Formalising Scaling Algorithms for Minimum Cost Flows

Thomas Ammer
joint work with Mohammad Abdulaziz

June 27, 2024

About this Project

- a specific combinatorial optimisation problem

  "Subfield of mathematical optimisation that consists of finding an optimal object from a finite set of objects [...]"

- formalise problem and mathematical results
- formalise algorithms and correctness proofs
- formalise a running time proof

This Talk

- introduce problem and algorithms
- mathematical insights: a technical lemma to prove a step
- 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

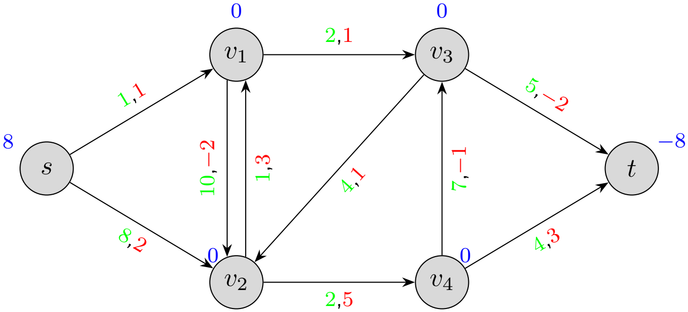
- 2005: maximum flows in Mizar by Lee
- 2019: maximum flows by Lammich and Sefidgar in Isabelle (see Isabelle AFP)
- no minimum costs flows yet

Main Reference on Theory

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

Network Flows

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



As a Linear Optimisation Problem

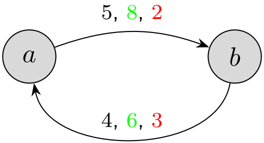
- a flow is a function f assigning positive reals to edges, i.e. $f : E \to \mathbb{R}_0^+$
- Definition $ex_f(v) = \sum_{e \in \delta^-(v)} f(e) - \sum_{e \in \delta^+(v)} f(e)$ (excess flow)
- Optimisation over f

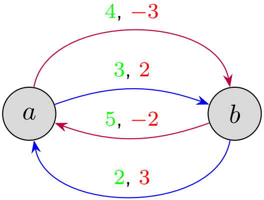
$$
\begin{array}{llll}
\text{min.} & \sum_{e \in E} f(e) \cdot c(e) & & \text{(minimise total cost)} \\
\text{s.t.} & -ex_f(v) & = b(v), & v \in V \ \text{(rspct. } b\text{)} \\
 & f(e) & \leq u(e) & e \in E \ \text{(rspct. } u\text{)}
\end{array}
$$

- excess and validness constraints straightforward in Isabelle.

Residual Network

- for any original edge e, introduce e' and $\overleftarrow{e}$
- forward residual edge same costs, negated for backward





- augmenting path: path of residual edges with positive residual capacity
- augmentation: along a residual path, change flow assigned to original edges

Translating to Isabelle

- 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 Edges

```
datatype 'b Redge = is_forward: F "('b × 'b)" |
                    is_backward: B "('b × 'b)"
```

Optimality

- flow f optimum iff $\nexists\ f'$ with less total cost while respecting capacity and balance constraints
- augmenting cycle: a closed augmenting path with negative total cost
- optimum flow iff $\nexists$ augmenting cycle (Klein 1967)

- 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.
- Invariant: flow is optimum for balance already distributed

Translating the Algorithm to Isabelle

- flag to express certain changes to the control flow
- program states (collection of program variables) as records

```
datatype return = success | failure | notyetterm
record 'b Algo_state = current_flow::"('b × 'b) ⇒ real"
                       balance::"'b ⇒ real"
                       return::return
```

- realise loops by recursion (Krauss' function package)
- assume locale functions with some properties for obtaining minimum weight augpaths

- Correctness: if $get\text{-}min\text{-}augpath$ returns some P, it is a minimum cost augmenting path from s to t
- Completenss: if there is a suitable source, one of these is returned
- Completeness: if there is a reachable target, same

```
locale SSP = residual + algo +
  fixes get_source::"('a ⇒ real) ⇒ 'a option" and
        get_reachable_target::"(('a × 'a) ⇒ real) ⇒ ('a ⇒ real)⇒ 'a ⇒ 'a option" and
        get_min_augpath::"(('a × 'a) ⇒ real) ⇒ 'a ⇒ 'a ⇒ (('a Redge) list)"
  assumes integral_u:  "⋀ e. e∈ ℰ ⟹ ∃ n::nat. u e = real n"
   and    integral_b:  "⋀ v. v∈ 𝒱 ⟹ ∃ n::int. b v =  n"
   and   is_balance_b: "is_balance b"
   and get_source_axioms: "⋀ b v. get_source b = Some v ⟹ b v > 0"
                          "⋀ b v. get_source b = Some v ⟹ v ∈ 𝒱"
                          "⋀ b. (∃ v ∈ 𝒱. b v > 0) ⟶ get_source b ≠ None"
```

```
 and get_reachable_target_axioms:
       "⋀ f s t b. get_reachable_target f b s = Some t ⟹ resreach f s t"
       "⋀ f s t b. get_reachable_target f b s = Some t ⟹ b t < 0"
       "⋀ f s t b. get_reachable_target f b s = Some t ⟹ t ∈ 𝒱"
       "⋀ f s b. (∃ t ∈ 𝒱. resreach f s t ∧ b t < 0)
                ⟶ get_reachable_target f b s ≠ None"
and get_min_augpath_axioms: True
       "⋀ f s t P. resreach f s t ⟹
            get_min_augpath f s t = P ⟹ is_s_t_path  f s t P"
       "⋀ f s t P. resreach f s t ⟹ (∄ C. augcycle f C) ⟹
            get_min_augpath f s t = P ⟹ is_min_path  f s t P"
and conservative_weights: "∄ C. closed_w ℰ C ∧ foldr (λ e acc. acc + c e) C 0 < 0"
```

Scaling Algorithm

set $b' = b$; $f = (\lambda\, e.\, 0)$;

$L = 2^{\lfloor \log_2 B \rfloor}$ where $B = \max\{1, \frac{1}{2} \sum_{v \in G} |b\ v|\}$

while $True$

while $True$

if $b' = 0$ **then** successful termination with current f

else find s, t with $\boxed{b\ s > L - 1,\ b\ t < -L + 1}$

find minimum weight s-t-augpath P with $\boxed{Rcap > L - 1}$

if there are none **break**;

else let $\gamma = \min\{b\ v, -b\ t, Rcap\ f\ P\}$.

augment along P by γ.

adjust b': $b'\ s = b'\ s - \gamma$ and $b'\ t = b'\ t + \gamma$.

if $L = 1$ **then failure** ($\nexists\, b$-flow)

else $\boxed{L = \frac{1}{2} \cdot L}$

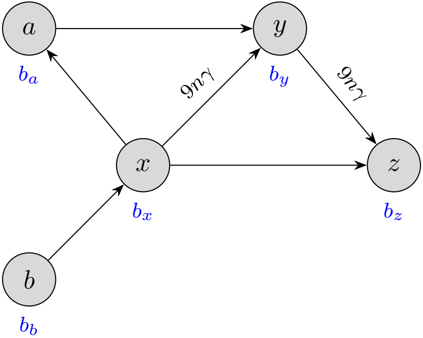
- yields a comparably fast running time (for infinite capacities):

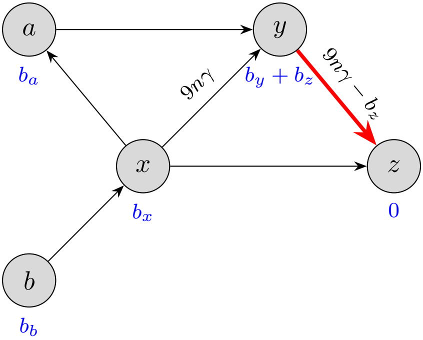
$$\mathcal{O}(n \cdot \log(\tfrac{1}{2} \sum |b\ v|) \cdot \text{time for finding a path})$$

- polynomial w.r.t. to input length
- RT not formalised

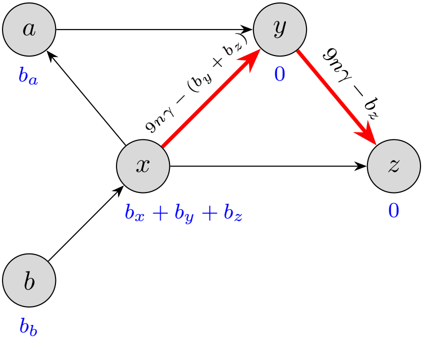
Orlin's Algorithm

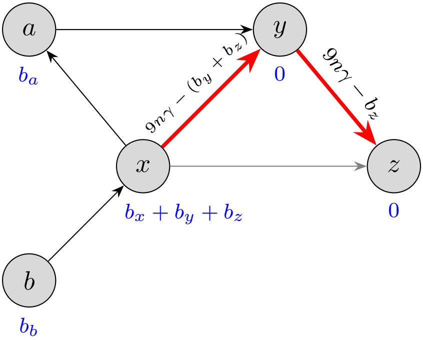
- building a graph of high-flow edges
- spanning forest
- concentrate the balance at single vertices
- use of deactivated edges forbidden


a
y
b_a
$gn\gamma$
b_y
$gn\gamma$
x
z
b_x
b_z
b
b_b



a
y
b
x
z
b_a
$b_y + b_z$
b_x
0
b_b
$9n\gamma$
$9n\gamma - b_z$


a
y
x
z
b
b_a
0
$b_x + b_y + b_z$
0
b_b
$9n\gamma - (b_y + b_z)$
$9n\gamma - b_z$



a
y
x
z
b
b_a
0
$b_x + b_y + b_z$
0
b_b
$9n\gamma - (b_y + b_z)$
$9n\gamma - b_z$

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 where we have to search for s, t and P

Orlin's Algorithm

Initialise;

while *True* **do**

    send flow from sources to targets;

    if $\forall$ *still active* $e.\ f(e) = 0$ **then**

        $\gamma \leftarrow \min\{\frac{\gamma}{2}, \max_{v \in \mathcal{V}} |b'(v)|\}$;

    else

        $\gamma \leftarrow \frac{\gamma}{2}$;

    maintain the forest;

Orlin's Algorithm: Send Flow around ('loopB')

while *True* **do**
- **if** $\forall v \in \mathcal{V}.\, b'(v) = 0$ **then return** current flow f;
- **else if** $\exists\, s.\, b'(s) > (1-\epsilon)\cdot\gamma$ **then**
  - **if** $\exists\, t\, .b'(t) < -\epsilon\cdot\gamma \wedge t$ *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;
    - $b'(s) \leftarrow b'(s) - \gamma$; $b'(t) \leftarrow b'(t) + \gamma$;
  - **else** no suitable flow exists;
- **else if** $\exists\, t.\, b'(t) < -(1-\epsilon)\cdot\gamma$ **then**
  - **if** $\exists\, s\, .b'(s) > \epsilon\cdot\gamma \wedge t$ *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;
    - $b'(s) \leftarrow b'(s) - \gamma$; $b'(t) \leftarrow b'(t) + \gamma$;
  - **else** no suitabe flow exists;
- **else break** and return to top loop;

Orlin's Algorithm: Maintaining the Forest ('loopA')

while $\exists$ *active* $e = (x, y)$ *not in the forest* $\mathcal{F}$. $f(e) > 8n\gamma$ **do**

- $\mathcal{F} \leftarrow \mathcal{F} \cup \{e, \overleftarrow{e}\}$; let $x' = r(x)$ and $y' = r(y)$;
- wlog. $|$component of $y| \geq |$component of $x|$;
- let Q be the path in $\mathcal{F}$ connecting x' and y';
- **if** $b'(x') > 0$ **then**
  - augment f along Q by $b'(x)$ from x' to y';
- **else**
  - augment f along $\overleftarrow{Q}$ by $-b'(x)$ from y' to x';
- $b'(y') \leftarrow b'(y') + b'(x')$; $b'(x') = 0$;
- **foreach** $d = (u, v)$ *still active and* $\{r(u), r(v)\} = \{x', y'\}$ **do**
  - deactivate d;
- **foreach** v *reachable from* y' *in* $\mathcal{F}$ **do**
  - set $r(v) = y'$;

In Isabelle

```
function (domintros) loopA::"'a Algo_state ⇒ 'a Algo_state" where
"loopA state = (let 𝔉 = 𝔉 state;
                    f = current_flow state;
                    b = balance state;
                    r = representative state;
                    E' = actives state;
                    to_rdg = conv_to_rdg state;
                    γ = current_γ state
                in (if ∃ e ∈ E'. f e > 8 * real N *γ
                    then let e = get_edge (λ e.  e ∈ E' ∧ f e > 8 * real N *γ);
                             x = fst e; y = snd e;
                             to_rdg' = (λ d. if d = e then F e
                                             else if prod.swap d = e then B d
                                             else to_rdg d);
                             (x, y) = (if card (connected_component 𝔉 x)
                                          ≤ card (connected_component 𝔉 y)
                                       then (x,y) else (y,x));
                              𝔉' = insert {fst  e, snd e} 𝔉;
                             x' = r x; y' = r y;
                             Q = get_path x' y' 𝔉';
```

```
function (domintros) orlins::"('a, 'b, 'd) Algo_state ⇒ ('a, 'b, 'd
"orlins state = (if return state = success then state
                 else if return state= failure then state
                 else (let f = current_flow state;
                      b = balance state;
                      γ = current_γ state;
                      E' = actives state;
                      γ' = (if ∀ e ∈ to_set E'. f e = 0 then
                             min (γ / 2) (Max { ¦ b v¦ | v. v ∈ 𝒱})
                           else (γ / 2));
                      state' = loopA (state ⦇current_γ := γ' ⦈);
                      state'' = loopB state'
                      in orlins state'')
```

- formalisation of outer loop starts with halving γ
- one iteration of $loopB$ in the beginning
- $loopB$ decides on termination

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

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

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'*).*

Then by Theorem 9.6 there is a circuit C in $G_{f'}$ with negative total weight. Consider the graph H resulting from $(V(G), E(C) \cup E(P))$ by deleting pairs of reverse edges. (Again, edges appearing both in C and P are taken twice.)

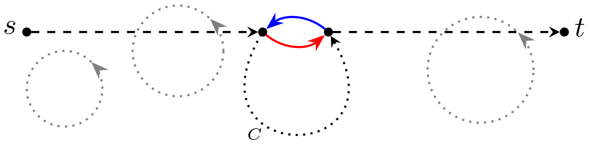
For any edge $e \in E(G_{f'}) \setminus E(G_f)$, the reverse of e must be in $E(P)$. Therefore $E(H) \subseteq E(G_f)$.

We have $c(E(H)) = c(E(C)) + c(E(P)) < c(E(P))$. Furthermore, H is the union of an s-t-path and some circuits. But since $E(H) \subseteq E(G_f)$, none of the circuits can have negative weight (otherwise f would not be a minimum cost b-flow).

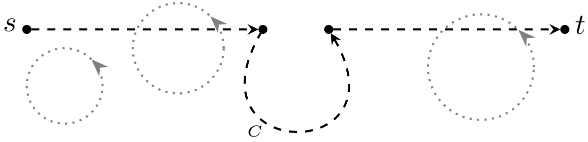
Therefore H, and thus G_f, contains an s-t-path of less weight than P, contradicting the choice of P. $\square$

- Pair of deleted reverse edges are *Forward-Backward-Pairs*.
- Claim: H consists of an s-t-path and some cycles.

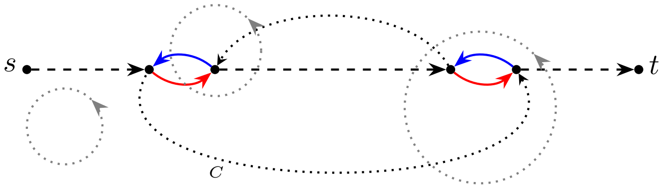
- simple case



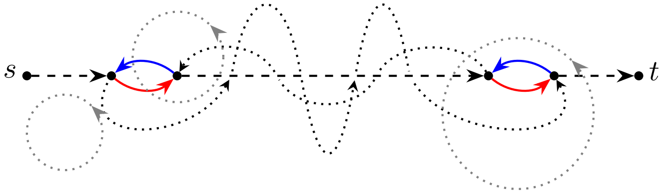
- simple case



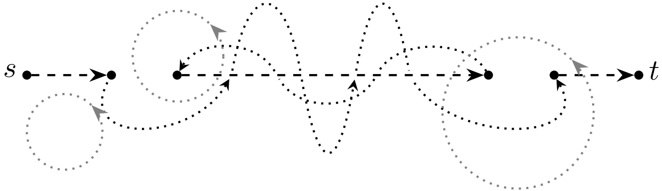
- nasty case, at least two FBPs, take first and last



- In fact, it might be

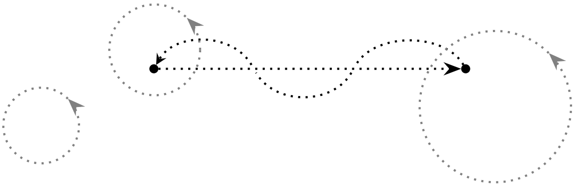


- Let's delete those two FBPs

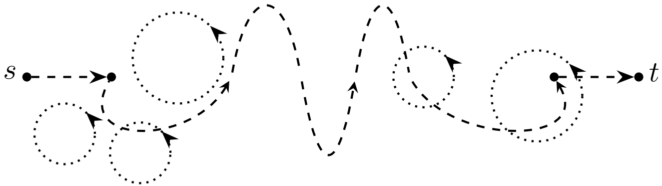


- results in a path

- and a *dirty* set of *dirty* cycles

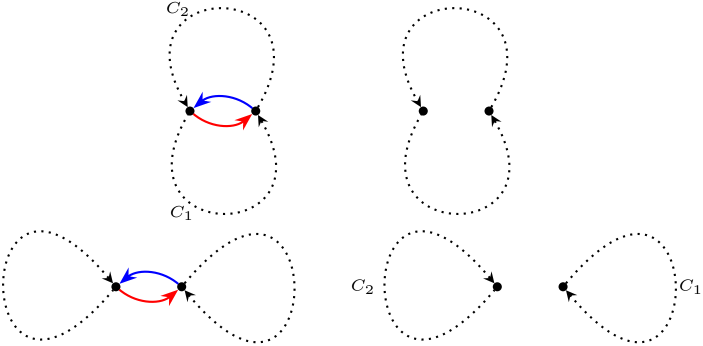


- suppose we can clean those cycles (i.e. eliminate all FBPs)



- Any remaining FBP is between the path and a cycle.
- Decreasing number of FBPs suggests induction

How to clean cycles?



Translating to Isabelle

- use functional constructors to express FBPs.

```
datatype 'b Hedge = is_path_edge: PP "('b Redge)"
                  | is_cycle_edge: CC "('b Redge)"
```

- lots of special and symmetric cases

Verifying Successive Shortest Path Algo in Isabelle

- Integrality of b' (remaining balance) and current f.
- Integrality allows for termination. Order by naturals $\sum |b' \ v|$.
- interesting invariant: f is always a minimum cost flow for $b - b'$. (Theorem 9.11)
- **If** the algorithm terminates with success, the result is optimum.
- **Else** (i.e. terminates with failure) $\nexists$ a b-flow.

Invariants for Correctness of Orlin's Algorithm

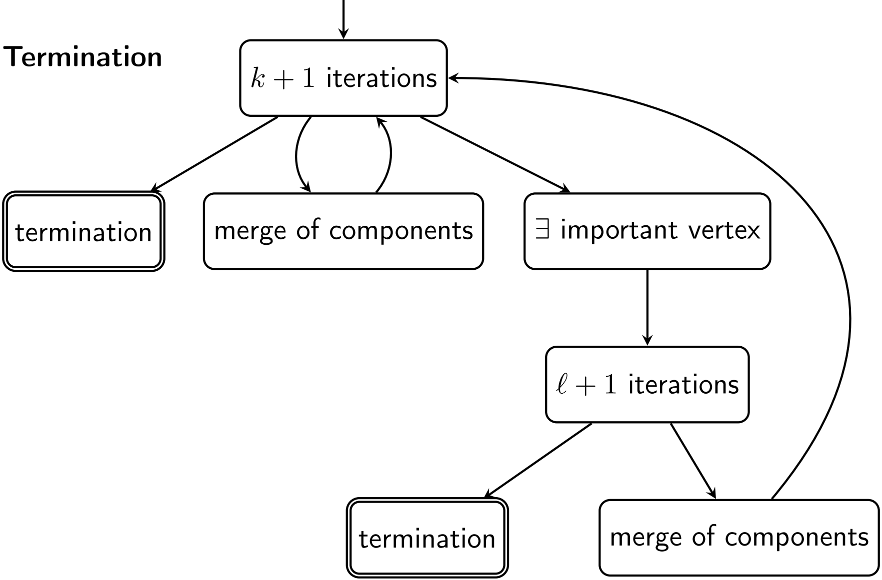
- a number of auxiliary invaraints

1. at any time (except the first iteration of $loopB$) $\gamma > 0$.
2. after leaving $loopB$, there is a vertex x with non-zero balance, i.e. $b'(x) \neq 0$.
3. also, any vertex will have its remaining balance $|b'(x)| \leq (1 - \epsilon) \cdot \gamma$ after leaving $loopB$.
4. any active edge e outside the forest $\mathcal{F}$ has a flow $f(e)$ that is a non-negative integer multiple of γ.
5. any edge e in the forest $\mathcal{F}$ has flow $f(e) > 4 \cdot n \cdot \gamma$ after finishing $loopB$.
6. The current flow f is always optimum for the balance $b - b'$.

Translating to Isabelle

- realise loops by recursion (Krauss' function package)
- loop is function mapping state to state
- assume functions with some properties to obtain paths

Termination



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

Formalising Termination

- notion of a sequence of states/
  state after i iterations of top loop
- some number of iterations followed by a desirable event
- proof by strong induction

Formalising the Running Time

- wait for at most k or l steps, resp. and we make progress with termination
- definition of single steps
- functions imitating the algorithm's structure to sum up times, e.g. $T(n) = 2 \cdot T(\frac{n}{2}) + c \cdot n \quad [\in \mathcal{O}(n \log n)]$
- assume times for basic parts
- simplification to/of sums
- Laminar Families

Formalising Running Time

```
lemma time_boundA:
  assumes "loopAtime_dom state"
          "aux_invar state"
  shows "fst (loopAtime state) =
     (card (comps 𝒱 (to_graph (𝔉_imp state)))
   -  card (comps 𝒱 (to_graph (𝔉_imp (snd (loopAtime state))))))*
                                  (t_Auf + t_AC + t_AB) + (t_Auf + t_AC)"

lemma time_boundB:
  assumes "invar_gamma state"
          "φ = nat (Φ state)"
  shows   "fst (loopBtime state) ≤
             φ * (t_BC + t_BB + t_Buf) + (t_BF + t_BC + t_Buf)"
```

Formalising Running Time

```
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
```

Executability

- Proofs with functions and sets
- vs. computation with functions and sets
- e.g. $[(x, x), (y, x), (z, x), (a, b), (b, b)]$ to implement representatives
- Abstract datatypes: specified behaviour, arbitrary implementation
  (Wirth 1971, Hoare 1972, Liskov and Zilles 1974)
- α denotes meaning of variables and states
- α is a homomorphism

```
locale Map =
fixes empty :: "'m"
fixes update :: "'a ⇒ 'b ⇒ 'm ⇒ 'm"
fixes delete :: "'a ⇒ 'm ⇒ 'm"
fixes lookup :: "'m ⇒ 'a ⇒ 'b option"
fixes invar :: "'m ⇒ bool"
assumes map_empty: "lookup empty = (λ_. None)"
and map_update: "invar m ⟹ lookup(update a b m) = (lookup m)(a := Some b)"
and map_delete: "invar m ⟹ lookup(delete a m) = (lookup m)(a := None)"
and invar_empty: "invar empty"
and invar_update: "invar m ⟹ invar(update a b m)"
and invar_delete: "invar m ⟹ invar(delete a m)"
```

```
locale Set =
fixes empty :: "'s"
fixes insert :: "'a ⇒ 's ⇒ 's"
fixes delete :: "'a ⇒ 's ⇒ 's"
fixes isin :: "'s ⇒ 'a ⇒ bool"
fixes set :: "'s ⇒ 'a set"
fixes invar :: "'s ⇒ bool"
assumes set_empty:    "set empty = {}"
assumes set_isin:     "invar s ⟹ isin s x = (x ∈ set s)"
assumes set_insert:   "invar s ⟹ set(insert x s) = set s ∪ {x}"
assumes set_delete:   "invar s ⟹ set(delete x s) = set s - {x}"
assumes invar_empty:  "invar empty"
assumes invar_insert: "invar s ⟹ invar(insert x s)"
assumes invar_delete: "invar s ⟹ invar(delete x s)"
```

```
locale algo_impl =
algo +
flow_map: Map  flow_empty "flow_update::('a × 'a) ⇒ real ⇒ 'f_impl ⇒
                 flow_delete flow_lookup flow_invar +
bal_map: Map  bal_empty "bal_update:: 'a ⇒ real ⇒ 'b_impl ⇒ 'b_impl"
               bal_delete bal_lookup bal_invar +
rep_comp_map: Map  rep_comp_empty "rep_comp_update::'a ⇒ ('a × nat) ⇒
              rep_comp_delete rep_comp_lookup rep_comp_invar +
conv_map: Map  conv_empty "conv_update::('a × 'a) ⇒ 'a Redge ⇒ 'conv_i
              conv_delete conv_lookup conv_invar +
not_blocked_map: Map  not_blocked_empty "not_blocked_update::('a × 'a) =
              not_blocked_delete not_blocked_lookup not_blocked_invar
```

Lift Correctness

$$\begin{array}{c} impl_1 \\ \big\downarrow{\scriptstyle \alpha} \\ state_1 \end{array}$$

Lift Correctness

$$
\begin{tikzcd}
impl_1 \arrow[r, "step"] \arrow[d, "\alpha"] & impl_2 \\
state_1 \arrow[r, "step"] & state_2
\end{tikzcd}
$$

Lift Correctness

$$\begin{tikzcd}
impl_1 \arrow[r, "step"] \arrow[d, "\alpha"] & impl_2 \arrow[d, "\alpha"] \\
state_1 \arrow[r, "step"] & state_2
\end{tikzcd}$$

Lift Correctness

$$
\begin{tikzcd}
impl_1 \arrow[r, "step"] \arrow[d, "\alpha"] & impl_2 \arrow[r, "step"] \arrow[d, "\alpha"] & impl_3 \arrow[d, "\alpha"] \\
state_1 \arrow[r, "step"] & state_2 \arrow[r, "step"] & state_3
\end{tikzcd}
$$

Lift Correctness

$$\begin{tikzcd}
impl_1 \arrow[r, "step"] \arrow[d, "\alpha"] & impl_2 \arrow[r, "step"] \arrow[d, "\alpha"] & impl_3 \arrow[r, "step"] \arrow[d, "\alpha"] & impl_4 \arrow[d, "\alpha"] \\
state_1 \arrow[r, "step"] & state_2 \arrow[r, "step"] & state_3 \arrow[r, "step"] & state_4
\end{tikzcd}$$

Lift Correctness

- abstraction of final state of implementation = final state of abstraction
- final state of abstraction has minimum cost flow
- abstraction of implementation's final flow is of minimum cost

Path Computation

- subprocedures select paths
- Depth-First Search for Forest Paths, already present (Abdulaziz)
- Bellman-Ford for Minimum Cost Paths (Ford 1956, Bellman 1958, Moore 1959):
  - Dynamic Programming (Bellman 1957)
  - Translation between residual graph and BF graph
  - Invariants
  - Path Recovery

Methodology

- different types of *refinement* (Wirth 1971, Hoare 1972)
- stepwise refinement (Wirth 1971, Hoare 1972)
- Abstract Datatypes (Wirth 1971, Hoare 1972, Liskov and Zilles 1974)
- Isabelle Locales (Ballarin)
- Locales for stepwise refinement (Nipkow 2015, Abdulaziz + Mehlhorn + Nipkow 1019, Maric 2020)
- model RT as recursive function (Nipkow et al.)

Methodology

- stepwise refinement (Wirth 1971):
  send flow from sources to targets;
  halve γ;
  maintain forest;

Methodology

- stepwise refinement:
  select source and target; select mincost path; augment;
  halve γ;
  select and insert edge; select forest path; concentrate balance;

Methodology

- stepwise refinement:
  select source and target; execute Bellman-Ford; augment;
  halve γ;
  select and insert edge; execute DFS; concentrate balance;
- continued when instantiating abstract datatypes

Methodology

- Refinement by abstract datatypes, Equivalent re-implementation
- Formalisation of RT:
  - separate maths and computation model
  - function modelling RT to do maths (Functional Algorithms, Nipkow et al.)
  - which bounds RT w.r.t. computation model (possibly IMP-, Wimmer, Haslbeck, Abdulaziz et al.)
  - yielding RT bound w.r.t. computation model
- locales (Nipkow 2015, Abdulaziz + Mehlhorn + Nipkow 2019, Maric 2020)

$$\mathsf{time}_{computation\text{-}model} \leq \mathsf{time}_{running\text{-}time\text{-}function} \leq \mathsf{time}_{\ graph}$$

- Isabelle functions do not have running time
- methodology scales to big algorithms