Maximum Network Flow with Floating Point Arithmetic

Ernst Althaus[*] Kurt Mehlhorn[†]

October 7, 1997

Abstract

We discuss the implementation of network flow algorithms in floating point arithmetic. We give an example to illustrate the difficulties that may arise when floating point arithmetic is used without care. We describe an iterative improvement scheme that can be put around any network flow algorithm for integer capacities. The scheme carefully scales the capacities such that all integers arising can be handled exactly using floating point arithmetic. For $m \leq 10^9$ and double precision floating point arithmetic the number of iterations is always bounded by three and the relative error in the flow value is at most 2^{-19}. For $m \leq 10^6$ and double precision arithmetic the relative error after the first iteration is bounded by 10^{-3}.

1 Introduction

Network algorithms, e.g., shortest paths or maximum network flow, are usually formulated for the *real number model* of computation in which all arithmetic operations are exact and incur no rounding error. Floating point arithmetic (FPA) incurs rounding error and it is therefore not surprising that network algorithms designed for the real number model of computation may malfunction when implemented with FPA. Malfunctioning can either mean non-termination or production of an incorrect result. We discuss this phenomenon in the context of the maximum flow problem.

Let $G = (V, E)$ be a directed graph, let s and t be vertices of G called the source and the sink, respectively, and let $c : E \to R_{\geq 0}$ be a non-negative real-valued capacity function on the edges. A flow f is a function from the edges to the real numbers satisfying

(1) the non-negativity constraint $f(e) \geq 0$ for all $e \in E$,

(2) the capacity constraint $f(e) \leq c(e)$ for all $e \in E$, and

(3) the flow conservation constraint $\sum \{ f(e) \ ; e \text{ ends in } v \} = \sum \{ f(e) \ ; e \text{ starts in } v \}$ for all vertices v distinct from s and t.

The value $|f|$ of a flow f is defined as the flow out of s minus the flow into s. The maximum flow problem asks for the computation of a flow of maximum value.

All algorithms for the maximum flow problem compute the flow incrementally; see [AMO93] for a survey of flow algorithms. The final flow across an edge e is computed as a sum of flow portions; flow portions may be positive or negative. In a floating point evaluation of this sum there may be cancellation. Figure 1 illustrates this point for the preflow-push algorithm of Goldberg and Tarjan [GT88]. Due to the cancellations the algorithm may not terminate, or terminate and return a function f which is not a flow (because it violates one of the constraints) or is a flow but not a maximal flow. The current paper was inspired by the observation that the implementation of the preflow-push algorithm distributed in LEDA [MN95, MNU97] does not always terminate when run with FPA.

What can be done to remedy the situation?

1. We may resort to arbitrary precision integer arithmetic. This will solve all problems, imply a certain loss of efficiency (since the integers might be quite large) and convenience (because of the necessary conversion), but otherwise solve all problems. We will not consider this solution any further since we want to stay within FPA.

[*]Max-Planck-Institute für Informatik, Im Stadtwald, 66123 Saarbrücken

[†]Max-Planck-Institute für Informatik, Im Stadtwald, 66123 Saarbrücken

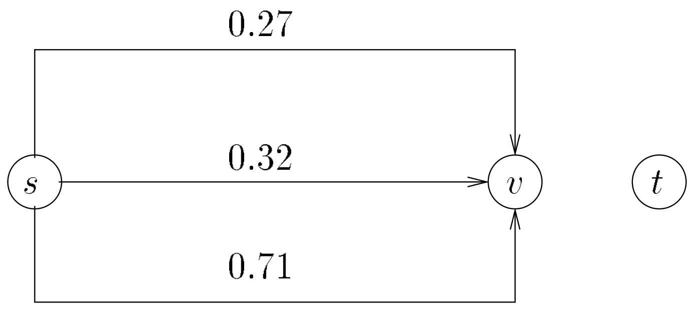

Figure 1: A network of three nodes and four edges. The capacities of the edges are as shown. The preflow push algorithm of Goldberg-Tarjan starts by saturating all edges out of s. This will create an excess of $0.27 + 0.32 + 0.71 = 1.3$ in v. In the course of the execution, the algorithm will determine that none of this excess can be forwarded to t and hence the excess will be shipped back to s by sending 0.27, 0.32, and .71, respectively, across the three edges (v, s). The final excess in v is $1.3 - 0.27 - 0.32 - 0.71 = 0$. Assume now that all calculations are carried out in a floating point system with a mantissa of two decimal places and rounding by cut-off. Then the excess in v after saturating all edges out of v will still be 1.3 as there is no cancellation in the summation. However, when the flow is pushed back to s the floating point computation the first subtraction $1.3 \ominus 0.29$ yields 1.1 as the last digit of 0.29 is dropped when the two summands are aligned for the subtraction. The effect of this is that v ends up with an excess of 0.09 but no outgoing edge across which to push flow. This may put the algorithm into an infinite loop.

2. We may change the algorithm, e.g., by replacing all tests for zero by comparisons with a carefully chosen epsilon (or maybe different epsilons for different tests). This is probably the most popular approach. However, several questions arise? How should the epsilons be chosen as a function of the input? What properties does the computed flow have, e.g, in what sense does it satisfy flow conservation and how close to the optimal flow value is the computed flow value? Is there a generic way to choose the epsilons that works for any flow algorithm or at least for a large class of flow algorithms? This approach was used in the LEDA implementation of the preflow-push algorithm mentioned above. Apparently the choice of the epsilons was not made carefully enough. We have no advice on what a careful choice is.

3. We may misuse floating point arithmetic as an implementation of integer arithmetic for integers whose absolute value is bounded by $M = 2^{ML+1} - 1$ where ML is the mantissa length of the floating point system. According to the IEEE standard [IEE87] we have ML equal 26 for single precision FPA and equal to 52 for double precision FPA. We follow the last approach in this paper. Throughout the paper we use M to denote the maximum integer that can be represented in the mantissa of the floating point system.

We describe an iterative algorithm that computes a flow f that approximates the optimal flow f^*. The algorithm uses any flow algorithm A for integral capacities as a subroutine. The subroutine is exercised with floating point arithmetic and the quality of approximation depends on the particular flow algorithm used. For many algorithms, e.g., all augmenting path algorithms and all push-relabel algorithms, we have

- $f = 0$ iff $f^* = 0$ and
- $(|\,f^*\,| - |\,f\,|)/|\,f^*\,| \leq 8m/M$ if $f^* \neq 0$.
- The scheme performs at most $2 + \lfloor \log m / \log(M/2m) \rfloor$ flow computations.

For $m \leq 10^9$ and double precision floating point arithmetic the number of iterations is always bounded by three and the relative error in the flow value is at most 2^{-19}. For $m \leq 10^6$ and double precision arithmetic the relative error after the first iteration is bounded by 10^{-3}.

The algorithm first computes the so-called bottleneck capacity c_B for s and t; this is the largest capacity such that there is a path from s to t all of whose edges have capacity at least c_B. We show that $c_B \leq |\,f^*\,| \leq m \cdot c_B$. We then use this estimate for the maximum flow to scale the capacities. The scaling depends on the algorithm A. We scale all capacities to integers such that A when run on the scaled capacities will only operate on integers whose absolute value is bounded by M. Thus the integer arithmetic required by A can be realized by floating point arithmetic. We run A on the scaled capacities and obtain a better estimate for the value of the maximum flow. We repeat until the relative error is less than $8m/M$ or until no further scaling is necessary. For $m \leq 10^9$ and double precision floating point arithmetic the number of iterations is always bounded by three.

2 Bottleneck Shortest Paths

The bottleneck capacity gives a crude approximation of $|\, f^* \,|$.

Lemma 1 $c_B \leq |\, f^* \,| \leq m \cdot c_B$

Proof: There is a path from s to t all of whose edges have capacity at least c_B. Thus $c_B \leq |\, f^* \,|$. Let S be the set of nodes that are reachable from s by paths all of whose edges have capacity more than c_B. Then $t \notin S$ and any edge in the cut $(S, V \setminus S)$ has capacity at most c_B. Thus the capacity of the cut is bounded by $m \cdot c_B$ and hence the value of the maximum flow is bounded by the same quantity. ∎

The bottleneck capacity can be computed in time $O(m + n \log n)$ by Dijkstra's algorithm, see [AMO93, exercise 4.37], or in time $O(m \log n)$ by sorting the edges by capacity and then performing an incremental connectivity computation. The details of the second approach are as follows. We sort the edges in order of decreasing cost, declare s reached, and all other vertices unreached. We then insert the edges one-by-one. When an edge $e = (v, w)$ is inserted we distinguish cases according to whether v is reached already or not. If v is not reached yet we simply add e to the list of outgoing of edges of v. If w is already reached we do nothing, and if v is was already reached but w was not we declare w reached and start a depth-first-search from w. We stop as soon as t is reached.

3 An Approximation Algorithm when an Upper Bound on the Flow is Known

In this section we show how to compute an approximate flow when an upper bound on the flow is known. The quality of the approximation depends on the quality of the upper bound.

Consider any network algorithm A for integral weights. Let $L(G, s, t, c)$ be the largest absolute value of any integer handled by A when it is run on input G with source s, sink t, and capacity vector c. Let C be the maximal capacity and let U be an upper bound on the value of the maximum flow, e.g. $m \cdot c_B$.

We give some examples.

1. Augmenting path algorithms compute a flow as a sum of "simpler" flows; the simpler flows are either flows along paths or so-called blocking flows. It is easy to see that all numbers handled by augmenting path algorithms are bounded by the maximum of $|\, f^* \,|$ and C. Thus $L(G, s, t, c) \leq \max(C, |\, f^* \,|) \leq \max(C, m \cdot c_B)$.

2. The first term in the upper bound in the previous item can be arbitrarily larger than the maximum flow. This is undesirable. The bound can be reduced to U by replacing all capacities larger than U by U.

3. Preflow-push algorithms deal with so-called preflows. Most of them start by saturating all edges out of s and then redistribute the excess created in the initialization step. Thus the numbers handled by the algorithms might be as large as the sum of the capacities of the edges out of s. Thus $L(G, s, t, c) \leq nC$.

4. The bound in the previous item can be improved to U as follows. As before, we replace all capacities larger than U by U. In addition, we add an artificial source s' and an edge (s', s) of capacity U to G and run the algorithm on the resulting network. All numbers handled will be smaller than U.

Let (G, s, t, c) be a flow problem with integral capacities and let U be an upper bound on the maximum flow; at the end of the section we show how to handle non-integral capacities. Let G' be obtained from G as in the examples above, i.e., by reducing all capacities larger than U and by maybe adding an artificial source. Call the resulting capacity vector c'. When A would be run on (G', s, t, c') the largest integer handled by the algorithm would be $L = L(G', s, t, c')$. Let l be the minimal non-negative integer such that $L2^{-l} \leq M = 2^{ML+1} - 1$ and define the capacity vector c'' by

$$c''(e) = \begin{cases} \lfloor c(e) * 2^{-l} \rfloor & \text{if } c(e) \leq U \\ \lfloor U * 2^{-l} \rfloor & \text{if } c(e) > U, \end{cases}$$

i.e, capacities larger than U are replaced by U and then the last l bits of all capacities are dropped.

How does algorithm A perform on input $G(', s, t, c'')$, in particular, what is $L(G', s, t, c'')$? We make the following assumption.

Scaling Assumption: $L(G', s, t, c'') \leq L(G, s, t, c')/2^l$.

The assumption is satisfied for all augmenting path and all preflow-push algorithms. We assume for the sequel that the assumption holds. Then all numbers handled by A on (G', s, t, c'') are bounded by M and hence A when executed with floating point arithmetic will compute the exact solution f' to the problem (G', s, t, c''). Let $f(e) = f'(e) \cdot 2^l$ for all e. Then f is a feasible solution for (G, s, t, c) since $c(e) \geq c''(e) \cdot 2^l$ for all e. Moreover, f satisfies the flow conservation constraints even in FPA since the multiplication by 2^l is only an adjustment of the exponent but does not affect the mantissa.

We next estimate the quality of f. Clearly, $|f| \leq |f^*|$.

To bound $|f^*|$ in terms of $|f|$ we consider a minimal s,t-cut (S, T) in (G, s, t, c''). Then

$$\begin{aligned} |f| &= |f'| \cdot 2^l \\ &= c''(S, T) \cdot 2^l \\ &= \sum_{e \in (S \times T)} c''(e) \cdot 2^l \\ &\geq \sum_{e \in (S \times T)} c(e) - 2^l \\ &\geq |f^*| - m2^l, \end{aligned}$$

where the next to last inequality follows from the fact that the flow across edge e is bounded by $\min(U, c(e))$ which in turn is bounded by $(c''(e) + 1)2^l$, and the last inequality follows from the fact that the value of any flow is bounded by the capacity of any cut.

We summarize our discussion. For simplicity we formulate the summary under the assumption that L is bounded by U.

Lemma 2 *If the scaling assumption holds and $L \leq U$ then the algorithm above computes a flow f with $f = f^*$ if $U \leq M$ and*

$$|f| \leq |f^*| \leq \lfloor |f| + 2mU/M \rfloor$$

otherwise. The algorithm works with FPA. The relative error $(|f^*| - |f|)/|f^*|$ is bounded by $2mU/(|f^*|M)$.

Proof: If $U \leq M$ then no scaling is necessary and the claim follows. If scaling is necessary then the error is bounded by $m2^l$ where l is minimal such that $U2^{-l} \leq M$. Then $l = \lceil \log U/M \rceil$ and hence the error is bounded by $2mU/M$. Since $|f^*|$ is integral we may round down to the next integer. ∎

With $U = m \cdot c_B$ and $c_B \leq |f^*|$ we obtain that the relative error is bounded by $2m^2/M$. In the case of double precision floating point arithmetic the relative error is bounded by 10^{-3} for $m \leq 10^6$. This suffices for most practical purposes. For single precision FPA the bound is unsatisfactory.

4 An Iterative Improvement Scheme

We describe an iterative improvement scheme that guarantees a relative error of $8m/M$. For this scheme we need the additional assumption that $\mu = 2m/M \leq 1/2$; this is no restriction for practical values of m.

Let $U_0 = m \cdot c_B$, let $f_0 = f$ be the flow computed above, and let $i = 0$. If $|f_i| \geq U_i/4$ or $l = 0$ was chosen in the i-th iteration stop and return f_i. Otherwise, let $U_{i+1} = \lfloor |f_i| + 2mU_i/M \rfloor$, compute f_{i+1} with the algorithm of the preceding section and upper bound U_{i+1}, increment i and repeat.

When the algorithm stops we either have $f = f^*$ or $|\, f_i \,| \geq U_i/4$. In the former case the relative error is zero and in the latter case the relative error can be bounded as

$$(|\, f^* \,| - |\, f_i \,|)/|\, f^* \,| \leq 2mU_i/(M|\, f^* \,|) \leq 2mU_i/(M|\, f_i \,|) \leq 8m/M.$$

If $|\, f_i \,| < U_i/4$ then $U_{i+1} \leq |\, f_i \,| + 2mU_i/M \leq U_i/2$ and hence a smaller l will be chosen in iteration $i + 1$ than in iteration i. This proves termination.

We want to bound the number of iterations. We claim that $U_i \leq (1 + 2\mu + \mu^i m)|\, f_i \,|$ for all $i \geq 1$. For $i = 1$ this follows from $U_1 \leq |\, f_0 \,| + \mu U_0$, $U_0 \leq m|\, f_0 \,|$, and $|\, f_0 \,| \leq |\, f_1 \,|$. For $i > 1$, we have

$$\begin{aligned} U_i &\leq |\, f_{i-1} \,| + (2m/M)U_{i-1} \\ &\leq (1 + \mu(1 + 2\mu + \mu^{i-1}m))|\, f_{i-1} \,| \\ &\leq (1 + 2\mu + \mu^i m)|\, f_i \,|. \end{aligned}$$

Assume now that a total of $i + 2$ flows are computed, i.e., flows $f_0, \ldots, f_{i+1}$ are computed. Then we have $|\, f_i \,| < U_i/4$ and hence $(1 + 2\mu + \mu^i m) > 4$. Thus $\mu^i m > 1$ and hence $i \leq \log m/\log(M/2m)$. Since i is an integer we may round down.

We summarize the discussion.

Theorem 1 *If the scaling assumptions holds, if $L = U$, and if $\mu = 2m/M \leq 1/2$ then the iterative improvement scheme computes a flow f such that $(|\, f^* \,| - |\, f \,|)/|\, f^* \,|$ is bounded by 4μ. The scheme performs at most $2 + \lfloor \log m/\log(M/2m) \rfloor$ flow computations. All flow computations are performed with FPA.*

For $m \leq 10^6 < 2^{20}$ and single precision arithmetic and for $m \leq 10^9 < 2^{30}$ and double precision arithmetic and all augmenting path and preflow-push algorithms the conditions of the theorem are satisfied. The number of iterations is bounded by $2 + \lfloor 20/6 \rfloor = 5$ in the first case and by $2 + \lfloor 30/22 \rfloor = 3$ in the second case.

For the development above we assumed that all capacities are integers. This assumption is easily removed. If the capacities are given as floating point numbers that are not necessarily integral we simply scale up all capacities by a suitable power of two and then apply the above.

References

[AMO93] R. K. Ahuja, T. L. Magnanti, and J.B. Orlin. *Network Flows.* Prentice Hall, 1993.

[GT88] A.V. Goldberg and R.E. Tarjan. A new approach to the maximum-flow problem. *JACM*, 35:921–940, 1988.

[IEE87] IEEE standard 754-1985 for binary floating-point arithmetic. *SIGPLAN Notices*, 2:9–25, 1987.

[MN95] K. Mehlhorn and S. Näher. Leda, a platform for combinatorial and geometric computing. *Communications of the ACM*, 38:96–102, 1995.

[MNU97] Kurt Mehlhorn, S. Näher, and Ch. Uhrig. The LEDA User Manual (Version R 3.5). Technical report, Max-Planck-Institut für Informatik, 1997. http://www.mpi-sb.mpg.de/LEDA/leda.html.