Formalizing the Edmonds-Karp Algorithm

Peter Lammich and S. Reza Sefidgar

TU München

August 2016

Maximum Flow Problem

- Network: Digraph with edge capacities and source/sink nodes
- Flow: From source to sink, not exceeding capacities
  - Kirchhoff's law: Inflow = Outflow for all nodes but source/sink
  - No inflow to source, no outflow from sink
  - Value: Flow transported from source to sink (=Outflow of source)
- Problem: Given a network, find a flow with maximal value

Ford-Fulkerson Method

- Consider network with flow
  - No antiparallel edges: $u \longrightarrow v \Longrightarrow v \not\longrightarrow u$
- Residual graph
  - For network edge $u \xrightarrow{c,f} v$, residual graph has

$$u \xrightarrow{c-f} v \text{ and } v \xrightarrow{f} u$$

  - Intuition: Flow that can be moved between nodes
    - By either increasing or decreasing flow on network edge
- Ford-Fulkerson
  - Flow is maximal iff no path from source to sink in residual graph
  - Corollary of min-cut max-flow theorem

Ford-Fulkerson Method

Flow is maximal iff no path from source to sink in residual graph.

- Greedy Algorithm

  Set flow to zero
  while *exists augmenting path*
    augment flow along path

- Partial correctness: obvious
- Termination: Only for integer/rational capacities
- Edmonds/Karp: Choose shortest augmenting path
  - $O(VE)$ iterations for real-valued capacities
  - Using BFS to find path: $O(VE^2)$ algorithm

Our Contributions

Verified in Isabelle/HOL

- Min-Cut Max-Flow theorem
  - Human-Readable Isar proof
  - Closely following Cormen et al.
- Ford-Fulkerson and Edmonds Karp algorithms
  - Human-readable presentation of algorithms
  - Proved Correctness and Complexity
- Efficient Implementation
  - Using stepwise refinement down to Imperative/HOL
  - Isabelle's code generator exports to SML
  - Benchmark: Comparable to Java (from Sedgewick et al.)

Human-Readable Proofs

- Tried to use Isar in readable way

Human-Readable Proofs

- Tried to use Isar in readable way
  Proof fragment from Cormen at al.:

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

Human-Readable Proofs

- Tried to use Isar in readable way
  Proof fragment from Cormen at al.:

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

Our Isar version:

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) **.**

And Automatic Proofs

- Cormen et al. also give more complicated proofs

First part of proof that $|f \uparrow f'| = |f| + |f'|$:

$$
\begin{aligned}
|f \uparrow f'| \\
&= \sum_{v \in V_1} (f(s,v) + f'(s,v) - f'(v,s)) - \sum_{v \in V_2} (f(v,s) + f'(v,s) - f'(s,v)) \\
&= \sum_{v \in V_1} f(s,v) + \sum_{v \in V_1} f'(s,v) - \sum_{v \in V_1} f'(v,s) \\
&\qquad - \sum_{v \in V_2} f(v,s) - \sum_{v \in V_2} f'(v,s) + \sum_{v \in V_2} f'(s,v) \\
&= \sum_{v \in V_1} f(s,v) - \sum_{v \in V_2} f(v,s) \\
&\qquad + \sum_{v \in V_1} f'(s,v) + \sum_{v \in V_2} f'(s,v) - \sum_{v \in V_1} f'(v,s) - \sum_{v \in V_2} f'(v,s) \\
&= \sum_{v \in V_1} f(s,v) - \sum_{v \in V_2} f(v,s) + \sum_{v \in V_1 \cup V_2} f'(s,v) - \sum_{v \in V_1 \cup V_2} f'(v,s) . \qquad (26.6)
\end{aligned}
$$

And Automatic Proofs

- Cormen et al. also give more complicated proofs
- We sometimes chose to use more automatic proofs

lemma augment_flow_value: Flow.val c s (f↑f') = val + Flow.val cf s f'
proof -
  interpret f'': Flow c s t f↑f' **using** augment_flow_presv[OF assms] **.**

And Automatic Proofs

- Cormen et al. also give more complicated proofs
- We sometimes chose to use more automatic proofs
    - Using some simplifier setup

```
note setsum_simp_setup[simp] =
  sum_outgoing_alt[OF capacity_const] s_node
  sum_incoming_alt[OF capacity_const]
  cf.sum_outgoing_alt[OF f'.capacity_const]
  cf.sum_incoming_alt[OF f'.capacity_const]
  sum_outgoing_alt[OF f''.capacity_const]
  sum_incoming_alt[OF f''.capacity_const]
  setsum_subtractf setsum.distrib
```

And Automatic Proofs

- Cormen et al. also give more complicated proofs
- We sometimes chose to use more automatic proofs
  - Using some simplifier setup
  - And auxiliary statements

```
have aux1: f'(u,v) = 0 if (u,v)∉E (v,u)∉E for u v
proof -
  from that cfE_ss_invE have (u,v)∉cf.E by auto
  thus f'(u,v) = 0 by auto
qed
```

And Automatic Proofs

- Cormen et al. also give more complicated proofs
- We sometimes chose to use more automatic proofs
  - Using some simplifier setup
  - And auxiliary statements
  - We reduce the displayed proof's complexity

have f''.val = ($\sum$ u$\in$V. augment f' (s, u) - augment f' (u, s))
  unfolding f''.val_def **by** simp
also have . . . = ($\sum$ u$\in$V. f (s, u) - f (u, s) + (f' (s, u) - f' (u, s)))
  — Note that this is the crucial step of the proof, which Cormen et al. leave as an exercise.
  by (rule setsum.cong) (auto simp: augment_def no_parallel_edge aux1)
also have . . . = val + Flow.val cf s f'
  unfolding val_def f'.val_def **by** simp
finally show f''.val = val + f'.val **.**

qed

Main Result

- Finally, we arrive at

context NFlow **begin**
...
theorem ford_fulkerson: **shows**
isMaxFlow f $\longleftrightarrow$ $\neg$ Ex isAugmentingPath

Ford-Fulkerson Method

- We use the Isabelle Refinement Framework

Ford-Fulkerson Method

- We use the Isabelle Refinement Framework
  - Based on nondeterminism monad + refinement calculus
  - Provides proof tools + Isabelle Collection Framework

Ford-Fulkerson Method

- We use the Isabelle Refinement Framework
  - Based on nondeterminism monad + refinement calculus
  - Provides proof tools + Isabelle Collection Framework

```
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
}
```

Correctness Proof

- First, we add some assertions and invariant annotations

```
definition fofu ≡ do {
  let f = (λ_. 0);

  (f,_) ← while^fofu_invar
    (λ(f,brk). ¬brk)
    (λ(f,_). do {
      p ← find_augmenting_spec f;
      case p of
        None ⇒ return (f,True)
      | Some p ⇒ do {
          assert (p≠[]);
          assert (NFlow.isAugmentingPath c s t f p);
          let f' = NFlow.augmentingFlow c f p;
          let f = NFlow.augment c f f';
          assert (NFlow c s t f);
          return (f, False)
        }
    })
    (f,False);
  assert (NFlow c s t f);
  return f
}
```

Correctness Proof

- First, we add some assertions and invariant annotations
- Then, we use the VCG to prove partial correctness

```
theorem fofu_partial_correct: fofu ≤ (spec f. isMaxFlow f)

unfolding fofu_def find_augmenting_spec_def
  apply (refine_vcg)
  apply (vc_solve simp:
    zero_flow
    NFlow.augment_pres_nflow
    NFlow.augmenting_path_not_empty
    NFlow.noAugPath_iff_maxFlow[symmetric])
  done
```

Correctness Proof

- First, we add some assertions and invariant annotations
- Then, we use the VCG to prove partial correctness
- This also yields correctness of the unannotated version

theorem (**in** Network) ford_fulkerson_method $\leq$ (**spec** f. isMaxFlow f)

Edmonds-Karp Algorithm

- Specify shortest augmenting path

definition find_shortest_augmenting_spec f $\equiv$ **assert** (NFlow c s t f) $\gg$
(**selectp** p. Graph.isShortestPath (residualGraph c f) s p t)

Edmonds-Karp Algorithm

- Specify shortest augmenting path
- This is a refinement of augmenting path

lemma find_shortest_augmenting_refine[refine]:
(f',f)$\in$Id $\Longrightarrow$ find_shortest_augmenting_spec f' $\leq$ $\Downarrow$Id (find_augmenting_spec f)

Note: This is verbose boilerplate for
find_shortest_augmenting_spec $\leq$ find_augmenting_spec

Edmonds-Karp Algorithm

- Specify shortest augmenting path
- This is a refinement of augmenting path
- Replace in algorithm

definition fofu $\equiv$ **do** {

...
p $\leftarrow$ find_augmenting_spec f;
...

Edmonds-Karp Algorithm

- Specify shortest augmenting path
- This is a refinement of augmenting path
- Replace in algorithm

definition edka_partial $\equiv$ **do** {

...
p $\leftarrow$ find_shortest_augmenting_spec f;
...

Edmonds-Karp Algorithm

- Specify shortest augmenting path
- This is a refinement of augmenting path
- Replace in algorithm
- New algorithm refines original one

lemma edka_partial_refine[refine]: edka_partial $\leq \Downarrow$Id fofu

unfolding find_shortest_augmenting_spec_def find_augmenting_spec_def
  apply (refine_vcg)
  apply (auto
    simp: NFlow.shortest_is_augmenting
    dest: NFlow.augmenting_path_imp_shortest)
  done

Total Correctness and Complexity

- Next, we define a total correct version

definition edka_partial $\equiv$ **do** {
...
(f,_) $\leftarrow$ **while**$^{\mathit{fofu\_invar}}$

...

Total Correctness and Complexity

- Next, we define a total correct version

definition edka $\equiv$ **do** {
...
(f,_) $\leftarrow$ $\textbf{while}_T{}^{\textit{fofu\_invar}}$

...

Total Correctness and Complexity

- Next, we define a total correct version
- And show refinement

theorem edka_refine[refine]: edka $\leq$ $\Downarrow$Id edka_partial

Total Correctness and Complexity

- Next, we define a total correct version
- And show refinement
- We also show $O(VE)$ bound on loop iterations
  - Instrumenting the loop with a counter

Towards Efficient Implementation

Several refinement steps lead to final implementation:

Towards Efficient Implementation

Several refinement steps lead to final implementation:

1. Update residual graphs instead of flows

Towards Efficient Implementation

Several refinement steps lead to final implementation:

1. Update residual graphs instead of flows
2. Implement augmentation (iterate over path twice)

Towards Efficient Implementation

Several refinement steps lead to final implementation:

1. Update residual graphs instead of flows
2. Implement augmentation (iterate over path twice)
3. Use BFS to determine shortest augmenting path

Towards Efficient Implementation

Several refinement steps lead to final implementation:

1. Update residual graphs instead of flows
2. Implement augmentation (iterate over path twice)
3. Use BFS to determine shortest augmenting path
4. Implement successor function on residual graph
    - Using pre-computed map of adjacent nodes in network

Towards Efficient Implementation

Several refinement steps lead to final implementation:

1. Update residual graphs instead of flows
2. Implement augmentation (iterate over path twice)
3. Use BFS to determine shortest augmenting path
4. Implement successor function on residual graph
   - Using pre-computed map of adjacent nodes in network
5. Imperative Data Structures
   - Tabulate capacity matrix and adjacency map to array
   - Maintain residual graph in array

Towards Efficient Implementation

Several refinement steps lead to final implementation:

1. Update residual graphs instead of flows
2. Implement augmentation (iterate over path twice)
3. Use BFS to determine shortest augmenting path
4. Implement successor function on residual graph
   - Using pre-computed map of adjacent nodes in network
5. Imperative Data Structures
   - Tabulate capacity matrix and adjacency map to array
   - Maintain residual graph in array
6. Export to SML code

Assembling Overall Correctness proof

- Correctness statement
  - As Hoare-Triple using Separation Logic

```
context Network_Impl begin
 theorem edka_imp_correct:
  assumes VN: Graph.V c ⊆ {0..<N}
  assumes ABS_PS: is_adj_map am
  shows
   <emp>
    edka_imp c s t N am
   <λfi. ∃_A f. is_rflow N f fi * ↑(isMaxFlow f)>_t
```

Assembling Overall Correctness proof

- Correctness statement
  - As Hoare-Triple using Separation Logic
- Proof by transitivity

```
proof -
    interpret Edka_Impl by unfold_locales fact

    note edka5_refine[OF ABS_PS]
    also note edka4_refine
    also note edka3_refine
    also note edka2_refine
    also note edka_refine
    also note edka_partial_refine
    also note fofu_partial_correct
    finally have edka5 am ≤ SPEC isMaxFlow .
    from hn_refine_ref[OF this edka_imp_refine]
    show ?thesis
      by (simp add: hn_refine_def)
qed
```

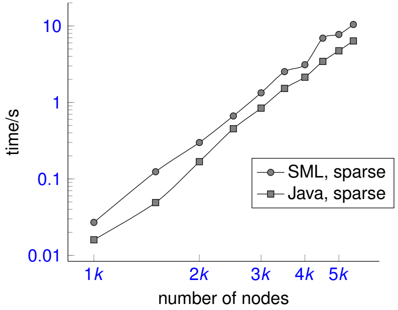
Assembling Overall Correctness proof

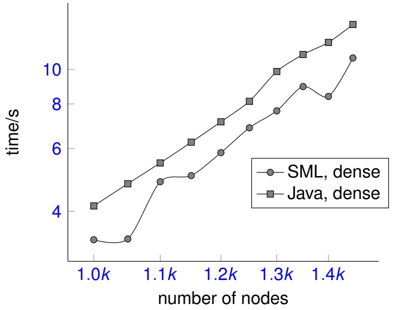
- Correctness statement
  - As Hoare-Triple using Separation Logic
- Proof by transitivity
- Also integrated with check for valid network
  - Input is list of edges, source, and sink

theorem
 fixes el **defines** c $\equiv$ ln_α el
 shows <emp> edmonds_karp el s t <λ
  None $\Rightarrow$ ↑(¬ln_invar el ∨ ¬Network c s t)
 | Some (N,cf) $\Rightarrow$
  ↑(ln_invar el ∧ Network c s t ∧ Graph.V c ⊆ {0..<N})
 * ($\exists_A$ f. is_rflow c N f cf * ↑(Network.isMaxFlow c s t f))>$_t$

Benchmarking

- Against Java version of Sedgewick et al., on random networks
  - Sparse graphs (density=0.02): Java is (slightly) faster
  - Dense graphs (density=0.25): We are (slightly) faster
  - Supposed reason: Different 2-dimensional array implementations





Conclusion

- Proof of Min-Cut/Max-Flow theorem
  - Human readable proofs following textbook presentation
  - Showing off Isar proof language
- Verified Edmonds-Karp algorithm
  - From abstract pseudo-code like version ...
  - ... down to imperative implementation
  - Showing off Isabelle Refinement Framework
- Our implementation is pretty efficient