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

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)

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 maximum value

Min-Cut/Max-Flow Theorem

- Consider network with flow
  - No antiparallel edges: $u \longrightarrow v \Longrightarrow v \not\longrightarrow u$

Min-Cut/Max-Flow Theorem

- Consider network with flow
  - No antiparallel edges: $u \longrightarrow v \Longrightarrow v \not\longrightarrow u$
- Residual graph
  - Intuition: flow that can be moved between nodes
    - By either increasing or decreasing flow on network edge
  - For network edge $u \xrightarrow{c,f} v$, residual graph has edges

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

Min-Cut/Max-Flow Theorem

- Consider network with flow
  - No antiparallel edges: $u \longrightarrow v \Longrightarrow v \not\longrightarrow u$
- Residual graph
  - Intuition: flow that can be moved between nodes
    - By either increasing or decreasing flow on network edge
  - For network edge $u \xrightarrow{c,f} v$, residual graph has edges

    $u \xrightarrow{c-f} v$ and $v \xrightarrow{f} u$
- Min-Cut/Max-Flow Theorem
  - Value of minimal cut = value of maximum flow
  - Corollary: maximum flow iff no source–sink path in residual graph

Ford-Fulkerson Method

Path from source to sink in residual graph is called *augmenting path*.
Flow is maximal iff there is no augmenting path.

- Greedy Algorithm to compute maximum flow

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

Ford-Fulkerson Method

Path from source to sink in residual graph is called *augmenting path*.
Flow is maximal iff there is no augmenting path.

- Greedy Algorithm to compute maximum flow

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

- Partial correctness: obvious

Ford-Fulkerson Method

Path from source to sink in residual graph is called *augmenting path*.
Flow is maximal iff there is no augmenting path.

- Greedy Algorithm to compute maximum flow

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

- Partial correctness: obvious
- Termination: only for integer/rational capacities

Ford-Fulkerson Method

Path from source to sink in residual graph is called *augmenting path*.
Flow is maximal iff there is no augmenting path.

- Greedy Algorithm to compute maximum flow

  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 proof
  - Closely following Cormen et al.

Our Contributions

Verified in Isabelle/HOL

- Min-Cut/Max-Flow Theorem
  - Human-readable proof
  - Closely following Cormen et al.
- Ford-Fulkerson and Edmonds Karp algorithms
  - Human-readable presentation of algorithms
  - Proved correctness and complexity

Our Contributions

Verified in Isabelle/HOL

- Min-Cut/Max-Flow Theorem
  - Human-readable 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

- Used Isar proof language

Human-Readable Proofs

- Used Isar proof language
  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

- Used Isar proof language
  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 . . . $\leq$ f(u,v) + f'(u,v) **using** f'.capacity_const **by** auto
also have . . . $\leq$ 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) $\leq$ 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:
  isMaxFlow f $\longleftrightarrow$ ($\nexists$ p. isAugmentingPath p)

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 f₀ = (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.augment_with_path c f p;
     assert (NFlow c s t f);
     return (f, False)
    }
  })
  (f₀,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 $\leq$ (**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:
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 ≡ **do** {

...
p ← find_augmenting_spec f;
...

Edmonds-Karp Algorithm

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

definition edka_partial ≡ **do** {

...
p ← 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 ≤ ⇓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 ≡ do {
...
(f,_) ← while^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 Graph.V c ⊆ {0..<N}
  assumes 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 $\leq$ SPEC isMaxFlow **.**
    from hn_refine_ref[OF this edka_imp_refine]
    show ?thesis
      by (simp add: hn_refine_def)
qed

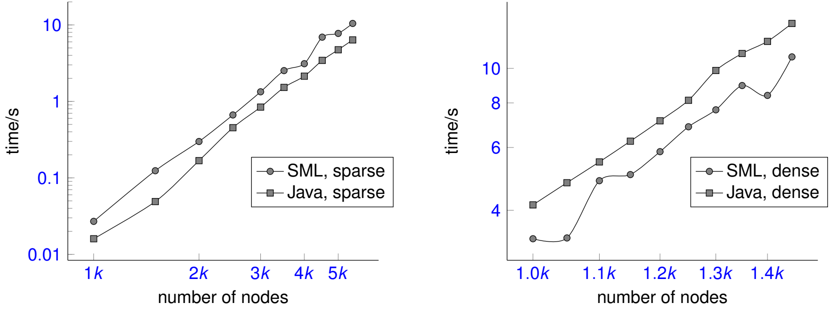
Assembling Overall Correctness Proof

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

```
theorem
 fixes el defines c ≡ ln_α el
 shows
  <emp>
   edmonds_karp el s t
  <
   λNone ⇒ ↑(¬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 s t N f cf * ↑(Network.isMaxFlow c s t f))
  >t
```

Benchmarking

- Against Java version of Sedgewick et al., on random networks
- Two data sets: Sparse ($D = 0.02$) and dense ($D = 0.25$) graphs
  - Sparse: Java is (slightly) faster
  - Dense: 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

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

Questions?