The Feasibility of Implementing Maximum Flow and Minimum-Cost Flow in Almost-Linear Time with an IPM

Adrian Valdemar Borup
adbo@itu.dk

Albert Rise Nielsen
albn@itu.dk

Supervisors:

Riko Jacob
rikj@itu.dk

Thore Husfeldt
thore@itu.dk

STADS code: KIREPRO1PE

Examination Group: V24KIREPRO1PE684

Research Project

IT University of Copenhagen

M.Sc. Computer Science

December 16, 2024

Abstract

A breakthrough algorithm for maximum flow and minimum-cost flow that runs in $m^{1+o(1)}$ time was presented in 2022 by Chen, Kyng, Liu, Peng, Gutenberg, and Sachdeva. However, the development is purely theoretical, and there has not yet been an attempt to implement it. The algorithm consists of two parts: 1) an interior-point method (IPM) to iteratively solve min-cost flow, and 2) a data structure to make each iteration efficient. We provide the first-known (static) implementation of their IPM that uses an inefficient oracle for solving the subproblem at each iteration. We evaluate the algorithm on its practicality, performance, and applicability, holding it up to the claims from the original paper. We conclude that it is currently infeasible to implement algorithms that use an IPM to solve flow problems, as it is too impractical, runs too slowly, and has issues that conventional flow algorithms do not encounter.

Contents

1 Introduction

In 2022, Chen, Kyng, Liu, Peng, Gutenberg, and Sachdeva (henceforth [CKLPGS22]) presented a breakthrough in the maximum flow and minimum-cost flow field by introducing a new algorithm that solves both in $m^{1+o(1)}$ time. Their algorithm is based on two key components:

1. An interior-point method (IPM) that reduces minimum-cost flow problems to a sequence of $m^{1+o(1)}$ undirected minimum-ratio cycle subproblems.
2. A dynamic graph data structure that solves these subproblems efficiently.

However, [CKLPGS22] is a purely theoretical paper, and we are not aware of any existing attempts to implement the algorithm. The goal of this research project is to answer the following research question:

"What are the most important components of the algorithm given by Chen, Kyng, Liu, Peng, Gutenberg, and Sachdeva for computing maximum flows and minimum-cost flows in almost-linear time, what are the challenges of a potential implementation, and how feasible is it to implement the algorithm so that it can be used in practice?" [1]

This project serves as an initial exploration and analysis, in which we focus on using the IPM for solving flow problems. The IPM forms the foundation of the algorithm and is responsible for iteratively solving the flow problem, while the data structure primarily makes each iteration computationally efficient. Understanding the IPM and its properties on its own is essential. If the IPM isn't useful in practice, then there is little point in implementing the data structure to make the IPM efficient—especially since the data structure is complex and requires much effort to implement.

As such, our main contributions include:

1. A detailed theoretical overview of the IPM. We provide explanations of equations and concepts from the paper that the authors skip over, examples, corrections of errors, and formulations of missing algorithmic components. Thus making the results of [CKLPGS22] more accessible.
2. An implementation of the IPM, using an oracle for solving the subproblem; the oracle may be inefficient.
3. An evaluation of the feasibility and practicality of using an IPM to solve flow problems.
4. Provide a scope for a master's thesis, whose goal is to implement the complete algorithm or related work.

1.1 Structure of the paper

The rest of this paper will be structured as follows:

Section 2 provides a background for minimum-cost flow problems, different approaches to solve them, interior-point methods, and minimum-ratio cycles. This includes a historical overview within the research fields.

Section 3 goes into depth with the theoretical aspects of the IPM, focusing on equations, concepts, and algorithms that are important to understand for an implementation. We also give a short, high-level overview of the dynamic graph data structure.

Section 4 covers existing solutions and alternatives to solve parts of the algorithm. We look at using linear program solvers for solving the IPM and implementation candidates for the minimum-ratio cycle oracle.

[1]The research question has been altered from the accepted problem statement by replacing "Chen et al." with the full set of authors.

Section 5 presents our implementation of the IPM and states the differences between our implementation and the original theoretical algorithm from [CKLPGS22]. We also cover the implementation of an oracle for finding minimum-ratio cycles.

Section 6 evaluates the implementation by analysing the correctness and running time of it. We design a test suite, create experiments, and present our experimental results alongside their implications. We also evaluate the feasibility of using the IPM for solving flow problems in practice.

Section 7 provides a potential scope for a master's thesis based on the previous sections.

2 Background

In this section, we provide a background for minimum-cost flow problems and a surface level overview of different approaches to solving said problem. We then provide a historical overview of continuous optimisation and interior-point methods, as well as minimum-ratio cycles. These are fundamental ideas and concepts that are required to understand the algorithm.

2.1 Minimum-cost flow

In this section, for the sake of understanding, we provide a high-level overview of the minimum-cost flow problem. In Section 3, we define it formally.

The minimum-cost flow problem can informally be stated as "the cheapest possible way of sending flow, within some capacities, through a flow network". An application for this problem could be finding the cheapest transport of goods through a network where every edge, such as a road or shipping lane, has some cost and capacity.

Minimum-cost flow instances can more formally be defined as a directed flow graph, where each edge has a lower and upper capacity along with a cost. Flow must then be sent through the graph in a way that satisfies edge constraints and minimises the total cost. Each edge contributes one unit of its cost per unit of flow that it sends.

Take for example the graph in Figure 1. This example shows a graph where some edges must have some flow, as defined by the lower capacities.

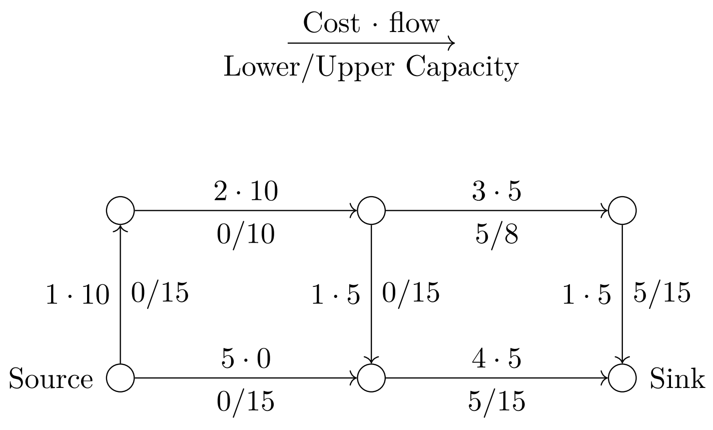


Figure 1: Example of minimum-cost flow with upper and lower capacities, costs, and an optimal solution. In this example, the minimum-cost flow that upholds the requirements routes 10 units of flow to the sink, with a total cost of 75.

Some problem instances also define a demand on vertices. This could, for example, represent the amount of goods that must be delivered to a certain vertex.

In this instance type, each vertex with a negative demand is a source, and each vertex with a positive demand is a sink. The sum of demands over the entire graph must be 0 in order for the problem to be solvable. This differentiates itself from the previous type of minimum-cost flow instance in that the latter only has a single source and sink.

The version of the problem with demands is also known as the transportation problem. Figure 2 contains an example of the transportation problem in a similarly structured graph as before.

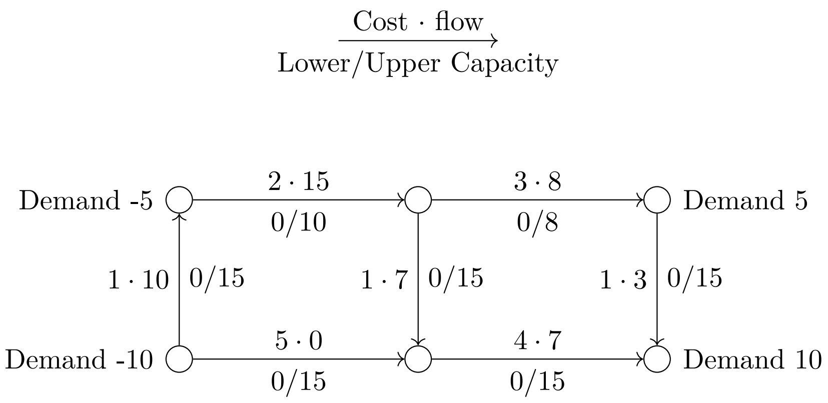


Figure 2: Example of the transportation problem with upper and lower capacities, costs, demands, and an optimal solution. The minimum-cost flow that upholds the requirements has a total cost of 102.

The minimum-cost flow problem is a fundamental graph problem, and many other problems can be reduced to minimum-cost flow. As such, efficient algorithms for minimum-cost flow may enable efficient solutions to many other problems. One such example is the maximum flow problem.

2.2 Solving maximum flow with minimum-cost flow

Utilising minimum-cost flow algorithms to solve maximum flow problems becomes trivial by turning the maximisation problem into a minimisation problem through a negative-cost edge.

First, assign a cost of 0 to all edges in the max flow instance. Then, create a new directed edge from the sink to the source with infinite capacity and a cost of -1.

Minimum-cost flow algorithms will minimise the overall cost of flow sent through the graph. Since the only edge with a cost in the graph has cost -1, an optimal solution will push the maximum possible amount of flow through that edge to minimise the cost.

In order for flow to originate from the source and pass through the new edge, it must to be routed through the entire original graph to the sink. Thus, the resulting flow on the new edge is the maximum possible flow through the input graph.

If capacities must be real numbers, it suffices to set the capacity of the new edge to the sum of outgoing capacities from the source, as that is the theoretical maximum flow in a network.

2.3 Approaches to solving minimum-cost flows

Minimum-cost flow has, since its conception, been solved using various linear programming methods. Though, many algorithmic paradigms to solve the problem have emerged. We provide a non-exhaustive overview here:

Cycle cancelling First introduced by Klein, cycle cancelling methods are fundamentally based on the idea that if you can route flow around a cycle, yielding a negative total cost, the minimum-cost flow has yet to be found [Kle67]. In this way, the minimum-cost flow problem is closely related to the negative-cycle problem. Klein's method is exponential, but Goldberg and Tarjan later proved that it can be improved to run in polynomial time by always selecting the cycle with the minimum average cost, known as the minimum-mean cycle [GT89].

Successive shortest path Also known as augmenting paths, successive shortest paths are possibly the most well known solution. The successive shortest path (SSP) method repeatedly finds the minimum-cost shortest path in a residual graph from s to t and routes flow along it. A residual graph is a graph where each edge has also been routed in the reversed direction, with its cost negated. The residual graph allows an "undo" action if the routed flow is not an optimal solution. Without the residual edges there might be "blocking flow", which is a routed flow that blocks the optimal solution from being found. There are many variations of SSP, especially as shortest path algorithm research evolves and introduces new shortest-path algorithms. The transportation problem can also be solved via SSP by modifying the algorithm such that it selects the minimum-cost path between a vertex with excess flow and a vertex with deficits. SSP is the minimum-cost flow analogy to the Ford-Fulkerson algorithm for maximum flow, which is based on the same idea but can use simpler path algorithms due to not having to account for cost. Much like Ford-Fulkerson for maximum flow, SSP is generally the simplest minimum-cost algorithm to implement.

Capacity Scaling Capacity scaling was devised to address a large potential inefficiency of the successive shortest path algorithm: the routed flows may be very small. This, in the worst case, causes a large number of edge updates. Capacity scaling functions much like standard SSP algorithms, with the constraint that it only selects paths with capacity at least x. When no such path is found, x is reduced (for example by halving x) and the algorithm repeats. Thereby ensuring the initially-routed flow is large. The algorithm is applied very similarly to solve the transportation problem, with the added constraints that the source has to be a vertex with demand satisfying $\geq x$, and the sink is a vertex with a demand satisfying $\leq -x$.

Push-relabel (Preflow-push) Push-relabel is a maximum flow algorithm, but we include it here as it aids in explaining cost scaling. Push-relabel solves a different inefficiency of Ford-Fulkerson: namely, it can happen that Ford-Fulkerson finds an augmenting path that updates *all* edges in the graph on every iteration, which, in the worst case, also only routes 1 unit of flow per iteration. Push-relabel addresses this with the concept of preflow, which routes flow equal to an edge's capacity, on every edge incident to the source vertex. This is intended to break flow conservation, as the next step is to route the excess flow to neighbouring vertices by selecting a vertex with excess flow. Push-relabel then places a label, also known as the height, on each vertex. The label governs which other vertices a vertex can push flow to, as a vertex can only push to vertices with a smaller label. As a visual example, imagine a hill with the source on top, then every other vertex is somewhere further down the hill. If some flow is stuck at a vertex, because there are no vertices with a smaller label that can accept flow, the algorithm moves it further up the hill by increasing its label. This is known as relabelling. On each iteration, the algorithm picks a vertex with excess flow and checks if there exists an edge to a vertex with a smaller label that it can push some or all of the excess through. If such an edge does not exist, it increases the label of the vertex. Repeat until there are no more vertices with excess. This is all done on a residual graph such that flow can be pushed back to previous vertices when required.

Cost Scaling The cost scaling algorithm defines the concept of ϵ-optimality, which is the maximum factor of which the costs are allowed to be violated. The algorithm first finds a feasible flow, and sets ϵ equal to the sum of costs in the network, by this, the feasible flow is ϵ-optimal. In each iteration, the algorithm improves the network by running the push-relabel algorithm, with the extra condition that edges and vertices can only be selected if they uphold the ϵ-optimality clauses (the exact clauses have been omitted for brevity), the rest of the push-relabel algorithm is left the same. Then the algorithm halves ϵ and repeats. For simplicity, one can think of the Cost Scaling algorithm as a Capacity Scaling algorithm, which uses cost instead of capacity, and

Push-relabel instead of augmenting paths. The algorithm is repeated until $\epsilon < 1/n$, as, the flow can be proved to be optimal when this condition is met. For clarity, take the visual example from before, but imagine that the pipes running down the hill costs something to use. We want to pay to use as few as possible, so we also move the nodes up the hill such that the flow delivered satisfies lower demands, but uses as little of the budget as possible.

Linear Programming The flow problem is expressed as a series of linear programming constraints and an objective. In theory, any linear programming solver can than be used to find an optimal solution. For example, network simplex [Dan51] and interior-point methods [Kar84] are both applicable. This paradigm is what led to the breakthrough we discuss in this project, and we explore it further in the next section.

2.4 Continuous optimisation and interior-point methods

This section outlines the history of using continuous optimisation to solve flow problems, particularly via interior-point methods. We first explain interior-point methods in general in Section 2.4.1 and then give an overview of previous work for solving flow problems using IPMs in Section 2.4.3.

2.4.1 Interior-point methods

Interior-point methods are ways to optimise linear programming problems. Linear programming problems are encoded as a set of constraints and an objective function, which [Kar84] expresses as:

$$\begin{aligned} \text{minimize} \quad & \boldsymbol{c}^\top \boldsymbol{x}, \qquad \boldsymbol{c}, \boldsymbol{x} \in \mathbb{R} \\ \text{subject to} \quad & A\boldsymbol{x} = \boldsymbol{b}, \quad \boldsymbol{x} \geq 0 \end{aligned} \tag{1}$$

where $\boldsymbol{c}^\top \boldsymbol{x}$ is the objective function and $A\boldsymbol{x} = \boldsymbol{b}$ is some way to encode the constraints. Any solution $\boldsymbol{x}$ that doesn't violate the constraints is called valid or feasible. The goal, then, is to find a feasible solution that minimises the objective function. Such a solution is called optimal.

Central to the understanding of interior-point methods is the *feasible region*: the solution space of all valid solutions to the problem. The problem constraints can be viewed as hyperplanes in the solution space. The feasible region is then the space bounded by these hyperplanes. An optimal solution will lie on the boundary—specifically at one of the corner points of the feasible region, which is illustrated on Figure 3.

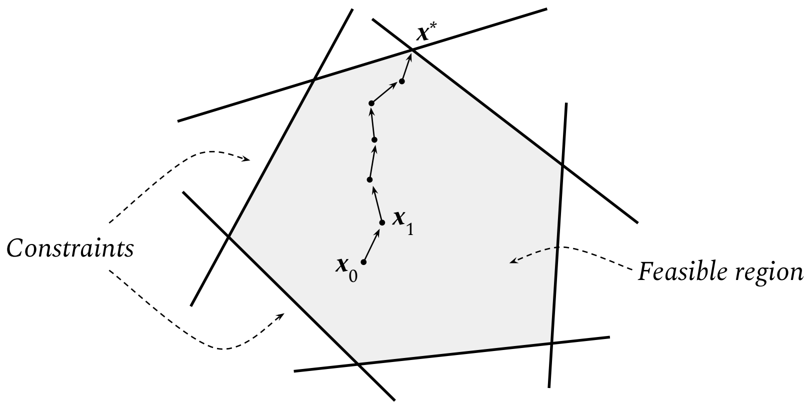


Figure 3: An example of a two-dimensional linear programming problem being solved with an interior-point method. The initial point is given at $\boldsymbol{x}_0$ and an optimal solution is given as $\boldsymbol{x}^*$.

Generally, IPMs consist of two phases:

1. Find an initial point, i.e. *some* point strictly within the interior of the feasible region.

2. Iteratively refine the feasible solution into an optimal solution.

Here, it's important to note that the interior of the feasible region does *not* include the boundary itself. The point has to be strictly inside the feasible region.

To iteratively refine the initial point, an IPM solves a sequence of subproblems, each yielding a point that decreases the objective function. Once the refined solution is sufficiently close to the optimal solution, the IPM terminates.

Note that IPMs work exclusively within the interior of the feasible region and will thus never reach the exact optimal solution, since it will be *on* the boundary. To obtain the optimal solution, the feasible solution can, for example, be rounded.

Multiple classes of IPMs exist, including path-following methods, potential-reduction methods, and primal-dual methods. [CKLPGS22] utilises a potential-reduction method, so the next section will focus exclusively on potential-reduction interior-point methods.

Linear programming problems can also be solved by the simplex method and the ellipsoid method. Interior-point methods differ from those in that the ellipsoid method bounds the feasible region from the outside, the simplex method traverses the boundary itself, and interior-point methods work from within the interior (hence the name).

2.4.2 Potential-reduction interior-point methods

The potential-reduction interior-point method, introduced by Karmarkar [Kar84], is a subclass of IPMs which solve linear programming problems in polynomial time. In fact, Karmarkar's paper is the one that initially introduced IPMs. Here, we give a high-level overview of it without delving too deeply into the underlying mathematical reasoning.

The idea of the potential-reduction method is that, instead of just minimising the objective $\boldsymbol{c}^\top \boldsymbol{x}$, you minimise a *potential function*, which measures the progress of the IPM. A potential function consists of two parts:

An objective. The function that should be minimised in order to find an optimal solution, e.g. $\boldsymbol{c}^\top \boldsymbol{x}$ from the previous section.

A barrier. A penalty function which ensures that the problem's constraints are not violated. This, for example, is a function that increases rapidly once a feasible solution gets close to a constraint, thus penalising that step direction.

Given an objective $\boldsymbol{c}^\top \boldsymbol{x}$, the potential function typically takes the following form:[2]

$$f(\boldsymbol{x}) = n \log(\boldsymbol{c}^\top \boldsymbol{x}) + \sum_{j}^{n} -\log \boldsymbol{x}_j \tag{2}$$

where n is the number of dimensions of $\boldsymbol{x}$. Here, the left side $n\log(\boldsymbol{c}^\top \boldsymbol{x})$ is an expression for the objective, and the right side $-\log \boldsymbol{x}_j$ is an expression of a barrier. The idea is that the potential $f(\boldsymbol{x})$ will decrease as the objective $\boldsymbol{c}^\top \boldsymbol{x}$ decreases, except if $\boldsymbol{x}_j$ gets too close to 0. In that case, $-\log \boldsymbol{x}_j$ will "explode" towards ∞ and rapidly increase the potential.

In the above, it is assumed that the optimal solution has $\boldsymbol{c}^\top \boldsymbol{x} = 0$. However, if the objective minimum is non-zero, you can instead use the objective $\boldsymbol{c}^\top \boldsymbol{x} - X^*$ where X^* is the minimum objective value. This also has the implication that the minimum value *must* be known in advance. If this is not the case, Karmarkar describes a "sliding objective function" variant of the algorithm, which can be used. This becomes relevant in Section 3.3.5.

For this to run in polynomial time, one simply has to prove that each iteration decreases the potential by at least some constant.

[2]The original expression of the potential function by Karmarkar is different, but modern algorithms typically write it in the way presented here. We show how to rearrange Karmarkar's version to the modern version in Appendix A.

In the context of minimum-cost flow, minimising the objective function would correspond to reducing the cost of a given flow, and the barrier function would prevent the flow from exceeding or receding edge capacities.

2.4.3 Previous work for solving flow problems with IPMs

In 1984, Karmarkar presented interior-point methods, which allowed for solving linear programming problems in polynomial time. Before [CKLPGS22], there has been substantial previous work in using interior-point methods to solve flow problems. In this section, we cover some of this work.

In 1992, Wallacher and Zimmermann solved network flow problems using an IPM. Their algorithm is based on cycle-cancelling, and each IPM subproblem is to find a minimum-ratio cycle, which can be done in polynomial time as per [GT89].

[ST04] came up with a new framework for flows based on continuous optimisation, in which they solve "electrical flow" in linear-time [BBST24]. Building on top of [ST04], [DS08] then reduced maximum flow to electrical flow by use of an IPM. This work standardised IPMs that solve electrical flow subproblems and motivated further research into IPMs for solving flow problems. We say that IPMs based on electrical flow are ℓ_2-norm minimisation problems.[3] Later research has focused on converging the IPM faster (thus minimising iterations) [Mad13; LS14; LS19] and using electrical flows within the augmenting paths framework [Mad16].

Since then, focus has shifted from reducing the number of iterations to minimising the cost per iteration, especially by utilising dynamic data structures to make efficient sparsifiers [DGGP19; BBGNSSS20; CGHPS20].

All this led to the work of [CKLPGS22], which we focus on in this project, that solves maximum flow and minimum-cost flow in $m^{1+o(1)}$ time by using an IPM.

[CKLPGS22] diverges from the traditional research on IPMs for flow problems: whereas an ℓ_2-based IPM with electrical flow subproblems is typically used for solving flow problems, [CKLPGS22] uses an ℓ_1 minimisation problem.

By [Kar84], a potential-reduction IPM using a suitable norm converges in $\widetilde{O}(m)$ iterations. Typically, the suitable norm would be the ℓ_2 norm, which is preferred to the ℓ_1 norm as it allows larger step sizes due to penalising large deviations harder than the ℓ_1 norm.

Later work by [Ren88] proved that $\widetilde{O}(\sqrt{m})$ iterations sufficed to give a high-accuracy solution in path-following IPMs with an ℓ_2 norm.

Despite this [CKLPGS22, p. 6] claim that they have been able to argue that the ℓ_1 norm can be used to solve $\widetilde{O}(m)$ subproblems and still achieve a high-accuracy solution, such that using an ℓ_1 norm instead of an ℓ_2 norm does not increase the number of iterations in a potential-reduction IPM. Using an ℓ_1 norm is a crucial advantage for the algorithm, as problems of the ℓ_1 norm must have optimal solutions in the form of simple cycles, which their data structure can efficiently compute (covered in Section 3.5).

In other words, despite having to solve $\widetilde{O}(m)$ subproblems, [CKLPGS22] solves each ℓ_1-based subproblem so efficiently that it outperforms a path-following IPM that solves $\widetilde{O}(\sqrt{m})$ ℓ_2-based subproblems.

2.5 Min-ratio cycles

[CKLPGS22] introduces a dynamic data structure to retrieve minimum cost-to-time ratio cycles (in short, min-ratio cycles) in amortized $m^{o(1)}$ time. In this section, we revisit the minimum ratio cycle problem and its history.

The problem, which is used extensively in combinatorial optimization, was first introduced by Dantzig, Blattner, and Rao [DBR66] and later by Lawler [Law66]. Informally, the problem associates each edge (u, v) in a graph G with two numbers: the "cost" $c_{u,v}$ and the "time" $t_{u,v}$ per unit of flow. The goal of the problem is then to find a cycle in G that has the minimum ratio of total cost to time.

[3]This simply refers to the fact that the problem being solved by electrical flow minimises the ℓ_2 norm of a vector.

Formally, let $\overline{x}_{u,v} = 1$ if (u, v) is an edge in some cycle and $\overline{x}_{u,v} = 0$ otherwise. Then let the total "time" be $T = \sum_{(u,v)\in E(G)} t_{u,v}\overline{x}_{u,v}$ similarly, let $C = \sum_{(u,v)\in E(G)} c_{u,v}\overline{x}_{u,v}$ be the total cost, then the ratio to minimize is C/T. Bringmann, Hansen, and Krinninger [BHK17] and Lawler [Law76] represent the problem more succinctly with C being a cycle consisting of a set of edges, then the minimized function is $q(C) = \sum_{e\in C} c(e)/\sum_{e\in C} t(e)$ where c is a function of the cost on the edge, and t is a function of the time on the edge.

There have been 3 types of algorithms proposed as solutions to this problem: weakly- and strongly-polynomial algorithms as well as pseudopolynomial running time algorithms. In his 1976 book, Lawler introduces a binary search algorithm for finding min-ratio cycles that performs $O(\log(nW))$ calls to a negative cycle detection algorithm [Law76], where $W = O(CT)$ when the costs are given as integers from $1 \ldots C$ and the time is given as $1 \ldots T$. Lawler's binary search algorithm runs in a combined $O(n^3 \log(nW))$ time, where n^3 stems from the negative cycle detection algorithm. Chatterjee, Ibsen-Jensen, and Pavlogiannis has found that one can bound the number of negative cycle detection calls to $O(\log(|a\cdot b|))$ when the value of the resulting minimum ratio cycle is $\frac{a}{b}$ [CIP15]. The family of weakly polynomial algorithms bases itself on this idea by Lawler, but improves the running time by replacing the negative cycle detection algorithm. The following is a list of such replacement algorithms and their running times, each of these running times must be multiplied by either $\log(nW)$ or $\log(|a\cdot b|)$:

- A variant of the Bellman-Ford algorithm runs in $O(mn)$ [BHK17] time.
- The all-pairs shortest paths algorithm by Chan and Williams runs in $n^3/2^{\Omega(\sqrt{\log n})}$ [CW15].
- The all-pairs shortest paths algorithm in matrix-multiplication-time by Sankowski runs in $\widetilde{O}(n^{\omega}W)$ [San05] where $\omega \leq 2.371552$ [WXXZ23].
- Goldberg's scaling algorithm, runs in $O(\sqrt{n}m \log W)$ [Gol95].
- The shortest paths algorithm based on interior-point methods by Cohen, Mądry, Sankowski, and Vladu runs in $\widetilde{O}(m^{10/7} \log W)$ [CMSV17].
- Karczmarz and Sankowski all-pairs shortest paths algorithm which in $\tilde{O}(nm/p)$ parallel time, assuming $p = \widetilde{O}(n^{1/3}m^{4/3})$ processors [KS21].

As for the group of strongly polynomial algorithms for finding the minimum-ratio cycle, we are aware of the following:

- Megiddo's algorithm which runs in $O(m \cdot n^2 \cdot \log n)$ [Meg79] and Megiddo's improvement on said algorithm which runs in $O(n^3 \log n + nm(\log n)^2 \log\log n)$ [Meg83] which can be further reduced to $O(n^3 \log n + nm(\log n)^2)$ due to [Col87].
- Burnss primal-dual algorithm, which runs in $O(mn^2)$ [Bur91]
- Bringmann, Hansen, and Krinninger introduced an algorithm which runs in $n^3/2^{\Omega(\sqrt{\log n})}$, and is the fastest currently known strongly polynomial algorithm [BHK17].

As for the last group of algorithms with a pseudopolynomial running time bound, the current fastest algorithm is by Hartmann and Orlin. This algorithm has a running time of $O(T(m + n \log n))$ which is dominated by the $O(T)$ shortest path calculations [HO93]. Other work has been done in this group, but no work has been completed that claims a better runtime than Hartmann and Orlin [Fox69; Gol82; BHK17]. This includes Howard's algorithm [CCGMQ98], which Dasdan, Irani, and Gupta improved to have a running time of $O(nm\alpha)$ with α being the number of cycles in the graph [DIG99].

Dasdan, Irani, and Gupta performed experiments that indicated Howard's algorithms to be the fastest algorithm in practice [DIG99]. The experiments compared the running times of a number of algorithms, including [Law76], [Bur91], Howard's algorithm, and [Kar78], along with Young, Tarjan, and Orlin's improvement on it [YTO91].

The latter two are both minimum-mean cycle algorithms. The minimum-mean cycle problem is a special case of the min-ratio cycle problem, where the algorithms for solving the minimum-mean cycle problem can also be used to solve the min-ratio problem, and vice versa. While Dasdan, Irani, and Gupta concluded Howard's algorithm to be fastest in 1999, Dasdan recreated the experiment with more parameters, larger graphs, and on newer hardware in 2004 [Das04]. In this experiment, the conclusion was that improvements in hardware and testing methods has improved such that the [YTO91] algorithm is faster in practice. However, the study also introduces parameters such as simplicity and robustness. In Section 5.2, we describe that we implement Howard's algorithm as it is nearly as fast as [YTO91] algorithm while being simpler in its data structure and pseudocode.

3 Overview of the algorithm

First, we define the minimum-cost flow problem instance. To reduce confusion, we use the same terminology and symbols as [CKLPGS22]. The instance is given as a directed graph $G = (V, E)$ with $|V| = n$ and $|E| = m$, upper and lower edge capacities $\boldsymbol{u}^+, \boldsymbol{u}^- \in \mathbb{R}^E$, edge costs $\boldsymbol{c} \in \mathbb{R}^E$, and vertex demands $\boldsymbol{d} \in \mathbb{R}^V$ that satisfy $\sum_{v \in V} \boldsymbol{d}_v = 0$.

The goal is to find a flow $\boldsymbol{f} \in \mathbb{R}^E$ that minimises the total cost, expressed as $\boldsymbol{c}^\top \boldsymbol{f}$, while respecting edge capacities ($\boldsymbol{u}^-_e \leq \boldsymbol{f}_e \leq \boldsymbol{u}^+_e$ for all $e \in E$) and vertex demands. To express the vertex demand constraint succinctly, [CKLPGS22] defines the edge-vertex incidence matrix $\mathbf{B} \in \mathbb{R}^{E \times V}$ as:

$$\mathbf{B}_{((a,b),v)} = \begin{cases} 1 & \text{if } a = v \text{ (edge exits } v\text{)} \\ -1 & \text{if } b = v \text{ (edge enters } v\text{)} \\ 0 & \text{otherwise} \end{cases}$$

The vertex demand constraint is then expressed as $\mathbf{B}^\top \boldsymbol{f} = \boldsymbol{d}$, meaning each vertex is left with exactly the amount of flow units that it requires.

Additionally, let $U \in \mathbb{R}$ bound the capacities such that $|\boldsymbol{u}^+_e|, |\boldsymbol{u}^-_e| \leq U$ and let $C \in \mathbb{R}$ bound the costs such that $|\boldsymbol{c}_e| \leq C$ for all $e \in E$.

For analysis, all upper/lower capacities, costs, and demands are assumed to be integral. However, the algorithm still works for non-integral values without loss of generality (see Section 3.2).

3.1 Key components

[CKLPGS22] has two primary contributions:

The first is a potential-reduction interior-point method, which reduces the minimum-cost flow problem to a sequence of $m^{1+o(1)}$ slowly-changing undirected minimum-ratio cycle subproblems. The cycle ratio is defined in terms of partial derivatives of the potential function, and thus the newly routed flow decreases the potential. At each iteration, the cycle is used to route flow in the graph, thus minimising the cost. The potential function itself is broken down in detail in Section 3.3.1.

The second contribution is a dynamic recursive and randomized data structure designed to efficiently compute an approximate minimum-ratio cycle in an unweighted, undirected graph, which undergoes edge updates over the course of the algorithm. Such updates include insertions/deletions and vertex splits. It does so by maintaining a sparse subgraph that embeds the original graph using short paths, called a spanner.

3.2 Assumptions

Before we cover the algorithm, we outline our assumptions:

Model of Computation. [CKLPGS22] claims that the algorithm works for fixed-point arithmetic. Specifically, they write: *for problem instances encoded with z bits, all algorithms work in fixed-point arithmetic where words have $O(\log^{O(1)} z)$ bits.* We cannot claim to fully understand what they mean by "problem instances encoded with z bits." We gather that the algorithm is designed to use fixed-point arithmetic on numbers that are bounded by the input size. The size and precision of the numbers grow slowly according to a polylogarithmic function of the total number of bits required to store the input. Arbitrary precision thus isn't required. We refer to Appendix C for our informal reasoning.

Basic arithmetic operations. We assume that foundational arithmetic operations run in constant time. These operations include: addition, subtraction, multiplication, division, exponentiation, and mathematical functions such as log.

Linear-algebraic operations. The algorithm makes use of linear-algebraic operations, such as the dot product between vectors, entry-wise multiplication, vector scaling, and ℓ_1 norms. Usually, these run in linear-time with respect to the number of dimensions in the vector.

In this algorithm, they are used to find and route flow along min-ratio cycles. In [CKLPGS22], the authors claim that each min-ratio cycle subproblem can be "computed and processed in amortized $m^{o(1)}$ time." During our research, we only scratched the surface of the theory behind the dynamic graph data structure—therefore, we simply accept this and assume that processing min-ratio cycle subproblems with their data structure takes amortized $m^{o(1)}$ time.

Specifically, the authors state in their Informal Theorem 1.5 that their data structure supports the following operations with high probability in amortized $m^{o(1)}$ time:[4]

1. *Return an $m^{o(1)}$-approximate min-ratio cycle.*
2. *Route a circulation along such a cycle.*
3. *Insert/delete edge e, or update $\mathbf{g}_e$ and $\boldsymbol{\ell}_e$.*
4. *Identify edges that have accumulated significant flow.*

The data structure builds a recursive hierarchy of graphs, where each step down in the hierarchy contains fewer vertices and edges—our assumption is that this "sparsification" results in the properties mentioned above. In other words, while the linear-algebraic operations run in linear time, we assume that the data structure only uses them with amortized $m^{o(1)}$ number of elements. Thus, each use of a linear-algebraic operation takes amortized $m^{o(1)}$ time.

3.3 IPM

The idea behind the potential-reduction interior-point method is to start from some initial feasible flow and use it to compute a sequence of flows, each of which decrease the potential.

To find a flow that decreases the potential, each iteration computes an approximate minimum-ratio cycle. The cycles minimise a ratio of gradients and lengths, which are computed based on the current flow, cost, and capacities in a way that causes good steps with the IPM while respecting capacity constraints.

After an almost-linear number of iterations, the potential function has been reduced to such a point that the final flow can be rounded to the optimal solution.

In the next sections, we first break down the potential function into its components. We then cover the min-ratio cycle subproblem, after which we explain how to combine the elements into a complete IPM that computes the minimum-cost flow.

[4]In order to outline the assumptions, this list refers to concepts and symbols that we define later in this section.

3.3.1 Breaking down the potential function

The algorithm by [CKLPGS22] is a potential-reduction interior-point method, in which the potential function is defined as

$$\Phi(\boldsymbol{f}) = \underbrace{20m\log(\boldsymbol{c}^\top \boldsymbol{f} - F^*)}_{\text{Objective}} + \underbrace{\sum_{e\in E}\left((\boldsymbol{u}_e^+ - \boldsymbol{f}_e)^{-\alpha} + (\boldsymbol{f}_e - \boldsymbol{u}_e^-)^{-\alpha}\right)}_{\text{Barrier}} \tag{3}$$

for a given flow $\boldsymbol{f}$ and $\alpha = 1/(1000\log mU)$. Starting with some feasible flow and iteratively minimising the potential function will, in the end, yield an approximately optimal solution to the minimum-cost flow problem. An exact solution can be obtained by rounding the approximately optimal solution.

The potential function balances the objective function (minimising the cost) and the barrier function that upholds the linear programming constraints (namely respecting $\boldsymbol{u}_e^- \le \boldsymbol{f}_e \le \boldsymbol{u}_e^+$ for all $e \in E$).

In the following sections, we deconstruct Equation 3, describing what its parts represent and how those parts balance objective and barrier function.

The potential function consists of the following internal parts:

$\boldsymbol{f}$ denotes the current flow.

$\boldsymbol{c}^\top \boldsymbol{f}$ denotes the cost of that flow. This is the metric we aim to minimise for minimum-cost flow.

F^* denotes the value of the optimal cost. Equation 3 operates under the assumption that F^* is known—a critical assumption that we explore further in Section 3.3.5.

$\boldsymbol{u}_e^+ - \boldsymbol{f}_e$ denotes how much capacity is left in edge e given the current flow.

$\boldsymbol{f}_e - \boldsymbol{u}_e^-$ denotes by how much the flow in an edge e exceeds the minimum capacity of that edge.

Note that, by definition of IPM, neither $\boldsymbol{f}_e - \boldsymbol{u}_e^-$ nor $\boldsymbol{u}_e^+ - \boldsymbol{f}_e$ may become negative, as that would violate the optimisation problem constraints ($\boldsymbol{f}$ would not be a point within the interior of the feasible region). In fact, $\boldsymbol{u}_e^+ - \boldsymbol{f}_e > 0$ and $\boldsymbol{f}_e - \boldsymbol{u}_e^- > 0$ must also hold, since solutions with saturated or empty edges lie on the boundary of the feasible region, not within its interior.

3.3.2 Balancing the objective and the barrier

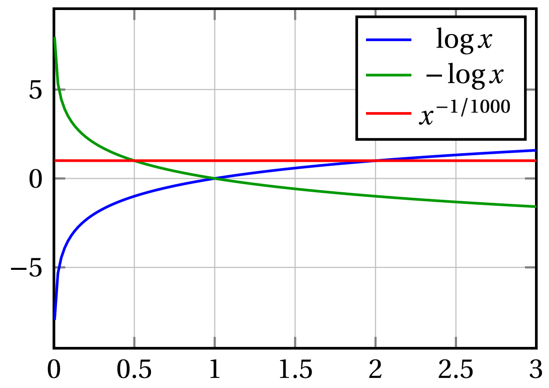
In this section, we describe how the objective function and barrier function impact the potential-reduction process—particularly by examining the growth rates of logarithmic functions and power functions.

The objective $\boldsymbol{c}^\top \boldsymbol{f} - F^*$ is scaled via the log function. As such, the value of the objective function decreases rapidly when the cost approaches the optimal cost, but otherwise decreases slowly as the cost is minimised.

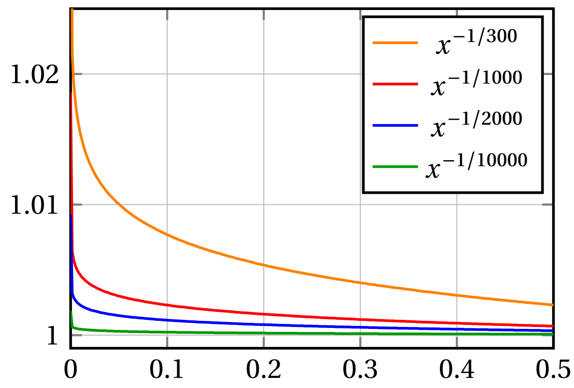
The barrier function utilises a power barrier of the form $x^{-\alpha}$. Because $0 < \alpha < 1$, $x^{-\alpha}$ is a function that decreases as its base grows. It is, according to [CKLPGS22], analogous to the more standard $-\log x$. Its purpose is to ensure that iterations remain within the feasible region by penalising solutions that get close to the boundaries. Additionally, the authors mention that the power barrier is used to guarantee that lengths and gradients calculated throughout the algorithm can be represented using $\widetilde{O}(1)$ bits.

A power barrier penalises solutions that come very close to violating constraints, more aggressively than the standard log barrier used by most other IPMs. The power barrier lets solutions come closer to the constraint before penalising them, and once the solution is "too close," it grows more rapidly than the log barrier, thus heavily discouraging the IPM from stepping further in the direction of the constraint. But when the solution is not close to violating a constraint, that constraint term contributes essentially nothing to the barrier.

To show this more intuitively, Figure 4 shows both a logarithmic barrier and a power barrier. In essence, the barrier function contributes virtually nothing, except for when a constraint is *very* close to being violated. Thus, minimising the potential function primarily means directly minimising the objective function, but never in a direction that gets very close to violating a constraint.



(a) A comparison of three functions: $\log x$, which is used to scale the objective function, $-\log x$, which is the log barrier used by most other IPMs, and the power barrier used by this IPM. Note that constant scaling factors have been omitted, as the growth rate is the focus of the comparison.



(b) Comparison of power barriers with different hypothetical values for α. It is shown that as the number of edges m and the capacity bound U grows, x can approach closer to 0 before the barrier explodes towards infinity.

Figure 4: Comparison of the growth rate of different functions that can be used for the barrier.

Naturally, the power barrier will restrict the flow in an edge from reaching either of its capacities *exactly*. But how much does the barrier restrict the accuracy of the final solution?

Consider the most "restrictive" power barrier: the smallest possible input instance has $m = 2$ and $U = 1$ and would maximise α. For the sake of maximising α, assume the slower-growing $\log_{10}$ rather than the standard $\log_2$. We then get $\alpha = 1/(1000 \log_{10} mU) \approx 1/300$. Even in such a case, the barrier function allows each edge to easily get within 10^{-100} of either of its capacities before exploding towards infinity. For instance, $\left(10^{-100}\right)^{-1/300} \approx 2.15$. The extra penalty of 1.15 compared to the baseline is easily dominated by the objective function.

As the input instance size grows, α decreases and the accuracy is allowed to be higher, as shown on Figure 4b. As such, any instance should be able to obtain at least the above accuracy.

In conclusion, in the final solution, $\boldsymbol{f}_e$ can be very close to $\boldsymbol{u}_e^+$ or $\boldsymbol{u}_e^-$—by far close enough for practical applications.

3.3.3 The min-ratio cycle subproblem

In each iteration of the IPM, we solve a subproblem in order to find a good step direction in the solution space. For this IPM, the subproblem is the undirected min-ratio cycle problem. In this section, we cover what the undirected min-ratio cycle problem is and why its solution provides a good step direction.

First, we establish the notion of edge lengths and gradients. Let $\boldsymbol{\ell} \in \mathbb{R}^E_{>0}$ denote (always-positive) edge lengths, defined as:

$$\boldsymbol{\ell}(\boldsymbol{f})_e = \left(\boldsymbol{u}_e^+ - \boldsymbol{f}_e\right)^{-1-\alpha} + \left(\boldsymbol{f}_e - \boldsymbol{u}_e^-\right)^{-1-\alpha} \tag{4}$$

The length of an edge $\boldsymbol{\ell}_e$, then, is a measure of how constrained an edge is. The first term refers to how close to the upper capacity the flow is, while the second term represents how close to the lower capacity the flow is. As the terms approach 0, due to $-1-\alpha$, they grow much like x^{-1}. In other words, they grow larger the closer to either capacity the flow is. The length is closely related to the barrier and exhibits similar behaviour.

Then, let $\boldsymbol{g} \in \mathbb{R}^E$ denote the (signed) gradient of the potential function, defined as $\boldsymbol{g}(\boldsymbol{f}) = \nabla\Phi(\boldsymbol{f})$, meaning:

$$\begin{aligned}\boldsymbol{g}(\boldsymbol{f})_e &= [\nabla\Phi(\boldsymbol{f})]_e & (5)\\ &= 20m(\boldsymbol{c}^\top \boldsymbol{f} - F^*)^{-1}\boldsymbol{c}_e + \alpha\left(\boldsymbol{u}_e^+ - \boldsymbol{f}_e\right)^{-1-\alpha} - \alpha\left(\boldsymbol{f}_e - \boldsymbol{u}_e^-\right)^{-1-\alpha} & (6)\end{aligned}$$

In other words, the gradient of an edge $\boldsymbol{g}_e$ is the partial derivative of the potential function $\Phi(\boldsymbol{f})$ with respect to the flow in the edge e. Consequently, $\boldsymbol{g}$ represents each edge's contribution to the rate of change of the potential function. For example, $\boldsymbol{g}_e < 0$ would indicate that changing $\boldsymbol{f}_e$ in the direction of the gradient would decrease the potential.

Because $\boldsymbol{g}$ and $\boldsymbol{\ell}$ are defined dependent on the current flow $\boldsymbol{f}$, which changes for each iteration of the IPM, $\boldsymbol{g}$ and $\boldsymbol{\ell}$ also need to be recomputed on each iteration.

Now, we define the undirected min-ratio cycle problem as follows: find a circulation $\boldsymbol{\Delta} \in \mathbb{R}^E$ in G that has the smallest ratio of gradients to lengths. Formally, find:

$$\min_{\mathbf{B}^\top \boldsymbol{\Delta}=0} \frac{\boldsymbol{g}^\top \boldsymbol{\Delta}}{\|\mathbf{L}\boldsymbol{\Delta}\|_1} \tag{7}$$

where $\mathbf{L} \in \mathbb{R}^{E\times E}$ is the diagonal length matrix defined as $\mathbf{L} = \operatorname{diag}(\boldsymbol{\ell})$. The $\mathbf{B}^\top\boldsymbol{\Delta} = 0$ constraint exists to ensure flow conservation.

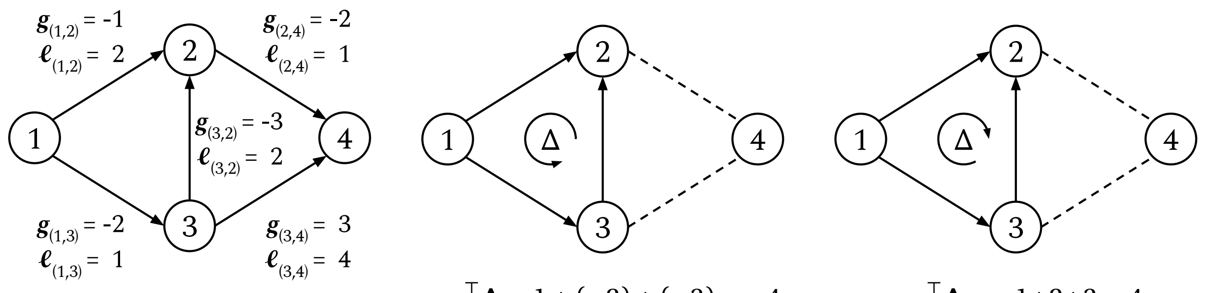
Here, "undirected" means that edges can be traversed in either direction, but the sign of the edge gradient changes depending on the direction traversed. Specifically,

$\boldsymbol{\Delta}_e > 0$ if $\boldsymbol{\Delta}$ sends flow in the same direction as e.

$\boldsymbol{\Delta}_e < 0$ if $\boldsymbol{\Delta}$ sends flow in the opposite direction of e.

$\boldsymbol{\Delta}_e = 0$ if $\boldsymbol{\Delta}$ sends no flow through e.

This means that the minimum ratio is always negative, because if $\boldsymbol{\Delta}$ is a circulation, then $-\boldsymbol{\Delta}$ is also a circulation [CKLPGS22]. Figure 5 demonstrates how to calculate the ratio of an undirected cycle and how the direction of flow in $\boldsymbol{\Delta}$ affects the sign of the ratio.



$\boldsymbol{g}^\top\boldsymbol{\Delta} = 1 + (-2) + (-3) = -4$

$\|\mathbf{L}\boldsymbol{\Delta}\|_1 = 2 + 1 + 2 = 5$

$\boldsymbol{g}^\top\boldsymbol{\Delta} = -1+2+3 = 4$

$\|\mathbf{L}\boldsymbol{\Delta}\|_1 = 2+1+2 = 5$

(a) An example graph with gradients $\boldsymbol{g}_e$ and lengths $\boldsymbol{\ell}_e$ defined for each edge.

(b) $\boldsymbol{\Delta}$ sends 1 unit of flow counterclockwise along the undirected cycle $(1, 3, 2)$.

(c) $\boldsymbol{\Delta}$ sends 1 unit of flow clockwise along the undirected cycle $(1, 2, 3)$.

Figure 5: An example showing how to calculate the ratio of an undirected cycle in a graph. It's shown that the orientation of flow through the cycle affects the sign of the contribution of the edge gradient. The min-ratio cycle of this graph is (1, 3, 2) with ratio $\boldsymbol{g}^\top\boldsymbol{\Delta}/\|\mathbf{L}\boldsymbol{\Delta}\|_1 = -4/5$.

A min-ratio cycle can be assumed to be a simple cycle, as it minimises the ratio. Assume that the min-ratio cycle algorithm returns a non-simple cycle. By definition of non-simple cycle, that cycle is a union of simple cycles, which we call sub-cycles. Then, three different cases can occur:

Case 1: Removing a sub-cycle improves the ratio. If that's the case, the returned cycle isn't a minimum-ratio cycle.

Case 2: Removing a sub-cycle worsens the ratio. Then that sub-cycle must have a smaller ratio than the returned cycle, meaning the returned cycle isn't a minimum-ratio cycle.

The intuition here is that the union of sub-cycles corresponds to the average ratio of the sub-cycles. If the ratio of any sub-cycle is different, one of them must be the minimum and smaller than the average.

Case 3: Removing the sub-cycle keeps the ratio unchanged. In that case, the sub-cycle has the same ratio and would thus also be a valid min-ratio cycle. Therefore any such non-simple min-ratio cycle could be reduced to a simple min-ratio cycle.

Min-ratio cycles provide good step directions in the IPM because, by definition, the ratio they minimise is the direction that drops the potential. As the numerator $\boldsymbol{g}^\top \boldsymbol{\Delta}$ decreases (and becomes more negative), the potential drops further. The denominator, $\|\mathbf{L}\boldsymbol{\Delta}\|_1$, increases as constraints are closer to being violated, thus disallowing the ratio to become very negative.

Another advantage of using circulations is that they preserve flow conservation. Thus, they are useful for maintaining feasibility across IPM iterations.

Note that the ratio of the cycle does not change with respect to the amount of flow routed by $\boldsymbol{\Delta}$. If $\boldsymbol{\Delta}$ is a simple cycle that routes $F \in \mathbb{R}$ units of flow, then $\boldsymbol{\Delta} \in \{-F, 0, F\}^E$—i.e. the absolute value of any $\boldsymbol{\Delta}_e$ is either F or 0. To demonstrate that the ratio does not change with respect to F, the ratio can be rearranged as follows:

$$\frac{\boldsymbol{g}^\top \boldsymbol{\Delta}}{\|\mathbf{L}\boldsymbol{\Delta}\|_1} = \frac{\sum_e \boldsymbol{g}_e \cdot \boldsymbol{\Delta}_e}{\sum_e |\boldsymbol{\ell}_e \cdot \boldsymbol{\Delta}_e|} = \frac{\sum_e \boldsymbol{g}_e \cdot (x \cdot \boldsymbol{\Delta}_e)}{\sum_e |\boldsymbol{\ell}_e \cdot (x \cdot \boldsymbol{\Delta}_e)|} = \frac{x \cdot \sum_e \boldsymbol{g}_e \cdot \boldsymbol{\Delta}_e}{x \cdot \sum_e |\boldsymbol{\ell}_e \cdot \boldsymbol{\Delta}_e|}$$

for any $x \in \mathbb{R} \setminus \{0\}$. This is important, because it means that, while $\boldsymbol{\Delta}$ provides a good step *direction*, it does not necessarily provide a good step *size*. Routing F units of flow may, for example, violate capacity constraints. This property becomes relevant when we iterate with the IPM in Section 3.3.4.

3.3.4 Iterating with the IPM

In this section, we combine the previous sections and describe how the IPM reaches an optimal solution from an initial feasible solution. This section is based on Theorem 4.3 of [CKLPGS22]. We use the notation $\boldsymbol{f}^{(t)}$ to denote the flow at iteration t, and write $\boldsymbol{f}$ as a shorthand for $\boldsymbol{f}^{(t)}$.

First, the IPM assumes that it is given an initial flow $\boldsymbol{f}^{(0)}$ that satisfies $\Phi\left(\boldsymbol{f}^{(0)}\right) \leq 200m \log mU$. Following the algorithm for finding an initial flow described in Section 3.4 yields one such flow.

Now, define the "approximation quality" as $\kappa \in (0, 1)$. Approximations are used to compute approximate min-ratio cycles efficiently. The algorithm will run for $\widetilde{O}(m\kappa^{-2})$ iterations; a better level of approximation increases the theoretical runtime. On each iteration t, do the following:

1. Define $\boldsymbol{f}$ as the current flow and $\widetilde{\boldsymbol{g}}, \widetilde{\boldsymbol{\ell}} \in \mathbb{R}^E$ as approximations of $\boldsymbol{g}(\boldsymbol{f})$ and $\boldsymbol{\ell}(\boldsymbol{f})$. $\widetilde{\boldsymbol{g}}$ and $\widetilde{\boldsymbol{\ell}}$ must be sufficiently good approximations, satisfying $\|\mathbf{L}(\boldsymbol{f})^{-1}(\widetilde{\boldsymbol{g}} - \boldsymbol{g}(\boldsymbol{f}))\|_\infty \leq \kappa/8$ and $\boldsymbol{\ell}(\boldsymbol{f})_e/2 \leq \widetilde{\boldsymbol{\ell}}_e \leq 2\boldsymbol{\ell}(\boldsymbol{f})_e$ for every edge e. Recall that the infinity norm $\|\boldsymbol{v}\|_\infty$ denotes the largest of the absolute values of $\boldsymbol{v}$.

2. Find some circulation $\boldsymbol{\Delta}$ that approximately solves the undirected minimum-ratio cycle problem, satisfying $\mathbf{B}^\top \boldsymbol{\Delta} = 0$ and $\widetilde{\boldsymbol{g}}^\top \boldsymbol{\Delta} / \|\widetilde{\mathbf{L}}\boldsymbol{\Delta}\|_1 \leq -\kappa$.

3. Augment the current flow using $\boldsymbol{\Delta}$. However, the naive update $\boldsymbol{f}^{(t+1)} \leftarrow \boldsymbol{f}^{(t)} + \boldsymbol{\Delta}$ is not guaranteed to be good nor valid because the ratio of the cycle is indifferent to the amount of flow routed by $\boldsymbol{\Delta}$. The step size $\|\boldsymbol{\Delta}\|_\infty$ may be large, in which case we might step so far in the optimal direction that the step worsens the potential function or even violates a capacity constraint. Conversely, if $\|\boldsymbol{\Delta}\|_\infty$ is too small, we make insufficient progress.

For $\mathbf{\Delta}$ to provide both a good step direction *and* step size, it must be scaled appropriately. We define the scaling factor η as:[5]

$$\eta = \frac{-\kappa^2}{50 \cdot \widetilde{\boldsymbol{g}}^\top \mathbf{\Delta}} \tag{8}$$

Now, the updated flow becomes $\boldsymbol{f}^{(t+1)} \leftarrow \boldsymbol{f}^{(t)} + \eta\mathbf{\Delta}$.

The idea behind η is to take into account both the amount of routed flow and the gradient. If $\|\mathbf{\Delta}\|_\infty$ is large, the denominator of η increases, causing η to decrease. Conversely, when $\|\mathbf{\Delta}\|_\infty$ decreases, η increases. This way, large values are scaled down, and small values are scaled up. The gradient is also taken into account. If the gradient is very steep, η decreases to take smaller, more careful steps. If the gradient is gentle, η increases and causes larger steps. The circulation should decrease the potential, so $\widetilde{\boldsymbol{g}}^\top \mathbf{\Delta}$ should be negative—this is why the numerator is negated.

[CKLPGS22] argues that once $\Phi\left(\boldsymbol{f}^{(t)}\right) \leq -200m \log mU$, the IPM can terminate because the solution is good enough. By "good enough," they mean that the solution has an error of at most $(mU)^{-10}$. Formally, $\boldsymbol{c}^\top \boldsymbol{f} \leq \boldsymbol{c}^\top \boldsymbol{f}^* + (mU)^{-10}$ where $\boldsymbol{f}^*$ is an optimal flow. This happens after $\widetilde{O}(m\kappa^{-2})$ iterations.[6]

Before continuing, we give a note on the little-o notation: $o(1)$ is a term that approaches 0 as the input grows. Thus,

- $m^{o(1)}$ approaches 1 from some number > 1 as m grows.
- $m^{-o(1)}$ approaches 1 from some number < 1 as m grows.
- $m^{1+o(1)}$ becomes closer to m as m grows (from some number > m).

By "an $m^{o(1)}$-approximation," we mean an approximation that is slightly worse than optimal, but whose error grows very slowly with respect to m.

To obtain an almost-linear number of iterations $m^{1+o(1)}$, the IPM must decrease the potential by at least $m^{-o(1)}$ per iteration. In their argument, they claim that if each iteration can find an $m^{o(1)}$-approximate min-ratio cycle, resulting in $\kappa \geq m^{-o(1)}$, then augmenting the flow with $\eta\mathbf{\Delta}$ decreases the potential by $\Omega(\kappa^2)$ (i.e. *at least* κ^2), which suffices.

They claim to achieve such approximations with their dynamic graph data structure, and thus they also obtain an almost-linear number of iterations. Additionally, each approximate min-ratio cycle can be found in amortized $m^{o(1)}$ time, meaning the total running time is also $m^{1+o(1)}$.

3.3.5 Finding the optimal cost

Equation 3 assumes that the optimal cost F^* is known in advance. In [CKLPGS22], the authors claim that they may assume F^* to be known, because running their own algorithm allows them to binary search for it. However, the details surrounding the binary search procedure are not specified.

The requirement of knowing the optimal objective value in advance stems from the original approach in [Kar84, p. 305], which [CKLPGS22] builds upon. We deduce that the binary search procedure is a variation of the *sliding objective function method* introduced by [Kar84], which can be deployed when the minimum value of the objective function is unknown.

Since [CKLPGS22] omits an algorithm for the binary search, we come up with the following algorithm: let F^* be the unknown optimal objective value and define F_{min} and F_{max} as lower and upper bounds for F^*, such that $F_{\text{min}} \leq F^* \leq F_{\text{max}}$. Then:

1. Let $F' = \lfloor (F_{\text{min}} + F_{\text{max}})/2 \rfloor$ be a guess for F^*.

[5]Note that Theorem 4.3 of [CKLPGS22] defines η differently, but we believe it is due to a typo. Rearranging an equation from Lemma 4.4 of [CKLPGS22] yields this version of η.

[6]In Theorem 4.3 of [CKLPGS22], the authors write $\widetilde{O}(m\kappa^2)$. However, reading their proof, this seems to be a mistake. It should have been $\widetilde{O}(m\kappa^{-2})$.

2. Run the algorithm with the guess F' in place of F^*, thus minimising the objective $\boldsymbol{c}^\top \boldsymbol{f} - F'$. This will yield a solution $\boldsymbol{f}'$. Two cases can occur:

   Case 1, $\boldsymbol{c}^\top \boldsymbol{f}' \geq F'$: the IPM terminated with a solution that has a cost of at least F'. We have $F^* \geq F'$. Therefore, update the lower bound $F_{\min} \leftarrow F'$.

   Case 2, $\boldsymbol{c}^\top \boldsymbol{f}' < F'$: the IPM terminated with a solution that has a lower cost than F'. We have $F^* < F'$. Therefore, update the upper bound $F_{\max} \leftarrow F'$.

3. Re-run from step 1. After $O(\log(mUC))$ iterations, we converge onto F^*.

Because flow can be routed through at most m edges with capacities bounded by U and costs bounded by C, the minimum cost must be bounded by mUC. This leads to easy initial guesses for $F_{\min}$ and $F_{\max}$ and $\log_2(mUC)$ iterations. Each iteration will invoke the IPM, which runs in $m^{1+o(1)}$ time, meaning the general algorithm for minimum-cost flow runs in $m^{1+o(1)} \log(mUC)$ time.

Maximum flow is a specialisation of minimum-cost flow, which has $C = 1$ (using the standard reduction from maximum flow to minimum-cost flow). This directly leads to a running time of $m^{1+o(1)} \log(mU)$.

We can also invert the logic and view the binary search condition from the perspective of maximum flow rather than minimum-cost flow, since the optimal cost is exactly the negation of the maximum flow. Recall that the standard reduction adds an extra edge with cost -1, meaning less flow has higher cost. Case 1 corresponds to the intermediate solution being less than the guess for the maximum flow, thus the maximum flow must be smaller (and the cost larger). Conversely, if the intermediate solution is larger than the guess, the maximum flow must be larger (and the cost lower).

Before we conclude this section, a disclaimer is in order: the algorithm presented here has only been checked empirically, *not* proven mathematically. It must also be noted that our running time claims are different from those claimed by [CKLPGS22]. They claim $m^{1+o(1)} \log(U) \log(C)$ and $m^{1+o(1)} \log(U)$, respectively, which suggests that our assumptions regarding the binary search are incorrect. Unfortunately, we have been unable to deduce why.

3.4 Obtaining an initial feasible flow

The IPM requires some initial feasible flow, which it can slowly update into an optimal flow. As explained in Section 3.3.4, this flow needs not be optimal, just be *feasible* and satisfy $\Phi(\boldsymbol{f}) \leq 200m \log mU$. [CKLPGS22] proposes an algorithm to find such a starting point in Lemma 4.12, which we cover in this section. This is also based on the proof in [CKLPGS22, Appendix B.1].

The high-level idea is to assign flow to each edge, such that the flow is the midpoint between the edge's upper and lower capacities. Very likely, this will break flow conservation. Then, for each vertex, calculate the difference between its ingoing and outgoing flow; vertices that are not balanced must then be balanced. To restore flow conservation, a new vertex v^* is added. This vertex is connected to each unbalanced vertex with an edge whose capacity and initial flow is proportional to the size of the imbalance. In other words, vertices with an excess of incoming flow send the excess flow to v^*, and vertices with a deficit of incoming flow receive flow from v^*. The cost of these new edges is set to be extremely large, such that they are very expensive to use. Being expensive influences the algorithm to avoid using them as much as possible, thereby balancing the flow such that it becomes feasible within the constraints of the initial graph.

Formally, consider the input graph G, then construct $\widetilde{G}$ with $\widetilde{v} = v, \widetilde{\boldsymbol{d}}_v = \boldsymbol{d}_v$ for all $v \in V(G)$ and $\widetilde{e} = e$, $\widetilde{\boldsymbol{u}}_e = \boldsymbol{u}_e$, $\widetilde{\boldsymbol{c}}_e = \boldsymbol{c}_e$ for all $e \in E(G)$. For each edge, $e \in \widetilde{G}$ set the initial flow $\boldsymbol{f}_e = (\boldsymbol{u}_e^- + \boldsymbol{u}_e^+)/2$, thereby ensuring that the initial flow respects edge capacities. Then, calculate the demands routed by this initial flow as $\overline{\boldsymbol{d}} = \mathbf{B}^\top \boldsymbol{f}$. This constructs a $|V|$-dimensional vector representing the flow routed to each vertex, which must then be balanced to the vertices' actual demands. Add a new vertex v^*, and for each vertex $v \in V(G)$, add an edge $e_v = (v \to v^*)$ if $\overline{\boldsymbol{d}}_v < \boldsymbol{d}_v$, with $\boldsymbol{u}_{e_v}^- = 0, \boldsymbol{u}_{e_v}^+ = 2(\boldsymbol{d}_v - \overline{\boldsymbol{d}}_v), \boldsymbol{f}_{e_v} = \boldsymbol{d}_v - \overline{\boldsymbol{d}}_v$. Otherwise, if $\overline{\boldsymbol{d}}_v > \boldsymbol{d}_v$, add an edge $e_v = (v^* \to v)$, with $\boldsymbol{u}_{e_v}^- = 0, \boldsymbol{u}_{e_v}^+ = 2(\overline{\boldsymbol{d}}_v - \boldsymbol{d}_v), \boldsymbol{f}_{e_v} =$

$\overline{\boldsymbol{d}}_v - \boldsymbol{d}_v$. In both cases, let the cost of all newly created edges be $\widetilde{\boldsymbol{c}}_{e_v} = 4mU^2$. Thereby, the newly constructed graph instance $\widetilde{G}$ contains an initial feasible flow within the boundaries of the problem space.

The described method diverges from the method described in the paper, as we believe to have discovered an error in the paper. Specifically, the change is that we flip the direction of the new edges, so in the case that $\overline{\boldsymbol{d}}_v < \boldsymbol{d}_v$ the paper creates the edge $e_v = (v^* \to v)$ while our method creates the edge $e_v = (v \to v^*)$. According to our testing, the original approach would add more flow to a vertex that already has too much flow. Examples of this difference can be found in Appendix B.

3.5 Data structure

The primary focus of this research project has been on thoroughly understanding and implementing the previously described IPM. For completeness, we still provide a high level overview of the data structure, with the caveat that it will be far less detailed than other sections of this report.

The goal of the proposed data structure is to compute a min-ratio cycle of a graph in $m^{o(1)}$ time. A new min-ratio cycle has to be computed on each iteration of the IPM, therefore, specific focus has been put not only on the computation time, but also the update time of the data structure. The overarching goal of the proposed data structure is to find a cycle that approximately solves $\min_{\mathbf{B}^\top \boldsymbol{\Delta}=0} \frac{\boldsymbol{g}^\top \boldsymbol{\Delta}}{\|\mathbf{L}\boldsymbol{\Delta}\|_1}$. Algorithmically, this is a two-step process:

1. Sample a random low-stretch spanning tree T
2. Return the best tree cycle in T by adding one off-tree edge, denoted as $\text{cycle}_T(e)$

Retrieving the best tree cycle is done by repeatedly and randomly picking an off-tree edge and computing the resulting cycles. Then, cancel every adjacent edge between this set of cycles. This results, with probability at least $\frac{1}{2}$ in a min-ratio cycle for said tree. The probability of finding the min-ratio cycle of the graph can be boosted by repeating the algorithm and returning the best candidate after $O(\log n)$ repetitions [LC22].

The goal of the data structure then is to maintain an $m^{o(1)}$ forest of "low-stretch" trees while supporting slowly-changing edges and fast queries. The algorithm achieves this by reducing the number of edges and vertices via sparsification such that the graphs properties remain. Note that in this context "sparsification" is a process by which the original graph is embedded in the sparse subgraph using shortest paths, such that the original graphs properties remain and can be rebuilt.

First, consider a graph G with m edges and vertices. Then the proposed approach is to first construct a randomly rooted forest F containing K rooted partial trees on G. These trees are then used to construct a core graph $C(G,F)$ with m edges and m/K vertices. Then, by utilising a special dynamic spanner structure, S a sparsified core graph $S(C(G,F))$ is constructed with m/K edges and m/K vertices.

When an edge e is updated its tree T_e is found, the edge is then split such that each endpoint is in two different trees, this of course means a new rooted tree is constructed. The new rooted tree T_e' requires a new vertex in $C(G,F)$ as well as new edges in $S(C(G,F))$. Specifically, one edge change recursively requires one vertex split and $m^{o(1)}$ edge changes in K time.

4 Existing solutions for the subproblems

This section serves as an overview of existing solutions or implementations to the subproblems of solving an IPM and finding min-ratio cycles, which we've considered before implementing our solutions.

4.1 Linear program solvers

An IPM is a well-known type of linear program, also known as a barrier method. Solvers like IBM's CPLEX and Gurobi both support barrier functions. Unfortunately, Gurobi requires a central path that the solver follows, and such a path does not exist per our IPM definition [CKLPGS22, p. 6]. Comparatively, the IBM CPLEX barrier method requires the problem to be in a primal-dual formulation. This is also not the case for our IPM formulation: [CKLPGS22, footnote 3 on p. 6] specifically states that the polynomial length/resistance guarantees that are usually present in a primal-dual method is not present in this IPM, as it does not maintain any dual variables, and therefore, by definition, is not a primal-dual optimality problem.

4.2 Min-ratio cycle algorithms

For each solution, we consider the following parameters:

Language Which programming language is the implementation in?

Integratability How easy would it be to integrate into our use case?

Multigraph support Our algorithm supports parallel edges, and therefore so must the min-ratio solver.

The, to our knowledge, primarily used and openly available min-ratio cycle algorithm implementation is that of the Boost C++ library [BP06]. Boost implements Howard's algorithm, with the improvements from Dasdan, Irani, and Gupta's 1999 study [DIG99]. It's implemented in C++ and is heavily dependent on other data structures implemented in Boost, such as their graph, maps, and queues. This makes it difficult to adapt to our use case, but it serves well as a reference implementation, especially as it supports multigraphs.

Alternatively, Dasdan, Irani, and Gupta provided the C++ implementations [Das15] used to conduct the experiments in [Das04]. These algorithms are naturally implemented to be highly experiment-friendly. Unfortunately, this also means that the implementations contain a large amount of compilation logic that makes them rather hard to work with as a library, and to reason about. We've tested it on a multigraph, and it failed to compute the proper cycle, therefore it would require some modification to work with our graphs.

A simple solution we considered would be to calculate every cycle in the graph and then computing the cycle with the smallest ratio. Such a method could be implemented via the Python library NetworkX [HSC08] using its *simple_cycles* method. The approach would be to create a graph with each original edge being represented as an undirected edge, calling the method, and then processing each returned cycle. The processing would first find the direction each edge was travelled, compute the positive or negative gradient based on edge direction, and then calculate the ratio.

5 Partial implementation

In this section, we outline our partial implementation of the algorithm by [CKLPGS22].[7]

We focus on using the IPM to solve flow problems via continuous optimisation. To simplify both implementation and analysis, we implement a static algorithm that omits the complex internal dynamic graph data structure for computing min-ratio cycles efficiently. Instead, we use an oracle for finding min-ratio cycles that is allowed to be inefficient. If the IPM seems to be applicable in practice, future work may include implementing the data structure.

To give an overview, Algorithm 1 shows the structure of the main IPM loop that drives the algorithm.

[7]The full Python-based source code can be found online at `https://github.com/adbo-ITU/maximum-flow-and-minimum-cost-flow-in-almost-linear-time`

Algorithm 1 The general (simplified) structure of the static algorithm, given that, F^* is known.

```
$\boldsymbol{f} \leftarrow$ find some initial feasible solution    ▷ See Section 3.4
while $\boldsymbol{c}^\top \boldsymbol{f} - F^* \geq$ threshold do
    Compute $\boldsymbol{g}(\boldsymbol{f})$ and $\boldsymbol{\ell}(\boldsymbol{f})$
    $\Delta \leftarrow$ find a min-ratio circulation
    $\boldsymbol{f} \leftarrow \boldsymbol{f} + \chi \cdot \eta\Delta$
end while
return round$(\boldsymbol{c}^\top \boldsymbol{f})$ and round$(\boldsymbol{f})$
```

To reduce confusion and avoid subtle bugs, we stick to the same notation and perform the exact steps outlined in [CKLPGS22] when possible in the code. To do this, we heavily utilise `numpy` to calculate dot products, matrix multiplications, norms, and other linear-algebraic operations.

We implement algorithms for both maximum flow and minimum-cost flow. To implement max flow, we utilise the reduction described in Section 2.2 from max flow to min-cost flow: assign cost 0 to all edges and add a new edge (sink, source) with cost -1 and unlimited capacity.

The maximum flow algorithm utilises the binary search logic from Section 3.3.5 to find the optimal cost. The minimum-cost flow routine currently requires F^* to be given, but can be extended in future work to utilise binary search.

5.1 Assumptions and differences

Our implementation differs slightly from the original algorithm as described in [CKLPGS22]. Here, we cover some of those differences.

- The original algorithm is based on computing *approximate* min-ratio cycles. We use an oracle that computes *exact* min-ratio cycles, albeit much less efficiently. We assume that a true min-ratio cycle will cause a more effective IPM step. This may decrease the number of iterations made by the IPM compared to the original algorithm.
- The dynamic graph data structure from the original algorithm approximates gradients and lengths. In our implementation, gradients and lengths are recomputed on each iteration, and are thus also exact. This may also improve the number of IPM iterations.
- We experiment with various parameters in ways that are not defined by the original algorithm. We will explicitly state this when applicable in Section 6.

  Most notably, Algorithm 1 shows the use of a scaling factor χ when augmenting the flow. This is something we introduce and does not stem from [CKLPGS22]. We use χ to take larger steps within the feasible region than the original algorithm allows, which can significantly reduce the running time (by a factor of χ). However, since χ is not part of any proof in [CKLPGS22], we have no proof of the correctness of using it. We evaluate the use of χ in Section 6.
- Early termination:
  1. If $\Phi(\boldsymbol{f})$ explodes towards ∞ or $-\infty$, we terminate early and yield the current flow as the result. This can happen when the potential gets so small that floating point arithmetic results in ∞.
  2. Rather than the threshold $(mU)^{-10}$ used by [CKLPGS22], we use a static threshold of 10^{-5}. When $(mU)^{-10}$ is used, the threshold becomes so small that the number of iterations either becomes infeasible or floating point arithmetic fails.
- We use the standard numeric data types provided by Python, alongside the `int_` and `float64` integer and floating point types provided by `numpy`. All flows, gradients, and lengths are stored as `float64`.

This differs from [CKLPGS22], which specifies a model of computation where "problem instances [are] encoded with z bits [and] all algorithms work in fixed-point arithmetic where words have $O\left(\log^{O(1)} z\right)$ bits." We assume that the use of standard 64-bit floating point numbers is essential for an implementation to be feasible and provide enough precision for our experiments.

- The binary search procedure for finding the optimal cost is not described in [CKLPGS22]. We have come up with it ourselves, and its correctness has not been proven mathematically. When binary searching in the maximum flow algorithm, it suffices to define the upper bound as the sum of outgoing edge capacities from the source.

5.2 An oracle for min-ratio cycles

We have implemented Howard's algorithm with improvements from [DIG99] using [BP06] as a reference implementation. We decided to use Howard's algorithm due to [Das04], from which we concluded Howard's to be the best tradeoff between runtime and simplicity to implement.

Our implementation differs slightly from existing implementations, as we've implemented it such that it finds undirected cycles. This is done by considering each edge as moving in both directions. However, when an edge is traversed in the opposite direction of its direction, its gradient is negative. In other words, take the edge $e = u \to v$ and its gradient $\boldsymbol{g}(e)$: when this edge is traversed as $u \to v$ its gradient is $\boldsymbol{g}(e)$, and when traversed as $v \to u$ its gradient is $-\boldsymbol{g}(e)$. This is exactly as [CKLPGS22] describes it.

The result of running the algorithm is neither a list of cycle vertices nor edges. To fit with the paper, it's an $|E|$-sized list l where $l_e = 1$ if $e = u \to v$ is traversed as $u \to v$, -1 if it's traversed as $v \to u$, and 0 if the edge is not in the cycle.

6 Experimental results and evaluation

6.1 Correctness

In this section, we cover how we investigate the correctness of the implementation alongside which insights we've gained.

Unless otherwise specified, we use parameters $\chi = 500$ and $\kappa = 0.9999$.

6.1.1 Test suite

To check the correctness of the algorithm, we implement a test suite. The test suite includes instances of maximum flow and minimum-cost flow problems with various sizes and configurations. Some have been hand-crafted, some have been randomly generated, and some have been adopted from the *Maximum Flow* problem on Kattis.[8] We mostly focus on maximum flow, so most test cases consist of maximum flow instances. For maximum flow tests, we verify the result by checking:

1. Is the amount of flow correct? Here, we compare the result to an implementation of a standard max flow algorithm. Since instances can have multiple valid optimal solutions, we only check the value of the maximum flow.

2. Various conditions on the routed flow: does the amount of flow out of s match the flow into t? Are all flows non-negative? Do all flows respect edge capacities?

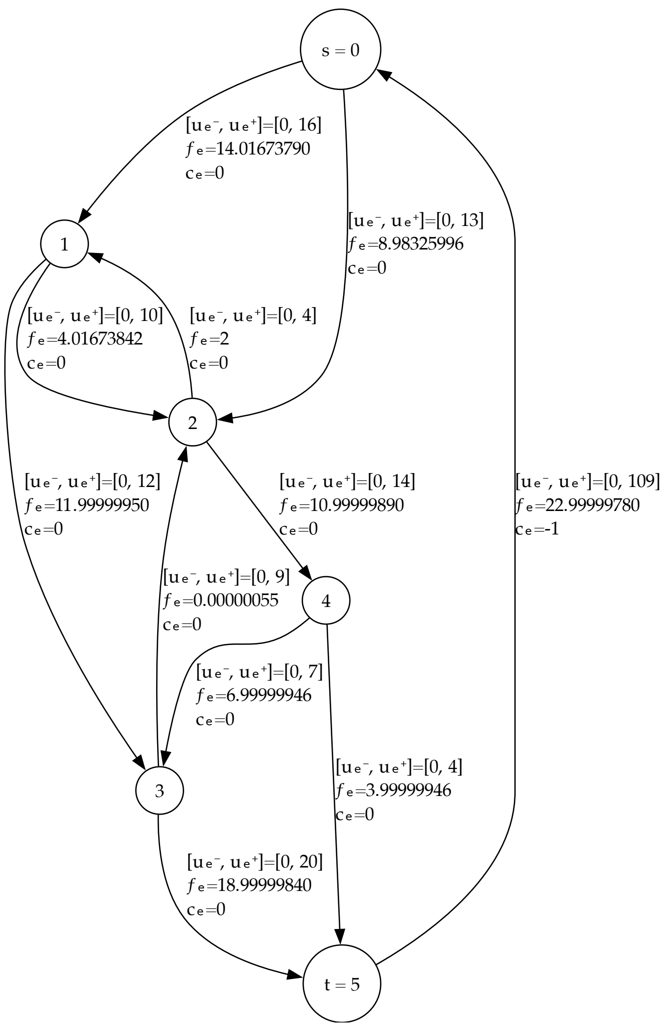
3. Is flow conservation respected? I.e. does any vertex send more/less flow than it receives?

[8]See `https://open.kattis.com/problems/maxflow`

With our implementation, most tests yield the correct value for the optimal cost (and thus also the maximum flow). However, some inputs with large capacities crash due to integer overflow. Additionally, some tests fail the flow conservation check (but yield a correct maximum flow value), which we describe in Section 6.1.2. Note that we have not been able to test instances with $m > 500$ due to slow computation of min-ratio cycles.

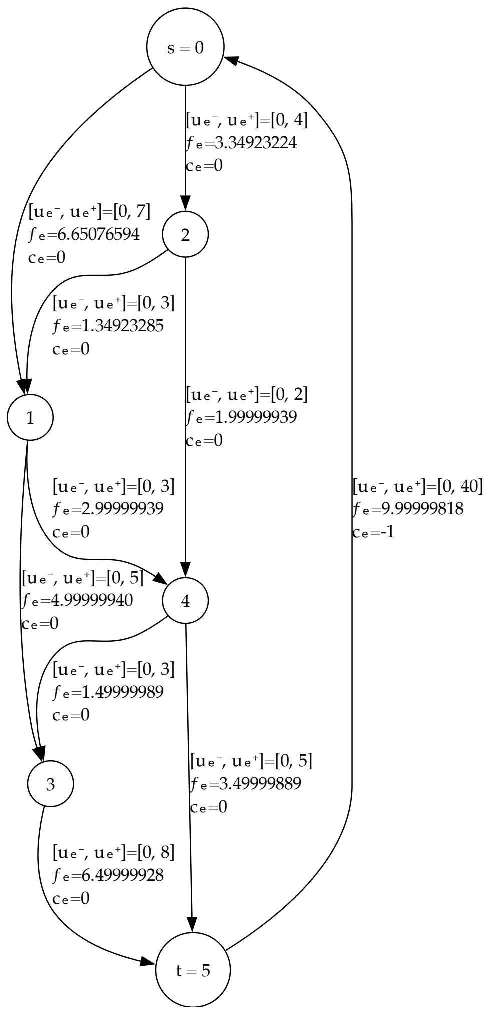
6.1.2 Individual edge flows

Often, it's useful to know how much flow is routed along each individual edge, not just the total cost or total amount of flow. However, the algorithm has a flaw in this aspect, which we cover here.



s = 0
[ue⁻, ue⁺]=[0, 16] fe=14.01673790 ce=0
[ue⁻, ue⁺]=[0, 13] fe=8.98325996 ce=0
1
[ue⁻, ue⁺]=[0, 10] fe=4.01673842 ce=0
[ue⁻, ue⁺]=[0, 4] fe=2 ce=0
2
[ue⁻, ue⁺]=[0, 12] fe=11.99999950 ce=0
[ue⁻, ue⁺]=[0, 14] fe=10.99999890 ce=0
[ue⁻, ue⁺]=[0, 109] fe=22.99999780 ce=-1
[ue⁻, ue⁺]=[0, 9] fe=0.00000055 ce=0
4
[ue⁻, ue⁺]=[0, 7] fe=6.99999946 ce=0
[ue⁻, ue⁺]=[0, 4] fe=3.99999946 ce=0
3
[ue⁻, ue⁺]=[0, 20] fe=18.99999840 ce=0
t = 5

(a) The correct result for the s-t max flow routes 23 units of flow. After rounding, the resultant flow along each edge on the graph is correct.



s = 0
[ue⁻, ue⁺]=[0, 4] fe=3.34923224 ce=0
[ue⁻, ue⁺]=[0, 7] fe=6.65076594 ce=0
2
[ue⁻, ue⁺]=[0, 3] fe=1.34923285 ce=0
[ue⁻, ue⁺]=[0, 2] fe=1.99999939 ce=0
1
[ue⁻, ue⁺]=[0, 3] fe=2.99999939 ce=0
[ue⁻, ue⁺]=[0, 40] fe=9.99999818 ce=-1
[ue⁻, ue⁺]=[0, 5] fe=4.99999940 ce=0
4
[ue⁻, ue⁺]=[0, 3] fe=1.49999989 ce=0
3
[ue⁻, ue⁺]=[0, 5] fe=3.49999889 ce=0
[ue⁻, ue⁺]=[0, 8] fe=6.49999928 ce=0
t = 5

(b) The correct result for the s-t max flow routes 10 units of flow. The total flow sent via the edge (t, s) is correct. However, after rounding, the graph does not respect flow conservation, meaning the flow graph is invalid.

Figure 6: Examples of graphs with upper/lower edge capacities, costs, source vertex s, and sink vertex t. The graphs are max flow instances that have been reduced to min-cost flow by adding an edge (t, s) with cost -1. The graphs show how much flow is routed along each edge as output by the algorithm *before* rounding.

Figure 6 shows some example instances of maximum flow and the result as given by our implementation. Figure 6a shows an instance, where the final result is correct. Figure 6b highlights a crucial flaw in the algorithm and shows an instance where the final result is incorrect due to rounding errors.

To demonstrate: after rounding the flow along each edge on Figure 6b, the following amount of flow is routed via vertex 4:

$$\text{round}\left(\boldsymbol{f}_{(1,4)}\right) = \text{round}(2.99999939) = 3 \qquad \text{round}\left(\boldsymbol{f}_{(2,4)}\right) = \text{round}(1.99999939) = 2$$
$$\text{round}\left(\boldsymbol{f}_{(4,3)}\right) = \text{round}(1.49999989) = 1 \qquad \text{round}\left(\boldsymbol{f}_{(4,5)}\right) = \text{round}(3.49999889) = 3$$

In other words, vertex 4 receives 5 units of flow and sends 4 units of flow (after rounding), meaning it has one leftover unit of flow, thus violating the flow conservation constraint.

This happens because (approximately) one unit of flow from vertex 4 is sent partially to vertex 3 and partially to vertex 5:

$$\boldsymbol{f}_{(4,3)} + \boldsymbol{f}_{(4,5)} = 0.49999989 + 0.49999889 = 0.99999878$$

In total, this would round to one unit of flow, but when split across two edges almost-evenly as in this case, each of them round to 0.

6.1.3 Scaling IPM step sizes

Most of the test suite succeeds with $\chi \in [1, 1000]$. However, $\chi > 1000$ starts to cause incorrect results. Some instances even support $\chi > 1750$, indicating that it depends on each given instance.

χ is significant for practicality, since a configuration like $\chi = 1000$ results in ~1000× speed-up compared to $\chi = 1$ as shown on Figure 7.

The fact that we are able to use χ to take larger steps indicates that the IPM takes too-small and careful steps. However, that may be a necessity as m grows. We do not claim that using χ is correct in generality, but we use it here to make the algorithm feasible wrt. practical running time.

6.1.4 Other known problems

Integer overflow. Using large capacities causes numeric overflows in multiple places. Most prominently is the initial feasible flow algorithm, where new edges are added with cost $4mU^2$, which very quickly overflows. We tested an instance with 490 edges having $U = 100,000,000$. In such a case, our implementation crashes due to overflow, which causes issues such as taking the logarithm of a negative number. We hereby note that the algorithm very quickly becomes non-practical without some modification of the input size.

Floating point numbers. Our implementation diverges from the fixed-point arithmetic used by [CKLPGS22] by using the less-predictable floating point arithmetic of Python. This causes two observed errors, firstly in some instances the flow to be routed becomes so small that the number is automatically rounded to 0, which in turn results in no flow being routed. This results in an infinite loop where no flow is ever routed. In the same vein, the amount of flow on an edge may approach a capacity so closely that an erroneous rounding error causes it to be equal to it, causing the potential function to explode.

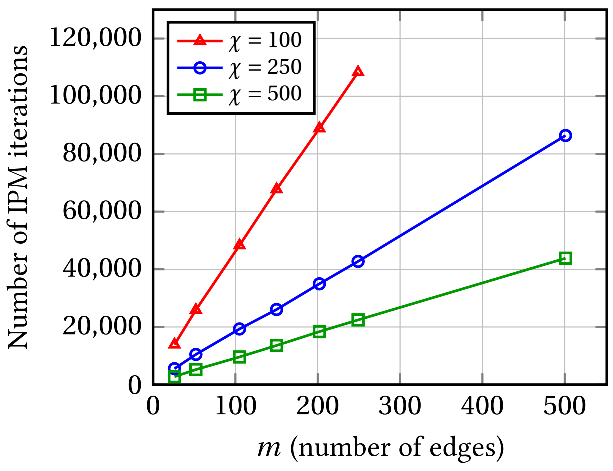
Off-by-one error in binary search. We have encountered an off-by-one error in our binary search algorithm that occurs in some specific instances. However, we believe that this stems from a silly bug that is mostly caused by our lack of understanding what we're doing. Slightly modifying the logic for the return value should fix the bug.

6.2 Running time

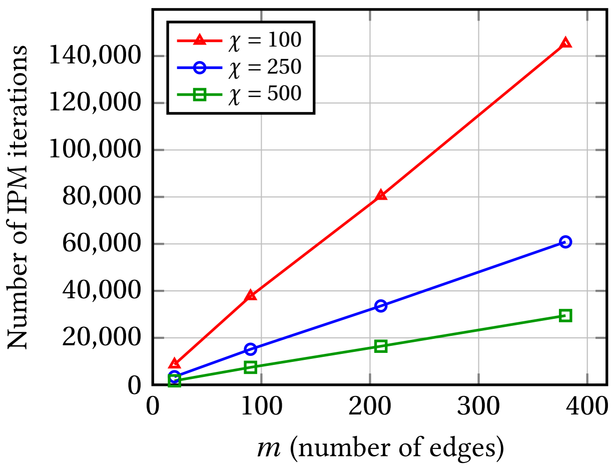
In this section, we investigate the running time of the algorithm in practice. We investigate the impact of m, χ, U, and the type of graph on the running time. The min-ratio cycles returned by our oracle are optimal, so we do not care about the approximation quality κ.

Since we use a slow implementation for the min-ratio cycle oracle, actual execution time is not representative of the algorithm's running time. Instead, we focus on the number of IPM iterations. According to [CKLPGS22], their data structure finds min-ratio cycles in amortized $m^{o(1)}$ time, meaning each IPM iteration will also run in amortized (almost) constant time.

First, we investigate the impact of the step size scaling factor χ. Figure 7 shows experimental results for varying χ.



(a) All graphs are directed acyclic graphs. The data point for $m = 500$ with $\chi = 100$ isn't included due to computation taking too long.



(b) All graphs are fully connected graphs.

Figure 7: A comparison of the number of IPM iterations before termination for different sizes of graphs (in terms of m) and scale factors for the step size. The algorithm is given F^*, so no binary search is performed.

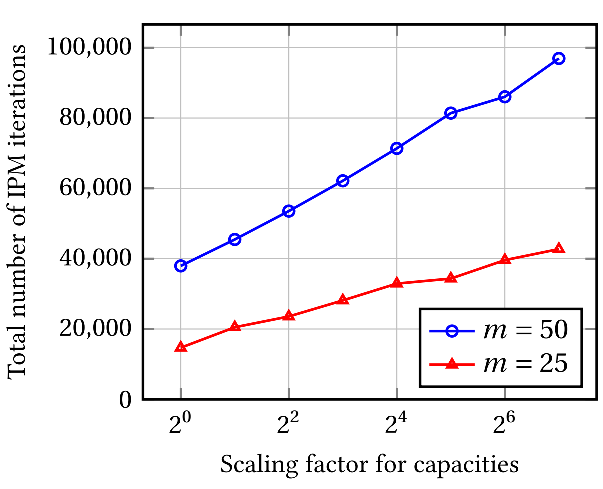
The results on Figure 7 indicate that the IPM indeed does run for an almost-linear number of iterations. More specifically, it's shown that almost-linearity in m is achieved for both directed acyclic graphs and fully connected graphs for various scaling factors χ.

It's also indicated that increasing the scale factor χ causes larger steps to be taken by the IPM, resulting in fewer overall iterations without loss of correctness. The number of iterations scales inversely and roughly linearly with χ. However, for the instances used in Figure 7, the algorithm starts to produce incorrect results for $\chi > 1400$.

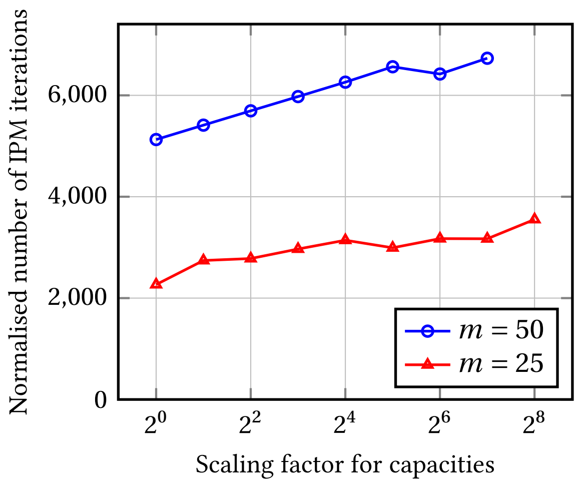
Omitting the scale factor χ from the algorithm will still yield a correct and almost-linear algorithm, but it results in too many iterations for a practical running time. It has been verified for small instances that this yields $\sim\chi$ times more iterations.

We now investigate the impact of U on the running time. The question we aim to answer is as follows: "when two instances differ only in the scale of their edge capacities, how does the running time between them differ?" The results for this experiment are shown on Figure 8.

The impact of U on the running time when binary-searching



(a) The total number of IPM iterations for each scaling factor, shown with a logarithmix x-scale.



(b) The number of IPM iterations, normalised by dividing by $\log_2(F_{\max})$ where $F_{\max}$ is the upper limit for the binary search.

Figure 8: Graphs that shows the impact of U on the running time. We keep the max flow instance fixed but scale the capacity of its edges by 1, 2, 4, … 128. The instances are the same as on Figure 7a. The original instances have capacities between 1 and 50. The algorithm is *not* given F^*, so it has to binary search for it. For concreteness: when binary searching, the instance $m = 50$ has an upper limit of 169 units of flow, which grows to 21,632 for scaling factor 128. The number of IPM iterations is accumulated over all binary search steps.

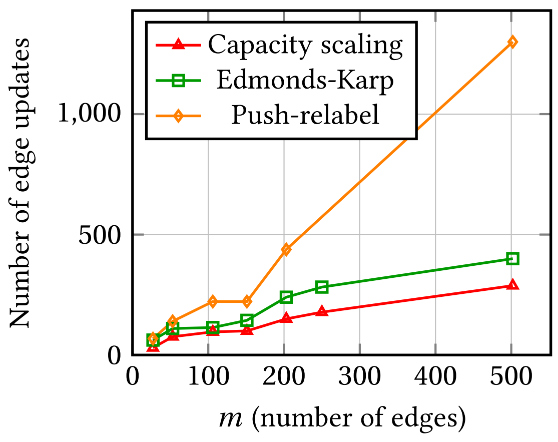
Recall that in [CKLPGS22], it's claimed that there exists an algorithm that computes the maximum flow between two vertices in $m^{1+o(1)} \log U$ time. The $m^{1+o(1)}$ term comes from the number of IPM iterations, and the $\log U$ term comes from binary searching for the optimal flow. In this experiment, the intention is to make the $m^{1+o(1)}$ term constant and see the impact of the $\log U$ term.

We would expect to see straight line on Figure 8a; the scaling factor doubles on each step, which should result in one extra binary search attempt per step. If the number of IPM iterations per attempt remains fixed, the line should be straight. Similarly, the normalised graph on Figure 8b should have a constant value and be parallel to the x-axis.

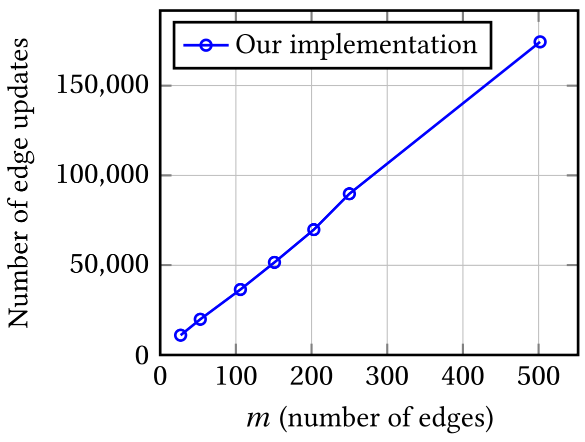
At a glance, it looks like Figure 8a grows similar to $m^{1+o(1)} \log U$. However, both instances on Figure 8b grow with respect to U. This indicates that, as U increases, the IPM needs to perform more iterations—which is not expressed by the running time of $m^{1+o(1)}$.

Thus, we cannot confirm that the algorithm runs in $m^{1+o(1)} \log U$ time. We acknowledge, however, that we may have made a mistake in the binary search implementation.

Now, we compare our implementation to standard max flow algorithms. An easy metric to measure for most flow algorithms is the number of edge updates being performed. Note, however, that the number of edge updates are *not* equivalent to the running time. An algorithm can, for example, process every edge in the graph but only update a small subset of edges. The number of edge updates does give us some comparative insight, though. The results are shown on Figure 9.



(a) Some standard max flow algorithms. Capacity scaling is implemented on top of Edmonds-Karp. The point at $m = 250$ has been omitted for push-relabel due to being an outlier.



(b) Our implementation of the max flow algorithm. F^* is given, so no binary search is performed.

Figure 9: The number of edge updates performed by various implementations of max flow algorithms with respect to m. We show our implementation on a separate graph because the number of edge updates are orders of a magnitude larger than for standard max flow algorithms. The max flow instances are the same as on Figure 7a.

Figure 9 shows that the number of edge updates grows almost-linearly with m for our implementation. However, while the selected standard flow algorithms do not necessarily grow linearly, the raw number of edge updates is orders of magnitude lower—at least for $m \leq 500$. We conclude that our implementation performs infeasibly many operations and is not practical.

We summarise this section with Table 1 by evaluating two key claims made by [CKLPGS22] about the running times of their algorithm.

Claim by [CKLPGS22]	**Our practical result**
The IPM terminates in $m^{1+o(1)}$ iterations	Yes, supported by our results
The maximum flow algorithm runs in $m^{1+o(1)} \log U$ time	We are unable to support this claim

Table 1: Evaluation of claims made by [CKLPGS22]. We assume that an iteration runs in amortized almost-constant time.

7 Discussion and scope for master's thesis

The work conducted throughout this project has suggested that the algorithm proposed by [CKLPGS22] is not a practical approach to solving flow problems efficiently, despite what the almost-linear asymptotic running time would suggest. This is largely due to the associated constants being far too large to be anywhere near practical. Problems that could be solved with ~60 iterations and hundreds of updates with traditional augmenting paths or push-relabel algorithms require ~45,000 iterations and ~170,000 updates with the IPM. Theoretically, this algorithm should be faster when the input size approaches infinity, but on practical datasets it fails to deliver.

These conclusions do fail to consider the dynamic graph data structure from [CKLPGS22]. However, it is a heavy and complex data structure that requires significant effort to implement. Future work may include implementing the dynamic graph data structure to fully investigate the scalability of the algorithm compared to current algorithms for minimum-cost flow.

Our aim was not to evaluate execution time, but rather to evaluate other properties such as number of iterations and edge updates, which would be indicative of running time. Therefore, other future work may also be to rewrite our implementation such that it utilizes low-level compute resources efficiently, since this could lead to better execution times. Though, we do not believe this to emit a useful algorithm without first completing the work of implementing the dynamic data structure.

We concede that on inputs with very large values of m, the algorithm may be faster when using the data structure to find min-ratio cycles. But with such input sizes, problems with integer sizes appear. One would need very large integers to compute expressions such as $4mU^2$, which would cause compute inefficiencies to appear.

It is important to note that we are unable to evaluate the scalability of the algorithm for $m > 500$ our current implementation, since our oracle for min-ratio cycles is not efficient enough to emit a fast algorithm.

One of our goals with this project was to investigate whether recent research on flow algorithms could provide a faster or more efficient practical algorithm for solving flow problems. It currently seems infeasible to use continuous optimisation (e.g. [CKLPGS22]) to solve such problems. Thus, we suggest that investigating another paradigm of flow algorithms may provide more practical implementations. The recent paper by Bernstein, Blikstad, Saranurak, and Tu [BBST24], which solves maximum flow via augmenting paths in $n^{2+o(1)} \log U$ time with high probability, could provide an algorithm that would reach our goal. Especially since the authors themselves believe the algorithm to be very implementable. They believe that their algorithm could be a "comeback" of combinatorial algorithms, as the trends and breakthroughs within flow algorithms has primarily been through continuous optimisation algorithms.

In conclusion, we believe that the algorithm described in [CKLPGS22] remains impractical, despite its theoretical runtime, and that switching focus to simpler combinatorial algorithms such as [BBST24] will have a higher probability of yielding a practically fast algorithm.

8 Conclusion

This research project sought to evaluate the feasibility of implementing the impressive and groundbreaking theoretical algorithm proposed by Chen, Kyng, Liu, Peng, Gutenberg, and Sachdeva, which solves maximum flow and minimum-cost flow in almost-linear time $m^{1+o(1)}$. The algorithm reduces minimum-cost flow to a linear programming problem and consists of two major components: a potential-reduction interior-point method which solves a series of min-ratio cycle subproblems, and a randomised dynamic data structure that computes approximate min-ratio cycles used to solve each subproblem efficiently. We focus on the IPM, as it is the logical foundation of the algorithm. Implementing the IPM without the data structure is significantly simpler and allows us to evaluate the practicality of the IPM on its own. There is little sense in improving the efficiency by using the data structure if the IPM is not useful.

Our main contributions are an in-depth explanation of the theoretical aspects of the IPM, a partial implementation of the algorithm where the data structure has been replaced with a less-efficient oracle for min-ratio cycles, and an evaluation of the practical aspects and properties thereof.

Through our evaluation of the IPM, we found that, while the algorithm achieves an impressive theoretical running time, the associated constants and computational requirements greatly limit its practical applicability. Theoretically, the algorithm should be efficient on inputs with large values of m, but due to both the large constants of the algorithm and the inefficiency of our oracle for min-ratio cycles, we have been unable to verify this hypothesis. We have been able to verify that the growth of the algorithm, across different parameters, is almost-linear in m.

Aside from the high number of required iterations, our implementation also identified several key limitations, such as issues with units of flow being spread across edges, alongside the requirement for large integers and high levels of precision to not either violate constraints or cause infinite computations. This makes the algorithm practically difficult to implement and causes computational inefficiencies.

We propose two directions for future work. First, implementing the dynamic data structure would allow for a more complete evaluation of the algorithm's scalability. Although, our results do not suggest this would improve the practicality of the algorithm. Second, and perhaps more promising within the goal of implementing the faster practical flow algorithm, is the recent approach to solving maximum-flow via augmenting paths by Bernstein, Blikstad, Saranurak, and Tu. Their algorithm appears more implementable and represents what they call a "comeback" of combinatorial approaches, contrasting with the continuous optimisation trend that led to the algorithm by [CKLPGS22].

In conclusion, while the theoretical advances made by [CKLPGS22] are impressive, our implementation and analysis suggests that it does not yield a practical algorithm. This underlines an important distinction between theoretical and practical advances in the design of algorithm—improvements in asymptotic complexity do not always translate to better real-world performance due to hidden constants and implementation details.

References

[BBGNSSS20] Aaron Bernstein, Jan van den Brand, Maximilian Probst Gutenberg, Danupon Nanongkai, Thatchaphol Saranurak, Aaron Sidford, and He Sun. *Fully-Dynamic Graph Sparsifiers Against an Adaptive Adversary.* arXiv:2004.08432 [cs]. Nov. 2020. DOI: 10.48550/arXiv.2004.08432. URL: http://arxiv.org/abs/2004.08432 (visited on 12/15/2024).

[BBST24] Aaron Bernstein, Joakim Blikstad, Thatchaphol Saranurak, and Ta-Wei Tu. *Maximum Flow by Augmenting Paths in $n^{2+o(1)}$ Time.* arXiv:2406.03648. June 2024. DOI: 10.48550/arXiv.2406.03648. URL: http://arxiv.org/abs/2406.03648 (visited on 11/13/2024).

[BHK17] Karl Bringmann, Thomas Dueholm Hansen, and Sebastian Krinninger. *Improved Algorithms for Computing the Cycle of Minimum Cost-to-Time Ratio in Directed Graphs.* arXiv:1704.08122. Apr. 2017. DOI: 10.48550/arXiv.1704.08122. URL: http://arxiv.org/abs/1704.08122 (visited on 11/13/2024).

[BP06] Dmitry Bufistov and Andrey Parfenov. *Boost Graph Library: Maximum (Minimum) cycle ratio - 1.41.0.* 2006. URL: https://www.boost.org/doc/libs/1_41_0/libs/graph/doc/howard_cycle_ratio.html (visited on 12/08/2024).

[Bur91] Steven Morgan Burns. "Performance analysis and optimization of asynchronous circuits". en. phd. California Institute of Technology, 1991. DOI: 10.7907/kez1-7q52. URL: https://resolver.caltech.edu/CaltechETD:etd-07092007-072640 (visited on 11/27/2024).

[CCGMQ98] Jean Cochet-Terrasson, Guy Cohen, Stéphane Gaubert, Michael McGettrick, and Jean-Pierre Quadrat. "Numerical Computation of Spectral Elements in Max-Plus Algebra". In: *IFAC Proceedings Volumes.* 5th IFAC Conference on System Structure and Control 1998 (SSC'98), Nantes, France, 8-10 July 31.18 (July 1998), pp. 667–674. ISSN: 1474-6670. DOI: 10.1016/S1474-6670(17)42067-2. URL: https://www.sciencedirect.com/science/article/pii/S1474667017420672 (visited on 11/27/2024).

[CGHPS20] Li Chen, Gramoz Goranci, Monika Henzinger, Richard Peng, and Thatchaphol Saranurak. *Fast Dynamic Cuts, Distances and Effective Resistances via Vertex Sparsifiers.* arXiv:2005.02368 [cs]. May 2020. DOI: 10.48550/arXiv.2005.02368. URL: http://arxiv.org/abs/2005.02368 (visited on 12/15/2024).

[Cha88] Bernard Chazelle. "A Functional Approach to Data Structures and Its Use in Multidimensional Searching". In: *SIAM Journal on Computing* 17.3 (June 1988), pp. 427–462. ISSN: 0097-5397. DOI: 10.1137/0217026. URL: https://epubs.siam.org/doi/10.1137/0217026 (visited on 12/12/2024).

[CIP15] Krishnendu Chatterjee, Rasmus Ibsen-Jensen, and Andreas Pavlogiannis. "Faster Algorithms for Quantitative Verification in Constant Treewidth Graphs". en. In: *Computer Aided Verification.* Ed. by Daniel Kroening and Corina S. Păsăreanu. Cham: Springer International Publishing, 2015, pp. 140–157. ISBN: 9783319216904. DOI: 10.1007/978-3-319-21690-4_9.

[CKLPGS22] 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.* arXiv:2203.00671. Apr. 2022. DOI: 10.48550/arXiv.2203.00671. URL: http://arxiv.org/abs/2203.00671 (visited on 11/13/2024).

[CMSV17] Michael B. Cohen, Aleksander Mądry, Piotr Sankowski, and Adrian Vladu. “Negative-Weight Shortest Paths and Unit Capacity Minimum Cost Flow in ? (m10/7 log W) Time (Extended Abstract)”. In: *Proceedings of the 2017 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. Proceedings. Society for Industrial and Applied Mathematics, Jan. 2017, pp. 752–771. DOI: 10.1137/1.9781611974782.48. URL: https://epubs.siam.org/doi/10.1137/1.9781611974782.48 (visited on 11/27/2024).

[Col87] Richard Cole. “Slowing down sorting networks to obtain faster sorting algorithms”. In: *J. ACM* 34.1 (Jan. 1987), pp. 200–208. ISSN: 0004-5411. DOI: 10.1145/7531.7537. URL: https://dl.acm.org/doi/10.1145/7531.7537 (visited on 11/27/2024).

[CW15] Timothy M. Chan and Ryan Williams. “Deterministic APSP, Orthogonal Vectors, and More: Quickly Derandomizing Razborov-Smolensky”. In: *Proceedings of the 2016 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. Proceedings. Society for Industrial and Applied Mathematics, Dec. 2015, pp. 1246–1255. DOI: 10.1137/1.9781611974331.ch87. URL: https://epubs.siam.org/doi/10.1137/1.9781611974331.ch87 (visited on 11/27/2024).

[Dan51] George B. Dantzig. “Application of the simplex method to a transportation problem”. In: *Activity Analysis and Production and Allocation* (1951). URL: https://cir.nii.ac.jp/crid/1571980075507143680 (visited on 12/10/2024).

[Das04] Ali Dasdan. “Experimental analysis of the fastest optimum cycle ratio and mean algorithms”. In: *ACM Trans. Des. Autom. Electron. Syst.* 9.4 (Oct. 2004), pp. 385–418. ISSN: 1084-4309. DOI: 10.1145/1027084.1027085. URL: https://doi.org/10.1145/1027084.1027085 (visited on 12/08/2024).

[Das15] Ali Dasdan. *optimum-cycle-ratio-algorithms*. en. 2015. URL: https://github.com/alidasdan/optimum-cycle-ratio-algorithms (visited on 12/08/2024).

[DBR66] George B. Dantzig, W. O. Blattner, and M. R. Rao. “FINDING A CYCLE IN A GRAPH WITH MINIMUM COST TO TIME RATIO WITH APPLICATION TO A SHIP ROUTING PROBLEM:” in: Fort Belvoir, VA: Defense Technical Information Center, Nov. 1966. DOI: 10.21236/AD0646553. URL: http://www.dtic.mil/docs/citations/AD0646553 (visited on 11/13/2024).

[DGGP19] David Durfee, Yu Gao, Gramoz Goranci, and Richard Peng. “Fully dynamic spectral vertex sparsifiers and applications”. In: *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*. STOC 2019. New York, NY, USA: Association for Computing Machinery, June 2019, pp. 914–925. ISBN: 9781450367059. DOI: 10.1145/3313276.3316379. URL: https://dl.acm.org/doi/10.1145/3313276.3316379 (visited on 12/15/2024).

[DIG99] A. Dasdan, S.S. Irani, and R.K. Gupta. “Efficient algorithms for optimum cycle mean and optimum cost to time ratio problems”. In: *Proceedings 1999 Design Automation Conference (Cat. No. 99CH36361)*. June 1999, pp. 37–42. DOI: 10.1109/DAC.1999.781227. URL: https://ieeexplore.ieee.org/document/781227 (visited on 11/20/2024).

[DS08] Samuel I. Daitch and Daniel A. Spielman. *Faster Approximate Lossy Generalized Flow via Interior Point Algorithms*. arXiv:0803.0988. Apr. 2008. DOI: 10.48550/arXiv.0803.0988. URL: http://arxiv.org/abs/0803.0988 (visited on 11/13/2024).

[Fox69] Bennett Fox. “Finding Minimal Cost-Time Ratio Circuits”. In: *Operations Research* 17.3 (1969), pp. 546–551. ISSN: 0030-364X. URL: https://www.jstor.org/stable/168388 (visited on 11/27/2024).

[Gol82] M. v. Golitschek. “Optimal cycles in doubly weighted graphs and approximation of bivariate functions by univariate ones”. en. In: *Numerische Mathematik* 39.1 (Feb. 1982), pp. 65–84. ISSN: 0945-3245. DOI: 10.1007/BF01399312. URL: https://doi.org/10.1007/BF01399312 (visited on 11/27/2024).

[Gol95] Andrew V. Goldberg. "Scaling Algorithms for the Shortest Paths Problem". In: *SIAM Journal on Computing* 24.3 (June 1995), pp. 494–504. ISSN: 0097-5397. DOI: 10.1137/S0097539792231179. URL: https://epubs.siam.org/doi/10.1137/S0097539792231179 (visited on 11/27/2024).

[GT89] Andrew V. Goldberg and Robert E. Tarjan. "Finding minimum-cost circulations by canceling negative cycles". In: *J. ACM* 36.4 (Oct. 1989), pp. 873–886. ISSN: 0004-5411. DOI: 10.1145/76359.76368. URL: https://dl.acm.org/doi/10.1145/76359.76368 (visited on 11/13/2024).

[HO93] Mark Hartmann and James B. Orlin. "Finding minimum cost to time ratio cycles with small integral transit times". fr. In: *Networks* 23.6 (1993), pp. 567–574. ISSN: 1097-0037. DOI: 10.1002/net.3230230607. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/net.3230230607 (visited on 11/27/2024).

[HSC08] Aric Hagberg, Pieter Swart, and Daniel Chult. *Exploring Network Structure, Dynamics, and Function Using NetworkX*. June 2008. DOI: 10.25080/TCWV9851.

[Kar78] Richard M. Karp. "A characterization of the minimum cycle mean in a digraph". In: *Discrete Mathematics* 23.3 (Jan. 1978), pp. 309–311. ISSN: 0012-365X. DOI: 10.1016/0012-365X(78)90011-0. URL: https://www.sciencedirect.com/science/article/pii/0012365X78900110 (visited on 11/13/2024).

[Kar84] Narendra Karmarkar. "A New Polynomial-Time Algorithm for Linear Programming-II". In: *Combinatorica* 4 (Dec. 1984), pp. 302–311. DOI: 10.1007/BF02579150.

[Kle67] Morton Klein. "A Primal Method for Minimal Cost Flows with Applications to the Assignment and Transportation Problems". In: *Management Science* 14.3 (Nov. 1967), pp. 205–220. ISSN: 0025-1909. DOI: 10.1287/mnsc.14.3.205. URL: https://pubsonline.informs.org/doi/10.1287/mnsc.14.3.205 (visited on 11/13/2024).

[KS21] Adam Karczmarz and Piotr Sankowski. *A Deterministic Parallel APSP Algorithm and its Applications*. arXiv:2101.02311. Jan. 2021. DOI: 10.48550/arXiv.2101.02311. URL: http://arxiv.org/abs/2101.02311 (visited on 11/27/2024).

[Law66] Eugene L. Lawler. "Optimal cycles in doubly weighted directed linear graphs". In: *Proc. Int'l Symp. Theory of Graphs*. 1966, pp. 209–232.

[Law76] Eugene L. Lawler. *Combinatorial optimization : networks and matroids*. eng. New York : Holt, Rinehart and Winston, 1976. ISBN: 9780030848667. URL: http://archive.org/details/combinatorialopt0000lawl (visited on 11/13/2024).

[LC22] Yang P. Liu and Li Chen. *Maximum Flow and Minimum-Cost Flow in Almost Linear Time*. Stanford, Mar. 2022. URL: https://yangpliu.github.io/pdf/Flows_Stanford.pdf.

[LS14] Yin Tat Lee and Aaron Sidford. "Path Finding Methods for Linear Programming: Solving Linear Programs in Õ(vrank) Iterations and Faster Algorithms for Maximum Flow". In: *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*. ISSN: 0272-5428. Oct. 2014, pp. 424–433. DOI: 10.1109/FOCS.2014.52. URL: https://ieeexplore.ieee.org/document/6979027 (visited on 11/13/2024).

[LS19] Yang P. Liu and Aaron Sidford. *Faster Energy Maximization for Faster Maximum Flow*. arXiv:1910.14276. Oct. 2019. DOI: 10.48550/arXiv.1910.14276. URL: http://arxiv.org/abs/1910.14276 (visited on 11/13/2024).

[Mad13] Aleksander Madry. *Navigating Central Path with Electrical Flows: from Flows to Matchings, and Back*. arXiv:1307.2205. Oct. 2013. DOI: 10.48550/arXiv.1307.2205. URL: http://arxiv.org/abs/1307.2205 (visited on 11/13/2024).

[Mad16] Aleksander Madry. *Computing Maximum Flow with Augmenting Electrical Flows.* arXiv:1608.06016. Aug. 2016. DOI: 10.48550/arXiv.1608.06016. URL: http://arxiv.org/abs/1608.06016 (visited on 11/13/2024).

[Meg79] Nimrod Megiddo. "Combinatorial Optimization with Rational Objective Functions". In: *Mathematics of Operations Research* 4.4 (1979), pp. 414–424. ISSN: 0364-765X. URL: https://www.jstor.org/stable/3689226 (visited on 11/13/2024).

[Meg83] Nimrod Megiddo. "Applying Parallel Computation Algorithms in the Design of Serial Algorithms". In: *J. ACM* 30.4 (Oct. 1983), pp. 852–865. ISSN: 0004-5411. DOI: 10.1145/2157.322410. URL: https://dl.acm.org/doi/10.1145/2157.322410 (visited on 11/13/2024).

[Ren88] James Renegar. "A polynomial-time algorithm, based on Newton's method, for linear programming". en. In: *Mathematical Programming* 40.1 (Jan. 1988), pp. 59–93. ISSN: 1436-4646. DOI: 10.1007/BF01580724. URL: https://doi.org/10.1007/BF01580724 (visited on 12/14/2024).

[San05] Piotr Sankowski. "Shortest Paths in Matrix Multiplication Time". en. In: *Algorithms – ESA 2005*. Ed. by Gerth Stølting Brodal and Stefano Leonardi. Berlin, Heidelberg: Springer, 2005, pp. 770–778. ISBN: 9783540319511. DOI: 10.1007/11561071_68.

[ST04] Daniel A. Spielman and Shang-Hua Teng. "Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems". In: *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*. STOC '04. New York, NY, USA: Association for Computing Machinery, June 2004, pp. 81–90. ISBN: 9781581138528. DOI: 10.1145/1007352.1007372. URL: https://doi.org/10.1145/1007352.1007372 (visited on 11/13/2024).

[WXXZ23] Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. *New Bounds for Matrix Multiplication: from Alpha to Omega.* arXiv:2307.07970. Nov. 2023. DOI: 10.48550/arXiv.2307.07970. URL: http://arxiv.org/abs/2307.07970 (visited on 11/27/2024).

[WZ92] C. Wallacher and U. Zimmermann. "A combinatorial interior point method for network flow problems". en. In: *Mathematical Programming* 56.1 (Aug. 1992), pp. 321–335. ISSN: 1436-4646. DOI: 10.1007/BF01580905. URL: https://doi.org/10.1007/BF01580905 (visited on 11/13/2024).

[YTO91] Neal Young, Robert Tarjan, and James Orlin. "Faster Parametric Shortest Path and Minimum Balance Algorithms". In: *Networks* 21.2 (Mar. 1991). arXiv:cs/0205041, pp. 205–221. ISSN: 0028-3045, 1097-0037. DOI: 10.1002/net.3230210206. URL: http://arxiv.org/abs/cs/0205041 (visited on 12/08/2024).

A Rewriting the potential function to the modern form

Given an objective $\boldsymbol{c}^\top \boldsymbol{x}$, the potential function is originally expressed as Equation 9 by [Kar84]. We show how to rearrange it to the more modern form of Equation 11.

$$f(\boldsymbol{x}) = \sum_j^n \ln\left(\frac{\boldsymbol{c}^\top \boldsymbol{x}}{\boldsymbol{x}_j}\right) \tag{9}$$

$$= \sum_j^n \left(\ln(\boldsymbol{c}^\top \boldsymbol{x}) - \ln \boldsymbol{x}_j\right) \tag{10}$$

$$= n \ln(\boldsymbol{c}^\top \boldsymbol{x}) + \sum_j^n - \ln \boldsymbol{x}_j \tag{11}$$

where n is the number of dimensions of $\boldsymbol{x}$.

B Examples of differences in feasible flow

Take the base graph in Figure 10.

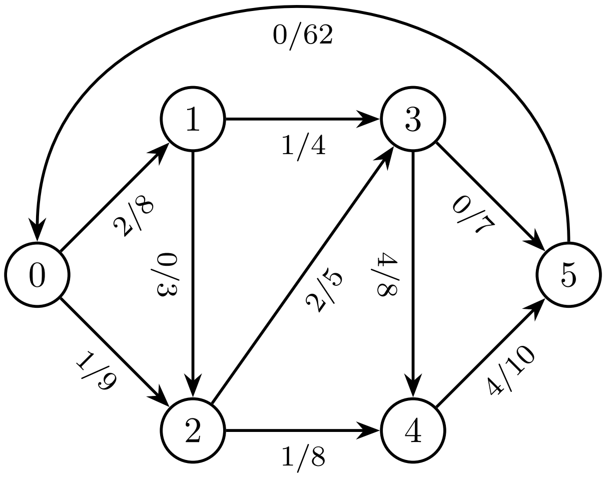
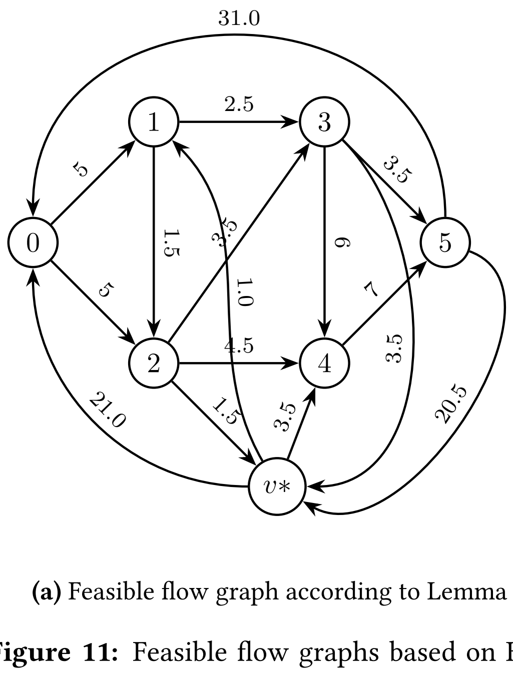


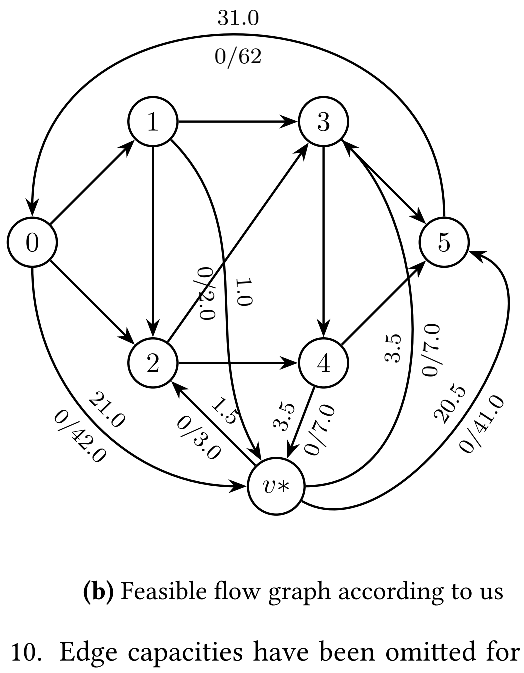
Figure 10: Base case

If one were to follow the method described in Lemma 4.12 of [CKLPGS22], the result would be that of Figure 11a. Focus on the vertex labelled 1, it has 5 units of incoming flow and 4 units of outgoing flow. In this case V^* adds an additional 1 incoming flow, therefore this graph does not uphold flow conservation.

Compare this to Figure 11b where the edges to and from V^* are reversed. In this case, the additional incoming flow is sent to V^* thereby upholding flow conservation.



(a) Feasible flow graph according to Lemma 4.12



(b) Feasible flow graph according to us

Figure 11: Feasible flow graphs based on Figure 10. Edge capacities have been omitted for visual clarity. The edge labels represent the initial flow on that edge

C Informal model of computation reasoning

This appendix attempts to explain our reasoning for the model of computation.

As per the overview: [CKLPGS22] claims that the algorithm works for fixed-point arithmetic. Specifically, they write: *for problem instances encoded with z bits, all algorithms work in fixed-point arithmetic where words have $O(\log^{O(1)} z)$ bits.* We do not claim that the following information is accurate, but it seems to be a reasonable explanation of the concept.

The model of computation defines two key aspects, firstly it defines a word size as a function of the bit size of the input. Secondly, it bounds the size of the stored numbers based on the bit size of the input. The bit size of the input describes the entire input. So for our graph, inputs z would be the total number of bits required to represent the upper- and lower-capacities, costs, demands, vertex numbers and edges. One could say z represents the storage size of the input, compared to factors such as n and, m which describe the length of the input. The word size then is defined as $O(\log^{O(1)} z)$ bits long. The word size defines how large, and small, the stored numbers can be, specifically they define the smallest possible number to be $\exp(-\log^{O(1)} z)$ and the largest to be $\exp(\log^{O(1)} z)$. Since z does not change based on $O(1)$, and $O(1)$ is a constant, one can conclude that the larger $O(1)$ becomes, the less numbers can be represented in the input, but they can be much larger, or preciser. On the other hand, if $O(1)$ is small, there are can be more numbers, or less precision.

The reason for including such a model of computation is explained well by Chazelle, which states that "It is all too easy to "improve" algorithms by encoding astronomical numbers in one computer word or allowing arbitrarily complex arithmetic"[Cha88]. In other words, by being explicit about the size and representation of the involved numbers you have to take computational time into account when deciding your runtimes, essentially, you are ensuring your theoretical algorithms remain practical and doesn't require either impractically large numbers, or unrealistic amounts of precision.

In the case of [CKLPGS22], the decision has been to use numbers of a reasonable size that grows very slowly as a function of the input size z.

In conclusion, the selected model of computation ensures that the algorithm does not "cheat" by requiring un-computably large numbers or precision, by bounding the numbers to a reasonable size and precision, as a function of the input.