Formalizing the Edmonds-Karp Algorithm

Peter Lammich, S. Reza Sefidgar

Technische Universität München, {lammich,sefidgar}@in.tum.de

Abstract. We present a formalization of the Ford-Fulkerson method for computing the maximum flow in a network. 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. We then use stepwise refinement to obtain the Edmonds-Karp algorithm, and formally prove a bound on its complexity. Further refinement yields a verified implementation, whose execution time compares well to an unverified reference implementation in Java.

1 Introduction

Computing the maximum flow of 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 Ford-Fulkerson method [10] describes a class of algorithms to solve the maximum flow problem. An important instance is the Edmonds-Karp algorithm [9], which was one of the first algorithms to solve the maximum flow problem in polynomial time for the general case of networks with real-valued capacities.

In this paper, we present a formal verification of the Edmonds-Karp algorithm and its polynomial complexity bound. The formalization is conducted in the Isabelle/HOL proof assistant [27]. Stepwise refinement techniques [33,1,2] allow us to elegantly structure our verification into an abstract proof of the Ford-Fulkerson method, its instantiation to the Edmonds-Karp algorithm, and finally an efficient implementation. The abstract parts of our verification closely follow the textbook presentation of Cormen et al. [7]. Using the Isar [32] proof language, we were able to produce proofs that are accessible even to non-Isabelle experts.

While there exists another formalization of the Ford-Fulkerson method in Mizar [23][1], we are, to the best of our knowledge, the first that verify a polynomial 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.

The rest of this paper is structured as follows: In Section 2 we give a short informal introduction to the Ford-Fulkerson method. In Section 3, we report on our formalization of the abstract method. Section 4 gives a brief overview of the Isabelle Refinement Framework [22,17], which supports stepwise refinement

[1] Section 8.1 provides a detailed discussion

based algorithm development in Isabelle/HOL. In Section 5, we report on our instantiation of the Ford-Fulkerson method to the Edmonds-Karp algorithm and the proof of its complexity. Section 6 reports on the further refinement steps required to yield an efficient implementation. Section 7 reports on benchmarking our implementation against a reference implementation of the Edmonds-Karp algorithm from Sedgewick et al. [31]. Finally, Section 8 gives a conclusion and discusses related and future work. The source code of our formalization is available at `http://www21.in.tum.de/~lammich/edmonds_karp/`.

2 The Ford-Fulkerson Method

In this section, we give a short introduction to the Ford-Fulkerson method, closely following the presentation by Cormen et al. [7].

A (flow) network is a directed graph over a finite set of vertices 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 vertices $s, t \in V$, 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 vertices except s and t, the sum of flows over all incoming edges is equal to the sum of flows over all outgoing edges. The value of a flow f is denoted by $|f|$, and 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 a 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 vertex of the network must be on a path from s to t. Note that any network can be transformed to a network with the aforementioned properties and the same maximum flow [7].

An important result is the relation between flows and cuts in a network. A *cut* is a partitioning of the vertices 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 Ford-Fulkerson 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 method 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

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

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 method 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 Ford-Fulkerson method 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 to increase the value of 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:

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

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 Ford-Fulkerson theorem, which is usually stated as the following three statements being 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 method does not specify how to find an augmenting path in the residual graph. There are several possible implementations with different execution times. The general method 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 [10,34]. When always choosing a *shortest* augmenting path, the number of iterations is bound 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 [9] with an overall running time of $O(VE^2)$.

3 Formalizing the Ford-Fulkerson Method

In this section, we provide a brief overview of our formalization of the Ford-Fulkerson method. 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}
  [...]
```

Moreover, we define basic concepts like (simple, shortest) paths, and provide lemmas to reason about them.

Networks are based on graphs, and add 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, formal proofs are conducted in the tactic language of the proof assistant, which is often some dialect of ML. Thus, many formal proofs are essentially programs that instruct the proof assistant how to conduct the proof. They tend to be inaccessible without a deep knowledge of the used proof assistant, in many cases requiring to replay the proof in the proof assistant in order to understand the idea behind it.

For the Isabelle/HOL proof assistant, the Isar proof language [32] 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. We use Isar to present our proof of the Ford-Fulkerson method such that it resembles the informal proof described by Cormen et al. [7].

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 ↑":

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

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

In addition,

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

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 [22,17]. For example, we define the Ford-Fulkerson method 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. We leave it to future work to define a set of combinators and appropriate syntax that allows for more concise presentation of pseudocode.

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 method, 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 method 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* [33], 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 [22,17]. 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 [16]. 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 $P \Longrightarrow m \leq \texttt{spec}\ r.\ Q\ r$ (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.

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 comes up with an initial version m_0 of the algorithm and its specification P,Q, and shows $P \Longrightarrow m_0 \leq \texttt{spec}\ Q$.

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 data refinement, one gets $P \Longrightarrow m_i \leq \Downarrow R_i \ldots R_1$ $\texttt{spec}\ Q$, 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 reflect the core idea of the proof precisely.

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 method 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 more tricky 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. Moreover, to make explicit the bound on the number of loop iterations, we instrument the loop to count its iterations, and assert the upper bound after the loop.

6 Refinement to Executable Code

In the previous section, we have presented our abstract formalization of the Edmonds-Karp algorithm, leaving open how to obtain a shortest augmenting path and how to implement the algorithm. In this section, we outline the further refinement steps that were necessary to obtain an efficient implementation.

6.1 Using Breadth First Search

A standard way to find a shortest path in a graph is breadth first search (BFS). Luckily, we had already formalized a BFS algorithm as an example for the Isabelle Refinement Framework. Unfortunately, this algorithm only computed the minimum distance between two nodes, without returning an actual path. For this project, we extended the formalization accordingly, and added an efficient imperative implementation, using the same stepwise refinement techniques as for the main algorithm. Note that the resulting BFS algorithm is independent, and can be reused for finding shortest paths in other applications.

Implementing shortest path finding by BFS in Edmonds-Karp algorithm yields a specification that algorithmically describes all major operations, but still leaves open the data structures used for implementation.

6.2 Manipulating Residual Graphs Directly

Next, we observe that the algorithm is phrased in terms of a flow, which is updated until it is maximal. In each iteration, the augmenting path is searched on the residual graph induced by the current flow. Obviously, computing the complete residual graph in each iteration is a bad idea. One solution to this problem is to compute the edges of the residual graph on the fly from the network and the current flow. Although this solution seems to be common, it has the disadvantage that for each edge of the residual graph, two (or even three) edges of the network and the flow have to be accessed. As edges of the residual graph are accessed in the inner loop, during the BFS, these operations are time critical.

After our profiling indicated a hot spot on accessing the capacity matrices of the network and the flow, we switched to an algorithm that operates on a representation of the residual graph directly. This resulted in a speed-up of roughly a factor of two. As the residual graph uniquely determines the flow (and vice versa), it is straightforward to phrase the operations directly on the residual graph. Performing a data refinement of the flow wrt. the refinement relation $\{(c_f, f) \mid f \text{ is a flow}\}$ then yields the desired algorithm.

6.3 Implementing Augmentation

In our abstract formalization, which matches the presentation in Section 2, we have formulated augmentation by first defining the residual capacity c_p of the augmenting path. Using c_p, we have defined the residual flow f_p, which was finally added to the current flow. In the refinement to operate on residual graphs, we have refined this to augment the residual graph. For the implementation, we compute the residual capacity in a first iteration over the augmenting path, and modify the residual graph in a second iteration. Proving this implementation correct is straightforward by induction on the augmenting path.

6.4 Computing Successors

In order to find an augmenting path, the BFS algorithm has to compute the successors of a node in the residual graph. Although this can be implemented on the edge labeling function by iterating over all nodes, this implementation tends to be inefficient for sparse graphs, where we would have to iterate over many possible successor nodes just to find that there is no edge.

A common optimization is to pre-compute an *adjacency map* from nodes to adjacent nodes in the network. As an edge in the residual graph is either in the same or opposite direction of a network edge, it is enough to iterate over the adjacent nodes in the network, and check whether they are actual successors in the residual graph. It is straightforward to show that this implementation actually returns the successor nodes in the residual graph.

6.5 Using Efficient Data Structures

In a final step, we have to choose efficient data structures for the algorithm.

We implement capacities as (arbitrary precision) integer numbers[3]. Note that an implementation as fixed precision numbers would also be possible, but requires additional checks on the network to ensure that no overflows can happen.

We implement nodes as 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 as 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 input network of the algorithm is represented as a function from network edges to capacities, which is tabulated into an array to obtain the initial residual graph. This gives us some flexibility in using the algorithm, as any capacity matrix representation can be converted into a function easily, without losing efficiency for read-access. Similarly, our implementation expects an adjacency

[3] 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.

map as additional parameter, which is then tabulated into an array. This is convenient in our context, where a preprocessing step computes the adjacency map anyway.

The output flow of the algorithm is represented as the residual graph. The user can decide how to compute the maximum flow from it. For example, in order to compute the maximum flow value, only the outgoing edges of the source node have to be computed, which is typically less expensive than computing the complete flow matrix. The correctness theorem of the algorithm abstractly states how to obtain the maximum flow from the output.

Note that 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. Fortunately, iteration over the shortest path seems not to significantly contribute to the runtime of our implementation, such that we leave this optimization for future work.

Note that we performed the last refinement step using our Sepref tool [19,20], which provides tool support for refinement from the purely functional programs of the Isabelle Refinement Framework into imperative programs expressed in Imperative/HOL [5]. The formalization of this refinement step consists of setting up the mappings between the abstract and concrete data structures, and then using Sepref to synthesize the Imperative/HOL programs and their refinement proofs. Finally, Imperative/HOL comes with a setup for the Isabelle code generator [14,15] to generate imperative programs in OCaml, SML, Scala, and Haskell.

6.6 Network Checker

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 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 implementation of the Edmonds-Karp algorithm with the network checker yields our final implementation, for which we can export code, and have proved the following theorem:

```
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})
    * (∃A f. is_rflow c N f cf * ↑(Network.isMaxFlow c s t f))>t
```

Note that this theorem is stated as a Hoare triple, using separation logic [30,21] assertions. 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.

7 Benchmarking

We have compared the running time of our algorithm in SML against an unverified reference implementation in Java, taken from Sedgewick and Wayne's book on algorithms [31]. We have used MLton 20100608 [26] and OpenJDK Java 1.7.0_95, running on a standard laptop machine with a 2.8GHz i7 quadcore processor and 16GiB of RAM.

We have done the comparison on 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. For sparse networks, we varied the number of nodes between 1000 and 5500, for dense networks between 1000 and 1450. The results are shown in Figure 1, in a double-logarithmic scale.

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. Note that the Java implementation operates on flows, while our implementation operates on residual graphs (cf. Section 6.2). Moreover, the Java implementation does not store the augmenting path in an intermediate list, but uses the predecessor map computed by the BFS directly (cf. Section 6.5). Finally note that a carefully optimized C++ implementation of the algorithm is only slightly faster than the Java implementation for sparse graphs, but roughly one order of magnitude faster for dense graphs. We leave it to future work to investigate this issue, and conclude that we were able to produce a reasonably fast verified implementation.

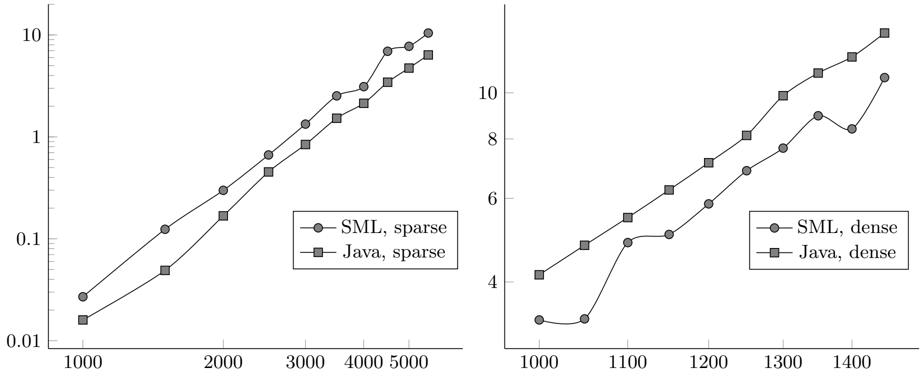

Fig. 1: Benchmark of different implementations. The x-axis shows the number of nodes, the y axis the execution time in seconds.

8 Conclusion

We have presented a verification of the Edmonds-Karp algorithm, using a stepwise refinement approach. Starting with a proof of the Ford-Fulkerson theorem, we have verified the generic Ford-Fulkerson method, specialized it to the Edmonds-Karp algorithm, and proved the upper bound $O(VE)$ for the number of outer loop iterations. We then conducted several refinement steps to derive an efficiently executable implementation of the algorithm, including a verified breadth first search algorithm to obtain shortest augmenting paths. Finally, we added a verified algorithm to check whether the input is a valid network, and generated executable code in SML. The runtime of our verified implementation compares well to that of an unverified reference implementation in Java.

Our formalization has combined several techniques to achieve an elegant and accessible formalization: Using the Isar proof language [32], we were able to provide a completely rigorous but still accessible proof of the Ford-Fulkerson theorem. The Isabelle Refinement Framework [22,17] and the Sepref tool [19,20] allowed us to present the Ford-Fulkerson method on a level of abstraction that closely resembles pseudocode presentations found in textbooks, and then formally link this presentation to an efficient implementation. Moreover, modularity of refinement allowed us to develop the breadth first search algorithm independently, and later link it to the main algorithm. The BFS algorithm can be reused as building block for other algorithms. The data structures are re-usable, too: although we had to implement the array representation of (capacity) matrices for this project, it will be added to the growing library of verified imperative data structures supported by the Sepref tool, such that it can be re-used 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 speed-up. 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 speed-up.

We conclude with some statistics: The formalization consists of roughly 8000 lines of proof text, where the graph theory up to the Ford-Fulkerson algorithm requires 3000 lines. The abstract Edmonds-Karp algorithm and its complexity analysis contribute 800 lines, and its implementation (including BFS) another 1700 lines. The remaining lines are contributed by the network checker and some auxiliary theories. The development of the theories required roughly 3 man month, a significant amount of this time going into a first, purely functional version of the implementation, which was later dropped in favor of the faster imperative version.

8.1 Related Work

We are only aware of one other formalization of the Ford-Fulkerson method conducted in Mizar [25] by Lee. Unfortunately, there seems to be no publication on this formalization except [23], 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. [24], 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 method. Termination is only proved for integer valued capacities.

Apart from our own work [18,28], there are several other verifications of graph algorithms and their implementations, using different techniques and proof assistants. Noschinski [29] verifies a checker for (non-)planarity certificates using a bottom-up approach. Starting at a C implementation, the AutoCorres tool [12,13] 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.

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].

8.2 Future Work

Future work includes the optimization of our implementation, and the formalization of more advanced maximum flow algorithms, like Dinic's algorithm [8] or push-relabel algorithms [11]. We expect both formalizing the abstract theory and developing efficient implementations to be challenging but realistic tasks.

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*, pages 31–43. 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*, pages 134–149. Springer, 2008.
6. A. Charguéraud. Characteristic formulae for the verification of imperative programs. In *ICFP*, pages 418–430. ACM, 2011.
7. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.
8. Y. Dinitz. Theoretical computer science. chapter Dinitz' Algorithm: The Original Version and Even's Version, pages 218–240. Springer, 2006.
9. J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, 1972.
10. L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian journal of Mathematics*, 8(3):399–404, 1956.
11. A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4), Oct. 1988.

12. D. Greenaway. *Automated proof-producing abstraction of C code.* PhD thesis, CSE, UNSW, Sydney, Australia, mar 2015.
13. D. Greenaway, J. Andronick, and G. Klein. Bridging the gap: Automatic verified abstraction of C. In *ITP*, pages 99–115. Springer, aug 2012.
14. F. Haftmann. *Code Generation from Specifications in Higher Order Logic.* PhD thesis, Technische Universität München, 2009.
15. F. Haftmann and T. Nipkow. Code generation via higher-order rewrite systems. In *FLOPS 2010*, LNCS. Springer, 2010.
16. A. Krauss. Recursive definitions of monadic functions. In *Proc. of PAR*, volume 43, pages 1–13, 2010.
17. P. Lammich. Refinement for monadic programs. In *Archive of Formal Proofs*. `http://afp.sf.net/entries/Refine_Monadic.shtml`, 2012. Formal proof development.
18. P. Lammich. Verified efficient implementation of Gabow's strongly connected component algorithm. In *ITP*, volume 8558 of *LNCS*, pages 325–340. Springer, 2014.
19. P. Lammich. Refinement to Imperative/HOL. In *ITP*, volume 9236 of *LNCS*, pages 253–269. Springer, 2015.
20. P. Lammich. Refinement based verification of imperative data structures. In *CPP*, pages 27–36. ACM, 2016.
21. 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.
22. P. Lammich and T. Tuerk. Applying data refinement for monadic programs to Hopcroft's algorithm. In *Proc. of ITP*, volume 7406 of *LNCS*, pages 166–182. Springer, 2012.
23. G. Lee. Correctnesss of ford-fulkerson's maximum flow algorithm1. *Formalized Mathematics*, 13(2):305–314, 2005.
24. G. Lee and P. Rudnicki. Alternative aggregates in mizar. In *Calculemus '07 / MKM '07*, pages 327–341. Springer, 2007.
25. R. Matuszewski and P. Rudnicki. Mizar: the first 30 years. *Mechanized Mathematics and Its Applications*, page 2005, 2005.
26. MLton Standard ML compiler. http://mlton.org/.
27. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
28. 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.
29. L. Noschinski. *Formalizing Graph Theory and Planarity Certificates.* PhD thesis, Fakultät für Informatik, Technische Universität München, November 2015.
30. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc of. Logic in Computer Science (LICS)*, pages 55–74. IEEE, 2002.
31. R. Sedgewick and K. Wayne. *Algorithms.* Addison-Wesley, 2011. 4th edition.
32. M. Wenzel. Isar - A generic interpretative approach to readable formal proof documents. In *TPHOLs'99*, volume 1690 of *LNCS*, pages 167–184. Springer, 1999.
33. N. Wirth. Program development by stepwise refinement. *Commun. ACM*, 14(4), Apr. 1971.
34. U. Zwick. The smallest networks on which the Ford-Fulkerson maximum flow procedure may fail to terminate. *Theoretical computer science*, 148(1):165–170, 1995.