An Isabelle/HOL Formalisation of Scaling Algorithms for Minimum Cost Flows

Thomas Ammer
joint work with Mohammad Abdulaziz
King's College London

September 10, 2024

About this Project

- formalise problem and mathematical results
- formalise algorithms and correctness proofs
- formalise a running time proof
- approx. two years
- builds on graph library (archive of graph formalisations)

This Talk

- introduce problem and algorithms
- how the algorithms and proofs can be formalised
- methodologies used

Theory of Network Flows
(selection of maxflow and mincost flow results)

- Ford and Fulkerson 1962
- Klein 1967
- Dinic 1970
- **Edmonds and Karp 1972**
- Hassin 1983
- Goldberg and Tarjan 1988
- **Orlin 1988**
- Orlin 2013

Work in Formalisation of Flows&Algos

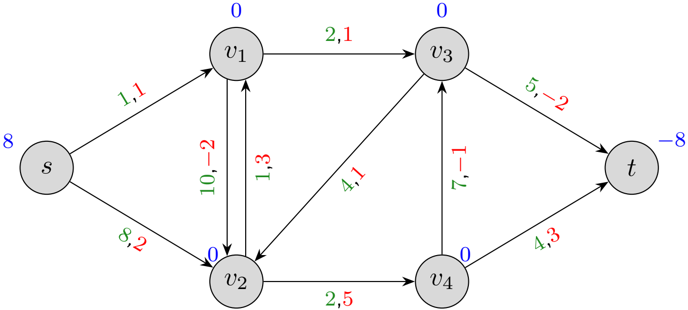
- 2005: maximum flows in Mizar by Lee
- 2017/2019: maximum flows by Lammich and Sefidgar in Isabelle/HOL (see Isabelle AFP)
- no minimum costs flows yet

Main Reference on Theory

- *Combinatorial Optimization* (1st + 5th ed.) by Korte and Vygen as main reference

Basics of Minimum Cost Flows

- find $f : E \to \mathbb{R}_0^+$ for directed Graph (V, E).
- edge capacities u
- per-unit costs c for sending flow through an edge
- vertex balances b ($b\,v > 0$ 'supply'; $b\,v < 0$ 'demand')
- flow of minimum total costs



- transport of liquid in pipeline system

Residual Network/Graph

- auxiliary structure derived from actual network and current flow
- augmentation: along an augmenting path (certain type of paths in residual graph), change flow assigned to edges in original graph.
- augmentation as a technique to send flow in the network.

Flow Network in Isabelle/HOL

- Graphs

```
locale residual =
  fixes ℰ::"'a dgraph"
   and  c::"'a × 'a ⇒ real"
   and  u::"'a × 'a ⇒ ereal"
 assumes u_non_neg: "⋀ u v. u (u,v) ≥ 0"
  and finite_E: "finite ℰ"
  and E_not_empty: "ℰ ≠ {}"
begin
```

- Residual Graph stored implicitly

A simple Algorithm

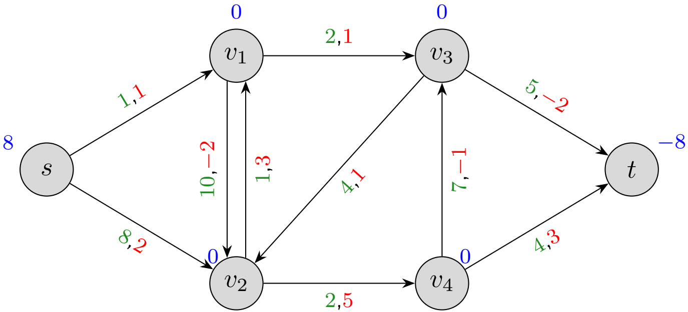
A While Loop:

- take s with $b(s) > 0$, t with $b(t) < 0$, and
- a minimum cost augmenting path P connecting them
- augment P by $\gamma \in \mathbb{R}^+$
- decrease supply/demand at s/t by γ
- able to detect infeasibility

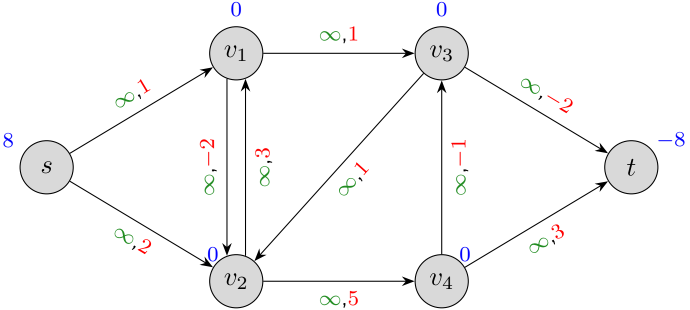
Capacity Scaling Algorithm

- infinite capacities
- two while loops (nested)
- sources + targets with high supply + demand
- fast progress
- sufficiently high balance: balance above threshold
- halve threshold if none remaining

let $\gamma = 1$

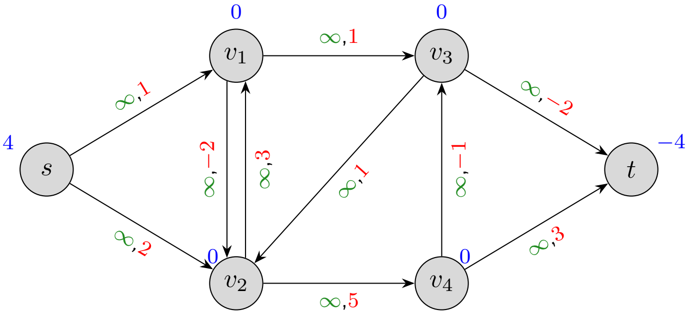


let $\gamma = 4$



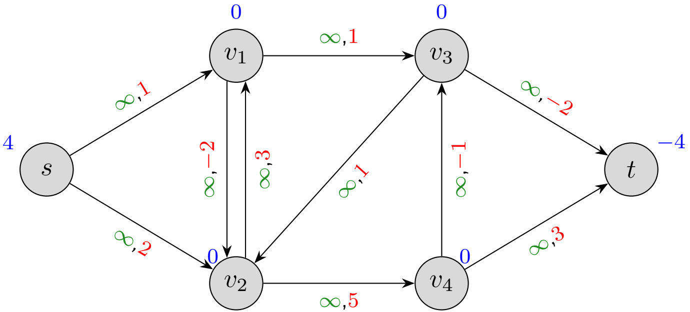
s
v_1
v_2
v_3
v_4
t
8
0
0
0
0
-8
∞,1
∞,2
∞,-2
∞,3
∞,1
∞,1
∞,5
∞,-1
∞,-2
∞,3

let $\gamma = 4$

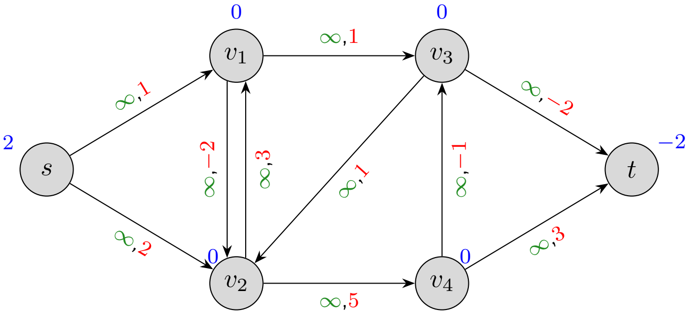


0
0
v_1
v_3
$\infty,1$
$\infty,1$
$\infty,-2$
4
s
$\infty,-2$
$\infty,3$
$\infty,1$
$\infty,-1$
t
-4
$\infty,2$
0
0
$\infty,3$
v_2
v_4
$\infty,5$

let $\gamma = 2$

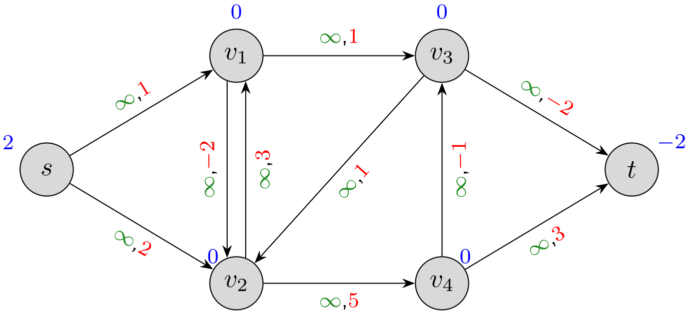


let $\gamma = 2$



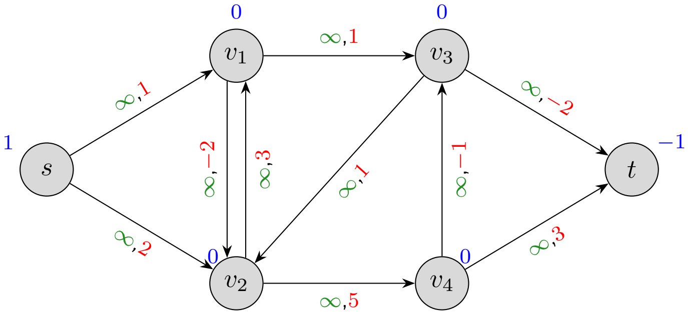
s
v_1
v_2
v_3
v_4
t
2
0
0
0
0
-2
∞,1
∞,2
∞,-2
∞,3
∞,1
∞,1
∞,5
∞,-1
∞,-2
∞,3

let $\gamma = 1$



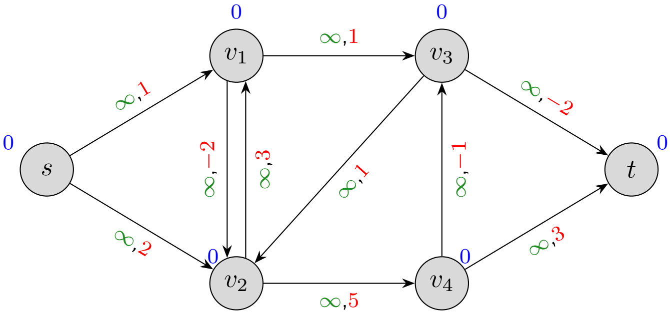
s
v_1
v_2
v_3
v_4
t
2
0
0
0
0
-2
∞,1
∞,2
∞,1
∞,-2
∞,3
∞,1
∞,5
∞,-1
∞,-2
∞,3

let $\gamma = 1$



s
v_1
v_2
v_3
v_4
t
1
0
0
0
0
-1
∞,1
∞,2
∞,-2
∞,3
∞,1
∞,1
∞,5
∞,-1
∞,-2
∞,3

let $\gamma = 1$



s
v_1
v_2
v_3
v_4
t
0
0
0
0
0
0
∞,1
∞,2
∞,1
∞,-2
∞,3
∞,1
∞,5
∞,-1
∞,-2
∞,3

Orlin's Algorithm

- build graph of high-flow edges (spanning forest)
- concentrate balance at single vertices (representatives)
- forest keeps growing

Why is that a good idea?

- sources and targets are representatives
- reduction by growing the forest
- time between component merges strongly polynomial
- strongly polynomial = polynomial in n and m
- c, b, u irrelevant
- reduce number of times when we have to search for s, t and P

Orlin's Algorithm

```
Initialise;
while True do
    while True do
        if ∀v ∈ 𝒱. b′(v) = 0 then return current flow f;
        else if ∃ s. b′(s) > (1 − ε) · γ then
            if ∃ t .b′(t) < −ε · γ ∧ t is reachable from s then
                take such s, t, and a connecting path P using active and forest edges only;
                augment f along P from s to t by γ;
                b′(s) ← b′(s) − γ; b′(t) ← b′(t) + γ;
            else no suitable flow exists;
        else if ∃ t. b′(t) < −(1 − ε) · γ then
            if ∃ s .b′(s) > ε · γ ∧ t is reachable from s then
                take such s, t, and a connecting path P using active and forest edges only;
                augment f along P from s to t by γ;
                b′(s) ← b′(s) − γ; b′(t) ← b′(t) + γ;
            else no suitabe flow exists;
        else break and return to top loop;

    if ∀ still active e. f(e) = 0 then
        γ ← min{γ/2, max_{v∈𝒱} |b′(v)|};
    else
        γ ← γ/2;

    while ∃ active e = (x, y) not in the forest ℱ. f(e) > 8nγ do
        ℱ ← ℱ ∪ {e, ←e}; let x′ = r(x) and y′ = r(y);
        wlog. |component of y| ≥ |component of x|;
        let Q be the path in ℱ connecting x′ and y′;
        if b′(x′) > 0 then
            augment f along Q by b′(x) from x′ to y′;
        else
            augment f along ←Q by −b′(x) from y′ to x′;

        b′(y′) ← b′(y′) + b′(x′); b′(x′) = 0;
        foreach d = (u, v) still active and {r(u), r(v)} = {x′, y′} do
            deactivate d;

        foreach v reachable from y′ in ℱ do
            set r(v) = y′;
```

Correctness

Major Invariant: The flow f in the program state

- satisfies the capacity constraints
- satisfies balance constraints for balance $b - b'$, i.e. balance that was already distributed
- is a flow of minimum costs satisfying the two conditions above

Correctness

Theorem 9.11 from Korte & Vygen

Theorem 9.11. (Jewell [1958], Iri [1960], Busacker and Gowen [1961]) *Let* (G, u, b, c) *be an instance of the* MINIMUM COST FLOW PROBLEM, *and let* f *be a minimum cost* b*-flow. Let* P *be a shortest (with respect to* c*)* s-t*-path* P *in* G_f *(for some* s *and* t*). Let* f' *be a flow obtained when augmenting* f *along* P *by at most the minimum residual capacity on* P*. Then* f' *is a minimum cost* b'*-flow (for some* b'*).*

Correctness

- involved graphical proof for a lemma, see paper for details
- many symmetric cases in the formalisation
- fruitful discussion with Jens Vygen
- own proof simplified: - 1000 LoP

Translating to Isabelle/HOL

- realise loops by recursion (Krauss' function package)
- loop is function mapping state to state
- state is record (collection of program variables)
- invariants are boolean predicates on states
- assume functions with some properties to obtain sources, targets and paths
- specify their behaviour by locales

A Loop

```
function (domintros) loopB::"('a, 'd, 'c) Algo_state
    ⇒ ('a, 'd, 'c) Algo_state" where
"loopB state = (let
                    f = current_flow state;
                    b = balance state;
                    γ = current_γ state
 in (if ∀ v ∈ 𝒱. b v = 0 then state ⦇ return:=success⦈
     else if ∃ s ∈ 𝒱. b s > (1 - ε) * γ then
          ( let s = get_source state
            in (if ∃ t ∈ 𝒱. b t < - ε * γ ∧ resreach f s t then
                   let t = get_target_for_source state s;
                       P = get_source_target_path_a state s t;
                       f' = augment_edges f γ P;
                       b' = (λ v. if v = s then b s - γ
                                  else if v = t then b t + γ
                                  else b v);
                       state' = state ⦇ current_flow := f', balance := b'⦈ in
                           loopB state'
```

A Loop

```
            else
                    state ⦇ return := failure⦈))
    else if ∃ t ∈ 𝒱. b t < - (1 -ε) * γ then
         ( let t = get_target state
           in (if ∃ s ∈ 𝒱.  b s >  ε * γ ∧ resreach f s t then
                  let s = get_source_for_target state t;
                      P = get_source_target_path_b state s t;
                      f' = augment_edges f γ P;
                      b' = (λ v. if v = s then b s - γ
                                 else if v = t then b t + γ
                                 else b v);
                      state' = state ⦇ current_flow := f', balance := b'⦈ in
                        loopB state'
            else
                      state ⦇ return := failure⦈)
         )
     else state ⦇ return := notyetterm⦈
```

- branches expressed by definitions and boolean conditions, e.g. $loopB$-$succ$, $loopB$-$call1$, or $loopB$-$succ$-$cond$, $loopB$-$call1$-$cond$, respectively.
- predefined simplification hard to understand

How to prove Invariants

```
theorem loopB_invar_isOpt_pres:
  assumes "loopB_dom state"
          "aux_invar state" "invar_gamma state" "invar_integral state"
          "invar_isOptflow state"
          "⋀ e. e ∈ ℱ state ⟹ current_flow state e ≥
                                  6*N*current_γ state - (2*N  - Φ state)*current_γ state"
        shows "invar_isOptflow (loopB state)"
```

by induction ...

```
lemma loopB_induct:
  assumes "loopB_dom state"
 "⋀ state. ⟦ loopB_dom state ;
            loopB_call1_cond state ⟹ P (loopB_call1_upd state);
            loopB_call2_cond state ⟹ P (loopB_call2_upd state) ⟧ ⟹ P state"
    shows "P state"
```

Induction Step

- case analysis: which branch?
- simplification

```
lemma loopB_simps:
  assumes "loopB_dom state"
  shows   "loopB_succ_cond state ⟹ loopB state = (loopB_succ_upd state)"
          "loopB_cont_cond state ⟹ loopB state = (loopB_cont_upd state)"
          "loopB_fail1_cond state ⟹ loopB state = (loopB_fail_upd state)"
          "loopB_fail2_cond state ⟹ loopB state = (loopB_fail_upd state)"
          "loopB_call1_cond state ⟹ loopB state = loopB (loopB_call1_upd state)"
          "loopB_call2_cond state ⟹ loopB state = loopB (loopB_call2_upd state)"
```

- single-step lemmas

```
lemma loopB_invar_isOptflow_call1:
  assumes "loopB_call1_cond state" "aux_invar state"
          "invar_gamma state" "invar_integral state"
          "invar_isOptflow state"
          "⋀ e. e ∈ ℱ state ⟹ current_flow state e ≥ current_γ state"
    shows "invar_isOptflow (loopB_call1_upd state)"
```

Formalising Running Time

- functions imitating the algorithm's structure to sum up times, e.g. Mergesort $T(n) = 2 \cdot T(\frac{n}{2}) + c \cdot n \quad [\in \mathcal{O}(n \log n)]$
- assume times for basic parts
- simplification to/of sums
- Laminar Families

Executability

- Proofs with functions and sets
- vs. computation with functions and sets
- Abstract Datatypes: specified behaviour, arbitrary implementation
  (Hoare 1972, Liskov and Zilles 1974)
- correctness of new version by equivalence
- use DFS and Bellman-Ford to implement path selection.

Synopsis of Formalisation Methodology

- while loops as recursive functions
- define + prove simplification and induction principle manually, combine single step lemmas
- different types of refinement
- stepwise refinement (Wirth 1971, Hoare 1972)
- later Abstract Datatypes (Hoare 1972, Liskov + Zilles 1974)
- refinement by equivalent reimplementation
- Isabelle Locales (Ballarin)
- Locales for stepwise refinement (Nipkow 2015, Abdulaziz + Mehlhorn + Nipkow 2019, Maric 2020)
- model running time as recursive function in Isabelle/HOL (Nipkow et al.)
- methodologies scale to big examples

Future Work

- multigraphs (minor)
- automatically generate definitions for execution branches + integrate them into lemmas
- refine to computation model
- other problems: different types of matchings (scaling)

THANK YOU

QUESTIONS?

Running Time

$$\mathcal{O}(n \log n \cdot (m + n \log n))$$

- if using Dijkstra's Algorithm
- requires some additional tricks

```
theorem running_time_initial:
  assumes "final = orlinsTime t_OC (loopB initial)"
  shows "fst final + fst (loopBtime initial) ≤
              (N - 1) * (t_Auf + t_AC + t_AB + t_BC + t_BB + t_Buf)
          +   (N * (l + k + 2) - 1)* (t_BF + t_BC + t_Buf +
                                     t_Auf + t_AC + t_OC + t_OB )
          +   ((l + 1) * (2 * N - 1)) *(t_BC + t_BB + t_Buf)
          +   (t_BF + t_BC + t_Buf ) + t_OC
```

$(l = \lceil \log(4 \cdot m \cdot n + (1 - \epsilon)) - \log \epsilon \rceil + 1$ and $k = \lceil \log n \rceil + 3)$