Formalizing the Edmonds-Karp Algorithm

S. Reza Sefidgar, Peter Lammich

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

Abstract. We present a formalization of the Ford-Fulkerson method for computing the maximum flow in a network. The formal proofs are done in Isabelle/HOL. They closely follow a standard textbook proof and are relatively easy to read, even without being an Isabelle expert.
We then use stepwise refinement to obtain the Edmonds-Karp algorithm and prove its upper complexity bound of $O(|V||E|^2)$. Further refinement yields a verified implementation, which is almost as fast as an unverified reference implementation in Java.

1 Introduction

The Ford-Fulkerson algorithm [cite] is one of the important results of the graph theory and is used to find a solution to the maximum flow problem in flow networks. Many important mathematical problems like the maximum-bipartite-matching problem, the edge-disjoint-paths problem, the circulation-demand problem, and many other scheduling and resource allocating problems can be reduced to the maximum flow problem. Hence, the Ford-Fulkerson algorithm has major application in the field of mathematical optimization.

Despite its importance, no formalization of the Ford-Fulkerson algorithm has been developed in modern proof assistants. The only similar work that the authors are aware of is formalization of the Ford-Fulkerson algorithm in Mizar [cite]. This formalization defines and proves correctness of the algorithm at the level of graph manipulations without providing concrete implementation of the algorithm. Providing such an implementation is specially important, as it could provide the basis for many verfied programs with practical importance.

This paper present a formalization of the correctness proof of the Ford-Fulkerson algorithm in Isabelle/HOL. Our proof is based on the informal proof of the algorithm which is presented in the book "Introduction to algorithms" [cite]. Due to practical importance of the algorithm, we also present a verified implementation of the algorithm. Isabelle/HOL provides some automation for generating the code corresponding to a verified algorithm, however, in order to use the code generating features, one needs to do the formalization with executability in mind. Being limited only to executable concepts in the formalization has the disadvantage of cluttering the proofs with implementation details. Such approach makes the proofs more complicated, and may even render proofs of medium complex algorithms unmanageable.

One solution for the aforementioned problem is refinement [cite]. In order to generate executable code of an algorithm using refinement, we first formulate

the algorithm on an abstract level. The abstract version of the algorithm has a clean correctness proof as it only captures the idea behind the algorithm. Next, we refine the abstract definition of the algorithm towards an executable implementation in possibly multiple refinement steps. During each step, we only need to prove the correctness of the implementation of a particular abstract concept. Hence, we can be sure about the correctness of the resulting executable program as each refinement step preserves the correctness.

There are several approaches to data refinement in Isabelle/HOL. We will be using the Autoref tool [cite], which has been used for proving more complex results such as formalized implementation of Hopcroft's DFA-minimization algorithm [cite]. Given an algorithm phrased over abstract concepts like sets and maps, it automatically synthesizes a concrete, executable algorithm and the corresponding refinement theorem. To make it applicable for the development of actual algorithms, Autoref is integrated with the Isabelle Refinement Framework [cite] and the Isabelle Collection Framework [cite].

2 Short Background on MinCutMaxFlow and FoFu

A flow network is a weighted digraph where the set of vertices (nodes) V is finite, and each edge (u, v) has a real capacity $c(u, v) \geq 0$. There are two distinct vertices s (source) and t (sink) in the flow network. In this case the network is called an s-t network. To make the formalization simple, some additional assumptions are added to the definition as following: 1) The source only has outgoing edges while the sink only has incoming edges; 2) If the flow network contains an edge (u, v) then there is no edge (v, u) in the reverse direction; and 3) every vertex of the flow network must be on a path from s to t. Although these assumptions seems restrictive, any network may be transformed such that it satisfies all the aforementioned properties.

An s-t flow on a weighted digraph is a function $: V \times V \to \mathbb{R}$ which satisfies the following conditions. 1) **Capacity constraint**: the flow of each edge is a non-negative value which is smaller or equal to the edge capacity; 2) **Conservation constraint**: For all vertices except s and t, the sum of flows over incoming edges of the vertex is equal to the sum of flows over outgoing edges of the vertex.

The value of an s-t flow f is denoted by $|f|$, and is computed by subtracting the sum of the flows over all incoming edges of s from the sum of the flows over all outgoing edges of s. Given an s-t network G, the maximum flow problem involves finding a flow f in G such that the value of f is greater or equal to the value of any other s-t flow in G [1].

One of the important results in mathematical optimization is the correlation between flows and cuts in flow networks. An s-t cut in a flow network with source s and sink t is a partition of vertices that puts s and t in different subsets. Then the capacity of a cut is defined as sum of the edge capacities over all edges going from the source's side to the sink's side. Consider a valid flow in a flow network

[1] Note that the value of the flow in our model of a flow network equals sum of flows over edges going out of s. Because the model prohibits edges coming into the source.

G. For any cut in G, the flow that crosses between the two subsets of the cut cannot exceed the capacity of the cut. So cuts define an upper bound for any flow in the flow network. 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 algorithm is the direct consequence of the Ford-Fulkerson theorem. It solves the maximum flow problem in a greedy approach. Assume an instance of the maximum flow problem in an s-t network. The algorithm starts with a zero flow: $f(e) = 0$ for all the edges e. In each iteration of the algorithm the value of f is increased by pushing flow along a path from s to t up to the limits imposed by the available edge capacities. The process would continue until no more paths can be found to push additional flow from s to t.

The total flow value is increased by manipulating the flow values along different edges. However, the flow along specific edges might decrease during this process. In order to perform these operations, the Ford-Fulkerson algorithm defines the residual graph of the flow network. Intuitively, the residual graph corresponding to a flow in a given flow network, has the same vertices as the flow network. It also and consists of edges with capacities representing how flow can be changed along the edges of the flow network.

Assume a flow f in an s-t network with capacity function c. An edge (u, v) of the network can admit an additional amount of flow equal to the edge's capacity minus the flow along that edge. If this value is positive, then a (u, v) edge is added to the residual graph with a residual capacity $c_f(u, v) = c(u, v) - f(u, v)$. So, the residual graph only contains those edges of the original flow network that have left-over capacities. The residual graph may also contain some edges that do not exist in the original flow network. For each edge (u, v) in the flow network which has a positive flow value, a (v, u) edge is added to the residual graph with the residual capacity $c_f(v, u) = f(u, v)$. The edge (v, u) indicates the possibility for pushing the flow backward along the edge (u, v) in the original flow network.

After each update of the flow f, the algorithm considers the new residual graph G_f corresponding to that flow. Then the algorithm looks for an augmenting path— a simple path from source to the sink— in the residual graph. If such a path could be found, it will be used to construct a flow in the residual graph. This flow provides a road map for modifying the current flow in the flow network. The process of increasing the current flow using an augmenting flow is called augmentation. Let f be the current flow in an s-t network $G = (V, E)$ with capacity function c, and f be a flow in the corresponding residual graph. The augmentation of f using f' is a function from $V \times V$ to $\mathbb{R}$, defined as following:

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

The intuition behind the above definition is based on the definition of the residual graph. The augment procedure increases the flow on (u, v) by $f(u, v)$ but decreases it by $f(v, u)$. This is due to the fact that pushing flow on the reverse edge on the residual graph signifies decreasing the flow in the original network. In each iteration of the algorithm we replace the current flow with the result of the

augment function, and the algorithm terminates if no more augmenting paths can be found in the residual graph. According to the Ford-Fulkerson theorem, the flow computed in this states is indeed the maximum flow in the flow network. The theorem states that following statements are equivalent:

- f is a maximum s-t flow in flow network G.
- there is no augmenting path in the residual graph G_f.
- there is an s-t cut C in G such that capacity of C is equal to the value of f.

The termination of the Ford-Fulkerson algorithm depends on how the augmenting path is found. For the networks with irrational edge capacities the algorithm might even fail to terminate. In practice, the maximum-flow problem often arises with integral capacities. Moreover, we may convert rational edge capacities to integral values by multiplying them with a big enough integer. If f^* denotes a maximum-flow in the transformed network, then the algorithm will iterate at most $|f^*|$ times, since the augmentation procedure increases the flow value by at least one unit in each iteration. Assuming that we use breadth-first-search (BFS) or depth-first-search (DFS) for finding augmenting paths makes the total execution time of the Ford-Fulkerson algorithm $O(E|f^*|)$, as BFS and DFS run in $O(E+V)$.

The Ford-Fulkerson algorithm is sometimes called a method instead of an algorithm because it does not fully specify the procedure for finding the augmenting path in the residual graph. There are several implementations with different execution times. For instance, the Edmond-Karp algorithm uses breadth-first-search (BFS) for finding augmenting paths and has $O(VE^2)$ running-time. The augmentings paths that are computed using BFS are also shortest paths connecting source to the sink in the residual graph. Edmond and Karp showed that in the Ford-Fulkerson algorithm, if each augmenting path is the shortest one, the length of the augmenting paths is non-decreasing and it always terminates. Which finally gave the polynomial algorithm for solving the maximum flow problem in generic case of real edge capacities.

3 Formalizing MinCutMaxFlow and Fofu

```
context NFlow
begin
  lemma augment_flow_presv_cap: "Flow cf s t f' ⟹ ∀e. 0 ≤ (augment
f') e ∧  (augment f') e ≤ c e"
proof -
      assume asm: "Flow cf s t f'"
      {
        fix e
        have "augment f' e ≤ c e"
          proof -
            obtain u v where obt: "e = (u, v)" by (metis nat_gcd.cases)
            thus ?thesis (is "?L ≤ _")
              proof (cases "(u, v) ∈ E")
```

```
                case True
                  have "f' (v, u) ≥ 0" using asm Flow_def by simp
                  then have "?L ≤ f (u, v)  + f' (u, v)"
                    unfolding augment_def using True obt by auto
                  moreover have "f' (u, v) ≤ cf (u, v)" using asm
Flow_def by auto
                  ultimately have fct: "?L ≤ f (u, v) + cf (u, v)"
by simp
                  have "cf (u, v) = c (u, v) - f(u, v)"
                    unfolding residualGraph_def using True by auto
                  thus ?thesis using fct obt by simp
                next
                  case False
                  then have fct: "augment f' (u, v) = 0" unfolding
augment_def by simp
                  thus ?thesis using cap_positive obt by simp
                qed
              qed
            moreover have "0 ≤ augment f' e"
              proof -
                obtain u v where obt: "e = (u, v)" by (metis nat_gcd.cases)
                then have "f' (u, v) ≥ 0" using asm Flow_def by simp
                moreover have "f (u, v) ≥ 0" using capacity_const by
simp
                ultimately show ?thesis unfolding augment_def using reverse_flow[OF
asm] obt by auto
              qed
            ultimately have "0 ≤ (augment f') e ∧  (augment f') e ≤ c
e" by blast
          }
          thus ?thesis by simp
        qedend
```

4 Refinement in Isabelle/HOL

When formalizing algorithms, there is often a trade off between verifiability and efficiency: An efficient algorithm that uses elaborate data structures and optimizations is often harder to verify than a simple but straightforward algorithm.

The idea of stepwise refinement [**?**] is to first specify a simple, abstract version of an algorithm, which can be easily verified, and then refine this algorithm towards an efficient implementation. Here, each refinement step can focus on a single aspect of the algorithm or its implementation, independently of the other refinement steps. This results in a greatly increased modularity of proofs. Refinement calculi [**?**] are used to systematically perform refinement in a Hoare-logic setting, and are particularly well-suited for implementation in theorem provers.

Note that it is important to support nondeterminism in a refinement setting, as this allows to defer implementation choices to the later refinement steps. For

example, if we want to specify an algorithm that returns a shortest path between two nodes of a graph, we do not want to fix a particular shortest path. Only later, when we decide to implement the specification by, e.g., breadth first search, we fix a particular path.

In Isabelle/HOL, stepwise refinement is supported by the Isabelle Refinement Framework [2]. It features a refinement calculus for programs phrased in a nondeterminism monad. The monad's type is a set of results plus an additional value that indicates a failure:

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

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

On nondeterministic results we define the *refinement ordering* 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 this ordering is a complete lattice, and bind is monotonic. Thus, we can define recursion using a fixed point construction [1]. 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 constants for loop constructs (while, foreach), assertions and specifications, and use a Haskell-like do-notation:

assert Φ = **if** Φ **then return** () **else fail**
spec *x*. $P\ x$ = *RES* $\{x.\ P\ x\}$
do $\{x \leftarrow m;\ f\ x\}$ = **bind** $m\ f$
do $\{m;\ m'\}$ = **bind** $m\ (\lambda\_.\ m')$

Correctness of a program m with precondition P and postcondition Q is expressed as $P \Longrightarrow m \leq$ **spec** *r*. $Q\ r$, 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**, while a nonterminating partial correct recursion yields **res** {}, which is the least element of the refinement ordering.

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

Finally, the Isabelle Refinement Framework provides a refinement calculus, which comes with a verification condition generator to simplify its usage.

In a typical program development using the Isabelle refinement framework, one first comes up with an initial version m_0 of the algorithm and its specification P, Q, and shows $P \Longrightarrow m_0 \leq$ **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 $P \Longrightarrow m_i \leq \Downarrow R_i \ldots R_1$ **spec** Q, showing the correctness of the refined algorithm.

xxx, ctd here ... Refinement itself can be modular ... sub-chains of refinement to refine one part of algo. Monotonicity to combine.

5 Edmonds-Karp Algorithm

The general Ford-Fulkerson method can only be shown to terminate if the edge capacities are integer numbers. An improvement over that is the Edmonds-Karp algorithm [], which is obtained when always choosing a shortest augmenting path. In this case, it can be shown that only $O(|V||E|)$ augmentations are performed until the algorithm terminates, even for real-valued capacities. A shortest augmenting path can be obtained by breadth first search in time $O(|E|)$, yielding an overall complexity of $O(|V||E|^2)$ algorithm.

In our formalization, we refine the specification of an augmenting path to a specification of a shortest augmenting path, which immediately yields an abstract version of the Edmonds-Karp algorithm which refines the Ford-Fulkerson method.

The larger part of the formalization is spent on proving the complexity bound. Note that the refinement framework does not have a notion of computational complexity, so we cannot even define the runtime of an algorithm. However, we can instrument the while-loop of the algorithm with a counter, which is incremented on each iteration, and prove an upper bound on this counter. Moreover, this also yields a termination argument for real-valued capacities.[2]

The idea of the complexity proof is as follows: Note that edges that are reverse to edges on a shortest augmenting path cannot lie on a shortest path itself. On every augmentation, at least one edge that lies on a shortest path is flipped. Thus, either the length of the shortest path increases, or the number of edges that lie on a shortest path decreases. As the length of a shortest path is at most $|V|$, there are no more than $O(|E||V|)$ iterations.

Formalizing the above intuitive argument is more tricky than it seems on first glance. While it easy to prove that, in a *fixed graph*, an edge and its reverse cannot both lie on shortest paths, generalizing the argument to a graph transformation which may add reverse edges and removes at least original edge, is tricky. Note that a straightforward induction on the length of the augmenting path must fail, as after flipping the first edge, the path is no longer augmenting.

We decided to generalize the statement as follows. Assume we have an original graph cf with a shortest augmenting path p, and a transformed graph cf' that has been created from cf by adding some flipped edges from p and removing some edges from p. Then, we consider an edge $(u, v) \in p$, a path p_1 from s to v in the original graph, and a path p_2 from u to t in the transformed graph.

By induction on the number of reversed edges in p_2, we show that $|p_1| + |p_2| > |p|$. In the induction step, we use the proof idea for the single flipped edge (v, u), and, if p_2 contains more flipped edges, we split p_2 at the first flipped edge. Then,

[2] As we restricted the algorithm to integer valued capacities at the very abstract level, this is actually not used.

the initial segment of p_2 contains no flipped edges, and thus also is a path in the original graph, and we can apply the induction hypothesis.

Proving that augmentation with a shortest augmenting path actually only adds flipped edges, and removes at least one original edge, required some more work, but was straightforward and yielded no surprises.

Finally, we phrase the complexity argument as a measure function $m = l * 2|E| + n$, where l is the length of the shortest augmenting path, and n is the number of edges that lie on any shortest path. We show that m is decreased by the loop body. Adding a special case for loop termination (there are no augmenting paths), and observing that m is bounded by $2|V||E|$, yields the desired upper bound on loop iterations. Refinement of the algorithm to add an explicit loop counter, and asserting the upper bound, is then straightforward.

6 Refinement to Executable Code

GOAL: Same as previous section

In the previous sections, we have described the Ford-Fulkerson and the Edmonds-Karp algorithm abstractly, leaving open how to obtain a (shortest) augmenting path. 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 refinement framework. However, the existing formalization could only compute the minimum distance between two nodes, without returning an actual path. We briefly report on our adapted formalization here, which is displayed in Figure **??**: TODO: Reference to figure (line numbers?) The abstract algorithm keeps track of a set C of current nodes, a set N of new nodes, and a partial map P from already visited nodes to predecessor nodes. Initially, only the start node is in C and N is empty. In each iteration of the loop, a node $u \in C$ is picked, and its successors v that have not yet been visited are added to N, and $P\ v$ is set to u. If C becomes empty, C and N are swapped. If the target node is encountered, the algorithm immediately terminates. Note that this implementation is a generalized version of the usual queue implementation of BFS: While a queue enforces a complete order on the encountered nodes, our implementation only enforces an ordering between nodes on the current level and nodes on the next level.

For the actual algorithm, we wrap this algorithm by a procedure to handle the special case where source and target nodes are the same, and to extract a shortest path from P upon termination. Moreover, we implement the abstract specification of adding the successor nodes of a node by a loop.

Finally, we prove the following theorem:

Obviously, the BFS algorithm refines the specification for obtaining a shortest path. Using this to refine the Edmonds-Karp formalization yields an algorithm that algorithmically specifies all major operations.

Down to executable code:

In a next step, we observe that the algorithm is phrased in terms of a flow which is updated until it is maximal. However, in order to update the flow, the

residual graph is used, which is a combination of the current flow and the network. Obviously, computing the complete residual graph each time before searching for an augmenting path would be a bad idea. One solution to this problem is to compute the edges of the residual graph on demand 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 especially time critical.

Thus, we chose to let the algorithm operate on a representation of the residual graph directly.

The shortest path specification and bottleneck computation are already phrased on residual graphs. What remains is the computation of the initial residual graph (which equals the Network), the augmentation of the residual graph, given an augmenting path and its bottleneck capacity, and the computation of the resulting maximum flow from the residual graph, at the end of the algorithm.

In order to refine these operations, we note that the residual graph uniquely determines the flow (and vice versa).

For a valid flow f, we have that

Thus, augmentation can be expressed directly on the residual graph:

Having these characterizations, a straightforward refinement yields an algorithm that operates on residual graphs.

In a next step, we have to specify iterative implementations for finding the bottleneck capacity and doing the augmentation. This is straightforward by folding over the path, exploiting that an augmenting path is simple, and thus never contains an edge and its flipped edge at the same time.

By combination of c and f, and pred-succ. TODO: Directly! All we need is successor function. Implemented by tabulating adjacent nodes of each node (pred-succ), and using c and f to filter out actual successor nodes. augpath-spec by BFS/DFS (which work on succ-functions) bottleneck and augment: Give iterative impl.

In order to find an augmenting path, the BFS algorithm has to compute the successors of a node. Although this can be implemented on the capacity matrix by iterating over all nodes, this implementation tends to be inefficient for sparser graphs. A common optimization is to pre-compute a map from nodes to adjacent nodes in the network. As an edge in the residual graph is either parallel or reverse to a network edge, it is enough to iterate over the adjacent nodes in the network, and check whether they are actually successors in the residual graph. It is straightforward to show that this implementation actually returns the successors in the residual graph:

Use efficient data structures: Now we have an abstract version of the algorithm that only relies on executable concepts. However, to yield an efficient algorithm, we have to choose efficient data structures.

We assume that the nodes are natural numbers less than N. Then we implement the capacity matrix of the residual graph by an array with row-major indexing. The adjacency map for network nodes is implemented as an array of

lists. In the BFS algorithm, we use two lists, one that holds the nodes of the current level, and another in which nodes of the next level are collected. The predecessor map is implemented as an array.

The input of the algorithm is still a function mapping network edges to capacities, which is tabulated once to obtain the initial residual graph. This ensures some flexibility in using the algorithm, without preventing efficient implementations (The function can be based on array lookup itself[3]).

The output of the algorithm is just the residual graph. The user can decide how he computes 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.

The last refinement step was performed using the Sepref tool, which supports data refinements from functional to imperative algorithms in Isabelle/HOL. [Few sentences on Sepref]

Arrays for c,f, and ps. c and f in row-major indexing.

List for augmenting path. [TODO/Future work: Iterator scheme]

In BFS, we use lists for the queues (we have split the queue into one for the current level, and one for the next level), and an array for the predecessor map.

The refinement is performed using the Sepref tool.

NetCheck We also implemented an algorithm that reads a list of edges and a source and target node, converts it 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 corresponding Network and adjacency function 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 algorithm, for which we can export code, and have proved the theorem: [...].

7 Benchmarking

What implementations did we compare: Authors? Where do implementations come from? [We must convince the reader that we did not intentionally chose bad implementations to compare us against!]

Comparsion of algorithms (Modified data structure: Flow vs Flow-like vs ResGraph) Computing successors in BFS: Filter by visited set first. -¿ Technically more challenging, when computing on flow: 1 access first vs 2 accesses first. On resGraph: No advantage in memory accesses (However, plain array access vs. matrix access) Computing bottleneck during BFS. Saves one iteration over the path, at the cost of one extra memory access per discovered node. For our

[3] Due to technical limitations of our tools, this function cannot depend on heap content itself, but it can use the efficient functional and pseudo-functional datastructures provided by the Collection Framework.

benchmarks, we observed that the BFS discovers considerably more nodes than the length of the ultimately returned path, and the overall code was faster with the extra iteration for bottleneck computation.

8 Conclusion

... and related work Mizar-Formalization: What did they do.

History of the formalization: Impl: From first impl that was hardly able to compute a 100 nodes graph, to current efficient one.

Future Work: Dinic. (Actually, our abstract scheme [almost] covers Dinic's algorithm!)

Contributions

Formal proof of mincut maxflow fofo-scheme inst to edmonds karp complexity analysis of edmonds karp refinement down to executable code. Roughly 2.5 times slower than Java. (What about OCaml) + NetCheck

What shines (its a Pearl) Min-Cut Max-Flow: Textboook like formal reasoning: Comprehensible proof, BUT machine checked (Present one (carefully worked) example in paperI. We could use lemma *augment_flow_presv_cap*)

Refinement based approach: Fofu-Scheme, instantiation to EdsKa. +1: Abstract Algo looks almost like pseudo-code you would expect in textbook. +2: Fofu-Scheme proved correct for all aug-path finders. EdsKa is instantiation of it. +3: Modularity: Fofu-scheme and pathfinder developed+proved independently of each other.

Down to executable code, plugging in efficient data structures.

Some minor contributions: Reusable BFS algorithm Imperative matrix data structure (really minor).

References

1. A. Krauss. Recursive definitions of monadic functions. In *Proc. of PAR*, volume 43, pages 1–13, 2010.
2. 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.