30 Aldwych, Bush House, London, WC2B 4BG, United Kingdom

A FORMAL ANALYSIS OF CAPACITY SCALING ALGORITHMS FOR MINIMUM-COST FLOWS

MOHAMMAD ABDULAZIZ [a] AND THOMAS AMMER [a]

Department of Informatics, King's College London
e-mail address: mohammad.abdulaziz@kcl.ac.uk, thomas.ammer@kcl.ac.uk

Abstract. We present formalisations of the correctness of executable 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 known running time for solving the problem of minimum-cost flow. We also include a formalisation of the worst-case running time argument for Orlin's algorithm. Our verified implementation of this algorithm, which is derived by the technique of stepwise refinement, is fully executable and was integrated into a reusable formal library on graph algorithms. Because the problems for which Orlin's algorithm works are restricted, we also verified an executable reduction from the general minimum-cost flow problem. We believe we are the first to formally consider the problem of minimum-cost flows and, more generally, any scaling algorithms. Our work has also led to a number of mathematical insights and improvements to proofs as well as theorem statements, compared to all existing expositions.

1. Introduction

Flow networks are some of 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 numerous connections to other important problems in computer science, most notably, the connection between maximum cardinality (maximum weight) bipartite matching and the problem of maximum (minimum-cost) 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 [EK72] 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 [Lee05], where the Ford-Fulkerson algorithm for maximum flow was verified. Later on, Lammich and Sefidgar [LS19] formally analysed the same algorithm and also the Edmonds-Karp algorithm [EK72], which is one of its polynomial worst-case running time refinements, in Isabelle/HOL.

We thank Jens Vygen for providing us with an elegant proof of Lemma 3.6, leading to a simplification of our own formalisation.

Preprint submitted to
Logical Methods in Computer Science

© M. Abdulaziz and T. Ammer
Creative Commons

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 targets, 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, such as shortest path, maximum flow, plus maximum weight and minimum weight perfect 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: (3a) successive shortest paths, which has an exponential worst-case running time, (3b) capacity scaling, which has a polynomial worst-case running time, (3c) Orlin's algorithm, which has a strongly polynomial worst-case running time, and (3d) the core of the running time argument for Orlin's algorithm. A noteworthy outcome of our work is that we present the first complete combinatorial proof for the correctness of Orlin's algorithm. For instance, an important property, namely, optimality preservation (Theorem 3.7), has a gap in all combinatorial proofs of which we are aware. We show that gaps existed in most combinatorial proofs (Lemma 3.6) and that some graphical arguments are complex when treated formally (Theorem 3.2, which needs so-called circulation decomposition (Lemma 2.9)), proving yet another example of complications uncovered in graphical and geometric arguments when treated in formal settings. This discrepancy between exact argument and graphical intuition is well known and is widely documented in the literature. An extreme case in another combinatorial optimisation algorithm formalised in our library is the Ranking algorithm for online bipartite matching, the majority of whose formalisation was spent on an argument originally described by one sketch [AM23]. Similar discrepancies were documented while formalising geometric arguments too [vMN23, IT20, AP19].

We use *abstract data types (ADT)* to model mathematical sets and functions in the formalisation of Orlin's algorithm. This allows us to obtain executable verified code, which still depends on functions selecting paths in the graph with certain properties. We also give implementations for those functions using the Bellman-Ford algorithm and Depth-First-Search (DFS). Because Orlin's algorithm only works on specific minimum-cost flow problems, we also formalised an executable reduction of general minimum-cost flow problems to the setting for which the algorithm works. To our knowledge, Orlin's algorithm is so far the most complex combinatorial optimisation algorithm for which there is verified executable code available.

Related Version. This work is an extension of a paper that we submitted to the 15th ITP conference in 2024 [AA24]. We added sections on the formal running time analysis (Section 5.4) and executability of Orlin's algorithm, the verification of an executable implementation of the Bellman-Ford algorithm (Section 6), as well as the formalisation of a reduction among flow problems (Section 8). The whole executability pipeline is new. We added background on ADTs (Section 2.3) and significantly extended the Sections 5, 5.1 and 5.2 (all rooted in the conference paper) with details on how to use ADTs for executability. Contrary to the conference paper, we now also work with multigraphs instead of simple graphs (see background section for terminology). We renamed the subprocedure *augment-edges* in the conference paper to *send-flow* in the current paper. We also refer to many Isabelle/HOL

code listings that were not mentioned in the conference versions which allows for a more detailed discussion of the formalisation. The conference paper already closed the gap of Lemma 3.6 for which we now present a simplified version.

Availability. The formalisation as presented here can be accessed on the repository `https://github.com/mabdula/Isabelle-Graph-Library` at the commit cfe2ab4b01a4bb11b3488555dfba1c3cfa4c0d25. In case of acceptance, a stable digital object identifier will be issued and referenced here.

Thomas: add zenodo with mincost flow only. Readme saying where which theorems/definitions from the paper can be found in the formalisation.

Presentation. In this paper, we have listings of Isabelle/HOL code taken from the formalisation. We sometimes removed technial preconditions or conclusions from theorem statements if the omissions are not interesting for the discussion. This is indicated by comments like `(*simplified*)`. Similarly, there can be symmetric cases in Isabelle/HOL where we sometimes write `(*analogous*)`. If a formalised statement has a named informal counterpart, we indicate the informal counterpart in a tag like [Lemma 2.6].

Thomas: perhaps add links to formalisation, comment here. Comment on (*simplified*)!

2. Background and Definitions

2.1. **Informal Definitions: Multigraphs and Flows.** We define a *simple directed graph* (henceforth 'simple graph') as a set $\mathcal{E}$ of ordered pairs, the so-called *edges*. Components of a pair e are the *endpoints* of the edge e. For the endpoints v_1 and v_2, we say that the edge (v_1, v_2) *leads* or *points* from v_1 to v_2. The endpoints of all edges in a graph is the set of the graph's vertices $\mathcal{V}$. In a simple directed graph, edges are unique, meaning there is at most one edge leading from v_1 to v_2 for all vertices v_1 and v_2. For example, $\mathcal{E} = \{(v_1, v_2), (v_2, v_3), (v_1, v_3)\}$ is a simple directed graph over the vertices $\mathcal{V} = \{v_1, v_2, v_3\}$.

On the contrary, a *directed multigraph* (henceforth 'multigraph') allows for multiple distinct edges pointing from one vertex to another. It is a set of objects, again called *edges*, together with two functions first and second mapping the edges to their first and second endpoint. If, for example, $(\cdot, \cdot)$ and $\langle \cdot, \cdot \rangle$ denote two distinct types of ordered pairs, then $\{(v_1, v_2), \langle v_1, v_2 \rangle, (v_2, v_3), (v_1, v_3), \langle v_3, v_1 \rangle\}$ is a multigraph involving the vertices v_1, v_2 and v_3. It is a multigraph because (v_1, v_2) and $\langle v_1, v_2 \rangle$ are two distinct edges leading from v_1 to v_2. For example, $\mathsf{first}(v_1, v_2) = v_1$, $\mathsf{first}\langle v_1, v_2 \rangle = v_1$ and $\mathsf{second}(v_2, v_3) = v_3$. We can interpret simple graphs as multigraphs if we define first and second to give the first and second element of an ordered pair, respectively.

We always assume directed multigraphs (henceforth 'directed graph'), that is, edges between vertices are not necessarily unique. In examples and figures, we often use the setting of a simple graph for the sake of simplicity.

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, we denote the set of all edges that have v as their first or second endpoint, meaning those that leave or enter the vertex v, by $\delta^+(v)$ or $\delta^-(v)$, respectively. Analogously to single vertices, we denote the set of leaving and entering edges of a set of vertices X by $\Delta^+(X)$ and $\Delta^-(X)$, respectively.

A *path over vertices* from v_1 to v_n is a sequence $v_1, ..., v_n \subseteq \mathcal{V}$ where there is an edge pointing from v_i to v_{i+1} for all $i < n$. A *path over edges* is a sequence $e_1, ..., e_n \subseteq \mathcal{E}$ where $\mathsf{second}(e_i) = \mathsf{first}(e_{i+1})$. It leads from $\mathsf{first}(e_1)$ (source) to $\mathsf{second}(e_n)$ (target). Any vertex-based path implies an edge-based path and the other way round, which is why we often identify the two notions in the informal analysis. A *cycle* is a path where source and target are identical.

A maximal set of vertices C, where there is, under omission of edge directions, a path between x and y 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 $x \in C$.

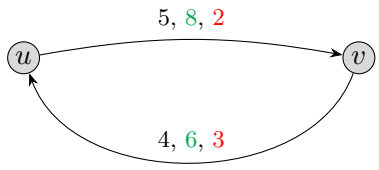
A *flow network* consists of a directed graph over edges $\mathcal{E}$ and vertices $\mathcal{V}$, and a capacity function $u : \mathcal{E} \to \mathbb{R}_0^+ \cup \{\infty\}$. If $u(e) = \infty$ for all $e \in \mathcal{E}$, the network is *uncapacitated*. The goal is to find a function called a *flow* $f : \mathcal{E} \to \mathbb{R}_0^+$ 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 *excess* of a flow f at the vertex v, $\mathrm{ex}_f(v)$, is the difference between ingoing and outgoing flow $\mathrm{ex}_f(v) \stackrel{\text{def}}{=} \sum_{e \in \delta^-(v)} f(e) - \sum_{e \in \delta^+(v)} f(e)$.

We call an ordered bipartition $(X, \mathcal{V} \setminus X)$ of the graph's vertices a *cut*. The (reverse) capacity of a cut X is the accumulated edge capacity of all (ingoing) outgoing edges $\mathrm{cap}(X) \stackrel{\text{def}}{=} \sum_{e \in \Delta^+(X)} u(e)$ ($\mathrm{acap}(X) \stackrel{\text{def}}{=} \sum_{e \in \Delta^-(X)} u(e)$). Edges going between X and $\mathcal{V} \setminus X$ are edges *crossing* the cut. If all edges going from X to $\mathcal{V} \setminus X$ are saturated for some flow, we say that the cut is saturated.

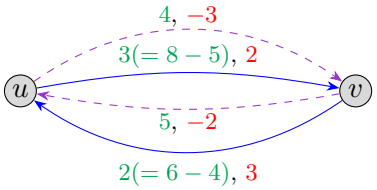
A *minimum-cost flow problem* consists of two further ingredients, apart from the edges and their capacities. We introduce *balances* $b : \mathcal{V} \to \mathbb{R}$ denoting the amount of flow that should be caught or emitted at every vertex. We call a flow satisfying the balance and capacity constraints *valid* or *feasible*. 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 (min-cost flow)* which is a feasible flow of least total costs. We call a network without cycles of negative total costs *weight-conservative*.

Example 2.1. An intuitive application of minimum-cost flows would be the transportation of products: at certain locations (vertices in the graph), for example, production and retail sites, there is a supply of and demand for a certain product P. We need to match demands and supplies to one another. Therefore, we must ship P along some ways of transport (edges in the graph). Each edge e comes with a transportation capacity $u(e)$ and costs $c(e)$ for each unit of G that we send along e. We can encode demands and supplies into the balances b, where $b(v) > 0$ is a supply and $b(v) < 0$ is a demand at the location v. A valid transportation plan means that (1) we remove all supply from all production facilities, (2) the demand at each retail site is exactly satisfied, and (3) the amount of G sent along e remains below $u(e)$ for each transportation link e. When modelling this as a flow problem, a minimum-cost flow f would be a valid transportation plan of minimum costs.

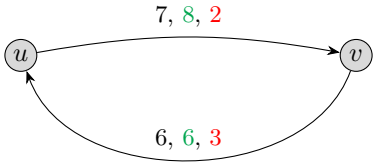
Given a flow f, we define the *residual network*: for any edge $e \in \mathcal{E}$ pointing from x to y, we have two *residual edges*, namely, the *forward* $F\,e$ and *backward* $B\,e$ edge pointing from x to y and from y to x, respectively. These form a pair of *reverse* edges. We write



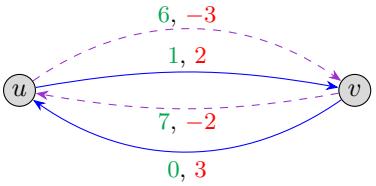
(1a) The original flow



(1b) Original Residual Graph



(1c) Flow after Augmentation



(1d) 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.

the reverse of a residual edge e as $\overleftarrow{e}$. We define the *residual cost* $\mathfrak{c}$ of a residual edge as $\mathfrak{c}(F\,e) = c(e)$ and $\mathfrak{c}(B\,e) = -c(e)$, 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 that we currently send through this edge: $\mathfrak{u}_f(F\,e) = u(e) - f(e)$. The capacity of a backward edge equals the flow assigned to the original edge: $\mathfrak{u}_f(B\,e) = f(e)$.

We define the residual capacity $\mathfrak{u}_f(p)$ of a path p as $\mathfrak{u}_f(p) = \min\{\mathfrak{u}_f(e)\,.\,e \in p\}$. We obtain the residual costs $\mathfrak{c}(p)$ by accumulating residual costs for the edges contained in p. We define the reverse $\overleftarrow{p}$ of a path p over residual edges as the reversal of each residual edge in p in reversed order. For example, the reverse of $[F\,e_1, B\,e_2, F\,e_3]$ is $[B\,e_3, F\,e_2, B\,e_1]$. From the definitions it follows that $\mathfrak{c}(\overleftarrow{p}) = -\mathfrak{c}(p)$.

The residual network is a multigraph itself regardless of whether the actual flow network is a simple graph. The residual network for a simple graph has two copies of the same edge, such as $F\,(u, v)$ and $B\,(u, v)$ (see the residual network in Figure 1b). Intuitively, a forward edge of the residual network indicates how much more could be added to the flow and a backward edge indicates how much could be removed from the flow.

Example 2.2. Figure 1a shows a flow network, where every edge is labelled by a flow, capacity, and a cost per unit of flow. The colouring convention from the caption applies. Figure 1b shows the residual network for the network in Figure 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 $B(x, y)$, going in the opposite direction. Each forward edge of the residual network is labelled by the residual capacity, indicating how much more flow can still go through the edge, without violating the edge's capacity constraint. Each backward edge is labelled by a capacity: intuitively this capacity represents how much flow we can remove from the edge and, accordingly, its value is the same as the flow going through the original edge. Each backward edge is labelled by a negative cost indicating that removing from the flow on that edge saves cost.

Example 2.3. In Figure 1b, the sequences of residual edges $[F\,(u, v),\ B\,(u, v)]$, $[F\,(u, v),$ $F\,(v, u), B\,(v, u), B\,(u, v)]$ and $[B\,(u, v), B\,(v, u)]$ are examples of edge-distinct paths in the

```
locale multigraph-spec =
  fixes ℰ :: 'edge set
  and fst :: 'edge ⇒ 'a and snd :: 'edge ⇒ 'a
  and create-edge :: 'a ⇒ 'a ⇒ 'edge
begin
...
definition delta-plus :: 'a ⇒ 'edge set (δ⁺) where
  δ⁺ v = {e. e ∈ ℰ ∧ fst e = v}
definition delta-minus :: 'a ⇒ 'edge set (δ⁻) where
  δ⁻ v =  {e. e ∈ ℰ ∧ snd e = v}
definition Delta-plus :: 'a set ⇒ 'edge set (Δ⁺) where
  Δ⁺ X = {e. e ∈ ℰ ∧ fst e ∈ X ∧ snd e ∉ X}
definition Delta-minus :: 'a set ⇒ 'edge set (Δ⁻) where
  Δ⁻ X = {e. e ∈ ℰ ∧ snd e ∈ X ∧ fst e ∉ X}
...
locale multigraph =
  multigraph-spec where ℰ = ℰ :: 'edge set for ℰ +
  assumes fst-create-edge: ⋀ x y. fst (create-edge x y) = x
  assumes snd-create-edge: ⋀ x y. snd (create-edge x y) = y
  assumes finite-E: finite ℰ
  assumes E-not-empty: ℰ ≠ {}
```

LISTING 1. Locales for Multigraphs

residual graph. Since the first and last vertices are identical (u, u and v, respectively), these even form cycles. The residual path capacities are 3, 2, 4, and the costs are 0, 0 and -5, respectively.

2.2. **Locales, Multigraphs and Minimum-Cost Flows in Isabelle/HOL.** In Isabelle/HOL, *locales* [Bal14] are named contexts that allow us to fix mathematical entities ('locale constants') and to specify/assume their properties. Inside the locale, we can use the constants and assumptions on them for definitions and proofs, respectively. We use locales to formalise multigraphs, as shown in Listing 1. The *specification locale* `multigraph-spec` fixes a set $\mathcal{E}$ of edges of type `'edge`, together with functions `fst` and `snd` from the edge to the vertex type. These are intended to return the first and second vertex of an edge, respectively. `fst` and `snd` correspond to the informal functions first and second that are associated with every multigraph. In the case of a simple graph, one would use the functions `fst` and `snd` that are predefined for ordered pairs in Isabelle/HOL. Furthermore, for every pair over the vertex type there should also be a possible edge of type `'edge` with the respective vertices as endpoints. We express this by the function `create-edge`. This edge does not have to be in $\mathcal{E}$, it only ensures the possibility of a connection between vertices, similar to the fact that we can construct an ordered pair for arbitrary components x and y.

The specification locale only fixes the constants while being ignorant of their behaviour. We have a second locale, the *proof locale* `multigraph` (see Listing 1), that extends the specification locale by making the assumptions that were informally stated in the previous paragraph. This distinction between a specification locale (fixing constants and types) and a proof locale (assuming the the constants' properties) is standard in our formalisation. The rationale behind this split follows from the mechanisms of *locale instantiation*.

We can later instantiate the constants fixed by a locale, as in the case of `multigraph` by giving a set $\mathcal{E}$ and implementations for `fst`, `snd` and `create-edge`. We must then prove all assumptions. For example, an instantiation of `multigraph` would require us to show finiteness of the $\mathcal{E}$ that is used for the instantiation. An instantiation turns every definition and theorem inside a locale into analogous definitions and theorems outside the locale. The latter involve the entities used to instantiate the locale constants.

Example 2.4. By instantiating, a definition `share-vertex e1 e2` $= \{$`fst e1`, `snd e1`$\} \cap \{$`fst e2`, `snd e2`$\} \neq \{\}$ inside `multigraph-spec` to check whether two edges share a vertex, becomes a constant `share-vertex` outside the locale that again takes two edges, but whose type and implementation of `fst` and `snd` is fixed according to the concrete instantiation, for instance, to the product type and the predefined `fst` and `snd`.

Isabelle/HOL also allows for the generation of executable code from definitions given in a locale, provided that the definitions are executable themselves. This requires a so-called *global interpretation*, which is a certain type of locale instantiation. Global interpretations must be outside of any context and can be generic with respect to all locale constants on which there are no non-trivial assumptions. For example, a global interpretation of `multigraph` only works for specific sets of edges, since some sets are finite and others are not. On the contrary, one can interpret `multigraph-spec` globally for all possible values of the locale constants because there are no assumptions.

As noted earlier, we distinguish between a specification locale, which fixes constants and types, and a proof locale, which makes assumptions about the constants' properties. We put every function definition that is meant to be executed into the specification locale. On the other hand, we place any reasoning regarding the correctness of an algorithm in the proof locale. By this, a single global interpretation will be sufficient to obtain code that would work for several input graphs. This code has the locale constants, such as `fst` or `snd` as function arguments. Correctness of the code for a specific input depends on a few simple properties, namely, that the input indeed satisfies the assumptions made by the proof locale.

Within the multigraph locale from Listing 1, we can already define basic concepts like δ^+, δ^-, Δ^+ and Δ^-.

In Listing 2, we use a functional data type to model residual edges. As an extension of `multigraph-spec`, we have a specification locale for flow networks with capacities (edges mapped to reals with infinity) but without costs. Within that locale, we formalise the concepts of flow excess, compliance with balance and capacity constraints, as well as, residual capacities. The proof locale for flow networks (`flow-network`) assumes that capacities are non-negative. The locale `cost-flow-spec` (Listing 3) extends `flow-network-spec` by assuming a cost function from edges to reals. In that locale, we define residual costs, and the notion of minimum-cost flows. `is-Opt b f` would mean that `f` is a minimum-cost flow for balances `b` and the edge capacities `u` fixed by the locale.

There are two reasons why it is beneficial to consider flows in multigraphs as formalised in Listing 1 instead of simple graphs formalised as a set of ordered pairs. Firstly, when modelling a real-world problem, such as the transportation of goods (Example 2.1), there can be multiple direct links between two sites v_1 and v_2, each of which has different per-unit costs. It is desirable to have algorithms that directly deal with that. The other reason is that we can easily interpret the residual graph as another flow network. The latter uses, however, the residual edges, residual capacities and residual costs. From time to time, flows in the residual graph are considered in informal proofs. Reflecting this in the formalisation

```
datatype 'type Redge = is-forward: F 'type | is-backward: B 'type

locale flow-network-spec =
  multigraph-spec where ℰ = ℰ :: 'edge set for ℰ  +
  fixes u :: 'edge ⇒ ereal
begin
...
definition ex :: ('edge ⇒ real) ⇒ 'a ⇒ real where
  ex_f v = (∑ e ∈ δ⁻ v. f e) - (∑ e ∈ δ⁺ v. f e)
definition isuflow :: ('edge ⇒ real) ⇒ bool where
  isuflow f ⟷ (∀ e ∈ ℰ. f e ≤ u e ∧ f e ≥ 0)
definition isbflow :: ('edge ⇒ real) ⇒ ('a ⇒ real)  ⇒ bool where
  f is b flow ⟷ (isuflow f ∧ (∀ v ∈ 𝒱 . (- ex f v) =  b v))

fun rcap :: ('edge ⇒ real) ⇒ 'edge Redge ⇒ ereal where
  u_f (F e) =  u  e - f e |
  u_f (B e) =  f e
end

locale flow-network =
  multigraph where ℰ = ℰ :: 'edge set +
  flow-network-spec where ℰ = ℰ :: 'edge set for ℰ +
assumes u-non-neg: ⋀ e. u e ≥ 0
```

LISTING 2. Locales and Definition for Flow Networks

```
locale cost-flow-spec =
  flow-network-spec where ℰ = ℰ :: 'edge set for ℰ   +
  fixes c :: 'edge ⇒ real
begin
...
fun 𝔠 :: 'edge Redge ⇒ real where
𝔠 (F e) = c e|
𝔠 (B e) = - c e
...
definition 𝒞 f = (∑ e ∈ ℰ. (f e) * (c e))
definition is-Opt b f = (f is b flow ∧ (∀ f'. (f' is b flow ⟶ 𝒞 f' ≥ 𝒞 f)))
```

LISTING 3. Locale and Definition for Flow Networks with Costs

is straightforward when using multigraphs from Listing 1. For example, the Isabelle/HOL script in Listing 4 yields theorems about flows in a specific residual network.

2.3. **Obtaining Executable Algorithms.** Functions and sets as they are available in Isabelle/HOL are already sufficient to formalise supporting theory and to describe the algorithms mathematically. Formalised in that way, algorithms are often not executable and, even if they are, the running time might be inflated. For example, in a mathematical description, one might perform updates to a function, that maps indices to values to model an array, by an `if`-expression. Due to the evaluation strategy, these build up to nested `if`s

```
interpretation residual-flow: cost-flow-network
  where fst = fstv and snd = sndv and create-edge = create-edge-residual
    and ℰ = 𝔈 and c = 𝔠 and u = λ -. PInfty
```

LISTING 4. Interpreting the Residual Graph as a Flow Network

`fstv` and `sndv` return the first and second endpoint a residual edge. $\mathfrak{E}$ and $\mathfrak{c}$ are the set of residual edges and the residual costs, respectively. Capacities are set to ∞ in this example.

resulting in a running time that linearly depends on the number of updates. For Orlin's algorithm, which is the most efficient algorithm presented in this paper, we aim for executable code of reasonable efficiency.

Remark 2.5. The running time analysis in the literature is done with respect to an imperative model of computation. Thus, we aim to bring the formalisation of Orlin's algorithm closer to the semantics of an imperative language. There is also some ongoing effort to implement an automated translation of Isabelle/HOL functions to IMP- [KHSA25], a deliberately simple but Turing-complete imperative language used to study computability and complexity theory. In the future, we might attempt this for Orlin's algorithm. Full executability is a prerequisite, however.

Definition of Abstract Data Types. One approach to obtain executable Isabelle/HOL functions with reasonable running time are so-called *abstract data types (ADT)*. These have been studied first by Wirth [Wir71] and Hoare [Hoa72], and are an example of Wirth's concept of stepwise refinement where a program instruction is decomposed into more detailed instructions. For ADTs, the scope of application is much wider than our own and immediately related work, as also Java, C or Haskell libraries use them to specify interfaces for sets or queues.

In the context of ADTs, for example, we can describe the behaviour of $\cup_{impl}$ (implementation of set union $\cup$) by $\alpha(A \cup_{impl} B) = \alpha(A) \cup \alpha(B)$ where α is an abstraction function denoting the semantics of a set implementation, that is, which implementations represent which mathematical sets. Mathematically speaking, α has to be a homomorphism from the implementations to mathematical sets with respect to the set operations. A data structure invariant, which is a well-formedness property that the operations have to preserve, is usually a precondition.

Abstract Data Types in Isabelle/HOL. The built-in standard library of Isabelle/HOL contains specifications of several ADTs by Nipkow, maps and sets, for example (see Listings 5 and 6). For example, a map modelling a (partial) function f would return `Some x` if $f(x)$ is defined and `None` otherwise. For a map `m`, we call the set of those `x` where `lookup m x` $\neq$ `None` the *domain* of `m`. Like flow networks or multigraphs, we formalise ADTs by means of locales. Within these and derived locales, we can use the properties of the assumed set or map implementations, provided that the well-formedness invariant `invar` holds. Later, we can instantiate the locales to obtain executable code.

For example, Abdulaziz [Abd25] used these locales to model graphs as adjacency maps, which are maps mapping vertices to set implementations of their neighbourhoods (sets of the adjacent vertices), and to verify and obtain executable implementations of

```
locale Set =
  fixes empty :: 's            and insert :: 'a ⇒ 's ⇒ 's
  and isin :: 's ⇒ 'a ⇒ bool and set :: 's ⇒ 'a set       and 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 invar-empty: invar empty
  assumes invar-insert: invar s ⟹ invar (insert x s)
```

Listing 5. Abstract Data Type Specification for Sets

```
locale Map =
  fixes empty :: 'm                    and update :: 'a ⇒ 'b ⇒ 'm ⇒ 'm
  and lookup :: 'm ⇒ 'a ⇒ 'b option and 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 invar-empty: invar empty
  and invar-update: invar m ⟹ invar (update a b m)
```

Listing 6. Abstract Data Type Specification for Maps

simple graph algorithms like Breadth-First-Search (BFS) or Depth-First-Search (DFS). Notably, he can mostly hide the fact that the algorithms work on assumed implementations instead of mathematical functions and sets by automation that applies the data structure semantics (Listings 5 and 6) in the background. Another example is the formalisation of algorithms for matroids and greedoids [AAMR25], where ADTs allow for deriving executable implementations of Kruskal's and Prim's algorithms, and maximum cardinality bipartite matching. We use the ADT approach in our work to obtain executable code for Orlin's algorithm from the formalisation.

2.4. **Fundamental Results in Flow Theory.** Proofs about flows often require some kind of decomposition of the network or the flow itself. Cuts allow us to view two parts X and $\mathcal{V} \setminus X$ of the network as single vertices connected by a pair of edges in opposite directions. For a flow f valid for balances b, the sum of b over X is the difference between the flow going from X to $\mathcal{V} \setminus X$ and the flow going from $\mathcal{V} \setminus X$ to X. We make this precise in the *Flow-Value Lemma*, a fundamental result in flow theory:

Lemma 2.6 (Flow Value Lemma)**.** *Let f be a flow that is valid for the balances b and let $(X, \mathcal{V} \setminus X)$ be a cut. Then*

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

Listing 7 contains the formalisation of the Flow-Value Lemma. We conduct the proof by replacing $b(v)$ with $-ex_f(v)$ and the observation that the flow $f(e)$ along edges e not crossing the cut is cancelled since it appears both as $f(e)$ and $-f(e)$ in the sum.

Cuts are an important tool for finding a bound for the amount of flow that we send between different parts of the network. For example, it follows from the definition of cuts

```
theorem flow-value: [Lemma 2.6]
  assumes f is b flow and X ⊆ 𝒱
    shows (∑ v ∈ X. b v) = (∑ e ∈ Δ⁺ X. f e) - (∑ e ∈ Δ⁻ X. f e)

corollary flow-less-cut-capacities: [Corollary 2.7]
 assumes f is b flow and X ⊆ 𝒱
   shows ∑ v ∈ X. b v ≤ Cap X and ∑ v ∈ X. b v ≥ - ACap X

theorem rescut-ingoing-zero-ingoing-saturated: [Lemma 2.8]
 assumes f is b flow and (Rescut f v) ⊆ 𝒱
   shows ∑ e ∈ (Δ⁺ (Rescut f v)). f e = Cap (Rescut f v)
     and ∑ e ∈ (Δ⁻ (Rescut f v)). f e = 0
```

Listing 7. Results on Cuts

and flow validity that we have $0 \leq \sum_{e \in \Delta^+(X)} f(e) \leq \mathrm{cap}(X)$ and $0 \leq \sum_{e \in \Delta^-(X)} f(e) \leq \mathrm{acap}(X)$ for any valid flow f and cut X. This implies together with the Flow-Value Lemma:

Corollary 2.7.

$$-\mathrm{acap}(X) \leq \sum_{v \in X} b(v) \leq \mathrm{cap}(X)$$

for any flow f valid for balances b and cut X.

The intuition behind this statement is that the sum of the balances in a cut is the amount of flow to be sent or removed from the cut. The cut capacities are a natural bound for the movement of flow. Listing 7 shows the formalisation of Corollary 2.7.

For a vertex v and a valid flow f, we call those vertices that are reachable from v by a path over residual edges with positive residual capacity a *residual cut* $Rescut_f(v)$. A simple and well-known fact follows from the definition of the residual cut:

Lemma 2.8. *For a residual cut X around a vertex v, every edge in $\Delta^+(X)$ is saturated, meaning the total flow leaving X is $\mathrm{cap}(X)$ and every edge in $\Delta^-(X)$ has zero flow.*

Listing 7 also shows a formalisation of this statement. If $f(e) < u(e)$ for $e \in \Delta^+(X)$ or if $f(e) > 0$ for some $e \in \Delta^-(X)$, there would be a residual edge with positive residual capacity leaving X, which contradicts the definition of the residual cut.

Apart from cuts, *flow decomposition* [Gal58, FF62, AGOT92] is another fundamental technique when reasoning about flows. There are several decomposition results, one of which is the decomposition of so-called circulations, which is also relevant to minimum-cost flows. We call a flow g that is valid for $b(v) = 0$, that is, $ex_g(v) = 0$ for all $v \in \mathcal{V}$, a *circulation*.

Lemma 2.9 (Decomposition of Circulations [AGOT92])**.** *For any circulation g, we can find a decomposition, which is that there is a set of cycles $\mathcal{C}$ and weights $w : \mathcal{C} \to \mathbb{R}^+$ where any edge e has $g(e) = \sum_{C \in \mathcal{C} \wedge e \in C} w(C)$. Moreover, if $g(e) \in \mathbb{N}$ for all e, then $w(C) \in \mathbb{N}$ for all $C \in \mathcal{C}$.*

This is illustrated by Figure 2.

Proof Sketch. We use a greedy search to find cycles: as long as there is e with $g(e) \neq 0$, there must be a vertex that has a leaving edge e with $g(e) > 0$. Since g is a circulation with zero

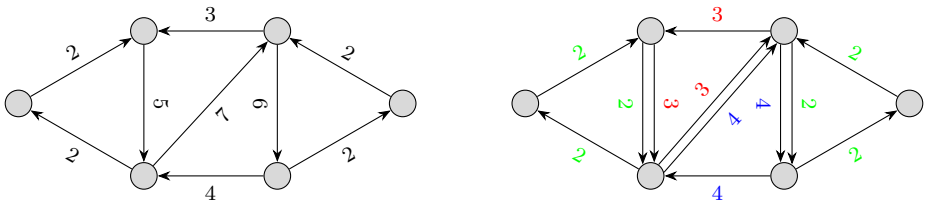


FIGURE 2. A Circulation and its Decomposition

excess everywhere, for every v and e_1 with $e_1 \in \delta^-(v)$ and $g(e_1) > 0$ we can find $e_2 \in \delta^+(v)$ with $g(e_2) > 0$. If we repeat this search $n = |\mathcal{V}|$ times, a vertex in the constructed path is repeated implying a cycle C. We assign $w(C) := \min_{e \in C} g(e)$, decrease $g(e)$ by $w(C)$ for every $e \in C$ and continue this process until $g(e) = 0$ for all e. □

Formalisation of Circulation Decomposition. Here, we encounter some difficulties with non-algebraic/graphical/geometric reasoning: The existence of a structure (a cycle) is intuitively obvious, yet we need to make its construction formally precise which is done by the function `find-cycle` (Listing 8). It searches for a path by modelling the "steps" from the path's informal construction. The formalisation of Lemma 2.9 is shown in the same Listing. The "continue this process" from the informal proof corresponds to formally proving the statement by induction on the cardinality of the support (set of edges with strictly positive flow). The hardest part is showing that the flow remains a circulation, that is, a flow with zero excess at every vertex, after decreasing the flow along one of its cycles. The central property, which one would barely pay attention to informally, is that for every cycle C found by `find-cycle` and for every vertex v in the graph, v either has no leaving or entering edge in C or exactly one leaving and one entering edge in C.

3. Towards a simple Algorithm

Augmentation is a principal technique in combinatorial optimisation by which we incrementally improve a candidate solution 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. We augment along a backward edge by decreasing the flow value of the original edge by γ. One can augment along a path 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 3.1. One can see the result of augmenting our example flow from Figure 1a in Figure 1c, where we augment the flow along the edges (u, v) and (v, u) by 2. Figure 1d shows the resulting new residual network. Please note the change in capacities in forward and backward edges.

Listing 9 contains the formalisation of augmentation: `augment-edge` augments the flow `f` along a single edge `e`, which `augment-edges` uses to augment along a sequence of residual edges. We can then show, for example, that augmenting a valid b-flow f by $\gamma \leq \mathfrak{u}_f(p)$ along an augmenting path P from a vertex s to a vertex t results a flow f' that is valid for the

```
fun find-cycle :: ('edge ⇒ real) ⇒ nat ⇒ 'a ⇒ 'a list where
  find-cycle g 0 v = [v] |
  find-cycle g (Suc n) v =
    v # (if (∃ e ∈ ℰ. g e > 0 ∧ fst e =  v)
         then (let e = (SOME e. g e > 0 ∧ fst e = v ∧ e ∈ ℰ)
                in find-cycle g n (snd e))
         else [])

theorem flowcycle-decomposition: [Lemma 2.9]
assumes g ≥_F 0 and 0 < Abs g and is-circ g
    and support g ≠ {} and card (support g) = n
  shows ∃ css ws.
 length css = length ws ∧ set css ≠ {} ∧ (∀ w ∈ set ws. w > 0) ∧
 (∀ cs ∈ set css. flowcycle g cs ∧ set cs ⊆ support g ∧ distinct cs) ∧
 (∀ e ∈ ℰ. g e = (∑ i < length css. (if e ∈ set (css ! i) then ws ! i else 0)))
 ∧ (is-integral-flow g ⟶ (∀ w ∈ set ws. ∃ n :: nat. w = real n))
```

Listing 8. Decomposition of Circulations

Explanations: `g` $\geq_F 0$ means `g(e)` ≥ 0 for all `e`. `Abs g` is the sum of $|$`g(e)`$|$ over all `e`. The support is the set of all `e` where `g(e)` is strictly positive. A `flowcycle` is a cycle along which the flow `g` is strictly positive. `g` is integral if and only if `g(e)` is integral for all `e`.

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

SSP1 initialise $b' \leftarrow b$ and $f \leftarrow 0$;
SSP2 **while** *True* **do**
SSP3 | **if** $\forall v \in \mathcal{V}.\ b'(v) = 0$ **then**
SSP4 | | **return** f as optimum flow;
SSP5 | **else**
SSP6 | | take some s with $b'(s) > 0$;
SSP7 | | **if** $\exists t$ *reachable from* $s \wedge b'(t) < 0$ **then**
SSP8 | | | take such a t and a minc-ost augpath P from s to t;
SSP9 | | | $\gamma = \min\{b(s), -b(t), \mathfrak{u}_f(P)\}$;
SSP10 | | | augment γ along P;
SSP11 | | | $b'(s) \leftarrow b'(s) - \gamma$; $b'(t) \leftarrow b'(t) + \gamma$;
SSP12 | | **else return** *infeasible*;

balances b' where $b'(s) = b(s) + \gamma$, $b'(t) = b(t) - \gamma$ and $b'(v) = b(v)$ for $v \neq s, t$ (Listing 9). The listing also shows that the costs change by $\gamma \cdot \mathfrak{c}(p)$ when augmenting along P by γ.

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. We keep the balances b generic, since these are subject to change during the computation. 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, that is, a minimum-cost path in the residual network

```
fun augment-edge where
  augment-edge f γ e = (λ d. if d = oedge e
                             then (case e of F d ⇒ f d + γ | B d ⇒ f d - γ)
                             else f d)
fun augment-edges where
  augment-edges f γ [] = f|
  augment-edges f γ (e#es) = augment-edge (augment-edges f γ es) γ e

corollary  augment-path-validness-b-pres-source-target-distinct:
  assumes f is b flow and 0 ≤ γ and  augpath f es  and γ ≤ Rcap f (set es) and
          set es ⊆ 𝔈 and distinct es and fstv (hd es) ≠ sndv (last es)
    shows (augment-edges f γ es) is (λ v. if v = fstv (hd es) then b v + γ
                                           else if v = sndv (last es) then b v - γ
                                           else b v) flow
lemma cost-change-aug:
  assumes set es ⊆ 𝔈 and distinct es
    shows 𝒞 (augment-edges f γ es) = (𝒞 f + γ * (∑ e ∈ set es. 𝔠 e))
```

Listing 9. Augmentation in Isabelle/HOL

Explanations: `oedge` denotes the graph edge a residual edge arises from, that is, e for $F\ e$ and $B\ e$. `set` takes a list (a finite sequenc)e and returns its elements. `Rcap` is the minimum residual capacity of a set of residual edges which is $\mathfrak{u}_f(p)$ for the edges in an augmenting path P. `hd` and `last` return the first and last element of a list. Here, $\mathcal{C}$ denotes the costs $c(f)$ of a flow f.

connecting s to t. Following that, it sends as much flow as possible, that is, as much as the minimum capacity of any residual edge in P or as the balances of s and/or t allow, from s to t along P. We lower and increase the balances at s and t by the same amount, respectively. We continue this until all balances reach zero or until we can infer infeasibility from the absence of an augmenting path.

Conceptually, the algorithm fixes the variables $\mathcal{E}$, $\mathcal{V}$, u, c and b, and works on *program states* consisting of the changing variables f (current flow) and b' (remaining balances). Invariants are predicates defined on states and might use the fixed variables, as well. If not all variables are relevant to an invariant, we say that only the involved variables satisfy the invariant.

Listing 10 displays the formalisation of Algorithm 1 as well as the program state. We represent the loop as a recursive function. The non-trivial termination argument requires the Isabelle function package [Kra09]. We use records to model program states. Most notation is standard functional programming notation. The main exception is record updates, such as `state(return := infeasible)` denoting `state`, but state with the variable `return` updated to `infeasible`. Formally, selecting reachable targets and minimum-cost paths corresponds to using functions (`get-source`, `get-reachable-target` and `get-min-augpath`, respectively) that compute those items non-deterministically. These are fixed by a locale called `SSP` that also makes assumptions about the functions' behaviour.

```
datatype return = success | infeasible | notyetterm
record ('b, 'edge) Algo-state = current-flow :: 'edge ⇒ real
                                balance :: 'b ⇒ real
                                return :: return

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 := success|)
      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 = s then b s - γ else
                                          if v = t then b t + γ else b v)
                           in SSP (state(|current-flow := f', balance := b'|)))))))
```

LISTING 10. Formalising the *SSP* Algorithm

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 the procedure *successive-shortest-path*$(\mathcal{G}, b)$ (Algorithm 1):

(ISSP-1) The flow f is a minimum-cost flow for the balance $b - b'$[1].
(ISSP-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.
(ISSP-3) The sum of b' over all vertices v is zero: $\sum_{v \in \mathcal{V}} b'(x) = 0$.

Invariant ISSP-1 is well known in the literature [KV12] and we made the two simple ones explicit by the formalisation.

Proving Invariant ISSP-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 3.2 (Optimality Criterion [Kle67])**.** *A flow f valid for balance b is optimum iff there is no augmenting cycle w.r.t. f.*

Proof Sketch. ($\Rightarrow$) An augmenting cycle is a possibility to decrease costs while still meeting the balance constraints.
($\Leftarrow$) Assume a valid flow f' with $c(f') < c(f)$. We define the flow g in the residual graph as $g(F\,e) = \max\{0, f'(e) - f(e)\}$ and $g(B\,e) = \max\{0, f(e) - f'(e)\}$. g is a flow in the residual graph with zero excess for every vertex, in other words, a circulation. Moreover, $\mathfrak{c}(g) = c(f') - c(f)$ and any residual edge e with $\mathfrak{u}_f(e) = 0$ has $g(e) = 0$. We can decompose the circulation g according to Lemma 2.9, implying 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

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

```
lemma diff-is-res-circ: [For Theorem 3.2]
  assumes f is b flow and f' is b flow
    shows residual-flow.is-circ (difference f' f)

lemma rcost-difference: [For Theorem 3.2]
  residual-flow.𝒞 (difference f' f) = 𝒞 f' - 𝒞 f

theorem no-augcycle-min-cost-flow: [For Theorem 3.2]
  assumes f is b flow and ∄ cs. augcycle f cs shows is-Opt b f

corollary is-opt-iff-no-augcycle: [Theorem 3.2]
  assumes f is b flow shows is-Opt b f ⟷ (∄ cs. augcycle f cs)
```

Listing 11. Formalisation of the Optimality Criterion

$0 > c(f') - c(f) = \mathfrak{c}(g) = \sum_{C \in \mathcal{C}} w(C) \cdot \mathfrak{c}(C)$, 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. □

To formalise Theorem 3.2, we interpret the residual graph as a flow network (Listing 4). We show that the difference flow is a circulation in the residual graph (Listing 11). This allows us to apply the formal counterpart of Lemma 2.9 to the difference flow. After reasoning about the costs of the difference flow, we can show the second direction of Theorem 3.2. The full formal statement for Theorem 3.2 is displayed in Listing 11.

For proving the mathematical main result of this section, namely, Theorem 3.7 [Jew58], one has to prove a topological property if certain edges are removed from a graph (Lemma 3.6). This is a gap in all published proofs of which we are aware and we document our reasoning here.

We consider colourings of the residual edges in red and blue. We count the same edge coloured once in red and once in blue as two distinct edges. We saw that for a graph G, the residual edges emerging from the edges in G form a multigraph. For any set R of residual edges (which is a multigraph with `fstv` and `sndv` in the sense of Section 2.1), the graph consisting of all edges in R both in red and blue is another multigraph, provided that `fstv` and `sndv` are defined to ignore the colors. The notions of paths and cycles as defined in Section 2.1 apply again.

Example 3.3. Assume a simple graph consisting of three vertices v_1, v_2 and v_3 which are all connected to one another. When taking colours into account, $p = [F\,(v_1, v_2), F\,(v_2, v_3), B\,(v_2, v_3), B\,(v_3, v_2), F\,(v_3, v_1), F\,(v_1, v_2)]$ is an edge-distinct path visiting v_1, v_2, v_3, v_2, v_3, v_1 and v_2. $F\,(v_1, v_2)$ and $F\,(v_1, v_2)$ are two distinct edges which is why the path is edge-distinct despite the repetition. If we drop the last edge, we obtain a cycle leading from v_1 to v_1. We see that for one edge e in the original graph, there would be four coloured residual edges: $F\,e$, $B\,e$, $F\,e$ and $B\,e$. (P does not use all of them, however).

We call pairs of residual edges of different colours, where one edge is the reversed one of the other, *forward-backward-pairs (FBP)*, such as $F\ e$ in blue and $B\ e$ in red. *FBP*s are involved in lemmas one can use to prove preservation of the optimality invariant of Algorithm 1. Subsequently, *disjointness* of paths and cycles means their edge-disjointness.

Example 3.4. In Example 3.3, $\{F\,(v_2, v_3), B\,(v_2, v_3)\}$ is the only FBP among p's edges.



FIGURE 3. Eliminating FBPs: Members of an FBP belong either to the same (left) or to two different cycles (right). When we drop the FBP on the left, we obtain two new cycles. On the right, FBP deletion results in a single new cycle. Disjointness is preserved.

Lemma 3.5. *Deleting all FBPs from a set of disjoint cycles yields another set of disjoint cycles.*

Proof Sketch. Proof by induction on the number of FBPs. For every FBP, the partners either belong to the same or to a different cycle, as one can see in Figure 3. Deleting an FBP results in two or a single new cycle, respectively. □

Lemma 3.6. *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. We add an edge e^* from t to s. By definition, let it be distinct from any other edge and not be member of any FBP. The result of adding this additional edge is an edge-disjoint set of edge-distinct cycles $\mathcal{C}$, to which we can apply Lemma 3.5. This yields another set of disjoint cycles $\mathcal{D}$ where exactly the FBPs have been removed. As e^* cannot be member of an FBP in $\mathcal{C}$, one of the cycles from $\mathcal{D}$ must still contain this edge. Removing e^* from $\mathcal{D}$ results in an s-t-path and some cycles. □

The following theorem implies the preservation of Invariant ISSP-1.

Theorem 3.7 (Optimality Preservation [Jew58])**.** *Let f be a minimum-cost flow for balances b. Take an s-t-path P of minimum residual costs and $\gamma \leq \mathfrak{u}_f(P)$. 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 3.2) contradicting the optimality of P. Assume the flow f' after the augmentation is not optimum. By Theorem 3.2, 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.

We now construct an augmenting cycle for f. First colour edges in P blue and edges in C red. Neither P nor C can contain any FBPs. We can therefore apply Lemma 3.6 to the union of P and C yielding another s-t-path P' and a set of cycles $\mathcal{C}'$. Their edges 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, we would have deleted this. 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, which, together with Theorem 3.2, contradict the assumption that f is optimal. □

Listing 12 shows the formal statement for Theorem 3.7.

```
theorem path-aug-opt-pres: [Theorem 3.7]
  assumes s ≠ t and is-Opt b f and γ ≤ Rcap f (set P)
      and 0 ≤ γ and is-min-path f s t P
      and f'-def: f' = augment-edges f γ P
      and b'-def: b' = (λv. if v = fstv (hd P) then b v + γ
                            else if v = sndv (last P) then b v - γ else b v)
    shows is-Opt b' f'
```

Listing 12. Formalisation of Theorem 3.7

Theorem 3.8 (Correctness of Algorithm 1 [EK72])**.** *Assume the sum of balances b over $\mathcal{V}$ is zero. An execution of successive-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 ISSP-1) initially true. The preservation of Invariant ISSP-1 follows from Theorem 3.7. Invariants ISSP-3 and ISSP-2 also hold initially. We can see their preservation 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 a further auxiliary result. We note that f is a valid flow w.r.t. $b - b'$. Lemma 2.6 (Flow-Value Lemma) and Lemma 2.8 yield for any b and any flow f valid w.r.t. b:

$$\sum_{x \in Rescut_f(v)} b(x) = \sum_{e \in \Delta^+(Rescut_f(v))} f(e) = \mathrm{cap}(Rescut_f(v)) \tag{3.1}$$

From the algorithm's control flow we can infer that, if the algorithm returns *infeasible* there must be an s with $b'(s) > 0$ without a reachable t where $b'(t) < 0$, that is, any x in the rescut has a $b'(x) \geq 0$. We show that there is a contradiction if there exists a flow f' valid w.r.t. b.

$$\begin{aligned}
\sum_{x \in Rescut_f(s)} b(x) &\leq \mathrm{cap}(Rescut_f(s)) && (f' \text{ is valid flow and Corollary 2.7}) \\
&= \sum_{x \in Rescut_f(s)} (b - b')(x) && (\text{Equality 3.1 for } f \text{ and } b - b') \\
&< \sum_{x \in Rescut_f(s)} b(x) && \square
\end{aligned}$$

Formalising Correctness of Algorithm 1. We introduced definitions to specify which execution branch is taken when doing an iteration of the loop body, such as `SSP-call-4-cond` indicating the recursive case from the function definition in Listing 10. It has the same structure as the loop body and returns `True` for exactly one branch and `False` for all others. Similarly, we model the effect of a single execution branch by expressions like `SSP-upd4 state`. We can show a simplification lemma and an induction principle for `SSP`, given in Listing 13. We prove the preservation of the invariants for single updates like the ones in Listing 14 and by the simplification and induction lemmas, we can lift this to a complete execution of `SSP`. Proof automation makes this process very smooth and convenient. We

```
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)

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 13. Customised Rules for *SSP*

```
definition invar-opt state =    [Invariant ISSP-1]
    (is-Opt (λ v. b v - balance state v) (current-flow state))

lemma invar-opt-holds-3: [For Theorem 3.8]
 assumes SSP-ret-3-cond state and invar-opt state
   shows invar-opt (SSP-ret3 state)

lemma invar-opt-holds-4: [For Theorem 3.8]
 assumes SSP-call-4-cond state and invar-opt state
   shows invar-opt (SSP-upd4 state)

lemma invar-opt-holds: [For Theorem 3.8]
   assumes SSP-dom state invar-opt state
   shows invar-opt (SSP state)
```

Listing 14. Single-Step Preservation of Optimality

```
lemma invar-opt-initial-state: [For Theorem 3.8]
 invar-opt (|current-flow = (λ e. 0), balance = b, return = notyetterm|)

theorem total-correctness: [Theorem 3.8]
 assumes final = SSP (|current-flow = (λ e. 0), balance = b, return = notyetterm|)
   shows return final = success ⟹ is-Opt b (current-flow final)
         return final = infeasible ⟹ ∄ f. is-Opt b f
         return final = notyetterm ⟹ False
```

Listing 15. Total Correctness. Note: for a function `f`, `f-dom input` means that `f` terminates on `input`.

show the invariants for the initial state and obtain total correctness (Listing 15). We follow the same formalisation methodology for all the algorithms we consider below.

Remark 3.9. Lemma 3.6 is a gap in all combinatorial proofs of the algorithm's correctness to which we have access, including the proof by Korte and Vygen [KV12]. The only other complete proof of which we are aware is a non-combinatorial proof by Orlin [Orl93, AMO93], in which he uses LP-theory. The main challenge is being able to show that, if after the

augmentation, f' is not optimal, then f must have also not been optimal. To construct an augmenting cycle for f, we need to extract one cycle from C that is disjoint from all other cycles in C. Otherwise the proof that one cycle has negative cost would be impossible due to the interference in summations of costs of members of C. To do that we had to devise the machinery of colouring the edges, FBPs, and the removal of FBPs. Other than that machinery, the proof boils down to one trick: adding an extra edge to turn the path P into a cycle makes the set of edges from C and P a union of disjoint cycles, which is more uniform, and which allows us to easily prove Lemma 3.6.

4. The Capacity Scaling Algorithm

The naive successive shortest path algorithm (Algorithm 1) arbitrarily selects sources, targets and min-cost paths for the augmentations. We refine this to *Capacity Scaling (CS)* by selecting those triples where the residual capacities and balances are above a certain threshold that we halve from one scaling phase to another (Algorithm 2). It was proposed by Edmonds and Karp [EK72]. As SSP, CS fixes $\mathcal{E}$, $\mathcal{V}$, u, c and b, and works on a state consisting of the changing variables 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. We can reapply the same three invariants. 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 $\min\{\mathtt{u}_f(p), b(s), -b(t)\}$. When no paths of the right cost remain, it halves the thresholds for treatment and continues with a more fine-grained analysis.

Partial Correctness. As we can see from Listing 16, a modified version of SSP realises the inner loop. The selection function returns paths that we assume 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 16 and the one from Listing 10 is the following: in the modified version, we set the flag to `notyetterm` which indicates that no more suitable paths exist and the decision on infeasibility is left to the outer loop.

We inherit most of the claims and proofs from the formalisation of SSP and therefore they are conditioned on termination for the respective input state. One can prove this from Invariant ISSP-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.

Termination. For SSP, we had the sum of the balances' absolute values as a termination measure decreasing in any iteration by at least 1. This is linear in the balances and therefore exponential with respect to 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 with respect to the greatest balance and thus, linear in terms of input length.

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

CS1 initialise $b' \leftarrow b$, $f \leftarrow 0$ and $\gamma = 2^{\lfloor \log_2 B \rfloor}$ where $B = \max\{1, \frac{1}{2} \sum_{v \in \mathcal{V}} b(v)\}$;
CS2 **while** *True* **do**
CS3 | **while** *True* **do**
CS4 | | **if** $\forall v \in \mathcal{V}.\ b'(v) = 0$ **then return** f;
CS5 | | **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**
CS6 | | | take such s, t and P; augment γ along P;
CS7 | | | $b'(s) \leftarrow b'(s) - \gamma$; $b'(t) \leftarrow b'(t) + \gamma$;
CS8 | | **else break**;
CS9 | **if** $\gamma = 1$ **then return** *infeasible*;
CS10 | **else** $\gamma \leftarrow \frac{1}{2} \cdot \gamma$;

```
function (domintros) 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 := success|)
    else (case get-source-target-path f b of
           None ⇒ state (|return := notyetterm|) |
           Some (s, t, P)  ⇒
             (let γ = min (min (b s) (- b t)) (Rcap f (set P));
                  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)
               in SSP (state (|current-flow := f', balance :=  b'|))))))

definition ssp (γ::nat) ≡ SSP.SSP ℰ u (get-source-target-path γ)

function (domintros) scaling :: nat ⇒ 'a Algo-state ⇒ 'a Algo-state where
 scaling l state =
  (let state' = ssp (2^l-1) state
    in (case return state' of
         success ⇒ state' |
         infeasible ⇒ state' |
         notyetterm ⇒ (if l = 0 then state' (|return := infeasible|)
                       else scaling (l-1) state')))
```

Listing 16. Formalisation of Scaling

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 $4n$ [KV12].

Remark 4.1. For an efficient path computation, CS even runs in $\mathcal{O}(n(n^2+m)\log B)$ [EK72, KV12], where B is the greatest absolute value of a balance. This is polynomial with respect to input length including the representation of balances. However, since its running time is not polynomial only in the number of vertices and edges, it is not a strongly polynomial algorithm.

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

```
O1 initialise b' ← b; f ← 0; r(v) ← v for any v; F ← ∅; actives = E; γ ← max_{v∈V} |b'(v)|;
O2 while True do
O3     (b', f, flag) ← send-flow(b', f, F, actives, γ);
O4     if flag = success then return f;
O5     if flag = infeasible then return infeasible;
O6     if ∀ e ∈ actives. f(e) = 0
O7     then γ ← min{γ/2, max_{v∈V} |b'(v)|};
O8     else γ ← γ/2;
O9     (b', f, F, r, actives) ← maintain-forest(b', f, F, actives, γ, r);
```

5. Orlin's Algorithm

Orlin's algorithm (Algorithm 3) allows for a strongly polynomial worst-case running time of $\mathcal{O}(n \log n(m + n \log n))$ [Orl93, AMO93]. It requires infinite capacities, and, like the other two algorithms, conservative weights (absence of negative cycles).

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 (*send-flow*). After each threshold decrease, the algorithm updates a continuously growing spanning forest $\mathcal{F}$. In the return value of *send-flow*, the $flag$ indicates whether the algorithm found an optimum flow or detected infeasibility. Otherwise, the computation continues.

We fix an uncapacitated flow problem ($u = \infty$) consisting of edges $\mathcal{E}$, vertices $\mathcal{V}$, costs c and balances b. The top loop works on a program state consisting of the changing variables b', f, $\mathcal{F}$, *actives*, γ and r. Each subprocedure has access to the fixed variables and expects the changing variables as arguments. The only exception is that r is not an argument for *send-flow* because we do not need r for that subprocedure. For the sake of brevity, we still sometimes write *send-flow*(*state*) although only *maintain-forest* takes the full state. The subprocedures return the changed variables.

Here, we can augment the flow on only *active edges* and forest edges, which is the purpose of *send-flow*. 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 increases to the spanning forest. Positivity ensures termination of *send-flow*.

For Algorithms 1 and 2, the running times depended on B. On the contrary, Orlin's algorithm avoids that using the continuously growing *spanning forest* $\mathcal{F}$ of edges. The crucial observation is that this forest is used such that we only consider one vertex (henceforth, the representative) per forest connected component (henceforth, $\mathcal{F}$-*component*) as a source or as a target in searching for augmenting paths. The algorithm achieves this reduction in augmentation effort by maintaining the forest, which is possible in time polynomial in n and m.

We show partial correctness of the algorithm from invariants on program states and properties of the subprocedures. Line specifications refer to the state *after* executing the respective line. For example, we write $state^i$ or var^i to refer to the state or variable var after executing Line i. Consider the following invariants about the execution of *orlins*$(\mathcal{E}, \mathcal{V}, c, b)$:

(IO-1) γ is strictly positive, except when $b = 0$.

(IO-2) Any active edge e outside $\mathcal{F}$ has a flow $f(e)$ that is a non-negative integer multiple of γ.
(IO-3) Endpoints of a deactivated edge belong to the same $\mathcal{F}$-component.
(IO-4) Only representatives can have a non-zero balance.
(IO-5) For states in Line O3, any edge $e \in \mathcal{F}^{O3}$ has a flow above $4n\gamma$, i.e. $f^{O3}(e) > 4n\gamma^{O3}$.
(IO-6) f is optimum for the balance $b - b'$.

With varying degree of explicity, the invariants are contained in the discussion by Korte and Vygen [KV12].

The *balance potential* Φ [KV12] is important for the number of augmentations during the subprocedures and their termination. We define it for b' and γ 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:

(PO-1) Let $(b'', f', \mathit{flag}) = \mathit{send\text{-}flow}(b', f, \mathcal{F}, \mathit{actives}, \gamma)$. For any $x \in \mathcal{V}$, we have $|b''(x)| \leq (1 - \epsilon) \cdot \gamma$. If $\mathit{flag} \neq \mathit{success}$ there is an x with $b''(x) > 0$. *send-flow* sets *flag* to *success* when it reaches $b'' = 0$.
(PO-2) Let $(b'', f', \mathit{flag}) = \mathit{send\text{-}flow}(b', f, \mathcal{F}, \mathit{actives}, \gamma)$. For any $e \in \mathcal{E}$, $f'(e) - f(e)$ is an integer multiple of γ.
(PO-3) If *state* satisfies Invariants IO-3 and IO-4, then *maintain-forest*(*state*) satisfies them, as well. Similarly, *send-flow*(*state*) satisfies Invariant IO-4, if *state* satisfies Invariant IO-4.
(PO-4) Let $(b'', f', \mathcal{F}', r', \mathit{actives}') = \mathit{maintain\text{-}forest}(b', f, \mathcal{F}, \mathit{actives}, \gamma, r)$. For any edge $e \in \mathcal{F}'$, the flow reduction $f(e) - f'(e)$ is at most $n\beta$ if β is some bound for the balances, i.e. $\forall\, v.\ |b'(v)| \leq \beta$. For edges $e \notin \mathcal{F}'$, $f'(e) = f(e)$.
(PO-5) $\Phi(\mathit{maintain\text{-}forest}(\mathit{state})) \leq \Phi(\mathit{state}) + n$.
(PO-6) Provided the other invariants hold, Invariant IO-6 (optimality) is preserved by either subprocedure. (For *maintain-forest* and *send-flow*, we have to replace $4n\gamma$ in Invariant IO-5 with $8n\gamma$ and $6n\gamma$, respectively.).
(PO-7) Let $(b'', f', \mathit{flag}) = \mathit{send\text{-}flow}(\mathit{state})$ where $\mathit{state} = (b', f, \mathcal{F}, \mathit{actives}, \gamma)$. The number of path augmentations during *send-flow*(*state*) is at most $\Phi(\mathit{state})$. For all $e \in \mathcal{E}$, $f(e) - f'(e) \leq \Phi(\mathit{state}) \cdot \gamma$, if $f(e) > 6n\gamma$ for all $e \in \mathcal{F}$ in *state*.
(PO-8) Assume all invariants hold on $\mathit{state} = (b', f, \mathcal{F}, \mathit{actives}, \gamma)$ and let $(b'', f', \mathit{flag}) = \mathit{send\text{-}flow}(\mathit{state})$. If $\mathit{flag} = \mathit{success}$, f' is optimum. If $\mathit{flag} = \mathit{infeasible}$, the flow problem is indeed infeasible.

Contrary to the invariants, which were taken from the literature, the properties of the subprocedures — and more generally, the separation between main loop and subprocedures — are a product of the formalisation. We examine how the invariants and properties yield partial correctness.

Theorem 5.1 (Partial Correctness)**.** *Assume the algorithm terminates on an uncapacitated instance with conservative weights. If it finds a flow, it is a min-cost flow, otherwise the problem is infeasible.*

Proof Sketch. $b = 0$ yields immediate termination with $f^{O3} = 0$ as correct result (from PO-1).

If $b \neq 0$, we show the invariants for the last state that we give as argument to *send-flow*. All invariants hold for the initialisation in the algorithm. The pseudocode and PO-1 imply

preservation of Invariant IO-1. Only *send-flow* changes the flow along edges outside $\mathcal{F}$ in Line O3 (from PO-4) and the change is an integral multiple of γ (from PO-2) implying preservation of Invariant IO-2. The arguments for Invariants IO-3 and IO-4 are simple (from PO-3).

Preservation of Invariant IO-5 is more difficult. Assume it holds in Line O3. We know $|b'^{O3}(x)| \leq (1-\epsilon) \cdot \gamma^{O3}$ for any x (from PO-1). After modifying γ (Lines O6 - O8), the flow along forest edges is above $8n\gamma^{O8}$, $\Phi(b'^{O8}, \gamma^{O8}) \leq n$ and $\forall x. |b'^{O8}(x)| < 2 \cdot \gamma^{O8}$. Executing *maintain-forest* can cause a decrease of $2n\gamma^{O9}$ for forest edges (from PO-4) and an increase in Φ by at most n (PO-5). At this point, we have $f^{O9}(e) > 6n\gamma^{O9}$ for all $e \in \mathcal{F}^{O9}$ and $\Phi(state^{O9}) \leq 2n$. *send-flow* in Line O3 is responsible for a further flow decrease of at most $2n\gamma^{O9}$ (from PO-7). The overall decrease along forest edges was at most $4n\gamma^{O9}$, hence again $f^{O3}(e) > 4n\gamma^{O3}$ for $e \in \mathcal{F}^{O3}$ in the next iteration. This implies preservation of Invariant IO-5.

By PO-6 we obtain preservation of Invariant IO-6 as well. The invariants hold for the last $state^{O9}$ before termination (state before the last usage of *send-flow*). PO-8 gives the claim. □

This proof refines proofs in the literature [KV12] and it was reconstructed from the formalisation.

Remark 5.2. We restrict ourselves to infinite edge capacities, which is insignificant because we can reduce problems to the uncapacitated setting with a linear increase in input length, as we will see later. Our theorems here require weight-conservativity which we inherit from Algorithm 1 and Algorithm 2 as a constraint. However, we drop the restriction to integral capacities and balances.

Formalisation. We formalise Algorithm 3 (Listing 17) in a locale `orlins`. Since the properties of the subprocedures *send-flow* and *maintain-forest* are quite complex (unlike the selection of paths), and since implementations of the subprocedures are not useful on their own, we do not specify them at the beginning of the locale `orlins` to be instantiated later. Instead, we set up two locales for the subprocedures (details later) and define the locale for the top level loop in Listing 17 as their conjunction.

Remark 5.3. We present Orlin's algorithm in a top-down manner to make it easier to understand, but the formalisation was developed bottom-up as we first formalised *send-flow* and *maintain-forest*. The top-down approach with specifying subprocedures in a locale and instantiating them later is only practicable when the subprocedures are meaningful on their own and their properties are comparably simple.

As in the case of `scaling` and `SSP`, we use records to model the program state. This time, however, the variables of the program state are abstract data types (ADTs), such as maps from vertices or edges to reals to model the flow or balances. To specify these ADTs, we reuse the locales from Listings 6 and 5 multiple times as a part of the locales for Orlin's algorithm, subject to some extensions. An example of an ADT operation shown in Listing 17 is `are-all`, which we use to check all elements of the data structure that models the active edges for a certain predicate. Operations like `abstract-flow-map` turn a mapping data structure `m` into a mathematical function $f_{\texttt{m}}$. When `m` is undefined for a certain input, $f_{\texttt{m}}$ returns a reasonable default value like 0 for reals.

```
definition new-γ state =
 (let f = current-flow state; b = balance state;
      γ = current-γ state; E' = actives state
   in (if are-all (λ e. (abstract-flow-map f e) = 0) E'
       then min (γ / 2) (get-max (λ x bx. |bx|) b)
       else (γ / 2)))

function (domintros) orlins where
 orlins state =
  (if return state = success then state
   else if return state = infeasible then state
   else (let γ' = new-γ state;
             state' = maintain-forest (state (|current-γ := γ'|));
             state'' = send-flow state'
         in orlins state''))

definition invar-integral state = (∀ e ∈ to-set (actives state). [Invariant IO-2]
   ∃ n :: nat. (a-current-flow state) e = n * (current-γstate))
definition invar-isOptflow state =    [Invariant IO-6]
   is-Opt (b - a-balance state) (a-current-flow state)

theorem initial-state-orlins-dom-and-results: (*simplified*) [Theorem 5.1]
  assumes state' = orlins (send-flow initial)
    shows orlins-dom (send-flow initial)
          return state' = success ⟹ is-Opt b (a-current-flow state')
          return state' = infeasible ⟹ ∄ f. f is b flow
          return state' = notyetterm ⟹ False
```

Listing 17. Top Loop of Orlin's Algorithm in Isabelle/HOL

`a-balance state` is an abbreviation for `abstract-bal-map (balance state)`. The same holds for `a-current-flow state`.

In addition to those variables mentioned above, the state contains some auxiliary ones that ease the implementation. We will give details in Section 5.2, which deals with *maintain-forest* where these variables are relevant. We also encapsulate the data structure invariants and some conditions on the domains, for example, that the domain of the flow map `current-flow state` is $\mathcal{E}$, into an invariant `implementation-invar` that the loops preserve.

Invariants are again boolean predicates on states. Listing 17 shows the formalisation of Invariants IO-2 and IO-6. There is a number of conceptually less interesting or even trivial properties, such as $\mathcal{F} \subseteq \mathcal{E}$, or that r is a valid representative function, that is, $r(u) = r(v)$ if and only if u and v are reachable from one another in $\mathcal{F}$. In the formalisation, we encapsulate these into the auxiliary invariant `underlying-invars`.

The Isabelle/HOL function `orlins` first decreases γ, followed by `maintain-forest` and `send-flow`, contrary to the pseudocode from Algorithm 3. We enter the loop body with the decrease of γ while preserving the relative order of the operations so that the invariants refer exclusively to the beginning/end of iterations. It is then much easier to reflect Invariant IO-5, which talks about the state after finishing Line O3, in the formalisation. Because of

Algorithm 4: *send-flow*$(b', f, \mathcal{F}, \mathit{actives}, \gamma)$

```
SF1  while True do
SF2    if ∀v ∈ V. b'(v) = 0 then return (b', f, success);
SF3    else if ∃ s . b'(s) > (1 − ε) · γ then
SF4      if ∃ t . b'(t) < −ε · γ ∧ t is reachable from s then
SF5        take such s, t, and a connecting path P with original edges from actives ∪ F;
SF6        augment f along P from s to t by γ;
SF7        b'(s) ← b'(s) − γ; b'(t) ← b'(t) + γ;
SF8      else return (b', f, infeasible);
SF9    else if ∃ t . b'(t) < −(1 − ε) · γ then
SF10     if ∃ s . b'(s) > ε · γ ∧ t is reachable from s then
SF11       take such s, t, and a connecting path P with original edges from actives ∪ F;
SF12       augment f along P from s to t by γ;
SF13       b'(s) ← b'(s) − γ; b'(t) ← b'(t) + γ;
SF14     else return (b', f, infeasible);
SF15   else return (b', f, please continue);
```

this, we write valid executions of Orlin's algorithm as `orlins (send-flow initial)` in the formalisation. Listing 17 also displays the final correctness theorem. However, proving it requires work that is discussed in the next subsections.

We now state the subprocedures and show that they satisfy the aforementioned properties, including both formal and informal proofs.

5.1. **Augmenting the Flow.** We now argue why *send-flow* (Algorithm 4) satisfies the properties stated in the previous section. PO-1, PO-2, PO-3, PO-6, PO-7 and PO-8 assert something about *send-flow*. We can infer PO-1 from the subprocedure's structure. The only possible amount to augment is γ which yields PO-2. PO-3 (Preservation of Invariant IO-4) also follows from the structure.

Proof Sketch for PO-6. We assume the invariants for hold for $\mathit{state} = (b', f, \mathcal{F}, \mathit{actives}, \gamma)$. Then f^{SF7} and f^{SF13} are minimum-cost flows for the balances $b - b'^{SF7}$ and $b - b'^{SF13}$, respectively, because of Theorem 3.7 and the fact that f^{SF2} is a minimum-cost flow for the balance $b - b'^{SF2}$ (Invariant IO-6). The restriction to active and forest edges (Lines SF5 and SF11) is unproblematic: simulate deactivated edges with forest paths which are of minimum-costs (see Lemma 5.9 in the next section where we discuss the forest). □

Lemma 5.4. *Assume $f(e) > 6n\gamma$ for all $e \in \mathcal{F}$. Any iteration of send-flow decreases Φ at least by 1, i.e. $\Phi(b'^{SF6}, \gamma) - \Phi(b'^{SF7}, \gamma) \geq 1$ and $\Phi(b'^{SF12}, \gamma) - \Phi(b'^{SF13}, \gamma) \geq 1$.*

Proof Sketch for PO-7. Assume $f(e) > 6n\gamma$ for all $e \in \mathcal{F}$. We have the decrease in Φ according to Lemma 5.4. Because of Lines SF3 and SF9, Φ cannot become negative. Thus, *send-flow* (state) performs at most $\Phi(\mathit{state})$ iterations and the flow decrease for an edge is at most $\Phi(\mathit{state}) \cdot \gamma$. Note that during the whole computation, $f(e) > 0$ for all $e \in \mathcal{F}$ meaning that we can always select a path using active and forest edges, as required in Lines SF5 and SF11. The proof of a strict decrease in Φ only works for $\epsilon > 0$. □

```
function (domintros) send-flow  where
 send-flow state =
  (let f = current-flow state;  b = balance state; γ = current-γ state
   in (if test-all-vertices-zero-balance state then state (| return :=success|)
       else (case get-source state of Some s ⇒
              (case get-source-target-path-a state s of Some (t, P) ⇒
                     (let f' = augment-edges-impl f γ P;
                          b' = move b γ s t;
                          state' = state (| current-flow := f', balance := b'|)
                       in send-flow state')
                   | None ⇒ state (| return := infeasible|))
        | None ⇒
            (case get-target state of Some t ⇒
              (case get-source-target-path-b state t of Some (s, P) ⇒
                     (let f' = augment-edges-impl f γ P;
                          b' = move b γ s t;
                          state' = state (| current-flow := f', balance := b'|)
                       in send-flow state')
                   | None ⇒ state (| return := infeasible|))
             | None ⇒ state (| return := notyetterm|)))))
 ...
definition get-source-target-path-a-cond state s t P b γ f = (
 get-source-target-path-a state s = Some (t, P) ∧ s ∈ 𝒱 ∧ underlying-invars state
 ∧ (∀ e ∈ ℱ state . abstract-flow-map f e > 0) ∧ b = balance state
 ∧ γ = current-γ state ∧ Some s = get-source state ∧ f = current-flow state
 ∧ invar-gamma state ∧ implementation-invar state ∧ invar-isOptflow state)
```

LISTING 18. Formalisation of *send-flow*

Proof Sketch for PO-8. If the algorithm found a flow, then $\forall v \in \mathcal{V}\,.\, b'^{SF2}(v) = 0$ (Line SF2). 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 SF3, SF4, SF9 and SF10. 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 3.8. The argument only works for $\epsilon \leq \frac{1}{n}$. □

By this, we have shown that *send-flow* satisfies all asserted properties.

Remark 5.5. We also saw the restriction $0 < \epsilon \leq \frac{1}{n}$. 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 might take exponentially many iterations, and its running time will depend on B. Interested readers should consult Section 4.2 of Orlin's original paper [Orl93].

Formalisation. We formalise the loop in a locale `send-flow-spec` that assumes selection functions, such as `get-source-target-path-a` to find the path required in Line SF5 of *send-flow*. Listing 18 shows the Isabelle/HOL code for the loop. We set up customised simplification and induction rules analogously to Listing 13. We also introduce definitions to

```
locale send-flow = send-flow-spec + algo +
assumes get-source-target-path-a-axioms:
  ⋀state ... f. get-source-target-path-a-cond state s t P b γ f ⟹
           is-min-path f s t P
  ⋀state ... f. get-source-target-path-a-cond state s t P b γ f ⟹
           oedge ` set P ⊆ to-set (actives state)∪ ℱ state
  ⋀ state ... f. get-source-target-path-a-cond state s t P b γ f ⟹
           t ∈ 𝒱 ∧ abstract-bal-map b t < - ϵ * γ
and get-source-target-path-b-axioms: (*analogous*)
and get-source-axioms:
  ⋀ s state b γ. get-source-cond s state b γ ⟹
     s ∈ 𝒱 ∧ abstract-bal-map b s > (1 - ϵ) * γ
  ⋀ state b γ. vertex-selection-cond state b γ ⟹
   ¬(∃ s ∈ 𝒱. abstract-bal-map b s > (1 - ϵ) * γ) ⟷ ((get-source state) = None)
and get-target-axioms: (*analogous*)
and impl-a-None:
  ⋀ state s b γ f.  impl-a-None-cond state s b γ f ⟹
 ¬ (∃ t ∈ 𝒱. abstract-bal-map b t < - ϵ * γ ∧ resreach (abstract-flow-map f) s t)
     ⟷ get-source-target-path-a state s = None
and impl-b-None: (*analogous*)
and test-all-vertices-zero-balance:
  ⋀ state b. ⟦b = balance state; implementation-invar state⟧ ⟹
     test-all-vertices-zero-balance state ⟷ (∀ v ∈ 𝒱. abstract-bal-map b v = 0)
```

LISTING 19. Proof Locale for *send-flow*

encapsulate the preconditions ensuring that the selection functions work properly. As example consider `get-source-target-path-a-cond` for `get-source-target-path-a`: among others, this condition requires edges in $\mathcal{F}$ to have positive flow. This is necessary to ensure the existence of an augmenting path that uses active and forest edges only, that is, no deactivated ones, as required in Line SF5 of the algorithm. We use those conditions to specify the behaviour of the selection functions in Listing 19. Since the preconditions like `get-source-target-path-a-cond` are of considerable complexity, it is necessary to encapsulate them in definitions, without which the locale `send-flow` would barely be readable. Whenever we use a function, it application precondition is provable allowing us to retrieve information about the result by referring to the specification from Listing 19.

In the locale `send-flow`, we formalise the proofs that we informally discussed in this section. Listing 20 shows examples of formalised results. `send-flow-call1-cond-Phi-decr` formalises that taking the first recursive step in `send-flow` (Lines SF5-SF7 in Algorithm 4) results in Φ being decreased by at least 1. `send-flow-Phi` is the formal counterpart to PO-7 where `invar-above-6Ngamma state` asserts $f(e) > 6n\gamma$ for all $e \in \mathcal{F}$ in `state`. Preservation of this invariant implies positive flow along forest edges (necessary for Lines SF5 and SF11 and precondition for `send-flow-call1-cond-Phi-decr`). `orlins-entry-after-send-flow` proves a part of PO-1 where `orlins-entry state` asserts that $|b(v)| \leq (1-\epsilon)\gamma$ in `state`. `send-flow-invar-isOpt-pres` contains the formal proof for PO-6. `send-flow-completeness` belongs to the formal counterpart of PO-8, where `b` refers to the balances of the actual flow problem.

```
lemma send-flow-call1-cond-Phi-decr: [For Lemma 5.4]
 assumes send-flow-call1-cond state and (*invariants*)
   shows Φ (send-flow-call1-upd state) ≤ Φ state - 1

theorem send-flow-Phi: [For Property PO-7]
  assumes invar-above-6Ngamma state and (*other invariants*)
    shows a-current-flow  state' e ≥
          a-current-flow state e - (Φ state - Φ state') * current-γ state'

lemma orlins-entry-after-send-flow: [For Property PO-1]
 assumes (*invariants*) and return (send-flow state) = notyetterm
 shows orlins-entry (send-flow state)

theorem send-flow-invar-isOpt-pres: [For Property PO-6]
 assumes (*other invariants*) and invar-isOptflow state
   shows invar-isOptflow (send-flow state)

theorem send-flow-completeness: [For Property PO-8]
  assumes (*invariants*) and return (send-flow state) = infeasible
    shows ∄ f. (f is b flow)
```

Listing 20. Some Lemmas on the Correctness of `send-flow`

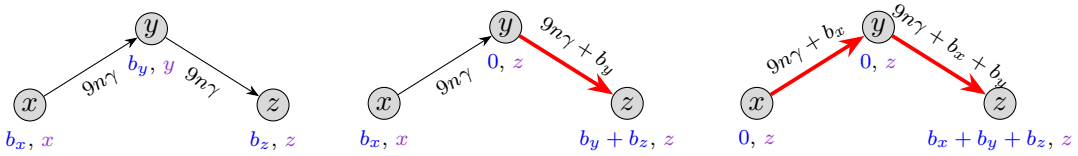


Figure 4. Merging Forest Components: remaining balances b', forest edges, flow values and representatives are blue, red, black and purple, respectively. Assume the balances are non-negative. Initially there are three components: $\{x\}$, $\{y\}$, and $\{z\}$. Then we merge $\{y\}$ and $\{z\}$ into one component, where z is the representative, inheriting the balance of both y and z. Then we add x to the component, and z is the representative of the final component. To make sure that the new balances are consistent, we move flow along the red edges to/from the representative vertex. The reduction in flow in each edge is bounded.

5.2. **Maintaining the Forest.** We now discuss the last major part of the algorithm, namely, maintaining the forest. Algorithm 5 shows the definition of *maintain-forest*. We add active edges with flow above $8n\gamma$ to $\mathcal{F}$, which inevitably changes the connected components of the forest – we create a new component for every added edge by merging the components to which the two end points of the edge belong. Since non-zero balance is only allowed for representatives, we must re-concentrate balances at one of the two representatives after merging two components.[2] Moreover, we deactivate all edges between them and update the representatives. For a forest $\mathcal{F}$, $\mathfrak{F}$ is the corresponding residual network consisting only of

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

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

MF1 **while** $\exists\, e = (x, y).\ e \in actives \wedge e \notin \mathcal{F} \wedge f(e) > 8n\gamma$ **do**
MF2 $\quad \mathcal{F} \leftarrow \mathcal{F} \cup \{e\}$;
MF3 $\quad$ **if** $|\mathcal{F}$-component of $y| < |\mathcal{F}$-component of $x|$ **then** exchange x and y;
MF4 $\quad$ let $x' = r(x)$ and $y' = r(y)$ the respective representatives;
MF5 $\quad$ take residual path $Q \subseteq \mathfrak{F}$ connecting x' and y';
MF6 $\quad$ **if** $b'(x') > 0$ **then** augment f along Q by $b'(x)$ from x' to y';
MF7 $\quad$ **else** augment f along $\overleftarrow{Q}$ by $-b'(x)$ from y' to x';
MF8 $\quad b'(y') \leftarrow b'(y') + b'(x')$; $b'(x') = 0$;
MF9 $\quad$ **foreach** v *reachable from* y' *in* $\mathfrak{F}$ **do** $r(v) \leftarrow y'$;
MF10 $\quad$ **foreach** $d = (u, v).\ d \in actives \wedge \{r(u),\ r(v)\} = \{y'\}$ **do** $actives \leftarrow actives \setminus \{d\}$;
MF11 **return** $(b', f, \mathcal{F}, actives, r)$;

forest edges. We perform the re-concentration by augmenting along paths in $\mathfrak{F}$. Figure 4 displays an example of how balances, flow and forest change.

Remark 5.6. The high-level reason for merging components and concentrating the balances at unique representatives is the controlled reduction of the number of non-zero vertices, resulting in a simpler problem. Only non-zero vertices need further treatment (by augmentations) to receive or emit more flow. The contraction of vertices is the foundation of several strongly polynomial time algorithms for the minimum-cost flown problem. The version by Korte and Vygen [KV12] that we follow simulates the contractions by maintaining the forest $\mathcal{F}$ and component representatives.

Only PO-3 - PO-6 make assertions about *maintain-forest* and we argue why they are indeed satisfied. PO-3 (preservation of Invariants IO-3 and IO-4) is easy to see since it is precondition for deactivation to have the same representatives (Line MF10).

Proof Sketch for PO-4. Invariants IMF-1 and IMF-2 bound the forest edges' flow decrease:

(IMF-1) For any x, the product of its $\mathcal{F}^{MF1}$-component's cardinality and β is a bound for $|b'^{MF1}(x)|$.
(IMF-2) For any $e \in \mathcal{F}^{MF1}$, we have $f^{MF1}(e) > \alpha - \beta \cdot |X|$ where X is the $\mathcal{F}^{MF1}$-component of e.

where α and β are initial bounds for the flow and the balances, respectively. The conjunction of IMF-1 and IMF-2 is preserved by *maintain-forest*. □

Remark 5.7. It is important to always concentrate the balances at the larger component's representative as it is done in the algorithm. Otherwise, the preservation proof for invariant IMF-2, which is omitted for the sake of brevity, would not work.

Any iteration merges two $\mathcal{F}$-components making $n - 1$ an upper bound for the number of iterations. PO-5 asserts $\Phi(\textit{maintain-forest}(state)) \leq \Phi(state) + n$. This holds because the potential cannot increase by more than 1 per iteration, as one can see in the following lemma.

Lemma 5.8. *The increase of* Φ *during a single iteration of maintain-forest is at most* 1.

Proof. Only Line MF8 is relevant for the potential since it is the only place where b' is changed. We look at $state^{MF7}$ (state before changing b') and $state^{MF8}$ (state after changing b'). It follows:

$$\begin{aligned}
\Phi(state^{MF8}) &= \sum_{v\in\mathcal{V}} \left\lceil \frac{|b'^{MF8}(v)|}{\gamma} - (1-\epsilon)\right\rceil \\
&= \sum_{v\in\mathcal{V}\setminus\{x',y'\}} \left\lceil \frac{|b'^{MF7}(v)|}{\gamma} - (1-\epsilon)\right\rceil + \left\lceil \frac{0}{\gamma} - (1-\epsilon)\right\rceil \\
&+ \left\lceil \frac{|b'^{MF7}(y') + b'^{MF7}(x')|}{\gamma} - (1-\epsilon)\right\rceil \\
&= \sum_{v\in\mathcal{V}\setminus\{x',y'\}} \left\lceil \frac{|b'^{MF7}(v)|}{\gamma} - (1-\epsilon)\right\rceil + \left\lceil \frac{|b'^{MF7}(y')| + |b'^{MF7}(x')|}{\gamma} - (1-\epsilon)\right\rceil \\
&\le \sum_{v\in\mathcal{V}\setminus\{x',y'\}} \left\lceil \frac{|b'^{MF7}(v)|}{\gamma} - (1-\epsilon)\right\rceil + \left\lceil \frac{|b'^{MF7}(x')|}{\gamma}\right\rceil + \left\lceil \frac{|b'^{MF7}(y')|}{\gamma} - (1-\epsilon)\right\rceil \\
&\le \sum_{v\in\mathcal{V}\setminus\{x',y'\}} \left\lceil \frac{|b'^{MF7}(v)|}{\gamma} - (1-\epsilon)\right\rceil + \left\lceil \frac{|b'^{MF7}(x')|}{\gamma} - (1-\epsilon)\right\rceil \\
&+ 1 + \left\lceil \frac{|b'^{MF7}(y')|}{\gamma} - (1-\epsilon)\right\rceil \\
&= \Phi(state^{MF7}) + 1
\end{aligned}$$

□

Lemma 5.9. *If all invariants hold in Line MF2, both Q and $\overleftarrow{Q}$ are minimum-cost paths.*

Proof. 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 3.2 and Invariant IO-6. The case for $\overleftarrow{Q}$ is analogous. □

Proof of PO-6. To apply Theorem 3.7 for Lines MF6 and MF7, we need optimality of the forest paths Q and $\overleftarrow{Q}$ that are used for augmentations. This follows from Lemma 5.9. □

This concludes our proofs that *maintain-forest* satisfies properties PO-3 - PO-6.

Formalisation. We formalise the loop as a function in a locale analogously to `send-flow`. This time, there is only one recursive case. Also, we only assume one path selection function `get-path`. We encapsulate the preconditions for specifying its behaviour into a definition `maintain-forest-get-path-cond`, similar to the definition from Listing 18. An important part of those preconditions are the auxiliary and implementation invariants.

The forest is specified as an ADT for an adjacency map [Abd25]. We add an undirected edge to the forest by adding two edges in opposite directions to the representing adjacency map, making the latter symmetric. For connected components of $\mathcal{F}$, we reuse Abdulaziz and Mehlhorn's [AM26] work. We assume the function `get-path` takes $\mathcal{F}$ as an adjacency map and returns a path as a sequence of vertices. We must transform these into paths over residual edges to perform an augmentation. We have a program variable `conv-to-redge`, again specified as an ADT for maps, that takes a pair of vertices (u, v) and returns a positive-capacity residual edge in $\mathfrak{F}$ pointing from u to v. `conv-to-redge state` is always

updated in a way such that residual edges returned by `conv-to-redge state (u, v)` and `conv-to-redge state (v, u)` originate from the same graph edge. Due to this, Q and $\overleftarrow{Q}$ have opposite costs as needed to show Lemma 5.9.

We deactivate edges in Line MF10 of *maintain-forest* if they are active and if their endpoints' representatives are within the same newly created forest component. The other loop *send-flow* requires paths that use active and forest edges only. To facilitate the computation of those paths later, we have the program variable `not-blocked` which is a map (specified as ADT, of course) from graph edges to booleans. `not-blocked state e` being true means that the edge `e` is not deactivated — in other words, it is either a forest or an active edge. In the formalisation, we assume the function `not-blocked-upd-all` in the respective ADT specification. We use it to iterate over the whole domain of `not-blocked state` (which is $\mathcal{E}$): if for an edge $e \in \mathcal{E}$ with endpoints x_1 and x_2, $\{r(x_1), r(x_2)\} = \{x', y'\}$ holds, we set `not-blocked state e` to `False`. We also remove these edges from `actives state` with a function `filter`, which comes from the ADT for `actives`. Like the list function of the same name, it iterates over all elements and keeps only those that satisfy a predicate.

To make the comparison in Line MF3 executable in constant time, we maintain cardinalities of each vertex' forest component. Since we have to update these in the same way as the representatives, we combine both into a program variable `rep-comp-card` which maps vertices to a pair of their representatives and $\mathcal{F}$-component cardinalities. Updates to this variable, which are required by Line MF9 of *maintain-forest*, are realised as follows: We only have to change representatives and component cardinalities for vertices whose $\mathcal{F}$-component grew by adding e. For a vertex v, this is the case if and only if its old representative was x' or y'. We use an iteration function `rep-comp-upd-all` (specified by the respective ADT, similar to `not-blocked-upd-all`) to iterate over the domain (set of vertices $\mathcal{V}$), to perform the required updates. The new representative is y' and the cardinality value is $|C_x| + |C_y|$ where C_x and C_y are the $\mathcal{F}$-components of x and y respectively.

Listing 21 contains the formal definitions for Invariants IMF-1 and IMF-2, the preservation lemma for invariant IMF-2, as well as, two lemmas relevant to PO-4. Since the program modifies all variables, the preservation proofs for the invariants, even for the auxiliary invariant `underlying-invars`, which talks about simple relationships among the program variables, are more verbose than in case of *send-flow*.

5.3. **Termination.** For the inner loops, we can build a termination measure out of the number of forest components and the value of Φ. Now, call a vertex v *important* iff $|b'(v)| > (1 - \epsilon) \cdot \gamma$ [KV12], that is, 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 $k = \lceil \log n \rceil + 3$.

Lemma 5.10 ([PT90, KV12])**.** *If there is an important vertex in an iteration, $\ell + 1$ additional iterations of the outer loop enforce its component being increased (or the algorithm terminates).*

Lemma 5.11 ([KV12])**.** *If we wait for $k + 1$ iterations, a vertex has become important (or the algorithm terminates).*

Theorem 5.12 (Termination of Orlin's Algorithm)**.** *After at most $(n - 1) \cdot (k + \ell + 2)$ iterations of the outer loop, Orlin's algorithm will terminate.*

```
definition invar-F1 thr state = (∀ v ∈ 𝒱. [Invariant IMF-1]
 |a-balance state v| ≤ thr * card (component (to-graph (𝔉 state)) v))
definition invar-F2 thr1 thr2 state = (∀ e ∈ ℱ state. [Invariant IMF-2]
  (a-current-flow state) e >
        thr1 - thr2 * card (component (to-graph (𝔉 state)) (fst e)))

lemma invar-F2-pres: [Preservation Invariant IMF-2, For Property PO-4]
  assumes underlying-invars state and 0 ≤ thr2 and invar-F1 thr2 state
      and invar-F2 thr1 thr2 state and thr2 ≤ 2 * current-γ state
      and thr1 ≤ 8 * real N * current-γ state and  implementation-invar state
    shows invar-F2 thr1 thr2 (maintain-forest state)

lemma mono-one-step-phi: [Lemma 5.8]
  assumes maintain-forest-call-cond state and underlying-invars state
      and invar-gamma state and implementation-invar state
    shows Φ (maintain-forest-upd state) ≤ Φ state + 1

lemma Phi-increase: [Consequence of Lemma 5.8]
  assumes underlying-invars state and invar-gamma state
      and implementation-invar state
    shows Φ (maintain-forest state) ≤
          Φ state + (card (comps 𝒱 (to-graph (𝔉 state)))) -
                       (card (comps 𝒱 (to-graph (𝔉 (maintain-forest state)))))
```

Listing 21. Correctness of `maintain-forest` (simplified)

Explanations: `component E x` is the set of vertices in the connected component around `x` with respect to the set of edges `E`. `comps V E` is the set of all `component E v` where `v ∈ V`.

Proof Sketch. *maintain-forest* and *send-flow* terminate due to the decrease in the number of forest components or Φ, respectively. Obviously, there can be at most $n-1$ component merges, which gives the statement together with Lemmas 5.10 and 5.11. Note that due to Invariant IO-4, there can be at most one important vertex per component. □

Formalisation. The usual way to prove termination in Isabelle/HOL is to find a termination measure which decreases in every loop iteration. This is not possible for the function `orlins` since we only know that after at most $k+\ell+2$ iterations, the number of forest components must have decreased. We introduce `orlins-one-step` and `orlins-one-step-check` to express a single iteration of the loop body in *orlins*. We define `orlins-one-step-check state` to execute `orlins-one-step state` if `flag state = notyetterm`. Otherwise, that is, if `flag state` is `success` or `infeasible`, the state remains unchanged. We iterate `orlins-one-step-check` (written as `compow i orlins-one-step-check` or `orlins-one-step-check ^^ i`) to express the state after `i` iterations. [3]

[3] If the algorithm terminates after i iterations, we say for all $j \geq i$ that the state after j iterations is the final state.

```
lemma succ-fail-term-same-dom:
 assumes compow (Suc k) orlins-one-step-check state = state'
     and return state' = success ∨ return state' = infeasible
   shows orlins-dom state and orlins state = state'

lemma important-or-merge-or-termination: [Lemma 5.11]
 assumes (*invariants*) and k = ⌈log 2 N⌉ + 3
   shows ∃ kk ≤ k + 1.
 return ((compow kk orlins-one-step-check) state) ≠ notyetterm ∨
 (∃ v. important (compow kk (orlins-one-step-check) state) v ∧
                 return ((compow kk orlins-one-step-check) state) = notyetterm) ∨
   (card (comps 𝒱 (to-graph (ℱ (compow kk (orlins-one-step-check) state))))
         < card (comps 𝒱 (to-graph (ℱ state))) ∧
                 return ((compow kk orlins-one-step-check) state) = notyetterm)

lemma if-important-then-comp-increase-or-termination: [Lemma 5.10]
 assumes important state z and ℓ = ⌈log 2 (4*M*N + (1 - ϵ)) - log 2 ϵ⌉ + 1
     and (*invariants*)
   shows ∃ k ≤ ℓ + 1. return ((compow k orlins-one-step-check) state)≠ notyetterm
          ∨ component (to-graph (ℱ (compow k (orlins-one-step-check) state))) z
              ⊃ component (to-graph (ℱ state)) z

theorem finally-termination: [For Lemma 5.12]
 assumes ℓ = ⌈log 2 (4*M*N + (1 - ϵ)) - log 2 ϵ⌉ + 1 and k = ⌈log 2 N⌉ + 3
     and (*invariants*) and I = card (comps 𝒱 (ℱ state))
   shows return ((orlins-one-step-check ^^ (I * (ℓ + k + 2))) state) ≠ notyetterm
```

Listing 22. Termination of Orlin's Algorithm (simplified)

`orlins` will terminate on the respective input state if the flag is `success` or `infeasible` after a strictly positive number of iterations (Lemma `succ-fail-term-same-dom` in Listing 22). We prove `finally-termination`, which shows that the flag is finally `success` or `infeasible`, by strong induction over the number of forest components. In the induction step, we apply the lemma `important-or-merge-or-termination`, which corresponds to Lemma 5.11, and obtain `k`, followed by a case analysis: (1) If the flag indicates termination in the next `k` steps, we are done. (2) If the number of components decreases, we can immediately apply the induction hypothesis. (3) Otherwise, there is an important vertex `z`, to which we apply `if-important-then-comp-increase-or-termination`, which corresponds to Lemma 5.10, to obtain ℓ. (3.1) If the flag indicates termination in the next ℓ steps, we are done. (3.2) Otherwise, `z`'s component is increased and we can apply the induction hypothesis. `succ-fail-term-same-dom` and `finally-termination` yield termination of `orlins` on the input state `initial`.

To prove invariants for the function `orlins`, we deviate from our standard approach by proving the invariants for `orlins-one-step-check ^^i state` instead. For the formal termination proof, we need that the invariants are satisfied after an arbitrary number of steps. In combination with `succ-fail-term-same-dom`, we obtain the invariants for the final state for free which is why we can deviate from the standard approach of an induction over the actual function.

The formalised results presented up to now yield total correctness from Listing 17.

5.4. **Running Time.** The last thing we consider for the theory behind the correctness of Orlin's algorithm is its worst-case running time analysis. The following argument is an informal reconstruction from our formalisation that follows and substantially refines Korte's and Vygen's argument [KV12].

We perform our analysis assuming elementary bounds for elementary computations performed by the algorithm. For instance, t_{FB} is an upper bound for the time that the algorithm consumes when executing the loop body in *maintain-forest* and t_{FC} is the time for checking the condition (analogously t_{SC} and t_{SB} for *send-flow*, and t_{OC} and t_{OB} for *orlins*). $\#comps$ means the number of $\mathcal{F}$-components. An i in the subscript means the ith iteration of the outer loop.

First, the following should be clear about the number of loop iterations in the subprocedures during iteration i:

- $\#itsF_i = \#comps_{i-1} - \#comps_i$. $\#comps_j$ is the number of components *after* iteration j.
- From Lemma 5.4 (proof of PO-7 for *send-flow*) we know $\#itsS_i \leq \Phi(state^{O2})_i$ where $state^{O2}$ is the state at the end of Line O2, that is, before calling *send-flow*. For $i \geq 2$, $\Phi(state^{O2})_i = \Phi(state^{O9})_{i-1} \leq \Phi(state^{O8})_{i-1} + \#comps_{i-2} - \#comps_{i-1}$ due to Lemma 5.8. For $i = 1$, let $\Phi(state^{O9})_{i-1}$ depend on the initialisation.

We obtain an upper bound for the running time of ν iterations:

$$\sum_{i=1}^{\nu}\Big[t_{OC} + t_{OB} + (t_{SC} + t_{SB}) \cdot \#itsS_i + t_{SC} + (t_{FC} + t_{FB}) \cdot \#itsF_i + t_{FC}\Big] \tag{5.1}$$

$$\leq\ \nu \cdot (t_{OC} + t_{OB} + t_{SC} + t_{FC}) + (t_{SC} + t_{SB}) \cdot \sum_{i=1}^{\nu} \Phi(state^{O8})_{i-1} +$$

$$(t_{FC} + t_{FB}) \cdot \sum_{i=1}^{\nu}(\#comps_{i-1} - \#comps_i)$$

$$+\ (t_{SC} + t_{SB}) \cdot \sum_{i=2}^{\nu}(\#comps_{i-2} - \#comps_{i-1})$$

The terms involving component numbers are telescope sums amounting to at most $n - 1$. The sum over the potentials remains to be bounded. Now, we call the pairs of iteration number i and a vertex that is important after Line O8 an *occurrence of an important vertex*. For $i = 0$, we define it as the number of important vertices after the initialisation. The sum over the potentials is the number of all occurrences, since a vertex' contribution to $\Phi(state^{O8})$ is either 1 (important) or 0 (not important) after modifying γ. We define $comps_i$ as the set of all forest components during the ith iteration which have an important vertex after Line O8 or after the initialisation, respectively. Note that $|comps_i| \neq \#comps_i$ in general. Since there is at most one important vertex per $\mathcal{F}$-component (Invariant IO-4), $\Phi(state^{O8})_i \leq |comps_i|$. We will look at the components instead of occurrences.

For a sequence $S_1, S_2, \ldots S_n \subseteq \mathcal{S}$ of sets and $x \in \mathcal{S}$ we call the difference between the first and last indices $1 \leq i \leq j \leq n$ where $x \in S_i$ and $x \in S_j$ the *lifetime* of x. We can bound the sum of the cardinalities $|S_i|$ by $|\mathcal{S}| \cdot \delta$ where δ is an upper bound for all lifetimes. Now we set $\mathcal{S} = \bigcup comps_i$. According to Lemma 5.10, the lifetime of one of those components is $\ell + 1$. It therefore remains to look at the overall number of $\mathcal{F}$-components that can occur. In combinatorics, a family of sets $\mathcal{X}$ is called *laminar* over a non-empty universe U iff

$\emptyset \subset X \subseteq U$ for any $X \in \mathcal{X}$, and for any $X \in \mathcal{X}$ and $Y \in \mathcal{X}$, we have that either $X \cap Y = \emptyset$, $X \subseteq Y$, or $Y \subseteq X$ hold. One can show that $\mathcal{S}$ is laminar over $\mathcal{V}$:

Lemma 5.13. *$\mathcal{S} = \bigcup_{i=1}^{\nu} comps_i$ is a laminar family.*

Proof. Assume $X, Y \in \mathcal{S}$ with $X \neq Y$. Moreover, suppose $X \cap Y \neq \emptyset$. We know these are forest components during some iterations i and j respectively. Wlog. assume $i \leq j$. Until iteration j, X might have been merged with other components. By X' we denote the (possibly new) component X is part of in iteration j. Of course, we have $X \subseteq X'$. This makes $X' \cap Y \neq \emptyset$ which means $X' = Y$ because they are from the same iteration. Hence, $X \subseteq Y$. □

The following lemma is a well-known result central to the theory of laminar families:

Lemma 5.14. *For a family $\mathcal{X}$ laminar over U, we have $|\mathcal{X}| \leq 2 \cdot |U| - 1$.*

Proof. We conduct an induction on $|U|$. The case $n = 1$ is trivial as $\mathcal{X}$ only contains a singleton set. Now, assume $|U| \geq 2$ and fix some $x \in U$. We define $\mathcal{Y} = \{X \setminus \{x\}.\ X \in \mathcal{X}\} \setminus \{\emptyset\}$. For any $(X \setminus \{x\}) \in \mathcal{Y}$, we call X the (not necessarily unique) *parent set*. $\mathcal{Y}$ is still a laminar family: assume neither $X \cap Y = \emptyset$, nor $X \subseteq Y$, nor $Y \subseteq X$ and the same hold for their respective parent sets which contradicts laminarity of $\mathcal{X}$. When the induction hypothesis is applied to $\mathcal{Y}$, we obtain $|\mathcal{Y}| \leq 2 \cdot (|U| - 1) - 1$. Now, we show $|\mathcal{X}| \leq |\mathcal{Y}| + 2$ which would already give the claim. Let us look how the sets in $\mathcal{X}$ and $\mathcal{Y}$ correspond to one another:

- $\{x\}$ may be in $\mathcal{X}$ but (by definition) not in $\mathcal{Y}$
- For any set $X \in \mathcal{Y}$ there is a parent set in $\mathcal{X}$. However, there could be two parent sets, namely, X and $X \cup \{x\}$. We claim that this can not happen more than once. Aiming for a contradiction, assume X, $Y \in \mathcal{Y}$ where $X \neq Y$ and X, Y, $X \uplus \{x\}$, $Y \uplus \{x\} \in \mathcal{X}$. Then we have $(X \cup \{x\}) \cap (Y \cup \{x\}) \supseteq \{x\}$; thus $X \cup \{x\} \subseteq Y \cup \{x\}$ or $Y \cup \{x\} \subseteq X \cup \{x\}$ as well, due to laminarity of $\mathcal{X}$. Since x is neither in X nor Y by definition, we conclude $X \subseteq Y$ or $Y \subseteq X$. Wlog. $X \subseteq Y$. By $X \neq Y$ we have $X \subset Y$. We infer $Y \cap (X \cup \{x\}) = X \neq \emptyset$, and $Y \not\subseteq X \cup \{x\}$ (as $x \notin Y$ and $X \subset Y$), and $X \cup \{x\} \not\subseteq Y$ (as $x \notin Y$) which finally contradicts the laminarity of $\mathcal{X}$. □

By Inequality 5.1, Lemmas 5.13 and 5.14, and the remarks about important vertices, we can bound the overall running time by the term

$$\begin{aligned} &(n \cdot (\ell + k + 2) - 1) \cdot (t_{OC} + t_{OB} + t_{SC} + t_{FC}) + \\ &(n - 1) \cdot (t_{FC} + t_{FB} + t_{SC} + t_{SB}) \ + \\ &(2n - 1) \cdot (\ell + 1) \cdot (t_{SC} + t_{SB}) + t_{SC} + t_{OC} \end{aligned} \tag{5.2}$$

provided that Orlin's algorithm terminates in the $(\nu + 1)$th iteration. To the last iteration we have to account a time of $(t_{SC} + t_{SB}) \cdot (\#comps_\nu - \#comps_{\nu-1}) + t_{SC} + t_{OC}$.

Elementary times depend on reachability and path selection for which well-known strongly polynomial algorithms exist. In this work, however, we only formalise the running time argument up to Term 5.2 without details on the selection procedures.

Remark 5.15. $\epsilon = \frac{1}{n}$ yields a strongly polynomial bound. Dijkstra's algorithm results in the best running time of $\mathcal{O}(n \log n(m + n \log n))$, but it requires preprocessing to keep edge weights positive. As long as the selection functions are strongly polynomial, the whole algorithm is strongly polynomial, as well.

```
function (domintros) orlins-time::
    nat ⇒ (...) Algo-state ⇒ nat × (...) Algo-state  where
 (orlins-time tt_OC state) =
  (if (return state = success) then (tt_OC, state)
   else if (return state = infeasible) then (tt_OC, state)
   else (let γ' = new-γ state;
             state'time = maintain-forest-time (state ⦇current-γ := γ'⦈);
             state''time = send-flow-time (prod.snd state'time)
         in ((t_OC + t_OB + prod.fst state'time + prod.fst state''time)
                      +++ (orlins-time tt_OC (prod.snd state''time)))))

definition laminar U 𝒳 = (
    (∀ X Y. X ∈ 𝒳 ⟶ Y ∈ 𝒳 ⟶ (X ⊆ Y ∨ Y ⊆ X ∨ X ∩ Y = {}))
  ∧ (∀ X ∈ 𝒳. X ≠ {} ∧ X ⊆ U) ∧ U ≠ {})

lemma laminar-family-number-of-sets: [Lemma 5.14]
 assumes n = card U and finite U and laminar U 𝒳
   shows card 𝒳 ≤ 2 * n - 1

theorem running-time-initial: [Term 5.2]
 assumes final = orlins-time t_OC (send-flow initial)
   shows prod.fst final + prod.fst (send-flow-time initial) ≤
         (N - 1) * (t_Fuf + t_FC + t_FB + t_sC + t_sB + t_suf)
     +   (N * (ℓ + k + 2) - 1)* (t_sF + t_sC + t_suf + t_Fuf + t_FC + t_OC + t_OB)
     +   ((ℓ + 1) * (2 * N - 1)) *(t_sC + t_sB + t_suf)
     +   (t_sF + t_sC + t_suf) + t_OC
    and prod.snd final = orlins (send-flow initial)
```

Listing 23. Running Time Formalisation (simplified)

Formalisation. We model algorithm running times as functions returning natural numbers, using an extension of Nipkow et al.'s approach [Nip25a]. In this approach, for every function $f : \alpha \to \beta$, we devise a functional program $f_{\text{Time}} : \alpha \to \mathbb{N}$ 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, we add the running times of the called functions 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.

Listing 23 shows an example of such a running time model. It takes a program state and returns a pair of a program state and a natural number, which models the running time of Orlin's algorithm. `orlins-time` uses `maintain-forest-time` and `send-flow-time` modelling the running times of the two lower-level loops. Without explicitly specifying them, we assume times like $\mathtt{t}_{SB}$ and $\mathtt{t}_{FB}$ in the locale containing the definition of `orlins-time`. We incorporate the running times of the path selection and augmentation functions into $\mathtt{t}_{SB}$ and $\mathtt{t}_{FB}$, respectively. In the formal proofs, we also have $\mathtt{t}_{Fuf}$ and $\mathtt{t}_{Suf}$ to model the time for unfolding the program state. In the informal analysis, we include these into t_{SB}, t_{FB}, t_{SC} and t_{FC}, respectively.

Listing 23 also contains the definition of laminarity and the formalisation of Lemma 5.14 whose above proof we devised specifically for this formalisation. To talk about the running

time consumed up to a specific number of iterations, we introduce `orlins-one-step-time` and `orlins-one-step-time-check` and use the function iteration `compow`, analogously to the previous subsection. `running-time-initial` in Listing 23 is the final formalised statement on the running time or Orlin's algorithm, which corresponds to Term 5.2 (plus the time for the last iteration).

6. Bellman-Ford Algorithm for Path Selection

To fully implement Orlin's algorithm, we have to provide suitable functions for path selection, which we currently only assume. To obtain paths as needed for *maintain-forest*, we reuse an existing DFS formalisation [Abd25]. For *send-flow*, which needs minimum-cost paths, we use the Bellman-Ford algorithm [Bel58, For56, Moo59], as costs of residual edges can be negative.[4] There is an Isabelle/HOL formalisation of Bellman-Ford [WHN18] by Wimmer et al., which, however, only computes the distances instead of actual paths. Furthermore, they formalised the vertices' distances to the source as a function, instead of a data structure. Nevertheless, it served as a useful inspiration. For the Bellman-Ford algorithm, we assume a simple graph whose edges $\mathcal{E}$ are ordered pairs. In that graph, we fix a single vertex s as the *source*.

Graph Format. To formalise the algorithm, we fix a list of the graph's edges and vertices (Listing 24), together with a cost/weight function `edge-costs`. `edge-costs u v` is the cost of directly going from `u` to `v`. Effectively, `edge-costs` takes the role of a weighted adjacency matrix. We iterate over edges and vertices by a simple functional `fold`. We model edges e not belonging to the graph by e having infinite weight. The algorithm works on a data structure `connections`, specified as ADT, mapping vertices x to a pair consisting of an `ereal` and a vertex. Recall that `ereal` is the Isabelle/HOL type for reals extended with $\{+\infty, -\infty\}$.

Remark 6.1. Here we do not reuse the standard graph representation like the `multigraph` loacle that was introduced in Section 2.2. A graph as set of edges is suitable for describing theory and algorithms for flows, but it is not suitable for the Bellman-Ford algorithm whose locales are tailored to implementing this algorithm.

Dynamic Programming. For every vertex v, the goal is to find an s-v-path whose accumulated edge weight is minium for all s-v-paths. Similar to the previous formalisation [WHN18], we formally define $OPT(l, s, v)$ as the least accumulated weight of an s-v-path of length $\leq l$ (number of vertices), as shown in Listing 25. We also prove a Bellman Equation (Listing 25). One can compute the weighted distances between s and all other vertices, which are the costs of the shortest paths, recursively via the Bellman Equation. We can simulate the recursive process with an iterative computation (dynamic programming). This is much more efficient since intermediate results can be shared among the computation paths that would appear during the recursion.

For all vertices v, the Bellman-Ford algorithm computes continuously improving approximations $\delta(v)$ for the weight of the shortest path from s to v. $\delta(v)$ is the weight of some

[4] One could introduce an auxiliary residual graph with non-negative weights, allowing for the faster Dijkstra's algorithm instead [KV12]. Then, Bellman-Ford is required only once in the beginning. We do not pursue this further for this formalisation.

```
locale bellman-ford-spec =
  fixes connection-update::'a ⇒ 'a option × ereal ⇒ 'con-impl ⇒ 'con-impl
  and connection-empty and connection-lookup
  and connection-invar and connection-delete
  assumes connection-map:
    Map connection-empty connection-update connection-delete
        connection-lookup connection-invar
  fixes es :: ('a × 'a) list and vs :: 'a list and edge-costs :: 'a ⇒ 'a ⇒ ereal

locale bellman-ford =
  bellman-ford-spec where connection-update =
    connection-update :: 'a ⇒ 'a option × ereal ⇒ 'con-impl ⇒ 'con-impl
  for connection-update +
  assumes no-MInfty-costs: ⋀ u v. edge-costs u v > -∞
  assumes edge-costs-outside-es: ⋀ u v. edge-costs u v < ∞ ⟹ (u, v) ∈ set es
  assumes v-non-empt: vs ≠ []
  assumes vs-set-es: set vs = dVs (set es)
  assumes distinct-vs: distinct vs
  assumes distinct-es: distinct es
```

Listing 24. Locales for Bellman-Ford Algorithm

s-v-path p. It also maintains the predecessor $\pi(v)$ of v, which is the second-to-last vertex in p. On well-defined inputs, in other words, weight-conservative graphs, the *predecessor graph* $\Pi = \{(\pi(v), v) | \, v. \, v \in \mathcal{V} \wedge \pi(v) \text{ is defined}\}$ is always a *directed acyclic graph (DAG)*.

In every iteration of the Bellman-Ford algorithm, we perform so-called *relaxations* for all edges e [CLRS09]: if for the edge $e = (u, v)$, the distance $\delta(s, v)$ is greater than $\delta(s, u) + c(e)$ for the cost function c, then store $\delta(s, u) + c(u, v)$ as $\delta(s, v)$ and record u as v's new predecessor $\pi(v)$. This can be seen in Listing 25 where the pairs of the π and δ values are stored in `connections`. By iterating over all edges while relaxing each of them, we consider paths of increasing length.

An induction over l and application of the Bellman Equation leads to the statement `bellman-ford-shortest` from Listing 25, which says that after relaxing all edges l times, $\delta(v) \leq OPT(l, s, v)$ for every vertex v.

Thomas: comment in intro about the dynamics of formalisation. Informal proofs in literature, invariants extracted before and during formalisation, now reconstructed informal proofs in this paper. THe ideas behind most invariants are not new. We make them explicit and isolate them as the important properties. We investigate the semnatic structure of the proof by formalising.

Invariants. We now have to prove that we will have equality for $l = n - 1$. While developing the formalisation, we identified the following invariants:

(IBF-1) For every vertex v, $\delta(s, v) \leq OPT(l, s, v)$ where $l < n$ is the iteration number (essentially `bellman-ford-shortest`).
(IBF-2) For every vertex v, $\delta(s, v) < \infty$ if and only if $\pi(v)$ is defined or $v = s$.
(IBF-3) If $\delta(s, s) \neq 0$ anymore, s has a defined predecessor.
(IBF-4) If a vertex v has a predecessor $\pi(v)$, there is a path p from s to $\pi(v)$ and $weight\,(pv) = \delta(s, v)$.

```
definition OPT l s t
  = Min ({weight (s#xs@[t]) | xs. length xs + 1 ≤ l ∧ set xs ⊆ set vs}
           ∪ {if t = s then 0 else ∞})

lemma Bellman-Equation: OPT (Suc l) s v =
  min (OPT l s v) (Min {OPT l s u + edge-costs u v | u. u ∈ set vs})

definition relax connections  u v =
 (let csu = snd (the (connection-lookup connections u));
      csv = snd (the (connection-lookup connections v)); cuv = edge-costs u v
   in (if csu + cuv < csv then connection-update v (Some u, csu + cuv) connections
       else connections))

fun bellman-ford :: nat ⇒ 'con-impl ⇒ 'con-impl where
 bellman-ford 0 connections = connections |
 bellman-ford (Suc n) connections =
   foldr (λ (x, y) done. relax done x y) es (bellman-ford n connections)

lemma bellman-ford-shortest:
 assumes s ∈ set vs
   shows snd (the (connection-lookup (bellman-ford l (bellman-ford-init s)) v))
           ≤ OPT l s v
```

Listing 25. Building the Bellman-Ford Algorithm

(IBF-5) If a vertex has a predecessor, the predecessor is s or it has a predecessor itself.
(IBF-6) If a vertex has a predecessor $\pi(v)$, then $\delta(s, v) \geq \delta(s, \pi(v)) + c(\pi(v), v)$.
(IBF-7) The predecessor graph Π is acyclic.

The only invariant Wimmer et al. use in their Bellman-Ford formalisation [WHN18] is Invariant IBF-1, however, as an equality instead. But, since, unlike them, we need to extract paths, we have to prove invariants about the predecessors map, which are Invariants IBF-2 - IBF-6. Invariants IBF-1 - IBF-6 do not need assumptions on the structure of the graph and these follow by simple computational inductions over `bellman-ford`. Invariant IBF-7 holds, provided that the graph is weight-conservative (absence of negative-weight cycles). Its proof is as follows:

Proof Sketch for Invariant IBF-7. Assume the graph over edges $\mathcal{E}$ has no negative cycle. We can lift the monotonicity property of Invariant IBF-6 to paths in Π, i.e. $\delta(s, v) \geq \delta(s, u) + c(p)$ for vertices u, v and an u-v-path p in Π. If the relaxation of $e = (x, y)$ adds a cycle to Π, $\delta(s, x) + c(x, y) < \delta(s, y)$ and there is a path p in Π leading from y to x. Then $\delta(s, y) + c(p) + c(x, y) < \delta(s, y)$ which would make the weight of px negative, resulting in a contradiction. □

From IBF-1 and IBF-4, one could infer that the distances computed by the algorithm will eventually equal the lengths of shortest paths.

How to Obtain a Path. Backward search along edges in Π yields paths of exactly this weight (function `search-rev-path` from Listing 26). It follows from the invariants that s is reachable by iterating the predecessor, which is equivalent to termination of `search-rev-path`. We can prove that searching from a vertex v yields a path in the

```
function (domintros) search-rev-path where
 search-rev-path s connections v =
  (if s = v then [s]
   else (let u = the (fst (the (connection-lookup connections v)))
          in v#search-rev-path s connections u))

lemma s-reachable-in-some-steps:
 assumes fst (the (connection-lookup connections v)) ≠ None and (*invariants*)
   shows ∃ i. (λ w. the (fst (the (connection-lookup connections w))))^^i v = s

lemma term-s-compow-reachable:
 assumes search-rev-path-dom (s, connections,v)
   shows ∃ i. ((λ w. the (fst (the (connection-lookup connections w))))^^i) v = s

lemma search-rev-path-is-in-pred-graph:
  assumes search-rev-path-dom (s, connections, v) and (*invariants*)
      and fst (the (connection-lookup connections v)) ≠ None
    shows vwalk-bet (pred-graph connections) s
                    (rev (search-rev-path s connections v)) v

definition vs-path s t xs =
  (length xs ≥ 2 ∧ hd xs = s ∧ last xs = t ∧ set xs ⊆ set vs)
definition opt-vs-path s t xs =
  (vs-path s t xs ∧ (∀ ys. vs-path s t ys ⟶ weight ys ≥ weight xs))

theorem computation-of-optimum-path: (*preconditions simplified*)
  assumes connections = bellman-ford (length vs - 1) (bellman-ford-init s)
      and fst (the (connection-lookup connections v)) ≠ None
    shows opt-vs-path s v (rev (search-rev-path s connections v))
```

Listing 26. Obtaining Paths from the State of Bellman-Ford (simplified)

predecessor graph leading from s to v. We also show that paths in the predecessor graph are better than optimum paths, making them optimum themselves.

7. Orlin's Algorithm Executable

After verifying Orlin's algorithm, which involved assuming path selection functions and abstract data types for the program variables, and after implementing and verifying the Bellman-Ford algorithm to find minimum-cost paths, we are finally ready to combine these to get an executable implementation of Orlin's algorithm. We instantiate the locale constants and prove the corresponding axioms.

Basic Data Structures. We use appropriate data structures to instantiate the variables like the flow or balances in the program state. In the locale headers, we assume structures that map keys/indices, such as vertices or edges, to values, such as reals. We use a red-black tree implementation of maps by Nipkow [Nip25b]. However, we add a number of special operations needed in our context, for example, to update all values simultaneously or to select the *argmax*. We implement the set of active edges as a list, which would achieve the

```
global-interpretation dfs: DFS
 where insert = neighb-insert (*plus many others*) and G = F and t = t::nat
   and s = s::nat for F t s
 defines dfs-initial-state = dfs.initial-state and dfs-impl = dfs.DFS-impl
     and neighbourhood =dfs.Graph.neighbourhood

definition dfs F t ≡ (dfs.DFS-impl F t)
definition dfs-initial s ≡ (dfs.initial-state s)
definition get-path u v E = rev (stack (dfs E v (dfs-initial u)))

lemma maintain-forest-axioms-extended:
  assumes maintain-forest-spec.maintain-forest-get-path-cond vset-empty vset-inv
            isin lookup t-set adj-inv get-path u v E p q
    shows vwalk-bet (digraph-abs E) u p v and distinct p
```

Listing 27. Use DFS to Select Forest Paths (simplified)

best possible asymptotic running time as we have to iterate over all edges in every iteration of *maintain-forest*.

Forest Paths. Forest paths are necessary for moving balance from one component's representative to the other when we join two components (Section 5.2). As we saw earlier, forest paths are always minimum-cost augmenting paths (Lemma 5.9), which is why we can select paths in the forest irrespectively of the costs, for exaple, by BFS or DFS.

We use the DFS formalisation by Abdualziz [Abd25] that computes a path in an adjacency map leading from a source to a target if there exists one. From the auxiliary and implementation invariants we prove that the forest, represented as an adjacency map, satisfies all necessary invariants, and that the represented graph is finite. These are the preconditions for DFS working correctly. Recall that we encapsulated such preconditions in definitions like `maintain-forest-get-path-cond`, which were provable inside the locales for Orlin's algorithm. Listing 27 shows the instantiation of DFS with red-black tree operations, the definition of the selection function `get-path` from the connectives the global interpretation gives us, and its correctness lemma which coincides with the assumptions on `get-path` in the locale `maintain-forest`.

Application of Bellman-Ford. The function *send-flow* sends flow from sources to targets along minimum-cost paths that use active and forest edges only. The residual graph in which we need paths may have multiple edges between two vertices (multigraph). On the contrary, the Bellman-Ford algorithm assumes a simple graph given by a weighted adjacency matrix. As weights for the matrix we take the costs of the cheapest edge that is available between the respective pair of vertices.

In greater detail, let us consider the case of path selection where we first select the source (Line SF5 of *send-flow*). Definition `get-edge-and-costs-forward` from Listing 28 shows the conversion to a problem on which we can use Bellman-Ford: for a flow `f`, 'not blocked' map `nb`, and two vertices `u` and `u`, we first look up all original graph edges from `u` to `v` (`ingoing-edges`) and from `v` to `u` (`outgoing-edges`), respectively. We then find the cheapest forward or backward residual edge pointing from `u` to `v`, alongside with the corresponding costs. We say that this residual edge realises the simple edge (`u`, `v`). We only

look at those edges that have positive residual capacity with respect to `f`, and that are not blocked (either forest or active edges). The central property of this function is that for every $e \in \mathit{actives} \cup \mathcal{F}$ with $\mathfrak{u}_f(e) > 0$, it finds $e' \in \mathit{actives} \cup \mathcal{F}$ with $\mathfrak{u}_f(e') > 0$ connecting the same pair of vertices where $\mathfrak{c}(e') \leq \mathfrak{c}(e)$. We can use the second component of the pair returned by `get-edge-and-costs-forward`, namely, the residual costs, as an input to the Bellman-Ford algorithm, alongside with a source `s`, as done by definition `bellman-ford-forward`.

For a program state, we obtain a path by the function `get-source-target-path-a` (Listing 29): `get-target-for-source-aux-aux` searches for a vertex `t` with `b(t)` $< -\epsilon \cdot \gamma$ such that `bf` found a path, that is, `t`'s distance to `s` is less than ∞. We then use `search-rev-path-exec` to go from predecessor to predecessor until we reach `s`. Edge by edge, we have to transform this path back into a path over residual edges, for which we use `pair-to-realising-redge-forward`, which gives for every simple edge a residual edge of minimum costs connecting the same vertices.

We give a proof sketch explaining why this function satisfies the locale assumptions `get-source-target-path-a-axioms` that we stated in Listing 19 as part of the locale `send-flow`.

Thomas: I assume the reviewer is talking about the proof sketch. shorten this?, merge it with previous discussion

Mohammad: I don't want to do this without discussing with you first because you insisted on the split/double discussion under different aspects.

Proof Sketch of Correctness of Minimum-Cost Path Selection. Assume that the state satisfies `get-source-target-path-a-cond` (henceforth C) from Listing 18. Let P be the path returned by `get-source-target-path-a`. Let P_1 be an arbitrary minimum-cost augmenting path from `s` to `t`. P_1 might use deactivated edges, i.e. edges that are blocked. Because of Invariant IO-3, which is entailed by C, the endpoints x and y of those edges e belong to the same forest components. The forest paths connecting x and y are optimum (Lemma 5.9), hence cheaper or have the same cost as e. Therefore, there is a P_2 that is cheaper than P_1 and uses active and forest edges only. Also because of C, all forest edges have positive flow, which means that P_2 is still augmenting. By the construction from Listing 28, there is a corresponding path P_3 in the graph we use for BF and that might be even cheaper. Since the optimality invariant is part of C, the graph we use for BF does not contain a negative cycle. From properties of BF, we know that P_3 is distinct. BF computes a distinct path P_4 from `s` to `t` that might even be cheaper. The function from Listing 28 also gives realising edges. We use this to transform P_4 into the path P in the residual graph, which has positive capacity and whose costs are equal to the costs of P_4. The costs of P are cheaper than those of P', making P a minimum-cost augmenting path. Also, P uses only forest and active edges. □

Line SF11 in *send-flow* is the other use case for minimum-cost path selection: given a target t, we need to find a source s reaching t and a minimum-cost path from s to t. We reverse all directions and use t as source for BF instead. Except for this change and the different balance-related properties required for t, the definitions for `get-source-target-path-b` and proofs that the specification in Listing 19 is satisfied are analogous.

Although the proofs are conceptually simple, the formalisation is comparably verbose due to the translations between different graph formats. We use a translation between multigraphs and simple graphs including weights, and the reduction of the single-target-shortest-path to the single-source-shortest-path problem.

```
definition get-edge-and-costs-forward nb (f::'edge ⇒ real) =
(λ u v.
(let ingoing-edges = (case (realising-edges-lookup realising-edges (u, v)) of
                        None ⇒ [] | Some list ⇒ list);
  outgoing-edges = (case (realising-edges-lookup realising-edges (v, u)) of
                        None ⇒ [] | Some list ⇒ list);
  (ef, cf) = find-cheapest-forward f nb ingoing-edges (create-edge u v) PInfty;
  (eb, cb) = find-cheapest-backward f nb outgoing-edges (create-edge v u) PInfty
 in (if cf ≤ cb then (F ef, cf) else (B eb, cb))))

definition bellman-ford-forward nb fs =
 bellman-ford-algo (λ u v. prod.snd (get-edge-and-costs-forward nb f u v)) es
                   (length vs - 1) (bellman-ford-init-algo vs s)
```

LISTING 28. Costs for Bellman-Ford Obtained from Program State Variables

```
definition pair-to-realising-redge-forward state = (λ e. prod.fst
(get-edge-and-costs-forward
(abstract-bool-map (not-blocked-lookup (not-blocked state)))
(abstract-real-map (flow-lookup (current-flow state))) (prod.fst e) (prod.snd e)))

definition get-source-target-path-a state s =
 (let bf = bellman-ford-forward
            (abstract-bool-map (not-blocked-lookup (not-blocked state)))
            (abstract-real-map (flow-lookup (current-flow state))) s
   in case (get-target-for-source-aux-aux bf
             (λ v. abstract-real-map (bal-lookup (balance state)) v)
              (current-γstate) vs) of
    Some t ⇒
     (let Pbf = search-rev-path-exec s bf t Nil;
          P = (map (pair-to-realising-redge-forward state) (edges-of-vwalk Pbf))
       in Some (t, P))|
    None ⇒ None)
```

LISTING 29. Using Bellman-Ford to Obtain Minimum-Cost Paths

8. Finite and Infinite Capacity Networks

Orlin's algorithm is currently the fastest known algorithm to solve the minimum-cost flow problem, when there are (1) no negative cycles (since the zero flow has to be a minimum-cost flow for $b - b$ in the beginning, see Invariant IO-6) and (2) no edges with finite capacity, which apparently pose significant limitations to the algorithm's usability. Fortunately, every finite and mixed capacity instance I can be reduced to a corresponding instance I' without capacities with a blow-up in graph size that is $\mathcal{O}(n + m)$ [Ord56, FF62]. The result I' is acyclic, hence without negative cycles, if there are no negative infinite-capacity cycles in I. If there are such cycles in I, however, flows of arbitrarily low costs exist. This reduction makes Orlin's algorithm applicable to networks with finite capacities, as well, provided that the optimisation problem is not unbounded.

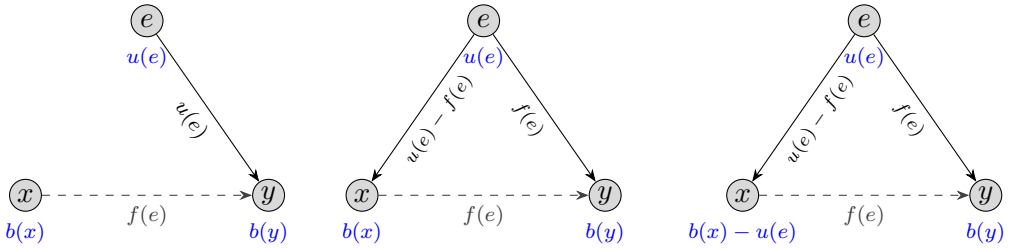


FIGURE 5. We replace an edge e from x to y with finite capacity $u(e) < \infty$ with a new vertex e with $b(e) = u(e)$ and two edges $e_1 = (e, x)$ and $e_2 = (e, y)$, each with infinite capacity. After first adding e_2, the flow along this new edge is exactly $u(e)$ although it should be equal to $f(e) \leq u(e)$. Adding the other edge e_1 makes this possible because we can then send $f(e)$ units of flow from e to y and $u(e) - f(e)$ units to x. The vertex x has now an abundant flow of $u(e)$ ($f(e)$ units do not leave x anymore and there is an additional ingoing flow of $u(e) - f(e)$). We need to adjust the balance $b(x)$ to again obtain a balance-complying flow.

In this section we describe our formalisation and verificiation of the functional correctness of this reduction. This is a generalisation of a method first given by Ford and Fulkerson [FF62], which is also presented by Korte and Vygen [KV12].

The Idea. For an original network with capacities, we construct a new network without capacities and modified balances such that a valid flow in one network allows us to easily find a valid flow in the other one. The idea is to encode edge capacities into vertex balances, which means we introduce vertices that replace edges in the network, as shown in Figure 5: for an edge e with endpoints x and y, and capacity $u(e) < \infty$, we need a corresponding edge e_2 that carries a flow $f'(e_2) = f(e) \leq u(e)$ despite e_2's infinite capacity. We can achieve this by deleting e from the network and replacing it with a vertex e with $b(e) = u(e)$ and an edge $e_2 = (e, y)$. Since this would enforce $f'(e_2) = u(e)$, the flow needs another way to leave the vertex e. We introduce $e_1 = (e, x)$ to remove the remaining flow from e (Figure 5). This results in $f'(e_2) = f(e) \leq u(e)$, $f'(e_1) = u(e) - f(e)$ and $f'(d) = f(d)$ for all other edges d as a possible flow in the modified network.

Flow validity is not only about capacities but about balances as well. Balance constraints that f satisfied before might not be satisfied by f'. While the excess of y is still unaltered, the modified flow violates the balance constraints at x: An amount of $f(e)$ is not leaving the vertex x anymore and there is an additional ingoing flow of $u(e) - f(e)$, which means that we must decrease $b(x)$ by $u(e)$ to maintain feasibility.

The Reduction. For the network $(\mathcal{V}, \mathcal{E})$ with costs $c : \mathcal{E} \to \mathbb{R}$, balances $b : \mathcal{V} \to \mathbb{R}$ and capacities $u : \mathcal{E} \to \mathbb{R}_0^+ \cup \{+\infty\}$, we define $\mathcal{V}'$, $\mathcal{E}'$, c', b': let $\delta_\infty^-(x)$ and $\delta_\infty^+(x)$ be the ingoing and outgoing edges of x with infinite capacity, respectively. Let $\mathcal{E}_\infty$ be set of edges in $\mathcal{E}$ with infinite capacity. We set $\mathcal{V}' = \mathcal{V} \cup (\mathcal{E} - \mathcal{E}_\infty)$. Let $\mathcal{E}_1 = \{(e, x).\ e \in \delta^+(x) - \delta_\infty^+(x) \wedge x \in \mathcal{V}\}$ and $\mathcal{E}_2 = \{\langle e, x\rangle.\ e \in \delta^-(x) - \delta_\infty^-(x) \wedge x \in \mathcal{V}\}$. By definition, let (a, b) and $\langle a, b\rangle$ denote two distinct pairs with components a and b. Now set $\mathcal{E}_3 = \mathcal{E}_\infty$ and $\mathcal{E}' = \mathcal{E}_1 \cup \mathcal{E}_2 \cup \mathcal{E}_3$. Let $c'(e, x) = 0$ for $(e, x) \in \mathcal{E}_1$, $c'\langle e, x\rangle = c(e)$ for $\langle e, x\rangle \in \mathcal{E}_2$ and $c'(e) = c(e)$ for $e \in \mathcal{E}_3$. For

$e \in \mathcal{E} - \mathcal{E}_\infty$ we set $b'(e) = u(e)$. For vertices $x \in \mathcal{V}$ we define

$$b'(x) = b(x) - \sum_{e \in \delta^+(x) - \delta^+_\infty(x)} u(e).$$

Theorem 8.1 (Reduction of mixed-capacity to infinite-capacity Problems)**.** *We assume a flow network $\mathcal{G}$ based on a directed multigraph. We construct $\mathcal{G}'$ with infinite capacities as sketched above. The following hold:*

1. *For a flow f feasible in $\mathcal{G}$, f' is feasible in $\mathcal{G}'$ where $f'(e,x) = u(e) - f(e)$ for $(e,x) \in \mathcal{E}_1$, $f'\langle e,x\rangle = f(e)$ for $\langle e,x\rangle \in \mathcal{E}_2$ and $f'(e) = f(e)$ for $e \in \mathcal{E}_3$. The costs correspond as well, i.e. $c(f) = c'(f')$.*
2. *For a flow f' feasible in $\mathcal{G}'$, f is feasible in $\mathcal{G}$ where $f(e) = f'\langle e,x\rangle$ for x being the second vertex of $e \in \mathcal{E} \setminus \mathcal{E}_\infty$. $f(e) = f'(e)$ for $e \in \mathcal{E}_\infty$. Again $c'(f') = c(f)$.*
3. *An optimum flow in one network defines an optimum flow in the other network.*
4. *$\mathcal{G}'$ contains a negative cycle iff $\mathcal{G}$ contains a negative cycle with infinite capacity.*

Formalisation. Endpoints of edges in the new graph are uniquely determined by the original edge the new edge e refers to and whether e is in $\mathcal{E}_1$, $\mathcal{E}_2$ or $\mathcal{E}_3$. Storing the vertices in the new graph's edges would be redundant. We encode whether an edge is in $\mathcal{E}_1$, $\mathcal{E}_2$ or $\mathcal{E}_3$ into a data type `hitchcock-edge` with constructors `outedge`, `inedge` and `vtovedge` (Listing 30), which are meant to realise edges in $\mathcal{E}_1$, $\mathcal{E}_2$ and $\mathcal{E}_3$, respectively. The new graph has edges of type `hitchcock-edge`. The definitions of `fst` and `snd` (not shown in the listing) take a `hitchcock-edge` and return an edge or vertex of the old graph. In the formalisation, the latter two are of different types which is why we wrap them in a `hitchcock-wrapper` to obtain vertices of uniform type (that is, type not depending on whether they wrap an edge or a vertex) as required by the `multigraph` locale. The `dummy` is necessary to ensure the possibility of an edge between every vertex, as required by the axioms of the `multigraph` locale. `get-edge` returns the edge in the original graph a `hitchcock-edge` refers to if there is any. Functions like `get-edge` are necessary to express the translation/corespondence between the two graphs. The formal proof of the statement in Listing 30 involves plenty of sum manipulation which is hinted by the informal definitions.

Solving several Types of Flows. We made the above reduction executable, as well, allowing us to solve different types of flow problems in strongly polynomial time: Orlin's algorithm itself finds minimum-cost flows in networks with infinite capacity. Our formalisation contains an instantiation for both simple graphs and multigraphs. We can compute mixed-capacity and finite-capacity minimum-cost flows by combining Orlin's algorithm with the reduction from Theorem 8.1. For all of those problems, there is now verified code.

Existence of Optimum Solutions. We can characterise the existence of an optimum solution for feasible problems by the absence of cycles with infinite capacities and negative total costs: after applying the reduction to such an instance, Orlin's algorithm will terminate and return an optimum flow, resulting in a constructive proof for one implication. Conversely, a negative cycle with infinite capacities allows for arbitrarily small costs by arbitrarily high/many augmentations. It is necessary to use Orlin's algorithm here as the simpler methods *successive-shortest-path* or *scaling* might not terminate. This characterisation of the existence of optimum solutions is the final theorem on minimum-cost flows in our formalisation.

```
datatype ('a, 'edge) hitchcock-edge =
 outedge 'edge | inedge 'edge | vtovedge 'edge |
 dummy ('a, 'edge) hitchcock-wrapper ('a, 'edge) hitchcock-wrapper
...
fun get-edge where
 get-edge (outedge e) = e | (*analogous for inedge and vtovedge*)
...
theorem reduction-of-min-cost-flow-to-hitchcock-general: [Theorem 8.1]
  fixes c b D
  assumes flow-network fstv sndv create-edge u ℰ
  defines fstv' ≡ new-fstv-gen fstv and sndv' ≡ new-sndv-gen fstv sndv
      and create-edge' ≡ new-create-edge-gen
      and make-pair' ≡ multigraph-spec.make-pair fstv' sndv'
  defines ℰ1 ≡ new-ℰ1-gen ℰ u and ℰ2 ≡ new-ℰ2-gen ℰ u
      and ℰ3 ≡ new-ℰ3-gen ℰ u
  defines ℰ' ≡ ℰ1 ∪ ℰ2 ∪ ℰ3
  defines u' ≡ (λ e. PInfty) and c' ≡ new-c-gen D fstv ℰ u c
      and b' ≡ new-b-gen fstv ℰ u b
shows flow-network fstv' sndv' create-edge' u' ℰ'
and ⋀ f f'.
 ⟦flow-network.isbflow fstv sndv u ℰ f b; f' = new-f-gen fstv ℰ u f⟧
  ⟹ flow-network.isbflow fstv' sndv' u' ℰ' f' b'
    ∧ (cost-flow-spec.𝒞 ℰ c f = cost-flow-spec.𝒞 ℰ' c' f')
and ⋀ f f'.
 ⟦flow-network.isbflow fstv' sndv' u' ℰ' f' b'; f = old-f-gen ℰ u f'⟧
  ⟹ flow-network.isbflow fstv sndv u ℰ f b
    ∧ (cost-flow-spec.𝒞 ℰ c f = cost-flow-spec.𝒞 ℰ' c' f')
and has-neg-cycle make-pair' ℰ' c' ⟷ has-neg-infty-cycle make-pair ℰ c u
and ⋀ f f' .
 ⟦f = old-f-gen ℰ u f'; cost-flow-network.is-Opt fstv' sndv' u' c' ℰ' b' f'⟧
  ⟹ cost-flow-network.is-Opt fstv sndv u c ℰ b f
and ⋀ f f' .
 ⟦f' = new-f-gen fstv ℰ u f; cost-flow-network.is-Opt fstv sndv u c ℰ b f⟧
  ⟹ cost-flow-network.is-Opt fstv' sndv' u' c' ℰ' b' f'
```

Listing 30. The Reduction from Theorem 8.1

Remark 8.2. Combining this reduction with Orlin's algorithm is still the most efficient method (in terms of worst-case running time) for general minimum-cost flows, that is, those with arbitrary finite or infinite capacities, and arbitrary balances and costs [KV12, Sch03]. Another method, that directly works on the capacitated network, achieves the same running time [Vyg02]. For restricted problems, weakly polynomial methods might be faster, for example, almost linear if the balances, capacities and costs are integral and polynomially bounded [CKL+22, BCK+23].

9. Discussion

Related Work. The first formalisation of maximum flows, namely, the max-flow-min-cut theorem and the Ford-Fulkerson algorithm, was in 2005 in Mizar by Lee [Lee05]. Lean's *mathlib* [mat24] does not contain network flows; the only substantial Lean development we are ware of is an unpublished AI-assisted formalisation of the max-flow-min-cut theorem [Jo26]. We are also not aware of any developments in Rocq.

Thomas: I know the references for lean are not good but there is nothing else and the reviewer asks for that.

The closest work to ours is Lammich and Sefidgar's work on maximum flows [LS19] in Isabelle. 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 with respect to the current flow maintained by the algorithm, and which Lammich and Sefidgar have already formalised [LS19]. Our development on min-cost flows is powerful enough to rederive most of the results about max-flows they have already formalised. A notable result contained in both formalisations is the Max-Flow-Min-Cut Theorem [FF56, DF57] and the max-flow analogue to Theorem 3.2. A result that we formalised for the first time is the *decomposition* of so-called *s-t-flows* [Gal58, FF56].

Results on augmenting paths (and their characterisation of optimality, like Theorem 3.2) have been formalised for the theory of optimisation of matching (Berge's Lemma) [AM26], and for matroid intersection [AAMR25].

We are, however, the first to formalise algorithms that involve scaling, which is used to design fast optimisation algorithms, including algorithms with the fastest worst-case running times for different variants of matching and shortest path problems [DPS17, Gab17, GT89, GT91, OA92], in addition to minimum-cost flows, which we consider here. Our work here provides a blueprint to formalise the correctness of those other scaling algorithms.

The work by Lammich and Sefidgar [LS19] also contains formalisations of the running time analyses. They too count loop iterations as the main indicator of running time. For Orlin's algorithm, however, the proofs involve properties of laminar families, which were never formalised before. These properties are necessary (and reusable) for arguing about running times of many other state-of-the-art combinatorial optimisation algorithms, including Edmonds' weighted matching algorithm [Edm65], the Micali-Vazirani algorithm [Vaz24], and Gabow's cost scaling weighted matching algorithm [Gab17].

We model multigraphs as sets of edges together with functions giving the two endpoints (Listing 1). Noschinski's Isabelle/HOL representation [Nos15] is very close to this, except that it treats the vertices as a distinct parameter. To make the reasoning smoother, our representation is purely edge-based, and thus, avoids invariants about relationships between edges and vertices. In Isabelle/HOL, Edmonds formalised undirected hypergraphs [EP24] as a set of sets (the edges) and a set of allowed elements (possible vertices). In Lean's *mathlib*, digraphs are an adjacency predicate over vertices, and (with dependent types) multigraphs are modelled by Quivers.

Mathematical Insights. A main outcome of our work, from a mathematical perspective, is our proof of Theorem 3.7, which is its first complete combinatorial proof. We have devised a

first attempt at a combinatorial proof for the gap (Lemma 3.6), described in the conference paper [AA24]. That proof, which involved a complex graphical construction, was later simplified by Vygen [Vyg24], who proposed a proof that depends on Eulerian circuits, which we then further simplified to one main insight: by adding a fresh edge, we can make the structure regular, or in other words, by making the set of edges form a set of disjoint cycles instead of cycles and one path. We use a similar approach of regularisation by adding an edge to prove the decomposition theorem for max-flows. Another less important and rather technical insight that came out of our formalisation is that we showed that PO-8 holds, even for problems with finite capacities. This is unlike Korte and Vygen's as well as Plotkin and Tardos's proof of this claim [PT90], which are the only two detailed proofs of this claim. This highlights the potential role of formalising deep proofs in filling gaps as well as in the generation of new proofs and/or insights. There were also other non-trivial claims in the main textbook we used as reference [KV12], for which no proof is given. For example, it is claimed that the accumulated augmentations through a forest edge are below $2(n-1)\gamma$. We could not formalise the textbook's proof as it also had gaps, and we devised Invariants IMF-1 and IMF-2 to be able to prove it. Other gaps like the proof of Lemma 5.8, which is merely asserted in the literature in a non-formalisable way, had to be closed for the Properties PO-5 and PO-7 which both involve the change in Φ. We also detailed a few technicalities for Lemma 3.8, and a few parts of the termination proof.

Thomas: should we say we often fixed technicalities? Or is that clear anyway?

Methodology. The formalisation of the algorithms presented here is around 36k lines of proof scripts (flow theory: 7k, min-cost flow algorithms and running time: 20k, instantiation of Orlin's algorithm and Bellman-Ford: 9k). Our methodology is based on using Isabelle/HOL's *locale*s to implement Wirth's notion of step-wise refinement [Wir71], thereby compartmentalising different types of reasoning. Many authors, such as Nipkow [Nip15], Maric [Mar20], and Abdulaziz [Abd25], used this locale-based implementation of refinement earlier. 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.

A limitation is that the locales become verbose quite soon, especially when employing several instances of the same ADT locale. This is a common issue when formalising an algorithm with many variables like Orlin's algorithm since each program variable comes with its own ADT. The verbosity of the approach could be improved using records, as shown by Lammich and Lochbihler [LL10].

We note that our focus here is more on formalising the mathematical argument behind the algorithm and less on obtaining the most efficient executable program. A notable alternative implementation of refinement is Peter Lammich's [Lam19] framework. It has the advantage of generating fast imperative implementations, but comes at the cost of a steep learning curve and somewhat fragile automation.

We would, however, always start with the least understood (sub-)algorithms and then go on to the better understood ones. This led us to using the locale-based approach in two ways: top-down and bottom-up. For instance, our approach to specifying `send-flow` 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 `send-flow` and `maintain-forest`, and then

started defining and proving the correctness of `orlins`, which calls both `send-flow` and `maintain-forest`.

We define procedures as recursive functions using Isabelle/HOL's function package [Kra09], program states as records, and invariants as predicates on program states. We derive theorems characterising different properties of recursive functions, and supply them to Isabelle/HOL's classical reasoning and simplification. In this approach, the automation handles proofs of invariants and monotone properties, like the growth of the forest or the changes in Φ. Again, one can use other approaches to model algorithms and automate reasoning about them, like using monads [LT12] or while combinators [BN02]. Both of those approaches allow for greater automation, which is particularly useful for reasoning about low-level implementations. However, those two approaches are not as useful where the majority of the correctness argument lies in background mathematical proofs, as in the case of Orlin's algorithm, because they add at least one layer of definitions, which is here a monad, between the theorem prover's basic logic and the algorithm's model.

A last noteworthy methodological point is the avoidance of code replication. This happens with most approaches to stepwise-refinement, for example with Lammich's framework [Lam19] or Nipkow's While-combinator [BN02]. In those approaches, one defines an algorithm a number of times, where each time a higher level of detail is added, such as that at one level one might use mathematical sets, the next a list-based implementation, and a third with an optimised array-based implementation. A somewhat extreme example is Nipkow's verification of a functional implementation of the Gale-Shapley algorithm, where the algorithm was defined 8 different times [Nip24], at an increasing level of detail. Here, we specify the algorithm only once using ADTs [Hoa72] and then obtain executable code by instantiating those ADTs with purely functional implementations. We adopted this approach, that was also used for matroid and greedoid algorithms [AAMR25], from earlier work on verifying graph traversal algorithms [Abd25]. The main advantage of this approach is that it avoids replication, but a risk is that starting directly with the ADTs could lead to too many low-level implementation-related details cluttering the verification process. In our experience here, however, Isabelle's automation, with minimal setup, can discharge most implementation related proof goals, allowing the user to focus on more interesting goals at the algorithmic level. 5We note that this methodology was used to verify matroid and greedoid algorithms [AAMR25] too.

Thomas: in third_ bib there are some urls with dois in it remaining because there is no doi available

References

[AA24] Mohammad Abdulaziz and Thomas Ammer. A Formal Analysis of Capacity Scaling Algorithms for Minimum Cost Flows. In *The 15th International Conference on Interactive Theorem Proving (ITP)*, 2024. doi:10.4230/LIPICS.ITP.2024.3.

[AAMR25] Mohammad Abdulaziz, Thomas Ammer, Shriya Meenakshisundaram, and Adem Rimpapa. A Formal Analysis of Algorithms for Matroids and Greedoids. In *The 16th International Conference on Interactive Theorem Proving (ITP)*, 2025. doi:10.4230/LIPIcs.ITP.2025.2.

[Abd25] Mohammad Abdulaziz. Graph Algorithms. In *Functional Data Structures and Algorithms: A Proof Assistant Approach*, volume 67, pages 273–298. Association for Computing Machinery, New York, NY, USA, 1 edition, September 2025. doi:10.1145/3731369.3731394.

[AGOT92] Ravindra K. Ahuja, Andrew V. Goldberg, James B. Orlin, and Robert Endre Tarjan. Finding minimum-cost flows by double scaling. *Math. Program.*, 53:243–266, 1992. doi:10.1007/BF01585705.

[AM23] Mohammad Abdulaziz and Christoph Madlener. A Formal Analysis of RANKING. In *The 14th Conference on Interactive Theorem Proving (ITP)*, 2023. doi:10.4230/LIPIcs.ITP.2023.3.

[AM26] Mohammad Abdulaziz and Kurt Mehlhorn. A Formal Correctness Proof of Edmonds' Blossom Shrinking Algorithm. *The Journal of Automated Reasoning*, 70(1), 2026. doi:10.1007/s10817-026-09747-y.

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. Network Flows: Theory, Algorithms, and Applications. 1993. URL: https://dl.acm.org/doi/10.5555/137406.

[AP19] Mohammad Abdulaziz and Lawrence C. Paulson. An Isabelle/HOL Formalisation of Green's Theorem. *Journal of Automated Reasoning*, 63(3):763–786, 2019. doi:10.1007/s10817-018-9495-z.

[Bal14] Clemens Ballarin. Locales: A Module System for Mathematical Theories. *J. Autom. Reason.*, 52(2):123–153, 2014. doi:10.1007/s10817-013-9284-7.

[BCK+23] Jan Van Den Brand, Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, Sushant Sachdeva, and Aaron Sidford. A deterministic almost-linear time algorithm for minimum-cost flow. In *64th IEEE Annual Symposium on the Foundations of Computer Science (FOCS)*, 2023. doi:10.1109/FOCS57990.2023.00037.

[Bel58] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, pages 87–90, 1958. doi:10.1090/qam/102435.

[BN02] Stefan Berghofer and Tobias Nipkow. Executing Higher Order Logic. In *Types for Proofs and Programs*, Berlin, Heidelberg, 2002. doi:10.1007/3-540-45842-5_2.

[CKL+22] Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Maximum Flow and Minimum-Cost Flow in Almost-Linear Time. In *63rd IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, 2022. doi:10.1109/FOCS54457.2022.00064.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, USA, 3rd edition, July 2009.

[DF57] G. B. Dantzig and D. R. Fulkerson. *On the Max-Flow Min-Cut Theorem of Networks*, pages 215–222. Princeton University Press, Princeton, 1957. doi:10.1515/9781400881987-013.

[DPS17] 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. doi:10.1137/1.9781611974782.50.

[Edm65] Jack Edmonds. Maximum matching and a polyhedron with 0,1-vertices. *Journal of Research of the National Bureau of Standards Section B Mathematics and Mathematical Physics*, 69(1 and 2):55–56, January 1965. doi:10.6028/jres.069B.013.

[EK72] Jack Edmonds and Richard M. Karp. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *J. ACM*, 19(2):248–264, 1972. doi:10.1145/321694.321699.

[EP24] Chelsea Edmonds and Lawrence C. Paulson. Formal Probabilistic Methods for Combinatorial Structures using the Lovász Local Lemma. In *The 13th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, London UK, January 2024. doi:10.1145/3636501.3636946.

[FF56] Lester Randolph Ford and Delbert R Fulkerson. Maximal flow through a network. *Canadian journal of Mathematics*, 8:399–404, 1956. doi:10.4153/CJM-1956-045-5.

[FF62] L. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962. doi:10.2307/jj.19783677.

[For56] L. R. Ford. *Network Flow Theory*. RAND Corporation, Santa Monica, CA, 1956. URL: https://www.rand.org/pubs/papers/P923.html.

[Gab17] Harold N. Gabow. The Weighted Matching Approach to Maximum Cardinality Matching. *Fundam. Informaticae*, 154(1-4):109–130, 2017. doi:10.3233/FI-2017-1555.

[Gal58] T. Gallai. Maximum-Minimum Sätze über Graphen. *Acta Mathematica Academiae Scientiarum Hungarica*, 9(3):395–434, September 1958. doi:10.1007/BF02020271.

[GT89] Harold N. Gabow and Robert Endre Tarjan. Faster Scaling Algorithms for Network Problems. *SIAM J. Comput.*, 18(5):1013–1036, 1989. doi:10.1137/0218069.

[GT91] Harold N. Gabow and Robert Endre Tarjan. Faster Scaling Algorithms for General Graph-Matching Problems. *J. ACM*, 38(4):815–853, 1991. doi:10.1145/115234.115366.

[Hoa72] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, December 1972. doi:10.1007/BF00289507.

[IT20] Fabian Immler and Yong Kiam Tan. The Poincaré-Bendixson theorem in Isabelle/HOL. In *The 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, 2020. doi:10.1145/3372885.3373833.

[Jew58] William S Jewell. Optimal flow through networks. *Operations research*, 6(4):633–633, 1958.

[Jo26] Sunghyeon Jo. Max-flow min-cut: a comprehensive proof process, 04 2026. URL: https://github.com/ainta/lean-algorithms/tree/main/combinatoric_optimization/max_flow_min_cut/MaxFlowMinCut.

[KHSA25] Kevin Kappelmann, Fabian Huch, Lukas Stevens, and Mohammad Abdulaziz. Proof-Producing Translation of Functional Programs into a Time & Space Reasonable Model, March 2025. arXiv:2503.02975, doi:10.48550/arXiv.2503.02975.

[Kle67] Morton Klein. A Primal Method for Minimal Cost Flows with Applications to the Assignment and Transportation Problems. *Management Science*, November 1967. doi:10.1287/mnsc.14.3.205.

[Kra09] Alexander Krauss. *Automating Recursive Definitions and Termination Proofs in Higher-Order Logic*. PhD thesis, Technical University Munich, 2009. URL: https://mediatum.ub.tum.de/681651.

[KV12] Bernhard Korte and Jens Vygen. *Combinatorial Optimization*, volume 21 of *Algorithms and Combinatorics*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. doi:10.1007/978-3-642-24488-9.

[Lam19] Peter Lammich. Refinement to Imperative HOL. *Journal of Automated Reasoning*, 62(4):481–503, 2019. doi:10.1007/s10817-017-9437-1.

[Lee05] Gilbert Lee. Correctnesss of ford-fulkerson's maximum flow algorithm. *Formalized Mathematics*, 13(2):305–314, 2005.

[LL10] Peter Lammich and Andreas Lochbihler. The Isabelle Collections Framework. In *Interactive Theorem Proving*, Berlin, Heidelberg, 2010. doi:10.1007/978-3-642-14052-5_24.

[LS19] Peter Lammich and S. Reza Sefidgar. Formalizing Network Flow Algorithms: A Refinement Approach in Isabelle/HOL. *J. Autom. Reason.*, 62(2):261–280, 2019. doi:10.1007/s10817-017-9442-4.

[LT12] Peter Lammich and Thomas Tuerk. Applying Data Refinement for Monadic Programs to Hopcroft's Algorithm. In *Interactive Theorem Proving*, Berlin, Heidelberg, 2012. doi:10.1007/978-3-642-32347-8_12.

[Mar20] Filip Marić. Verifying Faradžev-Read Type Isomorph-Free Exhaustive Generation. In *Automated Reasoning*, Cham, 2020. doi:10.1007/978-3-030-51054-1_16.

[mat24] Leanprover-community/mathlib4. leanprover-community, November 2024. URL: https://github.com/leanprover-community/mathlib4.

[Moo59] Edward R. Moore. The Shortest Path through a Maze. In *Proc. Int. Symp. Switching Theory 1957, Part II*, pages 285–292, 1959.

[Nip15] Tobias Nipkow. Amortized Complexity Verified. In *The 6th International Conference on Interactive Theorem Proving (ITP)*, 2015. doi:10.1007/978-3-319-22102-1_21.

[Nip24] Tobias Nipkow. Gale-Shapley Verified. *Journal of Automated Reasoning*, 68(2):12, 2024. doi:10.1007/s10817-024-09700-x.

[Nip25a] Tobias Nipkow. Basics. In *Functional Data Structures and Algorithms: A Proof Assistant Approach*, volume 67, pages 1–10. Association for Computing Machinery, New York, NY, USA, 1 edition, September 2025. doi:10.1145/3731369.3731371.

[Nip25b] Tobias Nipkow. Red–Black Trees. In *Functional Data Structures and Algorithms: A Proof Assistant Approach*, volume 67, pages 97–106. Association for Computing Machinery, New York, NY, USA, 1 edition, September 2025. doi:10.1145/3731369.3731378.

[Nos15] Lars Noschinski. A Graph Library for Isabelle. *Math. Comput. Sci.*, 9(1):23–39, 2015. doi:10.1007/S11786-014-0183-Z.

[OA92] James B. Orlin and Ravindra K. Ahuja. New scaling algorithms for the assignment and minimum mean cycle problems. *Math. Program.*, 54:41–56, 1992. doi:10.1007/BF01586040.

[Ord56] Alex Orden. The Transhipment Problem. *Manag. Sci.*, 2(3):276–285, 1956. doi:10.1287/mnsc.2.3.276.

[Orl93] James B. Orlin. A Faster Strongly Polynomial Minimum Cost Flow Algorithm. *Operations Research*, 41(2):338–350, April 1993. doi:10.1287/opre.41.2.338.

[PT90] Serge A. Plotkin and Éva Tardos. Improved Dual Network Simplex. In *Proc. First Annu. ACM-SIAM Symp. Discrete Algorithms SODA*, page 367–376. SIAM, 1990.

[Sch03] A. Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*. Number 24 in Algorithms and Combinatorics. Springer, Berlin ; New York, 2003.

[Vaz24] Vijay V. Vazirani. A Theory of Alternating Paths and Blossoms from the Perspective of Minimum Length. *Math. Oper. Res.*, 49(3):2009–2047, 2024. doi:10.1287/MOOR.2020.0388.

[vMN23] Floris van Doorn, Patrick Massot, and Oliver Nash. Formalising the h-Principle and Sphere Eversion. In *The 12th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, New York, NY, USA, January 2023. doi:10.1145/3573105.3575688.

[Vyg02] Jens Vygen. On Dual Minimum Cost Flow Algorithms. *Mathematical Methods of Operations Research*, 56(1):101–126, 2002. doi:10.1007/s001860200202.

[Vyg24] Jens Vygen. Communication citation, March 2024.

[WHN18] Simon Wimmer, Shuwei Hu, and Tobias Nipkow. Verified Memoization and Dynamic Programming. In *Interactive Theorem Proving*, Cham, 2018. doi:10.1007/978-3-319-94821-8_34.

[Wir71] Niklaus Wirth. Program Development by Stepwise Refinement. *Commun. ACM*, 14(4):221–227, 1971. doi:10.1145/362575.362577.

This work is licensed under the Creative Commons Attribution License. To view a copy of this license, visit https://creativecommons.org/licenses/by/4.0/ or send a letter to Creative Commons, 171 Second St, Suite 300, San Francisco, CA 94105, USA, or Eisenacher Strasse 2, 10777 Berlin, Germany