[5]

First, we show that the labels of each node remain smaller than $2|V|$. This bounds the number of relabel operations to $O(V^2)$. For the push operations, we define a push to be *saturating*, if the excess at the source node of the push is not less than the capacity of the edge. In this case, the edge is saturated by the push and disappears from the residual graph. However, the source node of the push may remain active. On the other hand, a non-saturating push deactivates the source node of the push, but the edge remains in the residual graph.

We count the number of saturating and non-saturating pushes separately. For saturating pushes, we regard the number of saturating pushes for each edge (u, v) during execution of the algorithm: After a saturating push, the edge has disappeared from the residual graph. Thus, before the next saturating push on the same edge, there must be an operation that restores the edge. This can only be a push on the reverse edge (v, e). However, this push can only occur after the height of v has increased. As the height of any node is bounded by $O(V)$, we

[4] I.e., $(u_0, [l_1, \ldots, l_n], u_n) \in R^*$ iff $\forall 0 \leq i < n.\ (u_i, l_{i+1}, u_{i+1}) \in R$.

[5] Using Cormen's technique, we would have been able to prove a bound of $17V^2E$.

get a bound of $O(VE)$ on the number of saturating push operations. Finally, the number of non-saturating pushes is bounded by defining the potential function $\Phi = \sum_{u \text{ is active node}} l(u)$. It is decreased by a non-saturating push. A saturating push or relabel operation increases the potential by $O(V)$. The initial potential is $O(V)$, and with the already established bounds on the other operations we get a bound of $O(V^3 + V^2E) = O(V^2E)$ on the number of non-saturating push operations. Combining the bounds for the different operation types with the costs for the operations then yields the desired time complexity bound of $O(V^2E)$.

The simple structure of the generic algorithm allowed us to conveniently define it as labeled transition system, such that the above informal proof could be formalized by inductive arguments over the runs of the algorithm. Moreover, the powerful Isabelle/HOL standard library, in particular the support for summations over finite sets, was very helpful to keep the formalization effort manageable. Nevertheless, the whole complexity proof is roughly 1000 lines of Isabelle text, and took a few weeks to complete. This includes the time that we spent on an initial termination proof that used simpler arguments, but did not derive an exact complexity bound.

7 Instantiating the Generic Push-Relabel Algorithm

The generic algorithm does not specify the order in which the operations are applied. Choosing the right order decreases the worst case time complexity, and, more importantly, the practical runtime.

We consider two instantiations of the generic algorithm: The relabel-to-front algorithm [10], and the FIFO push-relabel algorithm [16]. Both have a theoretical time complexity of $O(V^3)$, which is better than the $O(V^2E)$ bound of the generic algorithm. However, the more complicated structure of these algorithms prevents a convenient specification as labeled transition system, and thus a complexity proof in the style done for the generic algorithm. Instead, we use the Isabelle Refinement Framework to specify the algorithms and only prove total correctness. We leave it to future work to integrate complexity reasoning tools into the Refinement Framework and to prove the tighter complexity bounds for the specific implementations.

A general idea of the most efficient implementations is to *discharge* a node by repeatedly applying push operations until the node gets inactive or needs to be relabeled. The implementations mainly differ in the order in which the nodes are discharged.

Obviously, a sequence of discharge and relabel operations is a special instance of the generic algorithm, which gives us termination and invariant preservation. The main work of the correctness proofs for the refined algorithms goes into showing that they only terminate if no active nodes are left.

7.1 Relabel-to-Front Algorithm

Cormen et al. [10] present the relabel to front algorithm. It discharges a node until it gets inactive, applying relabel operations as necessary. Moreover, if a node was discharged without being relabeled, the already saturated edges during this discharge are stored, and not considered during the next discharge. This reduces the work required to find the next edge to push. The algorithm scans through a

list of all nodes, discharging every node as it is visited. If the node gets relabeled during discharging, it is moved to the front of the list, and the scan of the list begins anew. The algorithm terminates if it reaches the end of the list.

The main idea of the correctness proof is to show that all nodes left of the current position in the list are inactive. Thus, upon reaching the end of the list, all nodes are inactive, and the current flow is maximal. In order to show maintenance of this invariant, one has to strengthen it by adding that the list is a topological ordering of the *admissible network*, which consists of all edges (u, v) of the residual graph with $l(u) = l(v) + 1$.

Following the proof by Cormen et al., and using the Isabelle Refinement Framework to conveniently describe the abstract algorithm, the correctness proof offered no surprises. It is about 700 lines of Isabelle text plus 150 lines of general lemmas on topological orderings, and took only a few days to develop.

7.2 FIFO Push-Relabel Algorithm

While the relabel-to-front algorithm has a theoretical worst case complexity of $O(V^3)$, it seems to be rather inefficient in practice.[6] An alternative implementation is the FIFO push-relabel algorithm, which maintains a queue (FIFO) of the active nodes. In each iteration, it dequeues a node and tries to discharge it by only using push operations. If the node is still active, it is relabeled and enqueued.

The FIFO push-relabel algorithm has several advantages: It's correctness is straightforward to prove, as the only required additional invariant is that the queue contains exactly the active nodes. Moreover, it performs well in practice, and we have found several reference implementations of it [15,44].

Our algorithm follows the implementation that we found in the Stanford ACM-ICPC notebook [44], which uses an additional gap heuristics: If there is a value $0 < k < |V|$ such that there is no node that has height k, all nodes v with $k < l(v) < |V|$ may be relabeled to $|V| + 1$, preserving a valid labeling. This heuristics prevents unnecessary pushes of flow towards the sink when the path to the sink is already blocked. The correctness of the heuristics follows easily from the definitions of a valid height labeling. In total, the correctness proof of the FIFO push-relabel algorithm is about 400 lines of proof text and, building on the experience from relabel-to-front, it took only a day to complete.

8 Refinement to Executable Code

In the previous sections, we have presented our abstract formalization of the Edmonds-Karp and push-relabel algorithms. However, we left open several algorithmic aspects. For example, we did not specify how to find a shortest augmenting path. Moreover, efficient implementations of some operations require additional auxiliary data to be maintained. Finally, we have to specify the concrete data structures to be used for the implementation. In this section, we outline the further refinement steps that were necessary to obtain an efficient implementation.

[6] We have not found any implementation based on relabel-to-front, and our own experiments indicate a rather poor performance.

8.1 Algorithms

The most complex problem for which we specified no algorithm yet is how to find a shortest augmenting path in the Edmonds-Karp algorithm. We use breadth first search (BFS) for this. We adapted a BFS formalization from the examples collection of the Isabelle Refinement Framework to our needs and added an efficient imperative implementation. The resulting algorithm is independent of flow networks and can be reused for other applications.

Simpler problems for which we had to develop algorithms are augmentation of the flow along the shortest path, and the relabel and discharge operations. The abstract specification of augmentation (cf. Section 2) first maps the augmenting path to an augmenting flow, which is then added to the current flow. For the implementation, we skip the intermediate augmenting flow, and modify the current flow directly. The relabel operation has to determine the new height for the relabeled node. It is straightforward to implement this by iterating over the successor nodes in the residual graph. Refinement of the discharge operation is trickier: Abstractly, we nondeterministically select a new edge to push, until we find no more edges. In the concrete implementation, however, we fold over the list of successor nodes. This structural difference between abstract and concrete algorithm cannot be handled by the heuristics of the Refinement Framework's verification condition generator. Thus, we had to manually prove some specialized refinement rules.

8.2 Additional Data

An important optimization is to let the algorithms manipulate residual graphs instead of flows: Most operations can be implemented to only access one edge of the residual graph, while they would need to access multiple edges if phrased on flows. At the end of the algorithm, the flow can be computed from the residual graph. An initial version of our Edmonds-Karp algorithm worked on flows, and switching to residual graphs yielded a speedup of roughly factor 2. Technically, we relate the concrete operations on residual graphs to the abstract operations on flows by the refinement relation $\{(c_f, f) \mid f \text{ is a flow}\}$.

Moreover, for the push-relabel algorithms, we introduce a map from nodes to their current excess value: A single lookup in this map is sufficient to compute the flow for a push operation — instead of summing up all the incoming and outgoing flows. We combine the introduction of the excess map with the transition from preflows to residual graphs, using the refinement relation $\{((x_f, c_f), f) \mid f \text{ is a flow}\}$.

Another important optimization is to precompute an adjacency map of the network: To find the successors of a node in the residual graph, it is enough to search among the adjacent nodes in the network. As iterations over successor nodes occur in the inner loops of the algorithms, this optimization is quite effective, in particular for sparse graphs.

For the gap heuristics, we introduce a height frequency map, which maps each height value to the number of nodes labeled with this value. A single lookup in this map suffices to detect when we relabel the last node of a specific height such that the gap heuristics applies.

8.3 Using Efficient Data Structures

In a final step, we refine our algorithms to use efficient data structures. This step is partially automated by the Sepref tool [26, 27], which synthesizes imperative programs and the corresponding refinement proofs, based on a user defined mapping from abstract types to data structures. The imperative programs are specified in Imperative HOL [5], which defines a heap monad for Isabelle/HOL.

We implement capacities by (arbitrary precision) integer numbers[7]. Note that an implementation by fixed precision numbers would also be possible, but requires additional checks on the network to ensure that no overflows can happen.[8] Nodes are represented by natural numbers less than an upper bound N, and residual graphs are implemented by their capacity matrices, which, in turn, are realized as arrays of size $N \times N$ with row-major indexing, such that the successors of a node are close together in memory. The adjacency map of the network is implemented by an array of lists of nodes. An augmenting path is represented by a list of edges, i. e. a list of pairs of nodes. The height frequency map and the maps from nodes to height labels and excess values are implemented by arrays. The queue of the FIFO push-relabel algorithm and the list of the relabel-to-front algorithm are implemented by two stacks, allowing for amortized constant time operations.

There is still some optimization potential left in the choice of our data structures: For example, the BFS algorithm computes a predecessor map P. It then iterates over P to extract the shortest path as a list of edges. A subsequent iteration over this list computes the residual capacity, and a final iteration performs the augmentation. This calls for a deforestation optimization to get rid of the intermediate list, and iterate only two times over the predecessor map directly. Also the FIFO queue could be implemented by a ring buffer based on a fixed size array, yielding true constant time operations with less overhead. Fortunately, our profiling data shows that these operations do not significantly contribute to the runtime of our implementations. A more important optimization would be to refine our naive implementation of graphs as adjacency matrix. While this representation is well-suited for dense graphs, it wastes memory and cache locality for sparse graphs (cf. Section 9). We leave it to future research to verify graph representations that are more suited to sparse graphs.

Finally, we use the Isabelle code generator [19, 20] to extract our algorithm to an SML program, which we link to a simple parser and command line interface to get a stand-alone program for benchmarking.

8.4 Network Checker and Main Correctness Theorem

Additionally, we implemented an algorithm that takes as input a list of edges, a source node, and a target node. It converts these to a capacity matrix and

[7] Up to this point, the formalization models capacities as *linearly ordered integral domains*, which subsume reals, rationals, and integers. Thus, we could chose any executable number representation here.

[8] Actually, the unverified reference implementations of the push-relabel algorithms [15, 44] that we use in our benchmarks (cf. Section 9) lack such checks, and silently report erroneous maximum flow values in case of overflow. One implementation [44] has a parser that silently ignores overflows, while the other [15] uses a too small integer type for the excess flow.

an adjacency map, and checks whether the resulting graph satisfies our network assumptions. We proved that this algorithm returns the correct capacity matrix and adjacency map iff the input describes a valid network, and returns a failure value otherwise.

Combining the implementations of our algorithms with the network checker yields our final implementations for which we also export code. On any input, the implementation checks whether the input is well-formed and returns a maximal flow for well-formed inputs. We display the correctness theorem for the Edmonds-Karp algorithm here, the theorems for relabel-to-front and FIFO push-relabel are analogous.

```
theorem
  fixes el defines c ≡ ln_α el
  shows <emp> edmonds_karp el s t <λ
      None ⇒ ↑(¬ln_invar el ∨ ¬Network c s t)
    | Some (N,cf) ⇒
      ↑(ln_invar el ∧ Network c s t ∧ Graph.V c ⊆ {0..<N})
    * (∃ₐf. is_rflow c N f cf * ↑(Network.isMaxFlow c s t f))>ₜ
```

This theorem is stated as a Hoare triple, using separation logic [42,28] assertions.[9] There are no preconditions on the input. If the algorithm returns `None`, then the edge list was malformed or described a graph that does not satisfy the network assumptions. Here, `ln_invar` describes well-formed edge lists, i. e. edge lists that have no duplicate edges and only edges with positive capacity, and `ln_`α describes the mapping from (well-formed) edge lists to capacity matrices (note that we set `c` $\equiv$ `ln_`α `el`). If the algorithm returns some number `N` and residual graph `cf`, then the input was a well-formed edge list that describes a valid network with at most `N` nodes. Moreover, the returned residual graph describes a flow `f` in the network, which is maximal. As the case distinction is exhaustive, this theorem states the correctness of the algorithm. Note that Isabelle/HOL does not have a notion of execution, thus total correctness of the generated code cannot be expressed. However, the program is phrased in a heap-exception monad, thus introducing some (coarse grained) notion of computation. On this level, termination can be ensured, and, indeed, the above theorem implies that all the recursions stated by recursion combinators in the monad must terminate. However, it does not guarantee that we have not injected spurious code equations like $f\ x = f\ x$, which is provable by reflexivity, but causes the generated program to diverge.

9 Benchmarking

We have benchmarked our verified algorithms against unverified reference implementations. The SML code for our verified Edmonds-Karp and FIFO push-relabel algorithms has been compiled with MLton [37]. We have not included the relabel-to-front algorithm, as it was three orders of magnitude slower than FIFO push-relabel in preliminary tests.

[9] A Hoare triple is written as $<P>\ c\ <\lambda r.\ Q\ r>_t$, where P is the precondition, c the program, and Q the postcondition that also depends on the program's return value r. Pre- and postcondition are written as separation logic assertions, where emp describes the empty heap, $*$ is the separating conjunction, $\uparrow\Phi$ indicates a pure assertion, i. e. one that describes no heap content, and $\exists_A$ is the existential quantifier lifted to assertions.

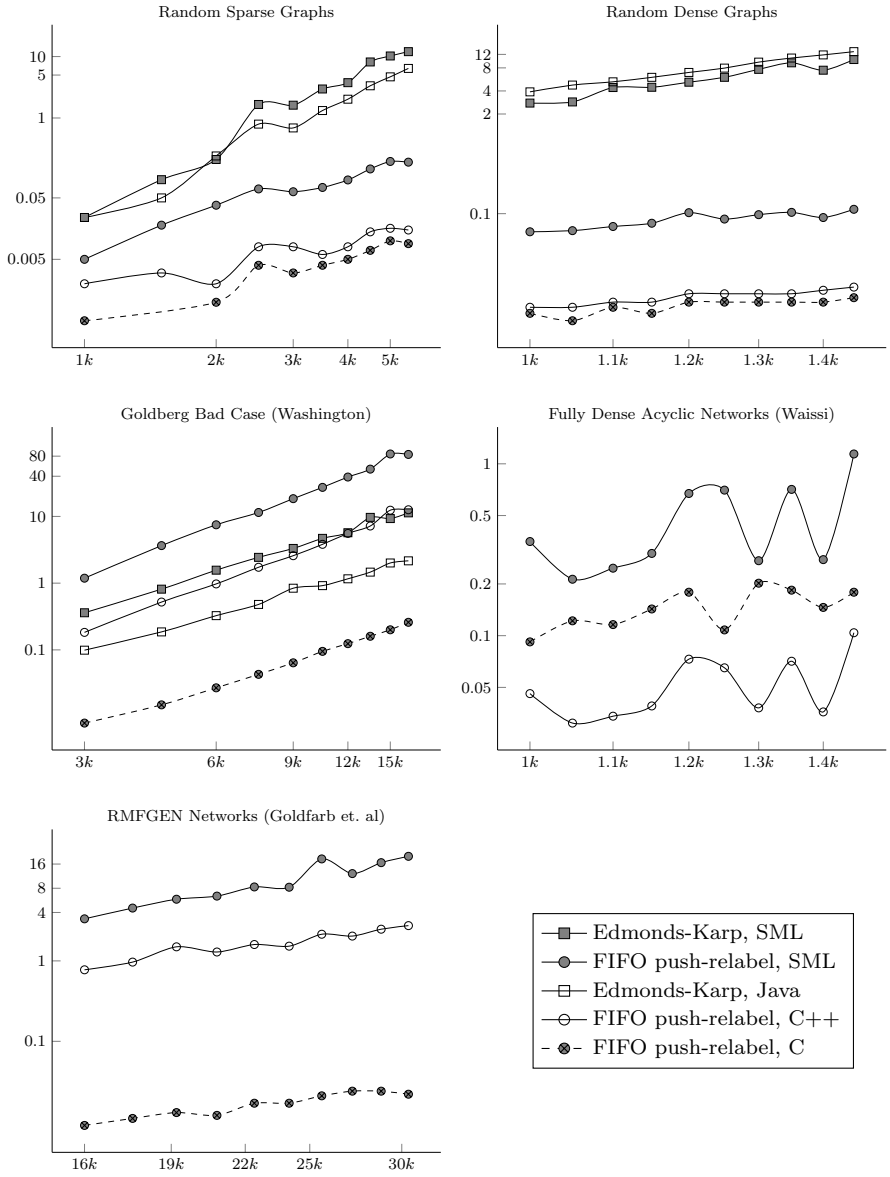

Fig. 1: Benchmark of different implementations. The x-axis shows the number of nodes, the y axis the execution time in seconds.

For the Edmonds-Karp algorithm, we use a reference implementation in Java from Sedgewick and Wayne's book on algorithms [43], which is compiled and run with OpenJDK Java 1.8.0. It is similar to our implementation except that it uses adjacency lists to represent the graph, operates directly on flows (cf. Section 8.2), and uses no intermediate list to store the augmenting path (cf. Section 8.3).

For the push-relabel algorithm, we use the C++ implementation from the Stanford ACM-ICPC notebook [44], which served as a template for our verified FIFO push-relabel algorithm. Moreover, we use the Q_PRF program version 1.8 from Andrew Goldberg's Network Optimization Library [15], which is a highly optimized C implementation of a FIFO push-relabel algorithm with various additional heuristics, originally developed by Cherkassky and Goldberg [9]. These were compiled using gcc 5.4 at optimization level 4. The tests were run on a standard laptop machine with a 2.7GHz i7 quadcore processor and 16GiB of RAM.

Our data set consists of randomly generated sparse and dense networks, the sparse networks having a *density* ($= \frac{E}{V(V-1)}$) of 0.02, and the dense networks having a density of 0.25. Note that the maximum density for networks that satisfy our assumptions is 0.5, as we allow no parallel edges. Moreover, we use some networks from the DIMACS implementation challenge [21]: The "Goldberg bad case networks" from the "Washington" generator, the "fully dense acyclic networks" of Waissi, and the RMFGEN networks of Goldfarb and Grigoriadis.

The results are displayed in Figure 1, using a log-log scale. Note that we only measure the time required to run the algorithms themselves, excluding the time required for parsing the input and running the network checker. Moreover, for the sake of readability, we excluded the Edmonds-Karp implementations from the diagrams for fully dense acyclic networks and RMFGEN networks, as they were orders of magnitude slower than the push-relabel implementations.

For the Edmonds-Karp algorithm, we observe that, for sparse graphs, the Java implementation is roughly faster by a factor of 1.6, while for dense graphs, our implementation is faster by a factor of 1.2. We conjecture that this is mainly due to the Java implementation using adjacency lists which are better suited for sparse graphs than our adjacency matrices. This is supported by the results for the even sparser (density ranges from 10^{-4} to 10^{-5}) Goldberg bad case networks, where the Java implementation is 4.8 times faster on average.

For the push-relabel algorithms, our verified algorithm is, depending on the test case, between 5 and 10 times slower than the unverified C++ implementation. The highly optimized C implementation is significantly faster in most cases, only for the "fully dense acyclic networks" it is slower than the simple C++ implementation. The performance gap between our verified SML code and the C++ code that implements essentially the same algorithm is not surprising: We compare a memory safe functional language with a slightly aged compiler used by only a few people against an unsafe language with a very recent and actively developed compiler and a huge user base.

The push-relabel implementations are significantly faster than the Edmonds-Karp implementations, except on the "Goldberg bad case" networks, which are designed as worst case problems for push-relabel algorithms: Here, the simple C++ implementation of FIFO push-relabel is slower than the Edmonds-Karp algorithm, while the highly optimized one in C shows no weaknesses.

10 Conclusion

We have presented a mechanical verification of the correctness and time complexity of maximum flow algorithms based on both the augmenting path and the push-

relabel approaches, as well as the min-cut-max-flow theorem, which underlies all these algorithms.

For the augmenting path algorithms, we start with a verification of the generic Ford-Fulkerson algorithm, specialize it to the Edmonds-Karp algorithm, and prove its time complexity of $O(VE^2)$. For the push-relabel approaches, we start with a verification of the generic push-relabel algorithm of Goldberg and Tarjan, show its time complexity of $O(V^2E)$, and instantiate it to both the relabel-to-front and the FIFO push-relabel algorithm. We then conduct several refinement steps to derive efficiently executable implementations of all three algorithm. The runtime of our verified implementations compares well to that of unverified reference implementations.

Our formalization has combined several techniques to achieve an elegant and accessible formalization: Using the Isar proof language [45], we were able to provide a completely rigorous but still accessible proof of the Ford-Fulkerson theorem. The Isabelle Refinement Framework [33, 24] and the Sepref tool [26, 27] allowed us to present the algorithms on a level of abstraction that closely resembles pseudocode presentations found in textbooks, and then formally link this presentation to efficient implementations. Moreover, modularity of refinement allowed us to independently develop subroutines like the breadth first search algorithm, which can be reused as building block for other algorithms. The data structures are reusable, too: although we had to implement the array representation of (capacity) matrices for this project, it is now part of the growing library of verified imperative data structures supported by the Sepref tool, such that it can be reused for future formalizations.

During this project, we have learned some lessons on verified algorithm development:

- It is important to keep the levels of abstraction strictly separated. For example, when implementing the capacity function with arrays, one needs to show that it is only applied to valid nodes. However, proving that, e. g. augmenting paths only contain valid nodes is hard at this low level. Instead, one can protect the application of the capacity function by an assertion — already on a high abstraction level where it can be easily discharged. On refinement, this assertion is passed down, and ultimately available for the implementation. Optimally, one wraps the function together with an assertion of its precondition into a new constant, which is then refined independently.
- Profiling has helped a lot in identifying candidates for optimization. For example, based on profiling data, we decided to delay a possible deforestation optimization on augmenting paths, and to first refine the algorithm to operate on residual graphs directly.
- "Efficiency bugs" are as easy to introduce as for unverified software. For example, out of convenience, we implemented the successor list computation by *filter*. Profiling then indicated a hot spot on this function. As the order of successors does not matter, we invested a bit more work to make the computation tail recursive and gained a significant speedup. Moreover, we realized only lately that we had accidentally implemented and verified matrices with column major ordering, which have a poor cache locality for our algorithm. Changing the order resulted in another significant speedup.

We conclude with some statistics: The formalization consists of roughly 14000 lines of proof text, where the graph theory up to the min-cut-max-flow theorem

requires 2500 lines. The abstract Edmonds-Karp algorithm and its complexity analysis contribute 800 lines, and its implementation (including BFS) another 1700 lines. The generic push relabel algorithm and its complexity analysis is 1700 lines, the abstract implementations of relabel-to-front and FIFO push-relabel is another 1100 lines, and their refinement to an efficient implementation takes 3000 lines.

The remaining lines are contributed by the network checker and some auxiliary theories. The development of the theories required roughly 5 man month, a significant amount of this time going into a first, purely functional implementation of the Edmonds-Karp algorithm, which was later dropped in favor of the faster imperative version.

10.1 Related Work

We are only aware of one other formalization of the Ford-Fulkerson algorithm conducted in Mizar [36] by Lee. Unfortunately, there seems to be no publication on this formalization except [34], which provides a Mizar proof script without any additional comments except that it "defines and proves correctness of Ford/Fulkerson's Maximum Network Flow algorithm at the level of graph manipulations". Moreover, in Lee et al. [35], which is about graph representation in Mizar, the formalization is shortly mentioned, and it is clarified that it does not provide any implementation or data structure formalization. As far as we understood the Mizar proof script, it formalizes an algorithm roughly equivalent to our abstract version of the Ford-Fulkerson algorithm. Termination is only proved for integer valued capacities.

Apart from our own work [25,40], there are several other verifications of graph algorithms and their implementations, using different techniques and proof assistants. Noschinski [41] verifies a checker for (non-)planarity certificates using a bottom-up approach. Starting at a C implementation, the AutoCorres tool [17,18] generates a monadic representation of the program in Isabelle. Further abstractions are applied to hide low-level details like pointer manipulations and fixed size integers. Finally, a verification condition generator is used to prove the abstracted program correct. Note that their approach takes the opposite direction than ours: While they start at a concrete version of the algorithm and use abstraction steps to eliminate implementation details, we start at an abstract version, and use concretization steps to introduce implementation details.

More recently, Chen and Levy [8] used Why3 [13] to verify a functional implementation of Tarjan's strongly connected components algorithm, with emphasis on proof readability.

Charguéraud [6] also uses a bottom-up approach to verify imperative programs written in a subset of OCaml, amongst them a version of Dijkstra's algorithm: A verification condition generator generates a *characteristic formula*, which reflects the semantics of the program in the logic of the Coq proof assistant [4].

We prove the complexity bounds of our algorithms on an abstract level, by instrumenting the loop with a counter to count abstract operations (in our case, loop iterations). This counter does not appear in the refined versions of the code, nor is there any formal model of runtime of the generated code. In this respect, our technique is similar to the approach used by Nipkow [38], where a complexity function is derived systematically but manually from the function to be analyzed.

A more precise approach is taken by Charguéraud and Pottier [7], who argue about the number of β-reduction steps taken by the program.

10.2 Future Work

The most obvious improvement to our formalization is to use a graph representation which is better suited for sparse graphs, or even switch between the two implementations depending on the input graph. Compared to the most advanced implementations of push-relabel algorithms, e. g. [9], our formalization lacks two important heuristics: First, the global relabeling heuristics periodically performs a breadth first search on the reversed residual graph, to determine exact node labels based on their distance to the sink. Second, the algorithm is split in two phases: The first phase ignores nodes with labels greater than $|V|$, and already determines the exact value of the flow. The second phase only pushes excess flow back to the source. The main focus of future work will be to verify these heuristics.

References

1. R.-J. Back. *On the correctness of refinement steps in program development.* PhD thesis, Department of Computer Science, University of Helsinki, 1978.
2. R.-J. Back and J. von Wright. *Refinement Calculus — A Systematic Introduction.* Springer, 1998.
3. C. Ballarin. Interpretation of locales in Isabelle: Theories and proof contexts. In J. M. Borwein and W. M. Farmer, editors, *MKM 2006*, volume 4108 of *LNAI*. Springer, 2006.
4. Y. Bertot and P. Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions.* Springer, 1st edition, 2010.
5. L. Bulwahn, A. Krauss, F. Haftmann, L. Erkök, and J. Matthews. Imperative functional programming with Isabelle/HOL. In *TPHOL*, volume 5170 of *LNCS*. Springer, 2008.
6. A. Charguéraud. Characteristic formulae for the verification of imperative programs. In *ICFP*. ACM, 2011.
7. A. Charguéraud and F. Pottier. Machine-checked verification of the correctness and amortized complexity of an efficient union-find implementation. In *Proc. of ITP*, 2015.
8. R. Chen and J.-J. Lévy. A semi-automatic proof of strong connectivity. VSTTE 2017, `moscova.inria.fr/~levy/pubs/17scct.pdf`.
9. B. V. Cherkassky and A. V. Goldberg. On implementing the push—relabel method for the maximum flow problem. *Algorithmica*, 19(4), 1997.
10. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.
11. Y. Dinitz. Theoretical computer science. chapter Dinitz' Algorithm: The Original Version and Even's Version. Springer, 2006.
12. J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2), 1972.
13. J.-C. Filliâtre and A. Paskevich. *Why3 — Where Programs Meet Provers.* Springer, 2013.
14. L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian journal of Mathematics*, 8(3), 1956.
15. A. V. Goldberg. Andrew goldberg's network optimization library. `http://www.avglab.com/andrew/soft.html`.
16. A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4), Oct. 1988.
17. D. Greenaway. *Automated proof-producing abstraction of C code.* PhD thesis, CSE, UNSW, Sydney, Australia, mar 2015.
18. D. Greenaway, J. Andronick, and G. Klein. Bridging the gap: Automatic verified abstraction of C. In *ITP*. Springer, aug 2012.
19. F. Haftmann. *Code Generation from Specifications in Higher Order Logic.* PhD thesis, Technische Universität München, 2009.

20. F. Haftmann and T. Nipkow. Code generation via higher-order rewrite systems. In *FLOPS 2010*, LNCS. Springer, 2010.
21. D. S. Johnson, C. C. McGeoch, et al. *Network flows and matching: first DIMACS implementation challenge*, volume 12. American Mathematical Soc.
22. A. V. Karzanov. Determination of maximal flow in a network by method of preflows. *Doklady Akademii Nauk SSSR*, 215(1), 1974.
23. A. Krauss. Recursive definitions of monadic functions. In *Proc. of PAR*, volume 43, 2010.
24. P. Lammich. Refinement for monadic programs. In *Archive of Formal Proofs*. `http://afp.sf.net/entries/Refine_Monadic.shtml`, 2012. Formal proof development.
25. P. Lammich. Verified efficient implementation of Gabow's strongly connected component algorithm. In *ITP*, volume 8558 of *LNCS*. Springer, 2014.
26. P. Lammich. Refinement to Imperative/HOL. In *ITP*, volume 9236 of *LNCS*. Springer, 2015.
27. P. Lammich. Refinement based verification of imperative data structures. In *CPP*. ACM, 2016.
28. P. Lammich and R. Meis. A separation logic framework for Imperative HOL. *Archive of Formal Proofs*, Nov. 2012. `http://afp.sf.net/entries/Separation_Logic_Imperative_HOL.shtml`, Formal proof development.
29. P. Lammich and S. R. Sefidgar. Formalizing the Edmonds-Karp algorithm. In *Proc. of ITP*, 2016.
30. P. Lammich and S. R. Sefidgar. Formalizing the Edmonds-Karp algorithm. *Archive of Formal Proofs*, Aug. 2016. `http://isa-afp.org/entries/EdmondsKarp_Maxflow.shtml`, Formal proof development.
31. P. Lammich and S. R. Sefidgar. Flow networks and the Min-Cut-Max-Flow theorem. *Archive of Formal Proofs*, June 2017. `http://isa-afp.org/entries/Flow_Networks.shtml`, Formal proof development.
32. P. Lammich and S. R. Sefidgar. Formalizing push-relabel algorithms. *Archive of Formal Proofs*, June 2017. `http://isa-afp.org/entries/Prpu_Maxflow.shtml`, Formal proof development.
33. P. Lammich and T. Tuerk. Applying data refinement for monadic programs to Hopcroft's algorithm. In *Proc. of ITP*, volume 7406 of *LNCS*. Springer, 2012.
34. G. Lee. Correctnesss of Ford-Fulkerson's maximum flow algorithm. *Formalized Mathematics*, 13(2), 2005.
35. G. Lee and P. Rudnicki. Alternative aggregates in Mizar. In *Calculemus '07 / MKM '07*. Springer, 2007.
36. R. Matuszewski and P. Rudnicki. Mizar: the first 30 years. *Mechanized Mathematics and Its Applications*, 2005.
37. MLton Standard ML compiler. http://mlton.org/.
38. T. Nipkow. Amortized complexity verified. In *Proc. of ITP*, 2015.
39. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
40. B. Nordhoff and P. Lammich. Formalization of Dijkstra's algorithm. *Archive of Formal Proofs*, Jan. 2012. `http://afp.sf.net/entries/Dijkstra_Shortest_Path.shtml`, Formal proof development.
41. L. Noschinski. *Formalizing Graph Theory and Planarity Certificates*. PhD thesis, Fakultät für Informatik, Technische Universität München, November 2015.
42. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc of. Logic in Computer Science (LICS)*. IEEE, 2002.
43. R. Sedgewick and K. Wayne. *Algorithms*. Addison-Wesley, 2011. 4th edition.
44. Stanford ACM-ICPC notebook. `https://github.com/jaehyunp/stanfordacm`.
45. M. Wenzel. Isar - A generic interpretative approach to readable formal proof documents. In *TPHOLs'99*, volume 1690 of *LNCS*. Springer, 1999.
46. N. Wirth. Program development by stepwise refinement. *Commun. ACM*, 14(4), Apr. 1971.
47. U. Zwick. The smallest networks on which the Ford-Fulkerson maximum flow procedure may fail to terminate. *Theoretical computer science*, 148(1), 1995.