A Formal Analysis of Capacity Scaling Algorithms for Minimum Cost Flows

Mohammad Abdulaziz
Department of Informatics, King's College London

Thomas Ammer
Department of Informatics, King's College London

Abstract

We present a formalisation of the correctness of algorithms to solve minimum-cost flow problems, in Isabelle/HOL. Two of the algorithms are based on the technique of scaling, most notably Orlin's algorithm, which has the fastest running time for the problem of minimum-cost flow. Our work uncovered a number of complications in the proofs of the results we formalised, the resolution of which required significant effort. Our work is also the first to formally consider the problem of minimum-cost flows and, more generally, scaling algorithms.

2012 ACM Subject Classification Theory of computation → Network flows; Theory of computation → Invariants; Theory of computation → Program verification

Keywords and phrases Network Flows, Formal Verification, Combinatorial Optimisation

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

Flow networks are some the most important structures in combinatorial optimisation and computer science. In addition to many immediate practical applications, flow networks and problems defined on them have many connections to other important problems in computer science, most notably, the connection between maximum weight bipartite matching and the problem of maximum flow. Because of this practical and theoretical relevance, network flows have been intensely studied, leading to many important milestone results in computer science, like the Edmonds-Karp algorithm [7] for computing the maximum flow between two vertices in a network. Furthermore, flow algorithms were some of the earliest algorithms to be considered for formal analysis. The first such effort was in 2005 by Lee in the prover Mizar [19], where the Ford-Fulkerson algorithm for maximum flow was verified. Later on, Lammich and Serfidgar [17] formally analysed the same algorithm and also the Edmonds-Karp algorithm [7], which is one of its polynomial worst-case running time refinements, in Isabelle/HOL.

In this work we formalise in Isabelle/HOL the correctness of a number of algorithms for the *minimum-cost flow problem*, which is another important computational problem defined on flow networks. Given a flow network, costs per unit flow associated with every edge, and a desired flow value between a number of sources and a number of sinks, a solution to this problem is a flow achieving that value, but for the minimum-cost. This problem can be seen as a generalisation of maximum flow, and thus many problems can be reduced to it, e.g. shortest path, maximum flow, and maximum weight bipartite matching.

More specifically, we formalise 1. the problem of minimum-cost flows, 2. the main optimality criterion used to justify most algorithms for minimum-cost flow, and 3. the correctness of three algorithms to compute minimum-cost flows: a. successive shortest paths, which has an exponential worst-case running time, b. capacity scaling, which has a polynomial worst-case running time, and c. Orlin's algorithm, which has a strongly polynomial worst-case running time. A noteworthy outcome of our work is that it uncovered gaps in the correctness proof of Orlin's algorithm in most textbook expositions. For instance, an important property, namely, optimality preservation, has a gap in all combinatorial proofs of which we are aware. We cover that gap (Lemma 3) using an involved graphical argument, which was at least 15% of our effort. The presence of this gap and the complexity of proving Theorem 1 is yet another example of complications uncovered in graphical and geometric arguments when formalising them, something which was documented by prior authors [1, 24, 13, 3].

2 Background and Definitions

A directed graph is defined as a set of ordered pairs. A maximal set of vertices C, where there is a path between x and y or y and x for all $x, y \in C$, is a *connected component*. A *representative function* $r : \mathcal{V} \to \mathcal{V}$ maps all vertices within a component C to the same vertex $r_C \in C$. Consequently, we call $r(x)$ the representative of component C for a vertex $x \in C$.

A *flow network* consists of a directed graph over edges $\mathcal{E}$ and vertices $\mathcal{V}$, and a capacity function $u : \mathcal{E} \to \mathbb{R}_0^+ \cup \{\infty\}$. If $u(e) = \infty$ for all $e \in \mathcal{E}$, the network is *uncapacitated*. The goal is to find a function, i.e. a *flow* $f : \mathcal{E} \to \mathbb{R}_0^+$ satisfying $f(e) \leq u(e)$ for any edge e. An edge is *saturated* if its flow $f(e)$ equals its capacity $u(e)$, otherwise the edge is *unsaturated*. The first vertex of an edge is called the *source* and the second one is the *target* of the edge, respectively. For a specific vertex v, the set of all edges entering or leaving this vertex is denoted by $\delta^-(v)$ or $\delta^+(v)$, respectively. The *excess* of a flow f at the vertex v, $\mathrm{ex}_f(v)$, is the difference between ingoing and outgoing flow $\mathrm{ex}_f(v) \overset{\text{def}}{=} \sum_{e \in \delta^-(v)} f(e) - \sum_{e \in \delta^+(v)} f(e)$.

Analogously to single vertices, the set of entering and leaving edges of a set of vertices X is denoted by $\Delta^+(X)$ and $\Delta^-(X)$, respectively. An ordered bipartition $(X, \mathcal{V} \setminus X)$ of the graph's vertices is called a cut. The (reverse) capacity of a cut X is the accumulated edge capacity of all (ingoing) outgoing edges $\mathrm{cap}(X) \overset{\text{def}}{=} \sum_{e \in \Delta^+(X)} u(e)$ $(\mathrm{acap}(X) \overset{\text{def}}{=} \sum_{e \in \Delta^-(X)} u(e))$.

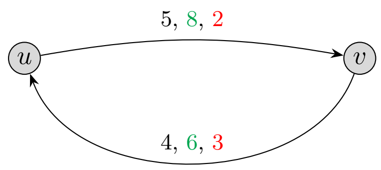
A *minimum cost flow problem* consists of two further ingredients. We introduce *balances* $b : \mathcal{V} \to \mathbb{R}$ denoting the amount of flow that should be caught or emitted at every vertex. A flow satisfying balance and capacity constraints is called *valid*. In addition, there is a function $c : \mathcal{E} \to \mathbb{R}$ telling us about the *costs* of sending one unit of flow through an edge. A flow's f *total costs* $c(f)$ are $c(f) = \sum_{e \in \mathcal{E}} f(e) \cdot c(e)$. The set of *feasible flows* is $\{f \mid \forall v \in \mathcal{V}.\ -ex_f(v) = b(v) \wedge \forall e \in \mathcal{E}.\ f(e) \leq u(e)\}$. We aim to find a *minimum cost flow* which is a feasible flow of least total costs. A network without cycles of negative total costs is called *weight-conservative*.

Given a flow f, we define the *residual network*: For any edge $(x, y) \in \mathcal{E}$ we have two *residual edges*, namely, the *forward* $F\,(x, y)$ and *backward* $B\,(y, x)$ edge pointing from x to y and from y to x, respectively. These form a pair of *reverse* edges. The reverse of a residual edge e is written as $\overleftarrow{e}$. We define the *residual cost* $\mathfrak{c}$ of a residual edge as $\mathfrak{c}(F\,(x, y)) = c(x, y)$ and $\mathfrak{c}(B\,(y, x)) = -c(x, y)$, respectively. For a flow f, we define the *residual capacities* $\mathfrak{u}_f$. On forward edges, this is the difference between the actual capacity and the flow currently sent through this edge: $\mathfrak{u}_f(F\,(x, y)) = u(x, y) - f(x, y)$. The capacity of a backward edge equals the flow assigned to the original edge: $\mathfrak{u}_f(B\,(y, x)) = f(x, y)$.

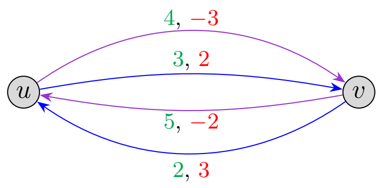
notation of e reverse can perhaps be deleted.

The residual capacity $\mathfrak{u}_f(p)$ of a path p is defined as $\mathfrak{u}_f(p) = \min\{\mathfrak{u}_f(e)\,.\ e \in p\}$. The residual costs $\mathfrak{c}(p)$ are obtained by accumlating residual costs for the edges contained in p.

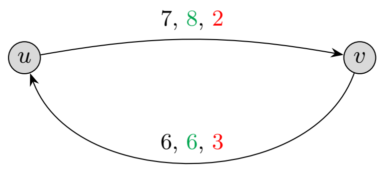
Note that the residual network can be considered a multi graph that has at most two copies of the same edge, e.g. $F(x, y)$ and $B(x, y)$ (see the residual network in Fig. 1b). Intuitively, a forward edge of the residual network indicates how much more could be added



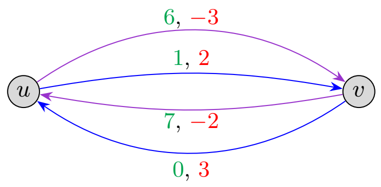
(a) The original flow



(b) Original Residual Graph



(c) Flow after Augmentation



(d) Residual Graph after Augmentation

Figure 1 Flows, (residual) capacities and (residual) costs are black, green and red, respectively. Forward edges are blue, backward edges purple.

to the flow and a backward edge indicates how much could be removed from the flow.

▶ **Example 1.** *Fig. 1a shows a flow network, where every edge is labelled by a flow, capacity, and a cost per unit flow. The colouring convention from the caption applies. Fig. 1b shows the residual network for the network in Fig. 1a. The residual network has, for every flow network edge (x, y), two edges: one is the forward edge $F(x, y)$, a copy of the original edge, and the second is the backward edge $B(y, x)$, going in the opposite direction. Thus, forward edges of the residual network are labelled by the residual capacity, indicating how much more flow can still go through the network. Backward edges are also labelled by a capacity but, as stated earlier, the capacity is the flow going through the original edge, and the costs are negative, indicating that removing from the flow saves cost.*

3 Towards a simple Algorithm

Augmentation is a principal technique in combinatorial optimisation in which a candidate solution is incrementally improved until an optimal solution is found. In the context of flows, augmenting (along) a forward edge by a positive real γ means to increase the flow assigned to the original edge by γ. Augmenting along a backward edge is done by decreasing the flow value of the original edge by γ. The augmentation along a path is done by augmenting along each edge contained. We call a path of residual edges along which the residual capacities are strictly positive an *augmenting path.* Closed augmenting paths p with $\mathfrak{c}(p) < 0$ are *augmenting cycles.*

▶ **Example 2.** *The result of augmenting our example flow from Fig. 1b is shown in Fig. 1c, where the flow is augmented along the edges (u, v) and (v, u) by 2. The resulting new residual network is shown in Fig. 1d, showing the change in capacities in forward and backward edges.*

For this and all subsequent sections, we fix a weight-conservative flow network $\mathcal{G} = (\mathcal{E}, \mathcal{V}, u, c)$. Unless said otherwise, costs and capacities refer to this network. The balances b are kept generic. Algorithm 1 is one of the most basic minimum cost flow algorithms. *successive-shortest-path*$(\mathcal{G}, b)$ repeatedly selects a source s with positive balance, a target t with negative balance and a *minimum cost augmenting path* P connecting s to t, i.e. a

Algorithm 1 *successive-shortest-path*$(\mathcal{E}, \mathcal{V}, u, c, b)$

```
initialise $b' \leftarrow b$ and $f \leftarrow 0$;
while $True$ do
    if $\forall v \in \mathcal{V}.\ b'(v) = 0$ then
        return $f$ as optimum flow;
    else
        take some $s$ with $b'(s) > 0$;
        if $\exists t$ reachable from $s \wedge b'(t) < 0$ then
            take such a $t$ and
            a mincost augpath $P$ from $s$ to $t$;
            $\gamma = \min\{b(s), -b(t), \mathfrak{u}_f(P)\}$ ;
            augment $\gamma$ along $P$;
            $b'(s) \leftarrow b'(s) - \gamma$; $b'(t) \leftarrow b'(t) + \gamma$;
        else return $infeasible$;
```

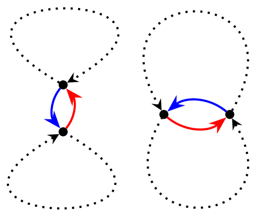


Figure 2 Eliminating FBPs: Members of an *FBP* belong either to the same (left) or to two different cycles (right). When the *FBP* is dropped on the left, we obtain two new cycles. On the right, *FBP* deletion results in a single new cycle. Disjointness is preserved.

minimum cost path in the residual network connecting s to t. Following that, it sends as much flow as possible, i.e. as much as the minimum capacity of any forward residual edge or as the balances of s and/or t allow, from s to t along P. The balances at s and t are lowered and increased by the same amount, respectively. This is done until all balances reach zero or infeasibility can be inferred from the absence of an augmenting path.

Conceptually, the algorithm is defined on *program states* consisting of variables $\mathcal{E}$, $\mathcal{V}$, u, c, balances b, remaining balances b' and the flow f. Invariants are predicates defined on states. If not all variables are relevant to an invariant, we say that only the involved variables satisfy the invariant.

Correctness of Algorithm 1. To prove that the algorithm is correct, we show that the following invariants hold for the states encountered during the main loop of *successive-shortest-path*$(\mathcal{G}, b)$ (Algorithm 1):

1. The flow f is a minimum cost flow for the balance $b - b'$[1].
2. If capacities u and balances b are integral, then $b'(v)$ and $f(e)$ are integral for any vertex $x \in \mathcal{V}$ and $e \in \mathcal{E}$, respectively.
3. The sum of b' over all vertices v is zero: $\sum_{v \in \mathcal{V}} b'(x) = 0$.

Proving Invariant 1 was the most demanding and we dedicate most of this section to it. The other two invariants easily follow from the algorithm's structure. Correctness of all non-trivial algorithms for minimum cost flows depends on the following optimality criterion:

▶ **Theorem 1** (Optimality Criterion [14]). *A flow f valid for balance b is optimum iff there is no augmenting cycle w.r.t f.*

Proof sketch. An augmenting cycle is a possibility to decrease costs while still meeting the balance constraints which gives one direction.

For the other direction, assume a valid flow f' with $c(f') < c(f)$. We define the flow g in the residual graph as $g(F(x, y)) = \max\{0, f'(x, y) - f(x, y)\}$ and $g(B(y, x)) = \max\{0, f(x, y) - f'(x, y)\}$. It can be shown that g is a flow in the residual graph with zero

[1] For any v, this is defined as $(b - b')(v) = b(v) - b'(v)$.

excess for every vertex, a so-called *circulation*. Moreover, $\mathfrak{c}(g) = c(f') - c(f)$ and any residual edge e with $\mathfrak{u}_f(e) = 0$ has $g(e) = 0$. The circulation g can be *decomposed*, i.e. there is a set of cycles $\mathcal{C}$ and weights $w : \mathcal{C} \to \mathbb{R}^+$ where any residual edge e has $g(e) = \sum_{C \in \mathcal{C} \wedge e \in C} w(C)$. Since $0 > c(f) - c(f) = \mathfrak{c}(g) = \sum_{C \in \mathcal{C}} w(C) \cdot \mathfrak{c}(C)$, there has to be a cycle $C \in \mathcal{C}$ where $\mathfrak{c}(C) < 0$. $\mathfrak{u}_f(C)$ must be positive making this an augmenting cycle w.r.t. f. ∎

Flow-decomposition [12, 8] as used in the proof of Theorem 1 is a fundamental technique in reasoning about flows. Now, pairs of residual edges where one is the reversed one of the other are called *forward-backward-pairs (FBP)*, e.g. $F(x, y)$ and $B(y, x)$. They are involved in lemmas one can use to prove preservation of the optimality invariant. Subsequently, *disjointness* of paths and cycles means their edge-disjointness.

▶ **Lemma 2.** *Deleting all FBPs from a set of disjoint cycles results yields another set of disjoint cycles.*

Proof Sketch. Proof by induction on the number of FBPs. See cases from Fig. 2. ∎

▶ **Lemma 3.** *Assume an s-t-path P and some cycles $\mathcal{C}$ where every FBP is between the path and a cycle, all items disjoint with one another. Deleting all FBPs results in an s-t-path and some cycles, again all disjoint.*

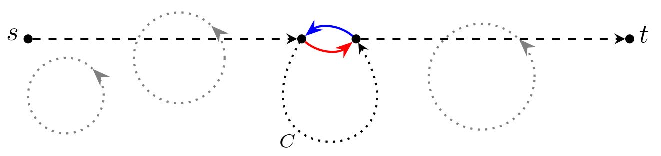
Proof Sketch. Proof by induction on the number of FBPs: Fix an arbitrary FBP which must be between the current path P and some cycle $C \in \mathcal{C}$. Now, we look at two cases.
Simple Case (Single FBP). It might be that this is the only FBP between P and C. By dropping it, we simply get a new s-t-path P' and may eliminate one cycle (Simple Case in Fig. 3). Still, there are no FBPs among or between cycles and the induction hypothesis can be applied immediately to P' and $\mathcal{C} \setminus \{C\}$.
Case (Several FBPs). We now consider the *first* and *last* FBPs between C and P according to the order given by P. By deleting those, we obtain a new s-t-path P' and a new cycle C' (Fig. 3a). Due to eliminating the first and last FBP, P' cannot have any FBPs within itself. But the set of cycles $\mathcal{D} = \mathcal{C} \setminus \{C\} \cup \{C'\}$ (Fig. 3b) may now contain FBPs, although they are still disjoint with one another. By Lemma 2, the FBPs can be deleted resulting in a set of disjoint cycles. This yields the path P' and cycles $\mathcal{C}'$ from Fig. 3c. The number of FBPs has decreased, the substructures are disjoint and any FBP is between P' and a cycle in $\mathcal{C}'$ to which the induction hypothesis may be applied. ∎

The following theorem implies the preservation of Invariant 1. This is perhaps not surprising since we always send the balance/flow along the cheapest augmenting paths.

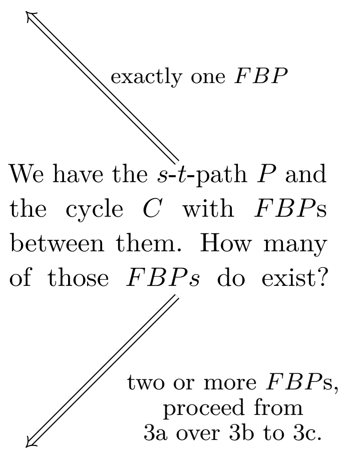
▶ **Theorem 4** (Optimality Preservation [14]). *Let f be a minimum cost flow for balances b. Take an s-t-path P of minimum residual costs and $\gamma \le \mathfrak{u}_f(P)$. If we augment f by γ along P then the result is still optimum for modified balances b' where $b'(s) = b(s) + \gamma$, $b'(t) = b(t) - \gamma$ and $b'(v) = b(v)$ for any other v.*

Proof Sketch. P is vertex-disjoint since any cycle in P would have positive costs (Theorem 1) contradicting the optimality of P. Assume the flow f' after the augmentation were not optimum. By Theorem 1, there exists an augmenting cycle C. Wlog. C is vertex-disjoint: Otherwise split C into vertex-disjoint cycles of which one has negative residual costs.

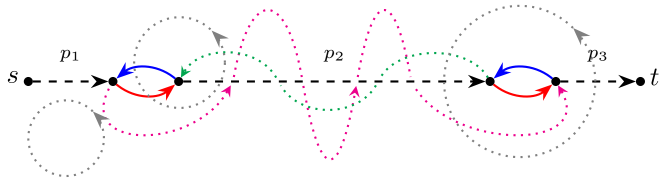
Due to vertex-disjointness, neither P nor C can contain any FBPs. We can therefore apply Lemma 3 to P and C yielding another s-t-path P' and a set of cycles $\mathcal{C}$. Their edges



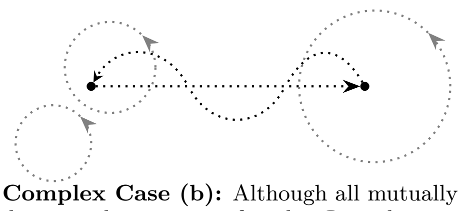
Simple case: Exactly one FBP between the s-t-path P and a cycle C. Deleting this simply forms a new s-t-path. Other cycles (grey) remain unaltered. Disjointness is preserved.



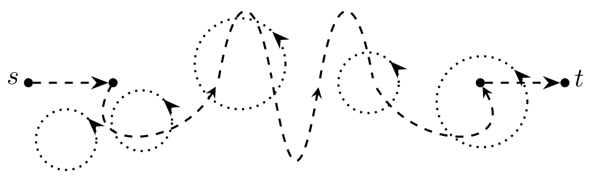
We have the s-t-path P and the cycle C with $FBPs$ between them. How many of those $FBPs$ do exist?



Complex Case (a): At least two FBPs between path and cycle. The path P is composed out of p_1, p_2, p_3 and the two red edges. The cycle consists of the blue edges, the green path and the pink path. When dropping the two fixed FBPs, the new path P' is formed out of p_1, p_3 and the pink path. A new cycle C' arises from p_2 and the green path. Uninvolved cycles are grey.



Complex Case (b): Although all mutually disjoint, the new set of cycles $\mathcal{D}$ might contain FBPs, need for elimination according to Lemma 2



Complex Case (c): After eliminating the FBPs from the set of cycles $\mathcal{D}$, we have a set of disjoint cycles $\mathcal{C}'$ without any FBPs. P' is still defined as in Fig. 3a.

Figure 3 Elimination of Forward-Backward-Pairs in the proof of Lemma 3

have positive residual capacity w.r.t. f: For any $e \in P' \cup \bigcup \mathcal{C}$ we have $\mathfrak{u}_f(e) > 0$ or $\mathfrak{u}_{f'}(e) > 0$. If only the latter holds, then $e \in C$ and $\overleftarrow{e} \in P$ which is an FBP. However, this would have been deleted. Since deleting $FBPs$ preserves costs, we have $\mathfrak{c}(P') + \mathfrak{c}(\mathcal{C}) = \mathfrak{c}(P) + \mathfrak{c}(C)$. Because $\mathfrak{c}(P') \geq \mathfrak{c}(P)$ (optimality of P) and $\mathfrak{c}(C) < 0$, there must be $D \in \mathcal{C}$ with $\mathfrak{c}(D) < 0$. This is an augmenting cycle w.r.t. f contradicting Theorem 1. ∎

Lemma 3 is a gap, which we cover with the construction above, in all published combinatorial proofs we are aware of, including the proof by Korte and Vygen [14]. The only other complete proof of Theorem 4 of which we are aware is a non-combinatorial proof by Orlin [22, 4], in which he uses advanced LP-theory.

▶ **Theorem 5** (Correctness of Algorithm 1). *Assume the sum of balances b over $\mathcal{V}$ is zero. An execution of sucessive-shortest-path($\mathcal{G}, b$) terminates, decides about the existence of a valid flow and returns one in case of existence.*

Proof Sketch. Due to weight conservativity, the zero flow is optimum for the zero balance making the optimality invariant (Invariant 1) initially true. Its preservation follows from Theorem 4. Invariants 3 and 2 also hold initially. Their preservation can be seen from the algorithm.

As the γ used for the augmentations will be a natural number, the sum of the absolute values of balances will decrease yielding a termination measure.

It remains to show that there is no valid flow if the procedure returns *infeasible*, a case for which we need some further auxiliary results. We note that f is a valid flow w.r.t. $b - b'$. We call the set of vertices X reachable from v in the residual graph its *residual cut*, denoted by $\mathit{Rescut}_f(v)$. It can be seen that any leaving edge must be saturated and any entering edge's flow is zero. The so-called *Flow-Value Lemma* says for any cut X, any b and any flow f valid w.r.t. b:

$$\sum_{v \in X} b(v) = \sum_{e \in \Delta^+(X)} f(e) - \sum_{e \in \Delta^-(X)} f(e)$$

Those two results yield for any b and any flow f valid w.r.t. b:

$$\sum_{x \in \mathit{Rescut}_f(v)} b(x) = \sum_{e \in \Delta^+(\mathit{Rescut}_f(v))} f(e) = \mathrm{cap}(\mathit{Rescut}_f(v)) \qquad \textbf{(Corollary 6)}$$

We have $0 \le \sum_{e \in \Delta^+(X)} f(e) \le \mathrm{cap}(X)$ and $0 \le \sum_{e \in \Delta^-(X)} f(e) \le \mathrm{acap}(X)$ for any valid flow f and cut X. If there is a valid flow, this implies together with the Flow-Value Lemma:

$$-\mathrm{acap}(X) \le \sum_{v \in X} b(v) \le \mathrm{cap}(X) \qquad \textbf{(Corollary 7)}$$

The sum of balances is the amount of flow to be sent to or removed from the cut. This has to be within the bounds given by the capacities in both directions which is the intuition behind Corollary 7 . From the algorithm's control flow we can infer that there must be an s with $b'(s) > 0$ without a reachable t where $b'(t) < 0$, i.e. any x in the rescut has a $b'(x) \ge 0$. We obtain a contradiction in case there exists a flow f' valid w.r.t. b.

$$\begin{aligned}\sum_{x \in \mathit{Rescut}_f(s)} b(x) &\le \mathrm{cap}(\mathit{Rescut}_f(s)) && (f' \text{ is valid flow and Corollary 7})\\ &= \sum_{x \in \mathit{Rescut}_f(s)} (b - b')(x) && (\text{Corollary 6 for } f \text{ and } b - b')\\ &< \sum_{x \in \mathit{Rescut}_f(s)} b(x)\end{aligned}$$

■

Formalisation. We represent loops as recursive functions. The non-trivial termination argument requires the Isabelle function package [15]. We use records to model program states. The formal version of the loop in Algorithm 1 is given in Listing 1. Most notation is standard functional prgoramming notation. The main exception is record updates, e.g. 'state(return := infeasible)' denotes 'state', but with state variable 'return' updated to 'infeasible'.

Formally, selecting reachable targets and minimum cost paths corresponds to using functions (*get-source* and *get-min-augpath*, respectively) that compute those items non-deterministically. Their existence and properties are assumed by a named context, a so-called *locale*. These allow us to fix constants and to make corresponding assumptions which are both available within the locale.

We introduced definitions to specify which execution branch is taken when doing an iteration of the loop body, e.g. *SSP-call-4-cond state* indicating the recursive case from the function definition above. It has the same structure as the loop body and returns *True* for exactly one branch and *False* for all others. Similarly, the effect of a single execution branch can be modelled, e.g. *SSP-upd4 state*. We can show a simplification lemma and an

Listing 1: Recursive function formalising *SSP*

```
function SSP:: 'a Algo-state ⇒ 'a Algo-state  where
 SSP state = (let b = balance state; f = current-flow state in
(if zero-balance b then state ⦇return := found⦈
 else (case get-source b of
      None ⇒ state ⦇ return := infeasible⦈ |
      Some s ⇒(case get-reachable-target f b s of
                   None ⇒ state⦇ return := infeasible⦈ |
                   Some t ⇒(let P = get-min-augpath f s t;
                                    γ = min( min(b s) (- b t)) (Rcap f(set P));
                                    f' = augment-edges f γ P;
                                    b' = (λ v. if v = sthenb s - γ else
                                                if v = t then b t + γ else b v)
                              in SSP(state⦇ current-flow := f', balance :=  b' ⦈))))))
```

Listing 2: Customised simplification for *SSP*

```
lemma SSP-simps: assumes  SSP-dom state
  shows  SSP-ret-1-cond state ⟹ SSP state = (SSP-ret1 state)
         SSP-ret-2-cond state ⟹ SSP state = (SSP-ret2 state)
         SSP-ret-3-cond state ⟹ SSP state = (SSP-ret3 state)
         SSP-call-4-cond state ⟹ SSP state = SSP(SSP-upd4 state)
```

Listing 3: Customised induction rule for *SSP*

```
lemma SSP-induct: assumes  SSP-dom state
  assumes ⋀state. ⟦SSP-dom state;
                    SSP-call-4-cond state ⟹ P(SSP-upd4 state)⟧ ⟹ P state
  shows  P state
```

Listing 4: Single-step preservation of Optimality

```
lemma assumes  SSP-call-4-cond state    invarOpt state
  shows  invarOpt(SSP-upd4 state)
```

induction principle for *SSP*, given in Listings 2 and 3, respectively. The preservation of the invariants is proven for single updates like the one in Listing 4 and by the simplification and induction lemmas this can be lifted to a complete execution of *SSP*. Proof automation makes this process very smooth and convenient. We follow the same formalisation methodology for all the algorithms we consider below.

4 The Capacity Scaling Algorithm

The naive Successive Shortest Path Algorithm (Algorithm 1) arbitrarily selects sources, targets and mincost paths for the augmentations. This is refined to *Capacity Scaling (CS)* by selecting those triples where the residual capacities and balances are above a certain threshold that is halved from one scaling phase to another (Algorithm 2). It was proposed by Edmonds and Karp [7]. As SSP, CS works on a state consisting of $\mathcal{E}$, $\mathcal{V}$, u, c, b, b' and f. The algorithm uses two nested loops: The outer one is responsible for monitoring the scaling and determining problem infeasibility. The inner one's purpose is to process every suitable path until none are remaining. It is also responsible for terminating the execution when a solution has been found. As this refines SSP, the proofs for correctness and termination do not differ significantly. The same three invariants may be reapplied. Note that capacities and balances must still be integral to ensure termination.

Intuitively CS behaves like SSP, but only greedily chooses large steps towards the optimal solution, thus hastening progress leading to a polynomial rather than exponential worst-case running time. Each of these steps is a minimum cost augmenting path p from a source s to a target t with a 'step size' of $\min\{\mathfrak{u}_f(p), b(s), -b(t)\}$ When no paths of the right cost remain, it halves the thresholds for treatment and continues with a more fine-grained analysis.

Algorithm 2 *capacity-scaling*$(\mathcal{E}, \mathcal{V}, u, c, b)$

```
initialise $b' \leftarrow b$, $f \leftarrow 0$ and $\gamma = 2^{\lfloor \log_2 B \rfloor}$ where $B = \max\{1, \frac{1}{2} \sum_{v \in \mathcal{V}} b(v)\}$;
while True do
    while True do
        if $\forall v \in \mathcal{V}.\ b'(v) = 0$ then return $f$;
        else if $\exists s\, t\, P.$ $P$ is $s$-$t$-path, $\mathfrak{u}_f(P) \geq \gamma$, $b'(s) \geq \gamma$ and $b'(t) \leq -\gamma$ then
            take such $s$, $t$ and $P$; augment $\gamma$ along $P$;
            $b'(s) \leftarrow b'(s) - \gamma$; $b'(t) \leftarrow b'(t) + \gamma$;
        else break;
    if $\gamma = 1$ then return infeasible;
    else $\gamma \leftarrow \frac{1}{2} \cdot \gamma$;
```

Listing 5: Formalisaton of Scaling

```
1   function  (domintros) SSP:: 'a Algo-state ⇒ 'a Algo-state  where
2    SSP state = (let b = balance state; f = current-flow state in
3     (if zero-balance b then state ⦇return := success⦈
4      else (case get-source-target-path f b of
5            None ⇒ state ⦇ return := notyetterm⦈ |
6            Some (s, t, P) ⇒( let γ = min (min (b s) (− b t)) (Rcap f (set P));
7                                   f' = augment-edges f γ P;
8                                   b' = (λ v. if v = s then b s − γ else
9                                                if v = t then b t + γ else b v)
10                              in SSP (state⦇ current-flow := f', balance :=  b' ⦈))))))
11
12  definition  ssp (γ::nat) ≡ SSP.SSP ℰ u (get-source-target-path γ)
13
14  function (domintros) Scaling:: nat ⇒ 'a Algo-state ⇒ 'a Algo-state  where
15   Scaling l state = (let state' = ssp (2^l−1) state in
16                     (case return state' of success ⇒ state'
17                        | failure ⇒ state'
18                        | notyetterm ⇒( if l = 0 then state' ⦇ return= failure⦈
19                                         else Scaling (l−1) state')))
```

Formalisation. As it can be seen from Listing 5, a modified version of SSP realises the inner loop. Paths returned by the selection function are assumed to have capacity above γ, which is enforced by the definition statement. In case of existence, *get-source-target-path* γ returns a source, a target and a minimum cost path with capacity above γ. The outer loop works on the logarithm of γ, denoted by l. The major difference between *SSP* from Listing 5 and the one from Listing 1 is Line 5: In the modified version, the flag is set to *notyetterm* which indicates that no more suitable paths were found and the decission on infeasibility is left to the outer loop.

Most of the claims and proofs are inherited from the formalisation of SSP and therefore they are conditioned on termination for the respective input state. This can be proven from Invariant 2. The outer loop is a function on two arguments, namely, the logarithm of the threshold and the program state. Its termination follows from the decrease in γ.

For SSP, we had the sum of the balances' absolute values as termination measure decreasing in any iteration by at least 1. This is linear in the balances and therefore exponential w.r.t. input length. On the contrary, CS halves the measure after a polynomial number of iterations resulting in fast progress. The number of scaling phases is logarithmic w.r.t. the greatest balance and thus, linear in terms of input length. The time for finding a minimum cost path is polynomial and an augmentation is $\mathcal{O}(n)$. Provided infinite capacities, the number of augmentations per phase is at most $4n$ [14]. For an efficient path computation, CS even runs in $\mathcal{O}(n(n^2 + m) \log B)$ [7, 14], where B is the greatest absolute value of a balance. This is polynomial w.r.t. input length including the representation of balances. It is not polynomial w.r.t. only the number of vertices and edges. Such an algorithm would be *strongly polynomial*.

5 Orlin's Algorithm

Orlin's Algorithm (Algorithm 3) allows for a strongly polynomial worst-case running time of $\mathcal{O}(n \log n(m + n \log n))$ [22, 4, 14]. Similar to Algorithm 2, we have an outer loop monitoring the threshold γ and an inner loop treating all paths with a capacity above that (*augment-edges*()). After each threshold decrease, a forest that is maintained by the algorithm is updated. In the return value of *augment-edges*(), the $flag$ indicates whether an optimum flow was found or infeasibility was detected. Otherwise, the algorithm continues. The top loop and subprocedures work on a program state consisting of the variables $\mathcal{E}$, $\mathcal{V}$, u, c, b, b', f, $\mathcal{F}$, $actives$, γ and r.

Here, the flow on only *active edges* and forest edges can be augmented, which is done using *augment-edges*(). All edges are active initially. Edges deleted from this set are *deactivated*. The subprocedures use a small positive constant ϵ. Its value influences the timespans between component merges. Positivity ensures termination of *augment-edges*().

For the previous algorithms, the running time depended on B. On the contrary, Orlin's Algorithm avoids that using a continuously growing *spanning forest* $\mathcal{F}$ of edges. The crucial observation is that this forest is used s.t. only one vertex (henceforth, the representative) per forest connected component (henceforth, $\mathcal{F}$-*component*) is considered as a source or as a target in searching for augmenting paths. This reduction in augmentation effort is achieved by maintaining the forest, which can be done in time polynomial in n and m.

Algorithm 3 *orlins*$(\mathcal{E}, \mathcal{V}, u, c, b)$

```
1 initialise $b' \leftarrow b$; $f \leftarrow 0$; $r(v) \leftarrow v$ for any $v$; $\mathcal{F} \leftarrow \emptyset$; $actives = \mathcal{E}$; $\gamma \leftarrow \max_{v \in \mathcal{V}} |b'(v)|$;
2 while $True$ do
3     $(b', f, flag) \leftarrow augment\text{-}edges(\mathcal{E}, \mathcal{V}, u, c, b', f, \mathcal{F}, actives, \gamma, r)$;
4     if $flag = found$ then return $f$;
5     if $flag = infeasible$ then return $infeasible$;
6     if $\forall e \in actives.\ f(e) = 0$
7     then $\gamma \leftarrow \min\{\frac{\gamma}{2}, \max_{v \in \mathcal{V}} |b'(v)|\}$;
8     else $\gamma \leftarrow \frac{\gamma}{2}$;
9     $(b', f, \mathcal{F}, r, actives) \leftarrow maintain\text{-}forest(\mathcal{E}, \mathcal{V}, u, c, b', f, \mathcal{F}, actives, \gamma, r)$;
```

Partial correctness of the algorithm is shown by invariants on program states and properties of the subprocedures. Line specifications refer to the state *after* executing the respective line. Consider the following invariants about the execution of *orlins*$(\mathcal{G}, b)$:

1. γ is strictly positive, except when $b = 0$.
2. Any active edge e outside $\mathcal{F}$ has a flow $f(e)$ that is a non-negative integer multiple of γ.
3. Endpoints of a deactivated edge belong to the same $\mathcal{F}$-component.
4. Only representatives can have a non-zero balance.
5. For states in Line 3, any edge $e \in \mathcal{F}$ has $f(e) > 4 \cdot n \cdot \gamma$.
6. f is optimum for the balance $b - b'$.

The *balance potential* Φ is important for the number of augmentations during subprocedures and their termination. For b' and γ it is defined as:

$$\Phi(b', \gamma) = \sum_{v \in \mathcal{V}} \left\lceil \frac{|b'(v)|}{\gamma} - (1 - \epsilon) \right\rceil$$

When writing $\Phi(\mathit{state})$, we refer to the respective b' and γ. We will later see in detail how the subprocedures work. For now, assume the subprocedures have the following properties:

P1. For the result of *augment-edges*(), we have $|b'(x)| \leq (1 - \epsilon) \cdot \gamma$ for any x. If $\mathit{flag} \neq \mathit{found}$ there is x with $b'(x) > 0$. *flag* is *found* when $b' = 0$ is reached.
P2. For any e, the change in $f(e)$ due to calling *augment-edges*() is an integer multiple of γ.
P3. Calling *maintain-forest*() preserves Invariants 3 and 4. Calling *augment-edges*() preserves Invariant 4.
P4. For any edge that is in $\mathcal{F}$ after calling *maintain-forest*(), the flow reduction incurred during this subprocedure is at most $n\beta$ where $\forall\, v.\ |b'(v)| \leq \beta$ before calling *maintain-forest*(). For edges e outside $\mathcal{F}$ after the call, there was no change in $f(e)$.
P5. $\Phi(\mathit{maintain\text{-}forest}(\mathit{state})) \leq \Phi(\mathit{state}) + n$.
P6. Provided the other invariants hold, Invariant 6 is preserved by either subprocedure.
P7. The number of path augmentations during $\mathit{augment\text{-}edges}(\mathit{state})$ is at most $\Phi(\mathit{state})$. The flow decrease along any edge during $\mathit{augment\text{-}edges}(\mathit{state})$ is at most $\Phi(\mathit{state}) \cdot \gamma$.
P8. Assume all invariants hold on *state*. If $\mathit{flag} = \mathit{found}$ in the result of $\mathit{augment\text{-}edges}(\mathit{state})$, f is optimum. If $\mathit{flag} = \mathit{infeasible}$, the flow problem is indeed infeasible.

We examine how the invariants and properties yield partial correctness.

▶ **Theorem 8** (Partial Correctness). *Assume the algorithm terminates on an uncapacitated instance with conservative weights. If a flow is found, it is a mincost flow, otherwise the problem is infeasible.*

Proof. $b = 0$ yields immediate termination with $f = 0$ as correct result (P1). If $b \neq 0$, we show the invariants for the last state on which *augment-edges*() is called. All invariants hold for the initialisation given in the algorithm. The pseudocode and P1 imply preservation of Invariant 1. The flow along edges outside $\mathcal{F}$ is only changed by *augment-edges*() (P4) and the change is integral multiple of γ (P2) implying preservation of Invariant 2. The arguments for Invariants 3 and 4 are simple (P3). Preservation of Invariant 5 is more difficult. Assume it holds in Line 3. We know $|b'(x)| \leq (1 - \epsilon) \cdot \gamma$ for any x (P1). After modifying γ (Lines 6 - 8), the flow along forest edges is above $8n\gamma$, $\Phi(b', \gamma) \leq n$ and $\forall x.\, |b'(x)| < 2 \cdot \gamma$. Executing *maintain-forest*() can cause a decrease of $2n\gamma$ for forest edges (P4) and an increase in Φ by at most n (P5). Calling *augment-edges*() is responsible for a further flow decrease of at most $2n\gamma$ (P7). The overall decrease along forest edges was at most $4n\gamma$. By P6 we obtain Invariant 6 for the state before the last call of *augment-edges*(). P8 gives the claim. ◼

Note: we restrict ourselves to infinite edge capacities, which is insignificant as problems can be reduced to the uncapacitated setting with a linear increase in input length [14]. Our theorems here require weight-conservativity which is inherited from Algorithm 1 and Algorithm 2 as a constraint. However, we drop the restriction to integral capacities and balances.

We now state the subprocedures and show that they satisfy the aforementioned properties.

Algorithm 4 *augment-edges*$(\mathcal{E}, \mathcal{V}, u, c, b', f, \mathcal{F}, actives, \gamma)$

```
A1  while $True$ do
A2      if $\forall v \in \mathcal{V}.\ b'(v) = 0$ then return $(b', f, found)$;
A3      else if $\exists\, s\,.\, b'(s) > (1-\epsilon)\cdot\gamma$ then
A4          if $\exists\, t\,.\, b'(t) < -\epsilon\cdot\gamma \wedge t$ is reachable from $s$ then
A5              take such $s$, $t$, and a connecting path $P$ with original edges from $actives \cup \mathcal{F}$;
A6              augment $f$ along $P$ from $s$ to $t$ by $\gamma$;
A7              $b'(s) \leftarrow b'(s) - \gamma$; $b'(t) \leftarrow b'(t) + \gamma$;
A8          else return $(b', f, infeasible)$;
A9      else if $\exists\, t\,.\, b'(t) < -(1-\epsilon)\cdot\gamma$ then
A10         if $\exists\, s\,.\, b'(s) > \epsilon\cdot\gamma \wedge t$ is reachable from $s$ then
A11             take such $s$, $t$, and a connecting path $P$ with original edges from $actives \cup \mathcal{F}$;
A12             augment $f$ along $P$ from $s$ to $t$ by $\gamma$;
A13             $b'(s) \leftarrow b'(s) - \gamma$; $b'(t) \leftarrow b'(t) + \gamma$;
A14         else return $(b', f, infeasible)$;
A15     else return $(b', f, please\ continue)$;
```

5.1 Augmenting the Flow

We now argue why *augment-edges*() satisfies the properties stated in the previous section. P1, P2, P3, P6, P7 and P8 assert something about *augment-edges*(). P1 can be inferred from the subprocedure's structure. The only possible amount to augment is γ which yields P2. P3 (Preservation of Invariant 4) also follows from the structure.

Proof Sketch for P6. We assume the invariants for hold *state* and we define $(b', f, flag) =$ *augment-edges*$(state)$. Then f is a minimum cost flow for balances $b-b'$ because of Theorem 4. The restriction to active and forest edges (Lines A5 and A11) is unproblematic: Simulate deactivated edges with forest paths which are of minimum costs. ■

Proof Sketch for P7. Show that any iteration decreases Φ at least by 1, thus *augment-edges*$(state)$ performs at most $\Phi(state)$ iterations and the flow decrease for an edge is at most $\Phi(state)\cdot\gamma$. The proof of a strict decrease in Φ only works for $\epsilon > 0$. ■

Proof Sketch for P8. If the algorithm found a flow, then $\forall v \in \mathcal{V}\,.\ b'(v) = 0$ (Line A2). Together with preservation of the optimality invariant, it gives the first subclaim.

If the algorithm asserts infeasibility, one can exploit information about b' from Lines A3, A4, A9 and A10. By employing residual cuts and the analogous definition where every direction is reversed, one can show infeasibility for both cases. The proof is similar to that of Theorem 5. The argument only works for $\epsilon \leq \frac{1}{n}$. ■

By this, we have shown that *augment-edges*() satisfies all asserted properties. We also saw the restriction $0 < \epsilon \leq \frac{1}{n}$. Note: one might think that ϵ could be assigned to 0. As we shall see later on, that would make it impossible for us to derive the worst-case running time bound. In essence, we need to allow vertices to be processed if they are 'slightly' below the threshold, otherwise the algorithm take exponentially many iterations, and its running time will depend on B. Interested readers should consult sec. 5.2 [22].

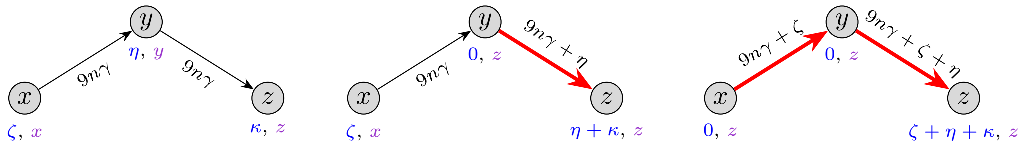


Figure 4 Merging forest components: Remaining balances b', forest edges, flow values and representatives are blue, red, black and purple, respectively. Assume the balances are non-negative.

Algorithm 5 *maintain-forest*$(\mathcal{E}, \mathcal{V}, u, c, b', f, \mathcal{F}, actives, \gamma, r)$

```
F1  while $\exists e = (x, y).\ e \in actives \wedge e \notin \mathcal{F} \wedge f(e) > 8n\gamma$ do
F2      $\mathcal{F} \leftarrow \mathcal{F} \cup \{e\}$;
F3      if $|\mathcal{F}$-component of $y| \geq |\mathcal{F}$-component of $x|$ then exchange $x$ and $y$;
F4      let $x' = r(x)$ and $y' = r(y)$ the respective representatives;
F5      take residual path $Q \subseteq \mathfrak{F}$ connecting $x'$ and $y'$;
F6      if $b'(x') > 0$ then augment $f$ along $Q$ by $b'(x)$ from $x'$ to $y'$;
F7      else augment $f$ along $\overleftarrow{Q}$ by $-b'(x)$ from $y'$ to $x'$;
F8      $b'(y') \leftarrow b'(y') + b'(x')$; $b'(x') = 0$;
F9      foreach $v$ reachable from $y'$ in $\mathfrak{F}$ do $r(v) \leftarrow y'$ ;
F10     foreach $d = (u, v).\ d \in actives \wedge \{r(u), r(v)\} = \{y'\}$ do $actives \leftarrow actives \backslash \{d\}$;
F11 return $(b', f, \mathcal{F}, actives, r)$;
```

5.2 Maintaining the Forest

We now discuss the last part of the algorithm, namely, maintaining the forest. The definition of *maintain-forest*() can be seen in Algorithm 5. We add active edges with flow above $8n\gamma$ to $\mathcal{F}$, which inevitably changes the connected components of the forest – a new component is created for every added edge by merging the components to which the two end points of edge belong. Since non-zero balance is only allowed for representatives, balances must be re-concentrated at one of the two representatives after merging two components.[2] Moreover, all edges between them are deactivated and the representatives must be updated. For a forest $\mathcal{F}$, $\mathfrak{F}$ is the corresponding residual network consisting only of forest edges. Re-concentration is done by augmentations along paths in $\mathfrak{F}$. An example of how balances, flow and forest change is displayed in Fig. 4.

Only P3-P6 make assertions about *maintain-forest*() and we argue why they are indeed satisfied. P3 (preservation of Invariant 3) is easy to see since it is precondition for deactivation to have the same representatives (Line F10).

Proof Sketch for P4. Invariants F1 and F2 bound the forest edges' flow decrease:
F1. For any x, $|b'(x)|$ is bounded by the product of its $\mathcal{F}$-component's cardinality and β.
F2. For any $e \in \mathcal{F}$, we have $f(e) > \alpha - \beta \cdot |X|$ where X is the $\mathcal{F}$-component of e.
Their conjunction is preserved by *maintain-forest*(). It is important to always concentrate the balances at the larger component's representative as done in the algorithm. ∎

Any iteration merges two $\mathcal{F}$-components making $n - 1$ an upper bound for the number of iterations. P5 asserts $\Phi(maintain\text{-}forest(state)) \leq \Phi(state) + n$. This holds because the potential cannot increase by more than 1 per iteration, as shown in the following lemma.

[2] Cardinalities of forest components in the pseudocode refer to the number of vertices in the component.

▶ **Lemma 9.** *The increase of* Φ *during a single iteration of maintain-forest*() *is at most* 1.

Proof. Let *state* be the program state in Line F1 and *state'* be the one in Line F9, respectively. Program variables refer to *state*. It follows:

$$\begin{aligned}
\Phi(state') &= \sum_{v \in \mathcal{V} \setminus \{x', y'\}} \left\lceil \frac{|b'(v)|}{\gamma} - (1-\epsilon) \right\rceil + \left\lceil \frac{0}{\gamma} - (1-\epsilon) \right\rceil + \left\lceil \frac{|b'(y') + b'(x')|}{\gamma} - (1-\epsilon) \right\rceil \\
&= \sum_{v \in \mathcal{V} \setminus \{x', y'\}} \left\lceil \frac{|b'(v)|}{\gamma} - (1-\epsilon) \right\rceil + \left\lceil \frac{|b'(y')| + |b'(x')|}{\gamma} - (1-\epsilon) \right\rceil \\
&\le \sum_{v \in \mathcal{V} \setminus \{x', y'\}} \left\lceil \frac{|b'(v)|}{\gamma} - (1-\epsilon) \right\rceil + \left\lceil \frac{|b'(x')|}{\gamma} \right\rceil + \left\lceil \frac{|b'(y')|}{\gamma} - (1-\epsilon) \right\rceil \\
&\le \sum_{v \in \mathcal{V} \setminus \{x', y'\}} \left\lceil \frac{|b'(v)|}{\gamma} - (1-\epsilon) \right\rceil + \left\lceil \frac{|b'(x')|}{\gamma} - (1-\epsilon) \right\rceil + 1 + \left\lceil \frac{|b'(y')|}{\gamma} - (1-\epsilon) \right\rceil \\
&= \Phi(state) + 1
\end{aligned}$$
■

Proof of P6. To apply Theorem 4 we need optimality of forest paths Q and $\overleftarrow{Q}$ used for augmentations. Suppose this were not true. There is P with $\mathfrak{c}(P) < \mathfrak{c}(Q)$ connecting the same vertices. Since $\mathfrak{c}(\overleftarrow{Q}) = -\mathfrak{c}(Q)$, $P\overleftarrow{Q}$ is a cycle with $\mathfrak{c}(P\overleftarrow{Q}) = \mathfrak{c}(P) + \mathfrak{c}(\overleftarrow{Q}) < 0$ and $\mathfrak{u}_f(P\overleftarrow{Q}) > 0$ contradicting Theorem 1 and Invariant 6. The case for $\overleftarrow{Q}$ is analogous. Due to Invariant 5, the flow in $\mathcal{F}$ is positive. Infinite edge capacities imply positive residual capacities for $\mathfrak{F}$. ■

This concludes our proofs that *maintain-forest*() satisfies properties P3-P6.

5.3 Termination and Running Time

For the inner loops we have termination measures decreasing in any iteration, namely, the number of components and Φ. For the outer loop, there is a maximum number of iterations until some desirable situation occurs. A vertex v is *important* iff $|b'(v)| > (1-\epsilon) \cdot \gamma$ [14], i.e. its contribution to Φ is positive. We repeatedly wait for the occurrence of an important vertex and ensure a merge of two forest components some iterations later. We define $\ell = \lceil \log(4 \cdot m \cdot n + (1-\epsilon)) - \log \epsilon \rceil + 1$ and $k = \lceil \log n \rceil + 3$. It can be shown that (a) if we wait for $k+1$ iterations, a vertex has become important or there is a component merge, and that (b) for an important vertex, $\ell + 1$ iterations enforce its component being increased. There can be at most $n-1$ such merges, which yields termination.

Concerning running time, we assume *atomic bounds* for basic parts of the algorithm. For instance, t_{FB} is an upper bound for the time consumed when executing the loop body in *maintain-forest*() and t_{FC} is the time for checking the condition (analogously t_{AC} and t_{AB} for *augment-edges*(), and t_{OC} and t_{OB} for *orlins*()). Time consumption can be bounded by the term

$$\begin{aligned}
&(n \cdot (\ell + k + 2) - 1) \cdot (t_{OC} + t_{OB} + t_{AC} + t_{FC}) + \\
&(n-1) \cdot (t_{FC} + t_{FB} + t_{AC} + t_{AB}) \quad + \quad (2n-1) \cdot (\ell + 1) \cdot (t_{BC} + t_{BB}) + t_{BC} + t_{OC}
\end{aligned}$$

The proof involves bounding the number of iterations of the subprocedures by n and Φ, respectively, a connection between the number of important vertices and Φ, and bounding the number of occurences of important vertices by results on so-called *Laminar Families*.

5.4 Formalisation

Listing 6: Locale to specify path selection for *augment-edges*()

```
locale augment-edges = algo +
  fixes get-source-target-path :: 'a Algo-state ⇒ 'a ⇒ 'a ⇒  'a Redge list  and
        get-vertex :: ('a ⇒ bool) ⇒ 'a
assumes get-source-target-path-axioms:
        ⟦get-source-target-path state s t = P ; s ∈ 𝒱 ; t ∈ 𝒱 ; s ≠ t
         aux-invar state ;(∀ e ∈ ℱ state . current-flow state e > 0) ;
         resreach(current-flow state) s t ⟧ ⟹
         (Rcap(current-flow state)(set P) > 0 ∧
         (invar-isOptflow state ⟶ is-min-path(current-flow state) s t P) ∧
         oedge ` set P ⊆ actives state ∪ ℱ state   ∧   distinct P
```

Listing 7: Formalisation of the Top Loop of Orlin's algorithm.

```
function(domintros) orlins:: 'a Algo-state ⇒ 'a Algo-state  where
 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 ∈ E'. f e = 0 then min(γ / 2)(Max {| b v|| v ∈ 𝒱})
                           else(γ / 2));
                      state' = maintain-forest(state ⦇current-γ:= γ' ⦈);
                      in orlins(augment-edges state')))
```

Listing 8: A function modelling the running time of *orlins*().

```
function(domintros) orlinsTime:: nat ⇒('a, 'b, 'd) Algo-state
                         ⇒ nat ×('a, 'b, 'd) Algo-state  where
 (orlinsTime tt_OC  state) =(if(return state = success) then(tt_OC, state)
                  else if(return state = failure) then(tt_OC, 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'time = maintain−forestTime(state ⦇current-γ:= γ' ⦈);
                      state''time = augment−edgesTime(snd state'time)
                      in ((t_OC + t_OB + fst state'time + fst state''time)
                         +++(orlinsTime tt_OC(snd state''time))))
)
```

General. We assume functions selecting paths non-deterministically via locales (see Listing 6). The locales were later instantiated suitably. As above, we model program states as records and use customised simplification and induction. However, here the algorithm's complexity is more substantial (see Listings 9 and 7). Also, we note that formal proofs about paths often need pairwise distinctness of the vertices or edges which is often neglected in an informal setting. **Forest.** For connected components, we reuse Abdulaziz et al.'s [2] formalisation modelling *undirected* edges as sets. Paths based on that must be transformed to paths over residual edges. Each direction is mapped to a residual edge pointing is this direction. Residual edges realising opposite directions originate from the same graph edge. This implies that converting a path over undirected edges and its reverse yields opposite costs as needed to show P6.

RIght now, we do not say anything on invariants and monotone properties.

Termination. The termination proof reasons about a fixed number of iterations. We introduced definitions expressing the effect of a single iteration, which can then be combined to a fixed number by the function iteration *compow*. As soon as executing a single step does not change anything, the recursive version would also terminate yielding an equal result.

Running Time. We model algorithm running times as functions returning natural numbers, using an extension of Nipkow et al.'s approach. In this approach, for every function $f : \alpha \to \beta$, we devise a functional program fTime $: \alpha \to \mathbb{N}$ with the same recursion and control-flow structure as the algorithm whose running time we measure. In its most basic form, for a given input $x : \alpha$, $f\text{Time}(x)$ returns the number of the recursive calls that f would perform while processing x. If f involves calls to other functions, the running times of the called functions are added to the number of recursive calls of f. To modularise the design of these running time models, we use locales to assume running times of the called functions. An example of such a running time model is shown in Listing 8, which models the running time of Orlin's algorithm. Here, there running times of the forest and path augmentation procedures are only assumed, without explicitly specifying them, in the locale containing the definition of *orlinsTime*. Mathematically, proving the upper bound on the running time requires basic results about *laminar families*, where the set of connected components of the forest is viewed as a laminar family.

Listing 9: Formalisation of *augment-edges*() and corresponding Induction Rule

```
function (domintros) augment-edges:: 'a Algo-state ⇒ 'a Algo-state  where
 augment-edges 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-vertex (λ s. b s > (1 − ϵ) ∗ γ ∧ s ∈ 𝒱)
        in (if ∃ t ∈ 𝒱. b t < − ϵ ∗ γ ∧ resreach f s t then
                 let t = get-vertex (λ t.  b t < − ϵ ∗ γ ∧ resreach f s t ∧ t ∈ 𝒱);
                     P = get-source-target-path state s t;
                     f' = augment-path 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
                         augment-edges state'
             else state ⦇ return := failure⦈))
      else if ∃ t ∈ 𝒱. b t < −(1 −ϵ) ∗ γ then
       (let t = get-vertex (λ t. b t < −(1 −ϵ) ∗ γ ∧ t ∈ 𝒱)
        in (if ∃ s ∈ 𝒱.  b s >  ϵ ∗ γ ∧ resreach f s t then
                 let s = get-vertex (λ s.  b s > ϵ ∗ γ ∧ resreach f s t ∧ s ∈ 𝒱);
                     P = get-source-target-path state s t;
                     f' = augment-path 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
                       augment-edges state'
             else state ⦇ return := failure⦈))
     else state ⦇ return := notyetterm⦈))

lemma augment-edges-induct: assumes  augment-edges-dom state
   ⋀ state. ⟦ augment-edges-dom state ;
               augment-edges-call1-cond state ⟹ P (augment-edges-call1-upd state);
               augment-edges-call2-cond state ⟹ P (augment-edges-call2-upd state) ⟧
                    ⟹ P state
     shows  P state
```

6 Discussion

The algorithms that we considered here share a number of features with other maximum flow algorithms that were formally analysed before, most notably the fact that they iteratively compute augmenting paths to incrementally improve a solution until an optimal solution is reached. Those algorithms also use residual graphs, which are intuitively graphs containing the remaining capacity w.r.t. the current flow maintained by the algorithm, and which have been formalised by Lammich and Sefidgar [17]. The most advanced one out of these is probably the Push-Relabel Algorithm by Goldberg and Tarjan. Another combinatorial optimisation algorithm that was also formalised is Edmonds' blossom shrinking algorithm for maximum cardinality matching in general graphs [2].

We are not stating any other differences

However, our work here is different from those previously studied algorithms for maximum flow in one crucial aspect: here we cover the algorithms which use *scaling*, a technique for designing fast optimisation algorithms, including algorithms with the fastest worst-case running times for different variants of matching and shortest path problems [6, 9, 10, 11, 23], in addition to minimum-cost flows. Our work here provides a blueprint to formalise the correctness of those other scaling algorithms. Furthermore, the running time proof here depends on properties of laminar families, making it one of the more advanced running time proofs to be formalised.

A main outcome of our work, from a mathematical perspective, is our proof of Theorem 4, which is its first complete combinatorial proof, and our simplified proof of Theorem 1. Those outcomes, especially the construction we devised for the former, highlight the potential role of formalising deep proofs in filling gaps as well as in the generation of new proofs and/or insights. Indeed, this was a recurring theme across the project, e.g. there were other non-trivial claims in the main textbook we used as reference [14], for which no proof is given. E.g. it is claimed that the accumulated augmentations through a forest edge are below $2(n-1)\gamma$. We could not formalise the proof of this in the textbook as it also had gaps, and we devised Invariants F1 and F2 to be able to prove it. Other gaps were in the proofs of Properties P7, P6, Lemma 5, Lemma 9, and a few parts of the termination proof, but we have to refer interested readers to the formalisation due to the lack of space.

Please have a look at TODO file and treat those points that are marked with a star, those with + were treated by myself.

The formalisation of the algorithms presented here is around 25K lines of proof scripts. Our methodology is based on using Isabelle/HOL's *locale*s to implement Wirth's notion of step-wise refinement [25], thereby compartmentalising different types of reasoning. This locale-based implementation of refinement was used earlier by many authors, e.g. by Nipkow [21], Maric [20], and Abdulaziz et al. [2]. In this approach, non-deterministic computation is handled by assuming the existence and properties of functions that compute non-deterministically, without assuming anything about the functions' implementation. Our formalisation is one further example showing that this approach scales to proving the correctness of some of the most sophisticated algorithms. We also note that our focus here is more on formalising the mathematical argument behind the algorithm and less on obtaining an executable program, which is nonetheless attainable using this locale-based approach. A notable alternative implementation of refinement is Peter Lammich's [16] framework.

We note that in our work, we have used the locale-based approach in two ways: top-down and bottom-up. Our approach to specifying *augment-edges* was top-down, where we assumed the existence of a procedure for finding shortest paths between sources and targets. On the other hand, for specifying *orlins* we went bottom-up, where we first defined and proved the correctness of *augment-edges* and *maintain-forest*, and then started defining and proving the correctness of *orlins*, which calls both *augment-edges* and *maintain-forest*. In the former case, we went top-down as we had a good a priori understanding of the functions to assume and their properties, while in the latter we went bottom-up because we had a poorer a priori understanding of the functions to assume and their properties.

Furthermore, we define procedures as recursive functions using Isabelle's function package [15], program states as records, and invariants as predicates on program states. We devise automation is based on manually deriving theorems characterising different properties of recursive functions, and combining those theorems with Isabelle's classical reasoning and simplification. In this approach, the automation handles proofs of invariants and monotone properties, e.g. the growth of the forest or the changes in Φ. Again, other approaches can be used to model algorithms and automate reasoning about them, like using monads [18] or while combinators [5]. Both of those approaches allow for greater automation, which is particularly useful for reasoning about low-level implementations. However, those two approaches are problematic for manual mathematical proofs, which form the majority of our effort, as they usually add a layer of concepts between the theorem prover's basic logic and the algorithm's model.

References

1 Mohammad Abdulaziz and Christoph Madlener. A Formal Analysis of RANKING. In *The 14th Conference on Interactive Theorem Proving (ITP)*, 2023. doi:10.48550/arXiv.2302.13747.

2 Mohammad Abdulaziz, Kurt Mehlhorn, and Tobias Nipkow. Trustworthy graph algorithms (invited paper). In *The 44th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, 2019. doi:10.4230/LIPIcs.MFCS.2019.1.

3 Mohammad Abdulaziz and Lawrence C. Paulson. An Isabelle/HOL Formalisation of Green's Theorem. In *The 7th International Conference on Interactive Theorem Proving (ITP)*, 2016. doi:10.1007/978-3-319-43144-4_1.

4 Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. Network flows: Theory, algorithms, and applications. 1993. URL: https://api.semanticscholar.org/CorpusID:12577796.

5 Stefan Berghofer and Tobias Nipkow. Executing Higher Order Logic. In *Types for Proofs and Programs*, 2002. doi:10.1007/3-540-45842-5_2.

6 Ran Duan, Seth Pettie, and Hsin-Hao Su. Scaling Algorithms for Weighted Matching in General Graphs. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, 2017. doi:10.1137/1.9781611974782.50.

7 Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, apr 1972. doi:10.1145/321694.321699.

8 L. R. FORD and D. R. FULKERSON. *Flows in Networks*. Princeton University Press, 1962. URL: http://www.jstor.org/stable/j.ctt183q0b4.

9 Harold N. Gabow. The Weighted Matching Approach to Maximum Cardinality Matching. *Fundam. Informaticae*, 2017. doi:10.3233/FI-2017-1555.

10 Harold N. Gabow and Robert Endre Tarjan. Faster Scaling Algorithms for Network Problems. *SIAM J. Comput.*, 1989. doi:10.1137/0218069.

11 Harold N. Gabow and Robert Endre Tarjan. Faster Scaling Algorithms for General Graph-Matching Problems. *J. ACM*, 1991. doi:10.1145/115234.115366.

12 Yon T. Gallai and Vorgelegt van G. HnjOs. Maximum-minimum sätze über graphen. *Acta Mathematica Academiae Scientiarum Hungarica*, 9:395–434, 1958. URL: https://api.semanticscholar.org/CorpusID:123953062.

13 Fabian Immler and Yong Kiam Tan. The Poincaré-Bendixson theorem in Isabelle/HOL. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, 2020. doi:10.1145/3372885.3373833.

14 Bernhard Korte and Jens Vygen. *Minimum Cost Flows*, pages 215–244. Springer Berlin Heidelberg, Berlin, Heidelberg, 2018. doi:10.1007/978-3-662-56039-6_9.

15 Alexander Krauss. *Automating Recursive Definitions and Termination Proofs in Higher-Order Logic*. PhD thesis, Technische Universität München, 2009.

16 Peter Lammich. Refinement to Imperative HOL. *Journal of Automated Reasoning*, 2019. doi:10.1007/s10817-017-9437-1.

17 Peter Lammich and S. Reza Sefidgar. Formalizing Network Flow Algorithms: A Refinement Approach in Isabelle/HOL. *J. Autom. Reason.*, 2019. doi:10.1007/s10817-017-9442-4.

18 Peter Lammich and Thomas Tuerk. Applying Data Refinement for Monadic Programs to Hopcroft's Algorithm. In *Interactive Theorem Proving*, 2012. doi:10.1007/978-3-642-32347-8_12.

19 Gilbert Lee. Correctnesss of ford-fulkerson's maximum flow algorithm1. *Formalized Mathematics*, 2005.

20 Filip Marić. Verifying Faradžev-Read Type Isomorph-Free Exhaustive Generation. In *Automated Reasoning*, 2020. doi:10.1007/978-3-030-51054-1_16.

21 Tobias Nipkow. Amortized Complexity Verified. In *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*, 2015. `doi:10.1007/978-3-319-22102-1_21`.

22 James Orlin. A faster strongly polynomial minimum cost flow algorithm. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, page 377–387, New York, NY, USA, 1988. Association for Computing Machinery. `doi:10.1145/62212.62249`.

23 James B. Orlin and Ravindra K. Ahuja. New scaling algorithms for the assignment and minimum mean cycle problems. *Math. Program.*, 1992. `doi:10.1007/BF01586040`.

24 Floris van Doorn, Patrick Massot, and Oliver Nash. Formalising the h-Principle and Sphere Eversion. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*, January 2023. `doi:10.1145/3573105.3575688`.

25 Niklaus Wirth. Program Development by Stepwise Refinement. *Commun. ACM*, 1971. `doi:10.1145/362575.362577`.