Faster Algorithms for Frobenius Numbers

Dale Beihoffer
Lakeville, Minnesota, USA
dbeihoffer@frontiernet.net

Jemimah Hendry
Madison, Wisconsin, USA
jhendry@mac.com

Albert Nijenhuis
Seattle, Washington, USA
nijenhuis@math.washington.edu

Stan Wagon
Macalester College,
St. Paul, Minnesota, USA
wagon@macalester.edu

Submitted: Oct 10, 2004; Accepted: May 3, 2005; Published: Jun 14, 2005
Mathematics Subject Classifications: 05C85, 11Y50

Abstract

The Frobenius problem, also known as the postage-stamp problem or the money-changing problem, is an integer programming problem that seeks nonnegative integer solutions to $x_1a_1 + \cdots + x_na_n = M$, where a_i and M are positive integers. In particular, the *Frobenius number* $f(A)$, where $A = \{a_i\}$, is the largest M so that this equation fails to have a solution. A simple way to compute this number is to transform the problem to a shortest-path problem in a directed weighted graph; then Dijkstra's algorithm can be used. We show how one can use the additional symmetry properties of the graph in question to design algorithms that are very fast. For example, on a standard desktop computer, our methods can handle cases where $n = 10$ and $a_1 = 10^7$. We have two main new methods, one based on breadth-first search and another that uses the number theory and combinatorial structure inherent in the problem to speed up the Dijkstra approach. For both methods we conjecture that the average-case complexity is $O(a_1\sqrt{n})$. The previous best method is due to Böcker and Lipták and runs in time $O(a_1n)$. These algorithms can also be used to solve auxiliary problems such as (1) find a solution to the main equation for a given value of M; or (2) eliminate all redundant entries from a basis. We then show how the graph theory model leads to a new upper bound on $f(A)$ that is significantly lower than existing upper bounds. We also present a conjecture, supported by many computations, that the expected value of $f(A)$ is a small constant multiple of $\left(\frac{1}{2}\,n!\,\Pi A\right)^{1/(n-1)} - \Sigma A$.

1. Introduction: Computing the Frobenius Number

Given a finite *basis* $A = \{a_1, a_2, \ldots, a_n\}$ of positive integers, an integer M is *representable* in terms of the basis if there exists a set of nonnegative integers $\{x_i\}$ such that

$$\sum_{i=1}^{n} a_i x_i = M. \tag{1}$$

It is well known and easy to prove [Owe03, Ram∞] that there exists a finite largest *unrepresentable* integer, called the *Frobenius number* $f(a_1, a_2, \ldots, a_n) = f(A)$, if and only if $\gcd(a_1, a_2, \ldots, a_n) = 1$, which we assume throughout this paper. We make no assumptions about the ordering of the basis except as explicitly stated, since the Frobenius number is the same for any ordering of A. The monograph [Ram∞], which surveys more than 400 sources, is a tremendous collection of results that will be invaluable to anyone interested in the Frobenius problem.

Computing the Frobenius number when $n = 2$ is easy: A result probably known to Sylvester in 1884 [Syl84] showed that $f(a_1, a_2) = a_1 a_2 - a_1 - a_2$. While no such simple formula is known for the Frobenius number for any $n > 2$, Greenberg [Gre88] (see also [Dav94]) developed a quadratic-time algorithm for computing the exact Frobenius number when $n = 3$; this method is easy to implement and very fast. For the general problem one runs into a familiar barrier: Ramírez Alfonsín [Ram96] proved that computing the Frobenius number in the general case is $\mathcal{NP}$-hard under Turing reduction.

Beck, Einstein, and Zacks [BEZ03] reported that "The fastest general algorithm we are aware of is due to" Nijenhuis [Nij79], who developed a shortest-path graph model for the Frobenius problem. Nijenhuis used Dijkstra's algorithm with a binary heap priority queue (see [CLRS01]) to find all single-source shortest paths in his graph, which immediately yields the Frobenius number. The information in the full shortest-path table, which is readily generated by the Nijenhuis approach, provides an almost-instantaneous solution for any particular instance of (1). In this paper we refer to this algorithm, with the heap implementation, as the *ND algorithm*. Recent work of Böcker and Lipták [BL04] contains a new induction-based algorithm that is quite beautiful; we call it RR for Round Robin. RR is significantly faster than ND, but not as fast as our new methods. Traditionally, researchers have focused on the case where n is small relative to a_1, say $n \sim \log a_1$. However, RR is arguably the first method that works very well in the case that n is very large relative to a_1 (e.g., $n = a_1$). Our algorithms also work very well in such cases. A new and powerful algorithm has been developed by Einstein et al [ELSW∞]; their algorithm works when $n \leq 10$ but with no limit on the size of a_1.

In §2 we describe the graph used in the Nijenhuis model, which we call a *Frobenius circulant graph*. In §3, we exploit the symmetry of Frobenius circulant graphs to formulate two new algorithms to find shortest paths in which the weights along each path are in decreasing order. The first is an extremely simple breadth-first search algorithm (BFD) that can be implemented in just a few lines of *Mathematica* code. The second algorithm (DQQD) shortens the average length of the Dijkstra heap by representing path weights as ordered pairs (q, r) of quotients and remainders (mod a_1). Our methods can compute the Frobenius number for bases with $\min(A) \sim 10^7$ on a desktop computer with 512 Mb of memory. Memory availability is the primary constraint for these algorithms.

Because computing the exact Frobenius number is $\mathcal{NP}$-hard, upper and lower bounds are also of interest (see, *e.g.,* [BDR02, EG72, Sel77]). Krawczyk and Paz [Kra88] used H. W. Lenstra's integer programming methods to establish the existence, for every fixed n, of a polynomial-time algorithm to compute B yielding the tight bounds $B/n \leq f(a_1, \ldots, a_n) \leq B$. Improvements to their results can be found in [ELSW∞]. Kannan [Kan89] used similar methods to establish, for every fixed n, the existence of a polynomial-time algorithm to compute the exact Frobenius number. However, Kannan's algorithm has apparently never been implemented. Moreover, the Kannan algorithm does not solve instances of equation (1). The general instance-solving problem has been solved by Aardal and Lenstra [AL02] by an algorithm that works quite well (see [ELSW∞]). The algorithms of this paper will also solve instances and in some situations ($a < 10^5$) are faster than those of Aardal and Lenstra.

In §4, we use the symmetric properties of a Frobenius circulant graph and Greenberg's algorithm to construct a polynomial-time upper bound for all n. Although our general upper bound is not as tight as the Krawczyk–Paz theoretical bound for fixed n, it is almost always better than any of the other previously published upper bounds, usually by orders of magnitude. Relaxing the requirement of polynomial running time allows further significant improvements in the accuracy of the bound. Our algorithms are based on a proof that we can always divide out the greatest common divisor d_j from a j–element subset $A_j \subseteq A$ to obtain a reduced basis $\bar{A}_j$ for which $f(A) \leq d_j f(\bar{A}_j) + f(\{d_j\} \cup (A \backslash A_j)) + d_j$.

In §5, we investigate how well a lower bound of Davison for triples estimates the expected size of $f(A)$ and find that it does very well. Then we generalize the simple formula of Davison to all n and carry out experiments to learn that the new formula, $L(A)$, does indeed work well to estimate the expected value of $f(A)$. Our experiments indicate that the asymptotic expected value of $f(A)$ is $c_n L(A)$, where c_3 is near 1.5, and c_4 is near 1.35.

Acknowledgements. We are grateful to David Einstein, Christopher Groer, Joan Hutchinson, and Mark Sheingorn for helpful comments.

2. The Frobenius Circulant Graph Model

The Model

We will work modulo a_1, so that (1) reduces to

$$\sum_{i>1} a_i x_i \equiv M \,(\operatorname{mod}\, a_1)\,. \tag{2}$$

For a basis $A = (a_1, a_2, \ldots, a_n)$, define the Frobenius circulant graph $\mathcal{G}(A)$ to be the weighted directed graph with vertices $\{0, \ldots, a_1 - 1\}$ corresponding to the residue classes mod a_1; thus $\mathcal{G}(A)$ has a_1 vertices. We reserve u and v for vertices of $\mathcal{G}(A)$, so $0 \le u, v \le a_1 - 1$. The graph $\mathcal{G}(A)$ has a directed edge from vertex u to vertex v if and only if there is $a_k \in A \backslash \{a_1\}$ so that

$$u + a_k \equiv v \,(\text{mod } a_1)\,; \tag{3}$$

the weight of such edge is a_k. A graph on a_1 vertices that satisfies the symmetry property (3) is a *circulant graph*. (The customary definition of a circulant graph is that there is a set of integers J so that if the vertices of the graph are $v_1, \ldots, v_n$, then the neighbors of v_i are v_{i+j} for $j \in J$, where subscripts are reduced mod n.) The Nijenhuis model is a circulant graph with the additional symmetry that the edge weights of the incoming and outgoing directed edges are the same at every vertex — one each of the weights $A \backslash \{a_1\}$ — so any vertex in the model looks like any other vertex.

Let $G = \mathcal{G}(A)$. If p is a path in G that starts at 0, let e_i be the number of edges of weight a_i in p and let w be the total weight of the path. If the weight of the path is minimal among all paths to the same endpoint, then the path is called a *minimal path*. For any path p from vertex 0, repeated application of (3) shows that its weight w determines the end-vertex v:

$$v \equiv \sum_{i>1} e_i a_i = w \,(\text{mod } a_1)\,. \tag{4}$$

This means that, assuming $\gcd(A) = 1$, G is strongly connected: to get a path from u to v just choose M so large that $v - u + Ma_1 > f(A)$. Then there exists a representation $\sum_i e_i a_i = v - u + Ma_1$. A path corresponding to the left side will go from u to v.

Setting $e_i = x_i$ and $v \equiv M \,(\text{mod } a_1)$ obviously establishes a correspondence between a solution $\{x_i\}$ in nonnegative integers to (1) and a path p in G from 0 to v having total weight $w = \sum_{i>1} e_i a_i \equiv v \,(\text{mod } a_1)$. Since every path from 0 of length w ends at the same vertex, the model is independent of the order in which edges are traversed.

Using the Model to Construct the Frobenius Number

Let S_v denote the weight of any minimal path from 0 to v in the Frobenius circulant graph $G = \mathcal{G}(A)$ and suppose that $M \equiv v \,(\text{mod } a_1)$. Then we assert that M is representable in terms of the basis A if and only if $M \ge S_v$. By (4), $v \equiv S_v \,(\text{mod } a_1)$. If $M \ge S_v$, then M is representable in terms of A because $M = S_v + ba_1$ for some nonnegative b and $S_v = \sum_{i>1} e_i a_i$ with $e_i \ge 0$. Conversely, if $\sum_{i=1}^n a_i x_i = M$, then $\sum_{i>1} a_i x_i \equiv v(\text{mod } a_1)$; if then p is a path from 0 to v corresponding to weights x_i, $i > 1$, we have $S_v \le \sum_{i>1} x_i a_i \le M$.

The largest nonrepresentable integer congruent to $v \,(\text{mod } a_1)$ is $S_v - a_1$. The maximum weight minimal path within a graph G is known as the *diameter* $D(G)$, and it follows that

$$f\,(a_1, a_2, \ldots, a_n) = D(G) - a_1. \tag{5}$$

Thus to compute the Frobenius number using a Frobenius circulant graph, all we need to do is find the minimal path weights from 0 to each vertex, take the maximum of such path weights, and subtract a_1.

We define a *decreasing path* to be a path in which the weights of the edges along such path are (non-strictly) decreasing; i.e., each edge weight is less than or equal to the immediately preceding edge weight. Since the model is independent of the order in which the edges are traversed, every path in the graph can be converted to a decreasing path without affecting either its end vertex or total path weight. Thus finding all decreasing paths from 0 of minimum total weight is sufficient to identify all minimum weight paths. This substantially reduces the number of edge weight combinations that must be examined, which is one key to the efficiency of our new algorithms.

We define the unique smallest edge weight a_j occurring in the set of all minimal paths from 0 to a vertex v to be the *critical edge weight* for v. There is a unique critical path to each vertex v in which the weight of the incoming edge to any intermediate vertex u along such path is the critical edge weight for u; the truncation of such a critical path at u is obviously the critical path to u. Each edge along a critical path is a *critical edge* for its target vertex, which we sometimes denote informally by using just the appropriate edge-weight index j. It is easy to verify that every critical path is a decreasing path of minimal weight.

We call a shortest-path tree with root vertex 0 in $\mathcal{G}(A)$ a *Frobenius tree*. If we preserve the data generated in a Frobenius tree, such data can be used to provide a specific representation of any number $M > f(a_1, a_2, \ldots, a_n)$ in terms of the basis. There can be many Frobenius trees for a given basis, but the unique *critical tree* consists entirely of critical paths from 0 to every other vertex. In fact, the critical tree is isomorphic to the graph derived in a natural way from a fundamental domain for a tiling that is central to an algebraic view of the Frobenius problem; see [ELSW∞] for details.

Another definition of interest is that of a *primitive reduction* of a basis. A basis entry is *redundant* if it can be expressed as a nonnegative integer linear combination of the other entries. Then the *primitive reduction* of A is simply A with all redundant entries deleted. The algorithms we present later are capable of finding the critical tree, and that tree yields the primitive reduction because of the following lemma.

Lemma 1. Given a basis A of distinct entries, the primitive reduction of A equals the weights of edges with source 0 in the critical tree. Therefore the size of the primitive reduction is 1 greater than the degree of 0 in the critical tree.

Proof. If a_j is the weight of an edge in the critical tree, then the edge is a critical edge to some vertex v, and so a_j cannot be redundant, for a representation of a_j would yield a minimal path to v that had an edge of weight less than a_j. Conversely, if a_j is not redundant in A and $a_j \equiv v \pmod{a_1}$ with v a vertex, then the edge from 0 to v having weight a_j must be in the critical tree. For otherwise use the path back from v to the root to construct a representation of a_j by the following standard technique: The representation consists of the weights in the path and then enough copies of a_1 to make up the difference between a_j and S_v, the sum of the weights in the path; this requires knowing that $S_v \leq a_j$, but this must be so because of the existence of the single-edge-path from 0 to v with weight a_j. $\square$

A Concrete Example

We illustrate these ideas with the classic Chicken McNuggets® example. McDonald's restaurants in the USA sell Chicken McNuggets only in packages of 6, 9, or 20; thus one cannot buy exactly 11 or 23 McNuggets. Figure 1 shows the Frobenius circulant graph $G = \mathcal{G}(\{6, 9, 20\})$. The blue edges have weight 9, and connect vertices that differ by 3 (because $9 \equiv 3 (\mathrm{mod}\ 6)$). The thicker red edges have weight 20, and connect vertices that differ by 2 ($20 \equiv 2 (\mathrm{mod}\ 6)$).

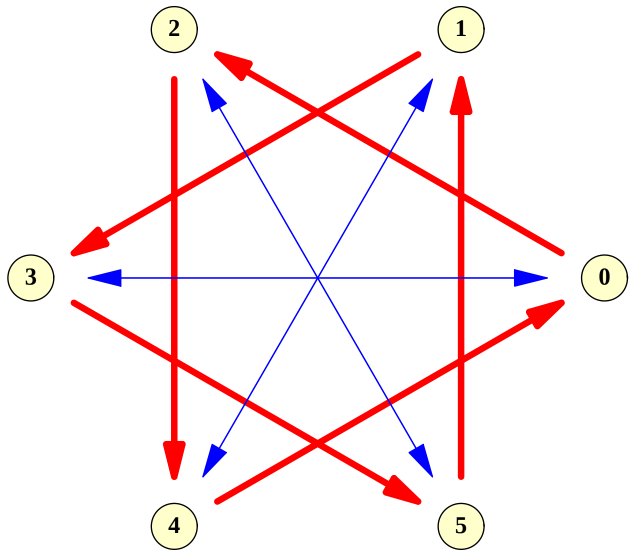

Figure 1: The circulant graph for the $(6, 9, 20)$ Frobenius problem. There are six red edges of weight 20 and six blue ones of weight 9.

Observe that $\gcd(6, 9) = 3$, and the edges of weight 9 partition G into the 3 disjoint sets of vertices $\{0, 3\}$, $\{1, 4\}$, and $\{2, 5\}$, which can be connected by edges of weight 20. Similarly, $\gcd(6, 20) = 2$, and the edges of weight 20 partition G into the 2 disjoint sets of vertices $\{2, 4, 6\}$ and $\{1, 3, 5\}$, which can be connected by edges of weight 9. We will make use of such partitions in §4 to develop new upper bound algorithms.

Figure 2 shows the minimal path $0 \to 2 \to 4 \to 1$ from vertex 0 to vertex 1, which includes $x_2 = 1$ edge of weight 9 and $x_3 = 2$ edges of weight 20, for a total path weight $S_1 = 49$; this particular path is decreasing, with weights 20, 20, 9. There are two other equivalent minimal paths (*i.e.*, $0 \to 3 \to 5 \to 1$ or $0 \to 2 \to 5 \to 1$), consisting of the same edge weights in a different order. Note that the path shown can be succinctly described as $(3, 3, 2)$ in terms of the weight indices.

The remaining minimal path weights are shown in Figure 2: $S_2 = 20$, $S_3 = 9$, $S_4 = 40$, and $S_5 = 29$. The Frobenius number is therefore $f(6, 9, 20) = D(G) - a_1 = 49 - 6 = 43$.

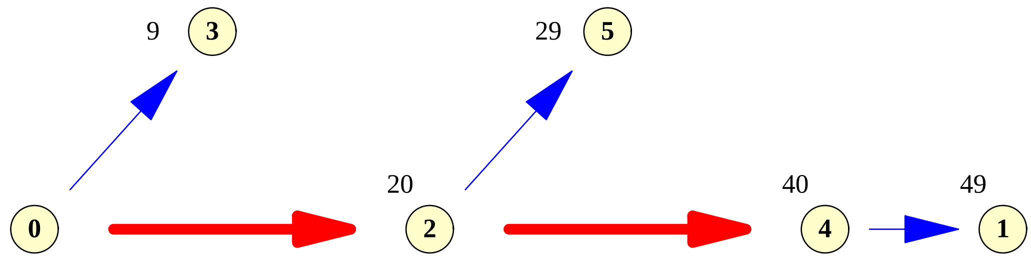

Figure 2: The critical tree for $\mathcal{G}(\{6, 9, 20\})$, showing minimal paths to each vertex. The diameter of the graph — the largest path-weight — is 49, so the "McNugget number" is $49 - 6$, or 43.

As noted in [Nij79], knowing a Frobenius tree for A allows one to solve any instance of (1). For suppose one has the vector containing, for each vertex, its parent in the minimum weight tree. This parent vector can then be used to get a solution (or confirmation that none exists) to any specific instance of $\sum_{i=1}^{n} a_i x_i = M$. Given M, let v be the corresponding vertex of the graph; that is $v \equiv M \pmod{a_1}$. Any representable M can be written as $x_1 a_1 + S_v$, so we can generate a solution by setting, $x_j (j \geq 2)$ equal to the number of edges of length a_j along a minimal path to vertex v, with $x_1 = (M - S_v)/a_1$. Here is a simple illustration using the McNugget basis, $A = \{6, 9, 20\}$. The parent vector is $\{-, 4, 0, 0, 2, 2\}$. For $6x_1 + 9x_2 + 20x_3 = 5417$, we have $5417 \equiv 5 \pmod 6$, so $v = 5$, $S_5 = 29$, and $5417 \geq 29$ (confirming that a solution exists). We trace the minimal path back up the tree from 5 to 0 ($5 \leftarrow 2 \leftarrow 0$) and count the number of uses of each edge length ($1 \cdot 9 + 1 \cdot 20 = 29$); then set $x_1 = (5417 - 29)/6 = 898$ to generate the solution $\{x_1, x_2, x_3\} = \{898, 1, 1\}$. The only time-consuming step here is the path-tracing: the worst-case would be a long path, but this typically does not happen and the computation is instantaneous once the tree data is in hand.

Existing Algorithms

The ND algorithm. The ND algorithm maintains a vertex queue for unprocessed vertices, which we implemented with a binary heap as described in [Nij79], and a list of labels $S = (S_v)_{v=0}^{a-1}$ of weights of paths. The vertex queue can be viewed as an ordered linear list such that S_v, the weight of the shortest path found so far to the vertex v at the head of the queue, is always less than or equal to S_u, the weight of the shortest path found so far to any other vertex u on the queue. Initially, the vertex in position 1 at the head of the queue is 0. Outbound edges from v are scanned once, when v is removed from the head of the queue, to update the queue. If the combined path of weight $w = S_v + a_i$ to vertex u is shorter than the current path weight S_u, then S_u is set to w and vertex u is added to the bottom of the queue (if it is not already in the queue) or moved within the queue, in either case by a leapfrogging process that moves it up a branch in a binary tree until it finds its proper place. No shorter path to v can be found by examining any subsequent vertex on the queue. The ND algorithm terminates when the queue is empty, at which point each of the $n - 1$ outbound edges from the a_1 vertices have been scanned exactly once.

The ND algorithm is a general graph algorithm and does not make use of any special features of the Frobenius context. We will show in section 3 how one can take advantage of the symmetry inherent in the Frobenius circulant graph to develop new algorithms, or to modify ND so that is becomes much faster. There is also the following recently published method that makes use of the special nature of the Frobenius problem.

The Round Robin method of Böcker and Lipták (RR [BL04]). This recently discovered method is very elegant and simple, and much faster than ND. We refer to their paper for details. For a basis with n elements, RR runs in time $O(a_1 n)$. All our implementations of RR include the redundancy check described in [BL04], which speeds it up in practice, though it does not affect the worst-case running time. One should consider two versions of this: the first, RR, computes the Frobenius number, but does not store the data that represents the tree; the second, which we will call RRTree, stores the parent structure of the tree. The second is a little slower, but should be used for comparison in the $n = 3$ case, where the tree is the only issue because Greenberg's method gets the Frobenius number in almost no time at all. The RRTree algorithm can find the critical tree, provided the basis is given in reverse sorted order (except that the first entry should remain the smallest).

3. New Algorithms

A Breadth-First Method

Throughout our discussion of algorithms we assume that the elements of A are distinct and given in ascending order; the time required to sort a basis is $O(n \log n)$.

Our simplest algorithm is a label-correcting procedure we call breadth-first decreasing (BFD) because the search is restricted to decreasing paths. We maintain a label vector $S = (0, S_1, S_2, \ldots, S_{a_1-1})$, in which each currently known minimal path weight to a vertex is stored. This vector is initialized by setting the first entry to 0 and the others to $a_1 a_n$ (because of Schur's bound; see §4). Vertices are processed from a candidate queue Q, starting with vertex 0, so initially $Q = \{0\}$. Vertices in the queue are processed in first-in-first-out (FIFO) order until the queue is empty. The processing of v consists of examining the outbound edges from v that might extend the decreasing path to v. Whenever a new shortest path (so far) to a vertex u is found (a "relaxation"), S_u is lowered to the new value and u is placed onto the queue provided it is not already there. The restriction to decreasing paths dramatically reduces the number of paths that are examined in the search for the shortest.

Here is a formal description of the BFD (breadth-first decreasing) algorithm. The restriction to decreasing paths is handled by storing the indices of the incoming edges in P and (in step 2b) scanning only those edges leaving v whose index is less than or equal to P_v.

BFD ALGORITHM FOR THE FROBENIUS NUMBER

Input. A set A of distinct positive integers $a_1, a_2, \ldots, a_n$.

Assumptions. The set A is given in sorted order and $\gcd(a_1, \ldots, a_n) = 1$. We take the vertex set as being $\{0, 1, ..., a_1 - 1\}$, but in many languages it will be more convenient to use $\{1, 2, \ldots, a_1\}$.

Output. The Frobenius number $f(A)$ (and a Frobenius tree of A).

Step 1. Initialize a FIFO queue Q to $\{0\}$; initialize $P = (P_v)_{v=0}^{a_1-1}$ a vector of length a_1, and set P_0 to n; let $S = (S_v)_{v=0}^{a_1-1}$ be $(0, a_1 a_n, a_1 a_n, \ldots, a_1 a_n)$; let $Amod$ be the vector A reduced mod a_1.

Step 2. While Q is nonempty:

- **a.** Set the current vertex v to be the head of Q and remove it from Q.
- **b.** For $2 \leq j \leq P_v$,
  - **i.** let u be the vertex at the end of the jth edge from v:
    $u = v + Amod_j$, and then if $u > a$, $u = u - a$.
  - **ii.** compute the path weight $w = S_v + a_j$;
  - **iii.** if $w < S_u$, set $S_u = w$ and $P_u = j$ and, if u is not currently on Q,
    add u to the tail of Q;

Step 3. Return the Frobenius number, $\max(S) - a_1$, and, if desired, P, which encodes the edge structure of the Frobenius tree found by the algorithm.

The queue can be handled in the traditional way, as a function with pointers to the head and tail, but we found it more efficient to use a list. We used h to always denote the head of the queue and t, the tail. Using Q_i for the ith element of the list, then Q_i is 0 if i is not on the queue and is the queue element following i otherwise. If t is the tail of the queue, $Q_t = t$. So enqueuing u (in the case that the queue is nonempty) just requires setting $Q_t = u$, $Q_u = u$, and $t = u$. Dequeuing the head to v just sets $v = h$, $h = Q_h$, and $Q_v = 0$. In this way $Q_v = 0$ serves as a test for presence on the queue, avoiding the need for an additional array. Because each dequeuing step requires at least one scan of an edge, the running time of BFD is purely proportional to the total number of edges scanned in 2b(i). We now turn to the proof of correctness.

Proof of BFD's Correctness. We use induction on the weight of a minimal path to a vertex to show that BFD always finds a minimal path to each vertex. Given a vertex v, let j be the index of the critical edge for v. Choose a minimal path to v that contains this critical edge, and sort the edges so that the path becomes a decreasing path to v; the path then ends with an edge of weight a_j. If u is the source of this edge, then the inductive hypothesis tells us that BFD found a minimal path to u. Consider what happens to P_u when S_u is set for the final time. At that time P_u was set to a value no less than j. For otherwise the last edge to u in the path that was just discovered would have been a_i, with $i < j$. But then the minimal path one would get by extending the path by the edge of weight a_j would have an edge smaller than a_j, contradicting the criticality of a_j. This means that when u is dequeued, it is still the case that $P_u \geq j$ (as there are no further resettings of P_u). So when the a_j-edge leaving u is scanned either (1) it produces the correct label for v, or (2) a minimal path to v had already been discovered. In either case, S_v ends up at the correct shortest-path distance to v. $\square$

The restriction to decreasing paths can be easily applied to the ND method by including a P-vector, keeping it set to the correct index, and using P_v to restrict the scan of edges leaving v leading to an NDD method; the preceding proof of correctness works with no change. While not as fast as BFD, it is still much faster than ND, as we shall see when we study running times and complexity.

Now we can describe an enhancement to BFD that turns out to be important for several reasons: (1) it is often faster; (2) it has a structure that makes a complexity analysis simpler; (3) it produces the critical tree. In BFD, the relaxation step checks only whether the new path weight, w, is less than the current label, S_u. But it can happen (especially when n is large relative to a_1) that $w = S_u$; then it might be that the last edge scanned to get the w-path comes from a_i, where $i < P_u$. In this case, we may as well lower P_u to i, for that will serve as a further restriction on the outbound edges when u is dequeued later. More formally, the following update step would be added as part of step 2b(iii). Note that we make this adjustment whether or not u is currently on the queue.

Update Step. If $w = S_u$ and $j < P_u$, set $P_u = j$.

This enhancement leads to an algorithm that is much faster in the dense case (meaning, n is large relative to a_1). At the conclusion of BFDU, the P-vector encodes the parent structure of the critical tree, which provides almost instantaneous solutions to specific instances of the Frobenius equation. Moreover, by Lemma 1, P gives us the primitive reduction of the basis.

With this update step included, the algorithm is called BFDU. And NDD can be enhanced in this way as well, in which case it becomes NDDU. The proof that BFDU finds the critical tree is identical to the proof of correctness just given, provided the inductive hypothesis is strengthened as follows: given vertex v, assume that for any vertex u whose minimal-path weight is less than that of v, BFDU finds the critical path to u. Then in the last line of the proof one must show that the path found to v is critical. But the update step guarantees that the critical edge to v is found when u is dequeued (whether or not the weight label of v is reset at this time).

The BFD algorithm is a variant of the well-known Bellman–Ford algorithm for general graphs. In [Ber93], the Bellman–Ford method is presented as being the same as BFD, except that all out-bound edges are examined at each step.

It takes very little programming to implement BFD or BFDU. For example, the following *Mathematica* program does the job in just a few lines of code. The queue is represented by the list `Q`, the While loop examines the vertex at the head of the queue and acts accordingly, the function `S` stores all the distances as they are updated, and the function `P` stores the indices of the last edges in the paths, and so needs only the single initialization at `P[a]`. We use `{1, 2, ..., a}` as the vertex set because set-indexing starts with 1. The weight of a scanned edge is `w` and its end is `e`; the use of Mod means that this could be 0 when it should be a, but there is no harm because `S[a]` is initialized to its optimal value, 0.

```
BFD[A_] := (Clear[S, P]; h = t = a = First[A]; b = Rest[A];
    Q = Array[0 & , a]; S[_] = a*A[[-1]]; S[a] = 0; P[a] = Length[b];
    While[h != 0, {v, Qh[[h]], h} = {h, 0, If[h == t, 0, Q[[h]]]};
      Do[e = Mod[b[[j]] + v, a]; w = b[[j]] + S[v];
        If[w < S[e], S[e] = w; P[e] = j;
           If[Q[[e]] == 0, If[h == 0, t = Q[[e]] = h = e,
                                     t = Q[[e]] = Q[[t]] = e]]],
            {j, P[v]}]];
    Max[S /@ Range[a - 1]] - a);

BFD[{6, 9, 20}]

43
```

For a simple random example with $a_1 = 1000$ and $n = 6$ this BFD code returns the Frobenius number about twice as fast as the ND algorithm (which takes much more code because of the heap construction). Figure 3 shows the the growth and shrinkage of the queue as well as the decrease in the number of live edges in the graph caused by the restriction of the search to decreasing paths. The total number of enqueued vertices is 1900.

```
BFD[{1000, 1476, 3764, 4864, 4871, 7773}] //Timing

{0.27 Second, 47350}
```

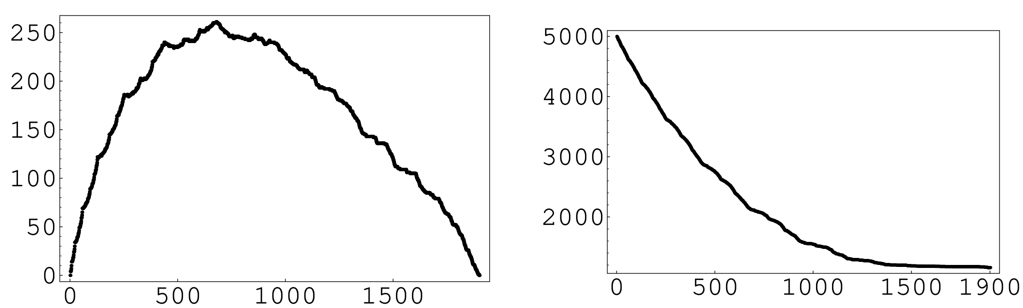

Figure 3: The left graph shows the rise and decline of the queue for an example with $a_1 = 1000$ and $n = 6$; a total of 1900 vertices were on the queue in all. The graph at the right shows the reduction in the live edges in the graph as the numbers of outbound edges become smaller because of the restriction to nonincreasing weights in the paths..

For a denser case the algorithm — BFDU in this case — behaves somewhat differently. Suppose $a_1 = 1000$ and $n = 800$. Then the graph has 799000 edges, but these get cut down very quickly by the restrictions P_u that develop (Figure 4, right).

The graph on the right in Figure 4 shows how quickly the graph shrinks. The average degree at the start is 799, but after 150 vertices are examined, the average degree of the

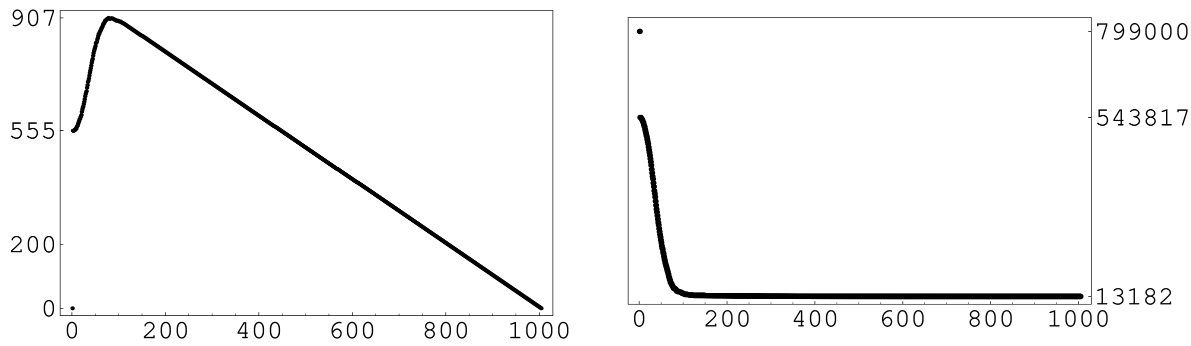

Figure 4: The left graph shows the rise and decline of the queue for BFDU working on an example with $a_1 = 1000$ and $n = 800$; a total of 1003 vertices were enqueued. The graph at the right shows the reduction in the live edges in the circulant graph; the shrinkage to a graph having average degree 13.2 occurs very quickly..

live graph is down to 15.4. A look at the distribution of the degrees actually used as each vertex is examined shows that half the time the degree is 6 or under, and the average is 14. This sharp reduction in the complexity of the graph as the algorithm scans edges and paths explains why BFDU is especially efficient in the dense case. One reason for this shrinkage is easy to understand. For the example given, after the first round, the neighbors of the root, 0, have restrictions on their outbound edges that delete $0, 1, 2, \ldots, 798$ edges, respectively, from the 799 at each vertex. So the total deletions in the first round alone are $(n-2)(n-3)/2$, or about 318000, a significant percentage of the total number of edges in this example (799000). Note that such shrinkage of the live graph occurs in ND as well, but in a linear fashion: as each vertex is dequeued, its edges are scanned and then they need never be scanned again. For this example, ND would remove 799 edges at each of the 1000 scanning steps.

Figure 5 shows the critical tree for the basis

$$\{200, 230, 528, 863, 905, 976, 1355, 1725, 1796, 1808\},$$

with nine colors used to represent the nine edge weights (red for the smallest, 230). The labels are suppressed in the figure, but the vertex names and their path-weights are enough to solve any instance of the Frobenius equation.

Bertsekas [Ber93] also presents several variations of the basic method, all of which try to get the queue to be more sorted, resulting in more Dijkstra-like behavior without the overhead of a priority queue. We tried several of these, but their performance was not very different than the simpler BFD algorithm, so we will not discuss them in any detail.

The BFD algorithm, implemented by the short code above, is capable of handling very large examples. It gets the Frobenius number of a 10-element basis with $a_1 = 10^6$ and $a_{10} \sim 10^7$ in about three minutes using *Mathematica*; ND works on such a case as well, but takes ten times as long.

Mathematica code for all the methods of this paper (ND, NDD, NDDU, BFD, BFDU, DQQD, DQQDU, and more) is available from Stan Wagon. As a final example, consider

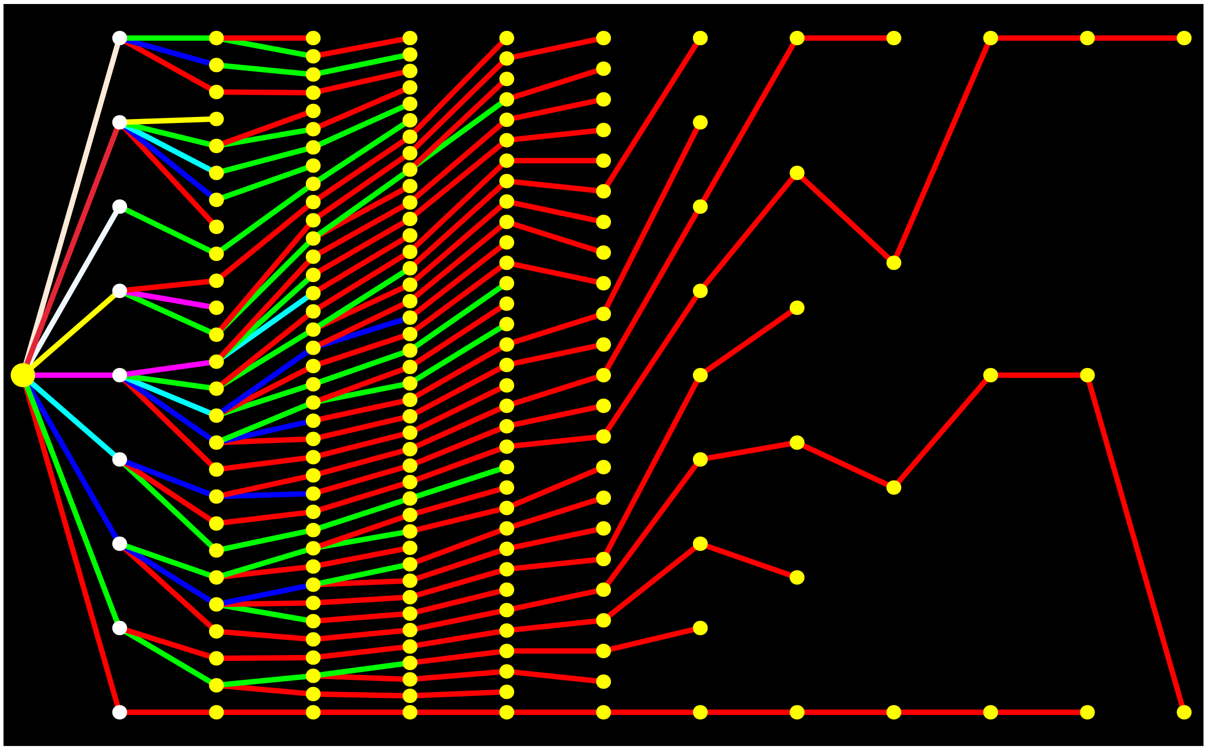

Figure 5: The critical tree for the basis $\{200, 230, 528, 863, 905, 976, 1355, 1725, 1796, 1808\}$, with different colors denoting different weights, red being the smallest (230). There are no redundant entries and this means that each weight occurs on an edge leaving the root. The white vertices are all the vertices in the graph $\mathcal{G}(A)$ that are adjacent to 0, the root.

a random basis A with $a_1 = 5000$ and having 1000 entries below 50000. Using BFDU to get the critical tree shows that most of the entries in such a basis will be redundant. In a 50-trial experiment the average size of the primitive reduction of A was 178 with a maximum of 196.

The Dijkstra Quotient Queue Method

The DQQD algorithm (Dijkstra quotient queue, decreasing) is a modification of the ND method that combines two new ideas: (1) a representation for the edge and path weights using ordered pairs of quotients and remainders (mod a_1); (2) a vertex queue based on the ordering by weight quotients of such ordered pairs. And of course we keep track of parents at the scanning step so that we continue to look only at decreasing paths.

1. Weight quotients as proxies for path weights. An edge weight $a_i = q_i a_1 + r_i$, with $2 \le i \le n$ and $0 \le r_i < a_1$, is represented by the ordered pair (q_i, r_i); we call q_i an *edge-weight quotient*. For any path from 0 of total weight $s = wa_1 + u$ and $0 \le u < a_1$, path weight s similarly can be represented by the ordered pair (w, u) with path weight quotient w; recall that in a Frobenius graph, such a path always ends at vertex u.

The current minimum weight discovered for a path from 0 to vertex u can be encoded implicitly by storing its weight quotient w as the entry in position u of the label vector S; retrieving $S_u = w$ in DQQD signifies that the current minimum weight path from 0 to u has weight $S_u a_1 + u = wa_1 + u$, so the weight quotient w can be used as a *proxy* for the actual path weight S_u. An important point is that if weight quotient w_1 is less than weight quotient w_2, then the corresponding path weights are in the same order, no matter which vertices are at the path-ends (i.e., regardless of the remainders r_i). This use of weight quotient proxies in the label vector S is not available in more general graphs, for which there is no consistent relationship between a vertex and the path to it.

All path weights encoded by weight quotients in S are unique. The same weight quotient w may appear in different positions S_u and S_v in S but the encoded path weights, for which S_u and S_v are proxies, are different: $S_v a_1 + v \neq S_v a_1 + u = S_u a_1 + u$. Finally, Schur's upper bound $f(A) \leq a_1(a_n - 1) - a_n$ (Corollary 4 to Theorem 1 in §4) allows us to use a_n as an upper bound for the maximum quotient value in S, so we can initialize $S = \{0, a_n, \ldots, a_n\}$.

In BFD, each path weight $w = S_v + a_i$ must be reduced $(\mathrm{mod}\ a_1)$ in order to identify vertex $u \equiv w \,(\mathrm{mod}\ a_1)$ for a comparison between w and the current path weight S_u. The equivalent operation in DQQD involves simpler addition/subtraction operations using the weight quotient proxy S_v and the (q_i, r_i) representation of a_i, but here two branches are needed to account for the possibility that $v + r_i$ may generate a "carry" $(\mathrm{mod}\ a_1)$. If $v + r_i < a_1$, the representation of p as (w, u) is satisfied by setting $u = v + r_i$ and $w = S_v + q_i$; then (w, u) correctly encodes the path weight $(S_v + q_i)a_1 + v + r_i = S_v + a_i = p$. On the other hand, if $v + r_i \geq a_1$, the condition $0 \leq u < a_1$ for the (w, u) representation of p requires that $u = v + r_i - a_1$ and $w = S_v + q_i + 1$; then (w, u) also correctly encodes the path weight $(S_v + q + 1)a_1 + u = (S_v + q)a_1 + v + r_i = S_v + a_i = p$. With this (w, u) representation, the path weight comparison by proxy in DQQD tests whether $w < S_u$.

2. A vertex queue based on weight quotients. The vertex queue in DQQD is constructed as a series of stacks in which each stack collects all vertices whose associated path weight quotients are the same: vertex v is pushed onto stack $Q(w)$ (for path weight quotient w) when a smaller-than-current-weight path from 0 to v is discovered of weight w. The current size of stack $Q(w)$ is stored as $L(w)$, so vertices can be easily pushed onto or popped off a stack. Initially, $Q(0) = \{0\}$ is the current (and only) nonempty stack, with $L(0) = 1$. The head of the vertex queue is the vertex at the top of the current stack; vertices are popped off for examination until the current stack is empty.

As with BFD, we maintain an auxiliary vector $P = (P_1, P_2, \ldots, P_{a_1 - 1})$ of indices of edges leading from parents to vertices, allowing us to consider only decreasing paths. If the examination of v identifies a path of weight quotient w to vertex u such that $w < S_u$, then vertex u is pushed onto stack $Q(w)$. When the algorithm is finished, P contains the information needed to construct a Frobenius tree.

DQQD also maintains a linear ordering for the nonempty stacks, in the form of an array Z of pointers, which is structured as an auxiliary priority queue. When the first vertex is pushed onto stack $Q(w)$ (*i.e.*, if $L(w) = 0$ at the start of the push), weight quotient w is inserted into Z. The head of priority queue Z is always the smallest remaining weight quotient for the nonempty stacks, and is used to identify the next stack to be processed. Initially, $Z = \{0\}$. The priority queue Z is shorter than the corresponding priority queue used in the ND algorithm, so the overhead costs of the DQQD priority queue are much smaller on average. As with ND, we use a binary heap for this auxiliary priority queue.

The combination of (a) the ordering of path weights by weight quotients, and (b) the processing of the stacks from smallest weight quotient to largest, is sufficient to ensure that no shorter path from an examined vertex v can be found by looking at any subsequent vertex on the stacks. This is the key feature of any label-setting algorithm (such as Dijkstra), in which each vertex needs to be processed only once. The priority queue in the ND algorithm identifies one such vertex at a time, while the DQQD priority queue identifies an entire stack of such vertices. This allows us to eliminate the update portion of the corresponding step in ND in which a vertex is moved to a new position on the vertex queue, which requires updating a set of pointers for the queue.

The complete vertex queue thus consists of the ordered set of nonempty stacks $\{Q(w_1), Q(w_2), \ldots\}$ for the remaining unprocessed $w_1 < w_2 < \cdots$ that have been placed on the auxiliary priority queue Z. Algorithm DQQD terminates when the vertex queue is empty and the last-retrieved stack $Q(h)$ has been processed, at which point the outgoing edges of each of the a_1 vertices have been scanned exactly once.

The complete DQQD method is formally described as follows. A proof of correctness for DQQD is essentially identical to that for BFD.

DQQD ALGORITHM FOR THE FROBENIUS NUMBER

Input. A set A of positive integers $a_1, a_2, \ldots, a_n$.

Assumptions. The set A is in sorted order and $\gcd(a_1, a_2, \ldots, a_n) = 1$. The vertex set is $\{0, 1, \ldots, a_1 - 1\}$.

Output. The Frobenius number $f(A)$ (and a Frobenius tree of A).

Step 1. Initialize

$S = (0, a_n, \ldots, a_n)$, a vector of length a indexed by $\{0, 1, \ldots, a_1 - 1\}$;
P, a vector of length a_1, with the first entry P_0 set to $n - 1$;
Q, a dynamic array of stacks, with $Q_0 = \{0\}$;
L, a dynamic array of stack sizes, all initialized to 0 except $L_0 = 1$;
$Z = \{0\}$, the auxiliary priority queue for weight quotients whose stack is nonempty;
Lists $Amod = \text{mod}(A, a_1)$, $Aquot = \text{quotient}(A, a_1)$.

Step 2. While Z is nonempty:

Remove weight quotient w from the head of Z;
While stack Q_w is nonempty:
a. Pop vertex v from Q_w;
b. For $2 \le j \le P_v$, do:
Compute the end vertex $u = v + Amod_j$
and its new weight quotient $w = S_v + Aquot_j$;

If $u \geq a_1$, set $u = u - a_1$ and $w = w - 1$;
If $w < S_u$,
push u onto stack Q_w;
set $S_u = w$ and $P_u = j$;
if w is not in the heap Z (i.e., $L_w = 0$), enqueue w in Z;
End While
End While

Step 3. Return $\max_v(S_v\, a_1 + v) - a_1$ and, if desired, P, the edge structure of the Frobenius tree found by the algorithm (with an adjustment needed for the root, 0).

This algorithm is a little complicated, but it can be viewed as a natural extension of the Dijkstra method that takes advantage of the number theory inherent in the Frobenius problem by its use of the quotient structure. The priority queue in DQQD tracks only distinct quotients, while the ND priority queue tracks all vertices, and this leads to considerable savings. Table 1 shows all the steps for the simple example $A = \{10, 18, 26, 33, 35\}$.

In the example of Table 1, the total number of quotients placed on the priority queue (q_{tot}) is 8, but the queue never contained more than 4 elements at any one time. The maximum size of the queue ($q_{\max}$) determines how much work is needed to reorganize the binary heap, which is $O(q_{\text{tot}} \log q_{\max})$. The number of vertices placed on the stacks is 11, as there was only one duplication (the 1 on stack indexed by 5). We will go into all this in more detail in the complexity section, but let us just look at two large examples. In a random case with $a_1 = 10^4$ and $n = 3$, $q_{\text{tot}} = 1192$ but $q_{\max} = 5$; the number of enqueued vertices was exactly 10000 and the total number of edges scanned was 10184 (compared to 20000 edges in the graph). In another case with the same a_1 but with $n = 20$, we get $q_{\text{tot}} = 28$, $q_{\max} = 9$, the number of enqueued vertices was 10399 and the number of edges examined was 17487 (out of 190,000). The following three points are what make the performance of DQQD quite good: (1) the heap stays small; (2) very few vertices are enqueued more than once; and (3) the number of outbound edges to be scanned is small because of the restriction to decreasing paths.

As with BFD, we can enhance DQQD to DQQDU by updating (lowering) P_v whenever a path weight exactly equal to the current best is found. But there is one additional enhancement that we wish to make part of DQQDU (but not DQQD), which makes DQQDU especially efficient for dense bases. Whenever the first examination of a vertex v is complete, we set $P_v = -P_v$. Then, whenever a vertex u is examined, we can check whether $P_u < 0$; if it is, then the edge-scan for u can be skipped because all the edges in question have already been checked when u was first examined (with the same weight-label and P-value on u, since those cannot change once the first examination of u is complete). At the end, the P-vector contains the negatives of the indices that define the tree. This enhancement will save time for vertices that appear more than once in the stacks (such as vertex 1 in the chart in Table 1). The proof of correctness of BFDU carries over to this case, showing that DQQDU finds the critical tree.

Contents of the priority queue for quotients	Examined vertex	P_v	Edges scanned. New weight quotients requiring stack entry are in subscript	Quotients indexing stacks	Stack bottom	Stack entry	Stack entry
0	0	4	$0 \to \{8_1, 6_2, 3_3, 5_3\}$	0	0		
$1, 2, 3$	8	1	$8 \to \{6\}$	1	8		
$2, 3$	6	2	$6 \to \{4_4, 2_5\}$	2	6		
$3, 4, 5$	5	4	$5 \to \{3, 1_6, 8, 0\}$	3	3	5	
$3, 4, 5, 6$	3	3	$3 \to \{1_5, 9_5, 6\}$	4	4		
$4, 5, 6$	4	1	$4 \to \{2\}$	5	2	1	9
$5, 6$	9	2	$9 \to \{7_7, 5\}$	6	1		
$5, 6, 7$	1	1	$1 \to \{9\}$	7	7		
$5, 6, 7$	2	2	$2 \to \{0, 8\}$				
$6, 7$	1	1	$1 \to \{9\}$				
7	7	1	$7 \to \{5\}$				
—							

Table 1: All the steps of the DQQD algorithm for $A = \{10, 18, 26, 33, 35\}$. The values P_v indicate how many edges to scan. Subscripts are assigned only to those yielding new weight quotients. Only one extra vertex is placed on the stacks (vertex 1). The graph has $10 \cdot 4 = 40$ edges, but only 21 of them are scanned. Of those 21, 10 led to lower weights. The final weight quotients (the subscripts) are $(0, 5, 5, 3, 4, 3, 2, 7, 1, 5)$. These correspond to actual weights $(0, 51, 52, 33, 44, 35, 26, 77, 18, 59)$, so the Frobenius number is $77 - 10 = 67$. The final P-vector is $(-, 1, 2, 3, 1, 4, 2, 1, 1, 2)$ which gives the index of the edge going backward in the final tree. Thus the actual parent structure is $(-, 3, 6, 0, 6, 0, 0, 9, 0, 3)$.

Running Time Comparisons

Previous researchers on Frobenius algorithms ([Gre99, AL02, CUWW97]) have looked at examples where $n \leq 10$, $a_1 \leq 50000$, and $a_n \leq 150000$ (except for the recent [BL04] which went much farther). For a first benchmark for comparison, we use the five examples presented by Cornuejols et al and the 20 of Aardal and Lenstra. Fifteen of these 25 examples were specifically constructed to pose difficulties for various Frobenius algorithms; the other 10 are random. Throughout this paper we call these 25 problems prob$_i$, with $1 \leq i \leq 5$ being the CUWW examples. Table 2 shows the running times (all times, unless specified otherwise, were using *Mathematica* (version 5) on a 1.25 GHz Macintosh with 512 MB RAM) for each of the algorithms on the entire set of 25 problems. The clear winner here is DQQD.

We learn here that the constructed difficulties of the CUWW examples wreak havoc on BFD, but cause no serious problems for the other graph-based methods, or for Round Robin. We have programmed some of these methods in C++ and the times there tend

Method	BFD	ND	NDD	RRTree	RR	DQQD
Time (secs)	1365	326	248	130	84	50

Table 2: Comparison of six algorithms on 25 benchmark problems. The Dijkstra-based ND is much slower than the others (except BFD). The Dijkstra Quotient Queue Decreasing algorithm is the fastest.

to be about 100 times faster. For example, ND in C++ took only 2.9 seconds on the benchmark. However, the ease of programming in *Mathematica* is what allowed us to efficiently check many, many variations to these algorithms, resulting in the discovery of the fast methods we focus on, BFD and DQQD. Recently Adam Strzebonski of Wolfram Research, Inc., implemented DQQDU in C so as to incorporate it into a future version of *Mathematica*. It is very fast and solves the entire benchmark in 1.15 seconds.

The computation of the Frobenius numbers was not the main aim of [AL02] (rather, it was the solution of a single instance of (1), for which their lattice methods are indeed fast), but they did compute all 25 Frobenius numbers in a total time of 3.5 hours (on a 359-MHz computer). On the other hand, our graph methods solve instances too, as pointed out earlier; once the tree structure is computed (it is done by the P-vector in all our algorithms and so takes no extra time), one can solve instances in an eyeblink. Using the million-sized example given earlier, it takes under a millisecond to solve instances such as $\sum_{i=1}^{10} x_i a_i = 10^{100} - 1$. A solution is $(10^{94} - 112, 7, 8, 1, 1, 2, 1, 1, 1, 1)$. As a final comparison, in [CUWW97] the first five problems of the benchmark were handled on a Sun workstation in 50 minutes.

The benchmark problems are not random, so now we turn our attention to random inputs, to try to see which algorithm is best in typical cases. Our experiment consisted of generating 40 random sorted basis sets $\{a_1, a_2, \ldots, a_n\}$ with $a_1 = 50000$ and $a_n \leq 500000$. When basis entries are randomly chosen from $[a_1, 10a_1]$, we say that the *spread* is 10; unless specified otherwise, we use this value in our experiments. This is consistent with examples in the literature, where the basis entries have roughly the same size. The RR algorithm is somewhat dependent on the factorization of a_1, so for that case we used random a_1 values between 47500 and 52500; for the other algorithms such a change makes no difference in running time. The size of the bases, n, ran from 3 to 78. If n is 2 or 3 and one wants only the Frobenius number, then one would use the $n = 2$ formula or Greenberg's fast algorithm, respectively. But the graph algorithms, as well as the round-robin method, all give more information (the full shortest path tree) and that can be useful. Figure 6 shows the average running times for the algorithms. As reported in [BL04], RR is much better than ND. But BFD and DQQD are clear winners, as the times hardly increase as n grows (more on this in the complexity section). For typical basis sizes, DQQ is a clear winner; for very short bases RR is fastest. The RR times in this graph are based on the simplest RR algorithm for computing only $f(A)$; RRTree requires a little more time, but produces the full tree.

The dense case is a little different, for there we should use the update variations of our algorithms (as described earlier) to take account of the fact that there will likely be

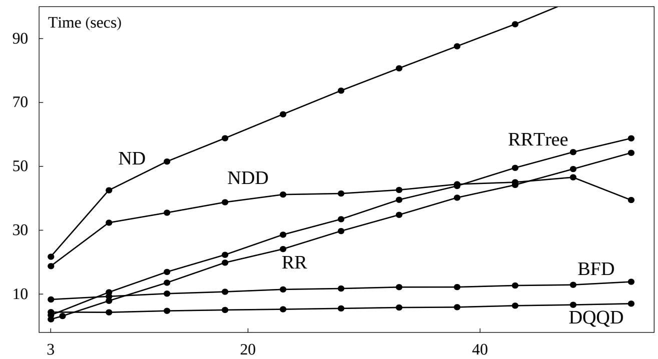

Figure 6: The average running times of several algorithms on random bases with $a_1 = 50000$ and spread 10, and with n between 3 and 53; 40 trials were used for each n. The flat growth of the times for NDD, BFD, and DQQD is remarkable.

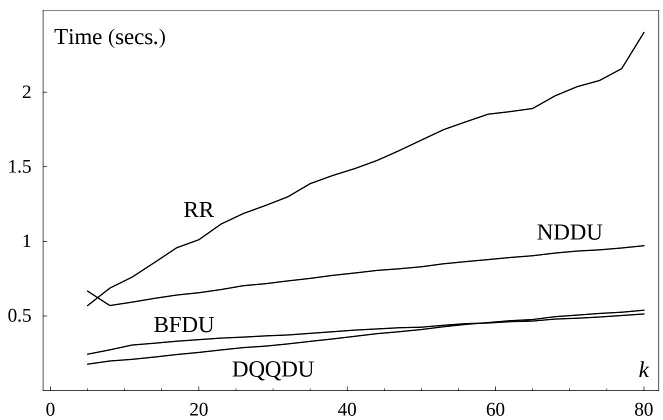

Figure 7: Timings in the dense case, where a_1 is near 1000 and n runs from $\lfloor 5 \log_2 a_1 \rfloor$ to $\lfloor 80 \log_2 a_1 \rfloor$. Even though RR can efficiently spot redundant basis entries, our graph methods are faster.

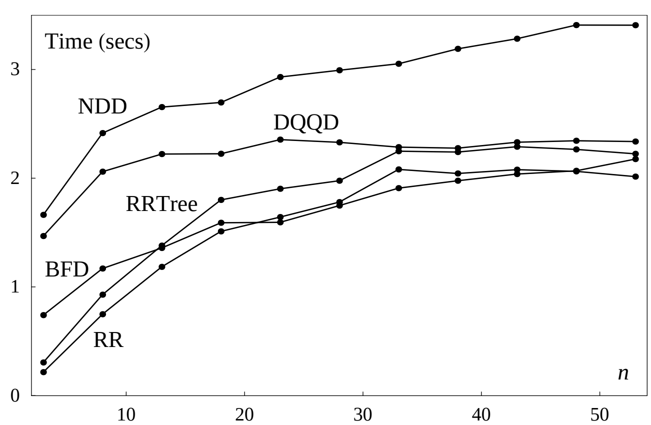

Figure 8: The average running times of several algorithms on random bases with $a_1 = 5000$ and the other entries between 5000 and 10^{100}.

several minimal paths having the same weight. No such update enhancement is available for RR. Figure 7 shows an experiment where the inputs were generated as follows: a_1 takes on all 100 values from 950 to 1049 with $n = k\lfloor \log_2 a \rfloor$, and k varying from 5 to 80; thus the basis size gets up to about 800. We used the exact same bases for the three algorithms. We see that BFDU and DQQDU are essentially tied, again showing a very small slope as the basis size increases.

Finally, one can wonder about very large spreads. To take an extreme case, suppose $a_1 = 5000$ with the other basis entries chosen between 5001 and 10^{100}. In this case the quotient structure that is the foundation of DQQD disappears, in the sense that all the quotients will likely be different; this means that the priority queue becomes in essence a priority queue on the vertices, and DQQD becomes essentially the same as NDD. Figure 8 shows the result of an experiment with 60 trials for each instance.

From these experiments, and many others, we learn that for normal spreads of around 10 the Round Robin algorithm is best when n is very small, DQQD is best for medium-sized n, and DQQDU and BFDU are essentially tied in the very dense case when n is large relative to a. As the spread grows, the performance of DQQD worsens. The most noteworthy feature of many of the timing charts is how slowly the running times of BFD and DQQD increase as n does.

Complexity

Here we use the "random access" model of complexity, as is standard in graph theory, whereby the sizes of the integers are ignored and the input size is taken to be the numbers of edges and vertices in the graph. This is inappropriate in number theory, where bit complexity is used and the input size is the true bit-length of the input. The Frobenius problem straddles the areas, so the choice of model is problematic. But since the integers that would be used in a shortest-path approach will almost always be under a billion, it seems reasonable to ignore issues relating to the size of the integers. Note that this would not be the right approach for Greenberg's algorithm when $n = 3$, since that algorithm is pure number theory and works well on integers with hundreds or thousands of digits; that algorithm is $O(N^2)$ in the bit-complexity sense, where N is the strict input length of the 3-element basis.

If we view the algorithms as finding only the Frobenius numbers, then all the algorithms discussed here have exponential time complexity, since they involve more than a steps (where we use a for a_1, the smallest entry of the basis). However, all the algorithms of this paper produce the entire Frobenius tree, which, in addition to giving $f(A)$, allows one to quickly determine, for an integer M, whether M is representable in terms of A, and, if so, to find a representation of M. Since the tree has a vertices, it makes sense to view the complexity as a function of a, the size of the output. Thus an algorithm that runs in time $O(a)$ can be viewed as running in linear time, which is best possible up to a constant. An algorithm running in time $O(ag(n, a))$ where $g(n, a)$ is small can be viewed as being nearly optimal. The updated U versions of our algorithms also determine, via the critical tree, all the redundant entries in a basis, something that cannot be done in fewer than a steps.

For all our experiments we use data that has distinct classes mod a. In particular, this means that $n \le a$. If we were given a purely random set of integers, a sort on the residues could be used to quickly eliminate duplications. The resulting tree is just as good in the sense of solving the main problems. However, such a reduction can lead to a loss of efficiency when computing representations. That is why we have 25¢ coins in addition to 5¢ and 10¢ coins.

When considering the complexities of our methods we all always use the updated "U" versions, since that always has fewer scans of outbound edges, despite a slower actual running time when n is small; thus we focus on NDDU, BFDU, and DQQDU. The complexity of these algorithms depends on the number of times they look at vertices or edges. Let σ denote the total number of scans of outbound edges in ND, NDDU, BFDU, or DQQDU. Let λ denote the number of times a weight label gets overwritten by a smaller number. For BFDU or DQQDU, λ is the number of vertices that are removed from the queue (stacks for DQQDU). For Dijkstra, λ has been called the number of updating steps in the literature, and is the number of times the heap needs to be reorganized. A final parameter is q, for the total number of quotients placed on the priority queue in DQQDU.

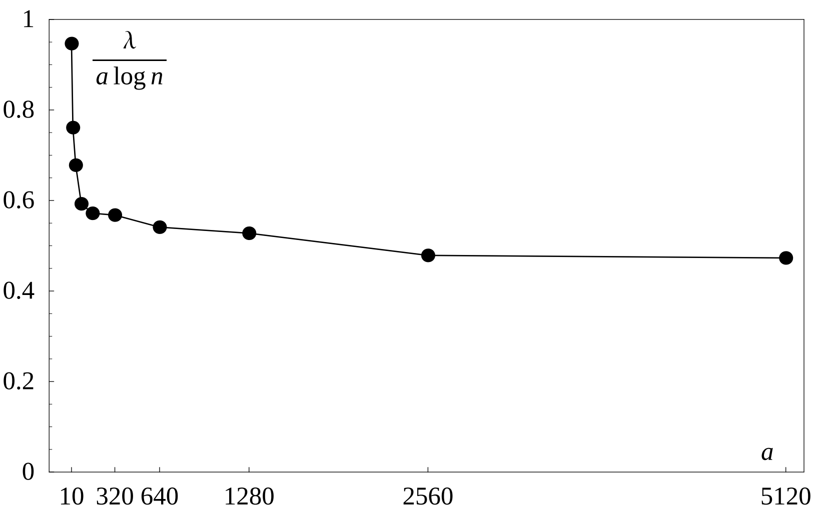

Figure 9: The average value of λ, the number of update steps, over 10000 trials for the Nijenhuis–Dijkstra algorithm, with n set to $\lfloor \log_2 a \rfloor$. The data points to the asymptotic average of λ being near $0.5a \log n$.

Nijenhuis–Dijkstra. The worst-case complexity of the binary heap form of Dijkstra's algorithm for a graph with v vertices and E edges is $O(E \log V)$, which is $O(na \log a)$ in the Frobenius situation [CLRS01]. We do not consider the theoretically faster Fibonacci heap version of Dijkstra, since that is difficult to implement. But Dijkstra is known to have average-case running time better than the worst case (see [Nos85], where the Dijkstra average problem for undirected graphs is discussed). A modest set of experiments shows that Dijkstra on random graphs with the same average degree as the Frobenius circulant graphs takes about twice as many updating steps (its λ is twice as large). This supports the view that the ND Frobenius algorithm should perform better than Dijkstra on general graphs. In an experiment with a random and n chosen from $[3, a]$, counts of the updating steps in the Frobenius case suggest that λ is asymptotically about $0.5a \log n$. Since the binary heap requires $O(\log a)$ steps for each update, this leads to an overall average time estimate of $c\sigma + k\lambda \log a = O(an + a \log a \log n)$; the first term comes from the necessary scan of all the edges ($\sigma = (n-1)a$) and the second comes from the updating steps. Figure 9 shows the results of an experiment with $n = \lfloor \log_2 a \rfloor$.

Round Robin. The Round Robin algorithm of [BL04] has a proved worst-case running time of $O(na)$, and so is a significant improvement on ND. The basic RR algorithm without a redundancy check always requires time $\Theta(na)$. The version that checks for redundancy will require time $\Theta(n_1 a)$, where n_1 is the size of the primitive reduction of A; in the densest case, when $n = a$, n_1 is well under n, but it is possible that, on average, the ratio of the two is asymptotically a constant.

Breadth-First Decreasing Updated (BFDU). The running time is directly proportional to σ, the number of edges scanned. The number of examined vertices, λ, is not relevant because σ increases by at least 1 each time a vertex is examined, but in fact the excess of λ over a is not great. This explains why BFD gets good performance: the excess vertex examinations is 0 in the ND cases but the heap manipulation is slow; the excess in BFDU is not far from 0 and the heap is eliminated. We conjecture that the asymptotic average of σ is bounded by $O((a+n)\sqrt{n})$. In many cases this seems too conservative, but this is the simplest form of a function that appears to bound σ in all cases. As evidence for this conjecture, we present the results of some experiments in Figures 10 and 11. Figure 10 shows that as n varies the linear dependence of σ on a is very strong. Figure 11 shows that the dependence of the slopes on n is linear in a log-log plot, and so follows a power rule with power near 0.4. A similar estimate for the residual after the linear term is subtracted from σ leads to an estimate of the constant term, and so one can say that σ behaves like $0.35an^{0.397} + 5.26n^{1.394}$. Since this is just an experiment, we may as well simplify it to $(a + n)\sqrt{n}$.

Figure 12 shows how this bound works in some special cases, such as $n = a$ or $n = \log_2 a$. The square root might well be an overestimate in some special cases (see the dashed curve in the logarithmic case), but in any case the complexity assertions are consistent with the running time experiments.

Nijenhuis–Dijkstra Decreasing Updated (NDDU). As with ND, each update of the binary heap requires $O(\log a)$ steps on average, but λ, the number of such updates, goes down dramatically with the restriction to decreasing paths. We always have $\lambda \geq a$, so let us call $\lambda - a$ the excess number of label-setting steps. In most experiments the excess was very close to n; for example, in 100 trials with $a = 1000$ and $n = 10$ the excess was between 2 and 86 with an average of 24. Yet this slowly grows as n and a do, and it can be quite large in the densest case, $n = a$. Thus we postulate $n\sqrt{a}$ as an upper bound on the average excess. In ND this excess is $a \log n$ (Fig. 9). As with BFDU, experiments to determine σ lead to the conclusion that on average $\sigma \leq (a + n)\sqrt{n}$. In fact, σ for

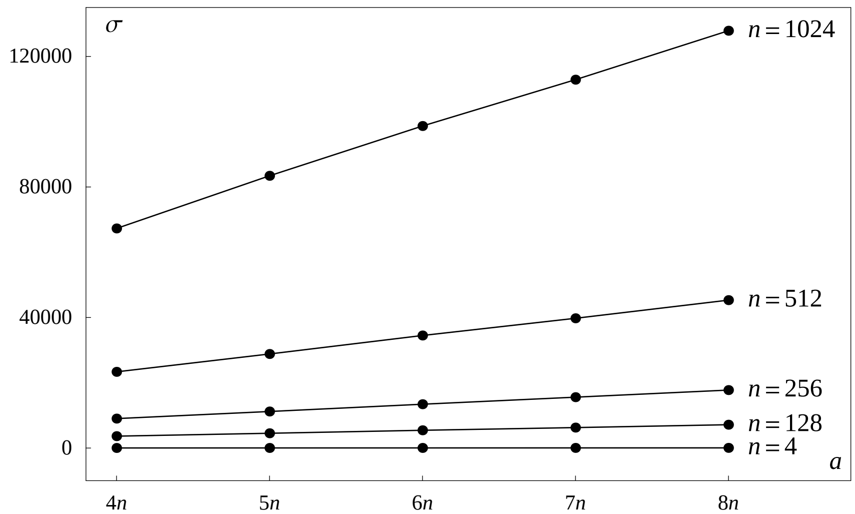

Figure 10: This plot shows σ as a function of a, where a varies from 4n to $8n$ and n varies from 4 to 1024. Each data point is the average of 300 trials. The linearity is clear, supporting a model of the form $\sigma \sim ma + b$ where m and b depend only on n.

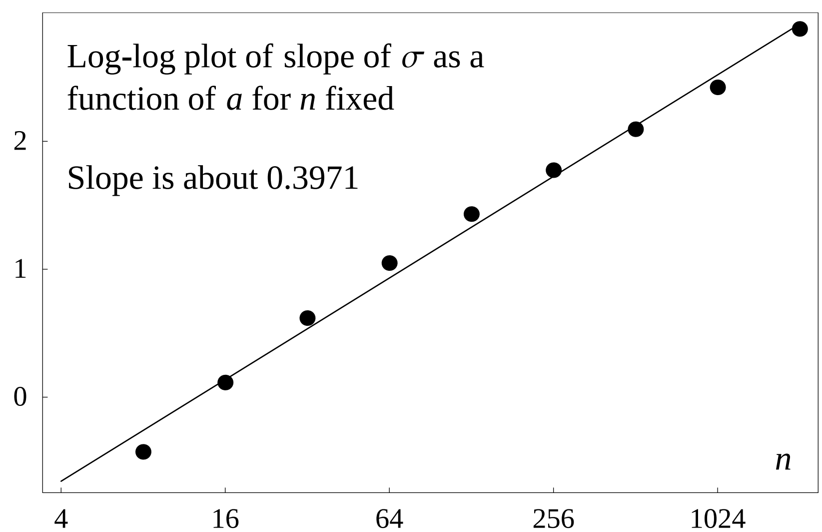

Figure 11: This double-log plot of the slopes of the lines in Figure 10 is close to linear, showing that the slopes depend on a power of n. The slope of the line is about 0.4, leading to a model of the form $\sigma \sim can^{0.4} + b$, where c is a constant and b depends on n.

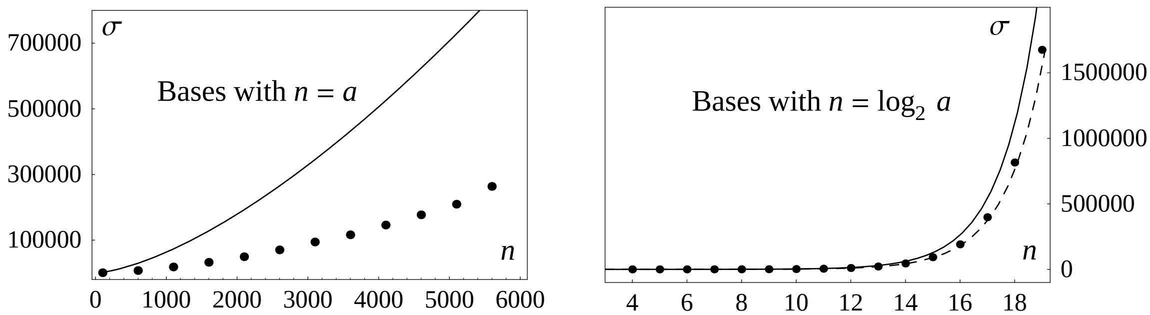

Figure 12: Two experiments showing that the model $(a+n)\sqrt{n}$ works as an upper bound on the average value of σ. The dots in the graph on the left show σ as a function of n for the densest case, where $n = a$. The dots on the right arise from the more typical case where $n = \lfloor \log_2 a \rfloor$. The dashed line on the right is a graph of $a \log n$ which also does a reasonable job in this case.

NDDU cannot be larger than than σ for BFFDU by the following argument. When a vertex v is pulled off the heap, the value of P_v is equal to its final value; this is because any vertex dequeued later has weight label no smaller than that of v, and so cannot lead to a path of minimal weight to v, which is the only way that P_v could be changed. This means that σ for NDDU, being the sum of the P-values, is exactly equal to the total of all the edge-indices in the critical tree. But the total of the edge indices in the critical tree is also a lower bound on what σ for BFDU can be, since in BFDU the value of P_v at the time v has its weight label set for the last time must be at least as large as its final (critical) value. Taking all terms into account, we find that his leads to an average time complexity estimate of $O(\sigma + \lambda \log a) = O((a+n)\sqrt{n} + (a + n\sqrt{a}) \log a)$.

Dijkstra Quotient Queue Decreasing Updated (DQQDU). The running time of DQQDU has the form $O(\sigma + (\lambda - a) + q \log q)$ where only the excess is considered in the label-correcting count because there are always at least a such steps and each of those leads to a vertex being pushed onto a stack and so at least one edge scan when it is popped; thus a of the label updates are accounted for in σ. The amount of work for the excess vertex pops is very small, because the algorithm will discover immediately that $P_v < 0$; nevertheless, each pop counts as one step. As with NDDU, this excess in λ — the number of times a label is set (and so a vertex is pushed onto a stack) beyond the minimum of a — is generally quite small but can get large in the densest case and again $n\sqrt{a}$ works as a conservative bound.

The work of the binary heap is measured by q_{tot}, which is the total number of quotients placed on the priority queue, and $q_{\max}$, which is the maximum number on the queue at any one time; the total work for the priority queue is $O(q_{\text{tot}} \log q_{\max})$. We will restrict our experiments to the case of spread 10, since as the spread grows the quotients are less likely to be distinct and DQQDU becomes essentially the same algorithm as NDDU. In the extreme case that $n = 2$, q_{tot} is always exactly $a - 1$ and $q_{\max} = 1$ (these are easy to see because every vertex has degree 1 and the paths all have weight $k \cdot b$). This case is of little interest, since there are simple ways to get $f(A)$ and the whole Frobenius tree for

$A = \{a, b\}$. When $n = 3$, the growth of q_{tot} becomes sublinear, and various experiments show that the following simple function serves as a bound: $\sqrt{a}\, a^{1/n}$. We need an estimate on $q_{\max}$ as well and it appears that, for any a and n, its expected value is not greater than 10. This means that the expected amount of heap work is simply $O(\sqrt{a}\, a^{1/n})$.

The edge-count σ must be the same for DQQDU as for NDDU. This is because, as with NDDU, once a vertex has been dequeued in DQQDU its P-value cannot change (because new path weights are greater by at least a_2, and so have greater weight quotients); therefore it has its final P-value and σ is again just the sum of edge-indices in the critical tree. Therefore we can use $(a+n)\sqrt{n}$ to bound σ and give an estimate of the average time complexity of DQQDU as $O((a+n)\sqrt{n}+n\sqrt{a}+a^{\frac{1}{2}+\frac{1}{n}})$. This simplifies to $O(a\sqrt{n}+n\sqrt{a})$, but because $n\sqrt{a} \le a\sqrt{n}$ (we are assuming $n \le a$), this becomes simply $O(a\sqrt{n})$.

Table 3 summarizes the complexity situation. All the functions refer to estimates of asymptotic average-case behavior, with the exception of Böcker and Lipták's RR, for which the $O(a\,n)$ result is proved worst-case behavior. For the traditionally studied case where n is small, say $n = \lfloor \log_2 a \rfloor$, the estimated complexity of BFDU and DQQDU is a significant improvement over that of ND or RR. In this case, and in the dense case, the complexity function for NDDU is at least as good as that for RR, and we saw earlier (Figs. 6, 7) that NDD and NDDU are in fact competitive with the other methods. Keeping in mind that it takes a steps just to write down a complete Frobenius tree, we see that DQQDU and BFDU do the job in near-optimal time when $n = \log a$. If n is fixed, then RR, DQQDU, and BFDU all work in time that is linear in terms of the output size.

Method	General	$n = a$	$n = \log a$	$n = 10$
ND	$O(an + a\log a \log n)$	$O(a^2)$	$O(a \log a \log\log a)$	$O(a \log a)$
RR	$O(an)$	$O(a^2)$	$O(a\log a)$	$O(a)$
NDDU	$O(a\sqrt{n} + a\log a + n\sqrt{a}\log a)$	$O(a^{3/2}\log a)$	$O(a\log a)$	$O(a\log a)$
DQQDU	$O(a\sqrt{n})$	$O(a^{3/2}\log a)$	$O(a\sqrt{\log a})$	$O(a)$
BFDU	$O(a\sqrt{n})$	$O(a^{3/2})$	$O(a\sqrt{\log a})$	$O(a)$

Table 3: The conjectured asymptotic average-case time complexity. The RR row has proved worst-case bounds. For DQQDU, it is assumed that the basis has spread 10 or less. In terms of the size of a Frobenius tree, the bounds for RR, DQQDU, and BFDU are optimal when n is a fixed integer. When $n = \lfloor \log_2 a \rfloor$ the bounds for DQQDU and BFDU are close to optimal.

Things might be different in the worst case. For example, the first test case of Cornuejols et al. [CUWW97] causes BFD to scan 3756443 edges, compared to the estimate of 27331. This is 137 times greater than the average behavior of σ, showing that BFD can perform badly in specific cases. The same might be true of our other algorithms, which is why we focus on average-case behavior.

Summary

Our methods are constrained by memory and currently work so long as $a < 10^7$, which is several orders of magnitude greater than previously published algorithms (except for

Round Robin, which can also handle such large numbers). For larger a, if $n \leq 10$ then the lattice methods of [ELSW∞] can be used. The comments that follow assume this condition on a. While our label-correcting algorithms were inspired by simple label-correcting algorithms for general graphs, such as those discussed in [Ber93], enhancements that make use of (a) the Frobenius symmetry, and (b) the quotient-residue structure of the graph yield large speedups. The algorithms can also compute the critical tree, solve any instance of the Frobenius equation, and eliminate all redundant entries from a basis.

Our guidelines as to which algorithms to use to find the Frobenius number are as follows.

1. When $n = 3$, Greenberg is by far the fastest.

2. When $n = 4$, Round Robin is best.

3. For intermediate values of n, such as those that appear in examples in the literature, DQQD is best.

4. For very dense data BFDU or DQQDU should be used.

5. For data that is not random, and might be specifically designed to be difficult (such as the CUWW examples) BFD should be avoided. Round Robin's worst-case time bound means that its performance is independent of the nature of the input. DQQD performed very well in the specific difficult cases from the literature.

6. Because of the binary heap, NDD and NDDU are not terribly fast, but neither are they terribly slow. They are simple in that they require only one or two very small enhancements to the classic ND scheme.

For Frobenius instances the Aardal–Lenstra technique is fast and has no limit on a, but experiments carried out by D. Lichtblau and A. Strzebonski (Wolfram Research, Inc.) indicate that an instance-solver based on DQQD is faster when $a < 10^5$ or when a is between 10^5 and 10^7 and $n \geq 20$.

4. Upper Bound Algorithms

A New Upper Bound

We use the following notation throughout our discussion of upper bounds. Let $G = \mathcal{G}(A)$ be the Frobenius circulant graph of $A = \{a_1, a_2, \ldots, a_n\}$; for $1 \leq j \leq n$, let $A_j = \{a_1, a_2, \ldots, a_j\}$ and $d_j = \gcd(A_j)$; let $\bar{A}_j = \{a_1/d_j, a_2/d_j, \ldots, a_j/d_j\}$; with associated graph $\mathcal{G}(\bar{A}_j)$. Note that $f(A) = f(A')$ for any permutation A' of A, so a_j can be any j-element subset of A: simply begin with an appropriate permutation.

Define an equivalence relation on the vertices of G as follows: $v \sim u$ if u is the endpoint of a path that starts at v and uses edges in $\mathcal{G}(A_j)$, an edge subgraph of G. The relation is reflexive and transitive and the circulant symmetry of G makes it symmetric. So this divides the vertices into disjoint equivalence classes C_v. Slightly abusing notation, we will consider each such set C_v as an induced subgraph of $\mathcal{G}(A_j)$; it is strongly connected since there is a path from any vertex to any other.

Lemma 2. With notation as above, and for some $j \le n$, let v and u denote vertices of G. Then **(a)** $D(C_v) = d_j D(\mathcal{G}(\bar{A}_j))$; and **(b)** $u \in C_v$ if and only if $u \equiv v \pmod{d_j}$; this

means that choosing $0 \le v < d_j$ gives a set of representatives for the equivalence classes of $\sim$.

Proof (a). We make use of the edge and weight relationships in C_v and in $\mathcal{G}(\bar{A}_j)$ to prove that C_v and $\mathcal{G}(\bar{A}_j)$ are isomorphic as unweighted directed graphs. Since $\gcd(\bar{A}_j) = 1$, $\mathcal{G}(\bar{A}_j)$ is a Frobenius circulant graph for the reduced basis $\bar{A}_j$. By the symmetry of G at every vertex, every C_v is isomorphic to C_0, so it is sufficient to show that C_0 is isomorphic to $\mathcal{G}(\bar{A}_j)$. Let h be a vertex of $\mathcal{G}(\bar{A}_j)$, so $0 \le h < a_1/d_j$. Define a function $\beta(h) = d_j \cdot h$ from $\mathcal{G}(\bar{A}_j)$ to G, so $0 \le \beta(h) \le a_1 - d_j \le a_1 - 1$, and $\beta(h)$ is a vertex of G. It is clear that β is one-one. The function β preserves edges: if there is an edge of weight a_i/d_j from vertex h_1 to h_2 in $\mathcal{G}(\bar{A}_j)$, then $h_2 \equiv h_1 + (a_i/d_j) \pmod{a_1/d_j}$; it follows that $d_j h_2 \equiv d_j h_1 + a_i \pmod{a_1}$, so there is an edge of weight a_i from $\beta(h_1)$ to $\beta(h_2)$ in G. Conversely, we can divide the congruence arising from G by d_j provided the modulus is also divided by d_j. So β is a graph isomorphism of $\mathcal{G}(\bar{A}_j)$ and its image $\beta(\mathcal{G}(\bar{A}_j))$. By the remark following (4), $\mathcal{G}(\bar{A}_j)$ is strongly connected, so there is a path of weight $w = \sum_{i=2}^{j} e_i a_i / d_j$ in $\mathcal{G}(\bar{A}_j)$ from 0 to h. Since the path from 0 in G of weight $d_j\, w = \sum_{i=2}^{j} e_i a_i$ consists entirely of edges with weights in $A_j \backslash \{a_1\}$, $\beta(h)$ is in C_0, so the image $\beta(\mathcal{G}(\bar{A}_j)) = C_0$. Thus, when considered as directed unweighted graphs, each C_v is isomorphic to $\mathcal{G}(\bar{A}_j)$. Each edge weight of C_v is d_j times the corresponding weight in $\mathcal{G}(\bar{A}_j)$, so $D(C_v) = d_j D(\mathcal{G}(\bar{A}_j))$.

(b). If $u \in C_v$, then there must be some path in C_v from v to u of weight $w = \sum_{i=2}^{j} e_i a_i$, with $u \equiv v + w \pmod{a_1}$. Since d_j divides a_1 and w, $u \equiv v \pmod{d_j}$. Conversely, suppose $u \equiv v \pmod{d_j}$. By the proof of (a) the unweighted graph C_v is isomorphic to $\mathcal{G}(\bar{A}_j)$, which is a Frobenius circulant graph containing a_1/d_j vertices. Therefore $|C_v| = a_1/d_j$, which means that each one of the a_1/d_j vertices u that are congruent to $v \pmod{d_j}$ must lie in C_v. $\square$

Theorem 1. Suppose $1 \le j \le n$ and $K = \mathcal{G}(\{d_j, a_{j+1}, \ldots, a_n\})$. Then $D(G) \le d_j D(\mathcal{G}(\bar{A}_j)) + D(K)$.

Proof. It suffices to show that for any vertex u of G, there is a path from 0 to u of weight at most $D(C_u) + D(K)$, since Lemma 2 states that $d_j D(\mathcal{G}(\bar{A}_j)) + D(K) = D(C_u) + D(K)$. So let u be a vertex of G, and let $\bar{u}$ be the mod-d_j reduction of u. Then 0 and $\bar{u}$ are vertices in K, so we can let $w = \sum_{i=j+1}^{n} e_i a_i$ be the minimum weight of a path p in K from 0 to $\bar{u}$; this means $w \equiv \bar{u} \pmod{d_j}$ and $w \le D(K)$. Now if $\hat{w}$ is the mod-a_1 reduction of w, then $\hat{w} \equiv \bar{u} \pmod{d_j}$ and so $\hat{w} \in C_u$ (see Fig. 13). Since the edge weights in K are all edge weights in G, we can interpret path p as a path $\hat{p}$ in G of weight w. The path $\hat{p}$ ends at vertex $\hat{w}$, a vertex of G in C_u. Since there is a path q in C_u from $\hat{w}$ to u

of weight at most $D(C_u)$, we have the desired path — $\hat{p}$ followed by q — from 0 to u of weight at most $D(K) + D(C_u)$. $\square$

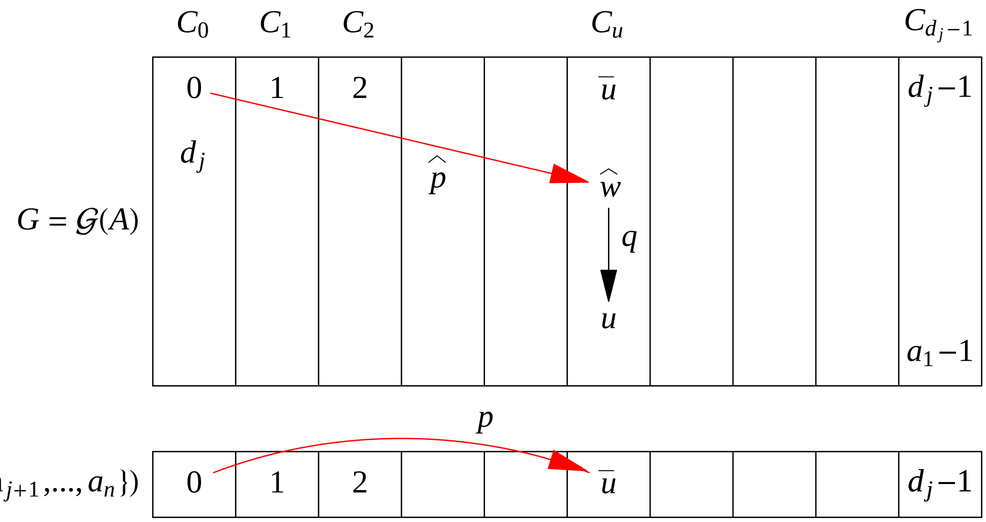

Figure 13: The diameter of G is bounded by the sum of the diameter of K and the diameter of any equivalence class C_u . The weight of each of the paths p and $\hat{p}$ is w, and $w \leq D(K)$. Graphs G and K have different structures: while path p in K ends at vertex $\bar{u} \pmod{d_j}$, the corresponding path $\hat{p}$ in G does not end at vertex $\bar{u} \pmod{a_1}$. But $\hat{p}$ does end in the same equivalence class $C_{\bar{u}}$, and the required weight of path q is no greater than $D(C_{\bar{u}}) = D(C_u)$.

Corollary 1. If $j \leq n$ then $f(A) \leq d_j f(\bar{A}_j) + f(d_j, a_{j+1}, \ldots, a_n) + d_j$.

Proof. This follows immediately from Theorem 1 and the relationship (5) between the graph diameter and the Frobenius number. $\square$

Corollary 2. If $j \leq n$ and if $a_g \in A \backslash A_j$ such that $\gcd(d_j, a_g) = 1$, then $f(A) \leq d_j f(\bar{A}_j) + (d_j - 1)a_g$.

Proof. By Corollary 1, $f(A) \leq d_j f(\bar{A}_j) + D(K)$, so we need only show that $D(K) \leq (d-1)a_g$. But $D(K) \leq D(\mathcal{G}(\{d_j, a_g\})) = (d_j - 1)a_g$, where the last equality comes from the formula for $f(d_j, a_g) = d_j a_g - d_j - a_g$. $\square$

Brauer and Shockley [Bra62] proved that the bound of Corollary 2 can be made exact, for the specific case $j = n - 1$, by adding a_n as an element of the reduced basis, thereby improving an earlier result by Johnson [Joh60]. Ramírez Alfonsín [Ram∞] gives a clear statement as his Lemma 3.1.7: Let $d = \gcd(a_1, \ldots, a_{n-1})$. Then $f(A) = df(a_1/d, \ldots, a_{n-1}/d, a_n) + (d-1)a_n$. The equivalent form of this equation in our notation is $f(A) = d_{n-1} f(\bar{A}_{n-1} \cup \{a_n\}) + (d_{n-1} - 1)a_n$.

Corollary 3. If j≤n and $a_{\max} = \max(A \backslash A_j)$, then $f(A) \leq d_j f(\bar{A}_j) + (d_j - 1)a_{\max}$.

Proof. Since K is connected, $D(K) \leq (d_j - 1)a_{\max}$. $\square$

Corollary 4 (Schur). If A is sorted, then $f(A) \leq a_1 a_n - a_1 - a_n$.

Proof. This follows from Corollary 3 with $j = 1$. $\square$

If A has two coprime entries i and j then $f(A) \leq f(a_i, a_j) = a_i a_j - a_i - a_j$. But it can happen that no such pair exists; Corollary 4 shows that the related formula $\min(A)\max(A) - \min(A) - \max(A)$ is always an upper bound.

In the Chicken McNuggets example, if we take the permutation $A' = \{6, 20, 9\}$, then $A_2 = \{6, 20\}$ and $d_2 = 2$. Referring to Figure 1, the triangular subgraph C_0 is obtained by using edges of weight 20 along the path $0 \to 2 \to 4 \to 0$ in graph $G = \mathcal{G}(\{6, 20, 9\})$, and C_0 is isomorphic to $H = \mathcal{G}(\{3, 10\})$ in which all edges are of weight $a_2/d_2 = 20/2 = 10$. Subgraph C_1 isomorphic to C_0 contains vertices $\{1, 3, 5\}$, so all vertices of G are in $\{C_0, C_1\} = \mathcal{G}(A_2)$. Another example is presented in Figure 14.

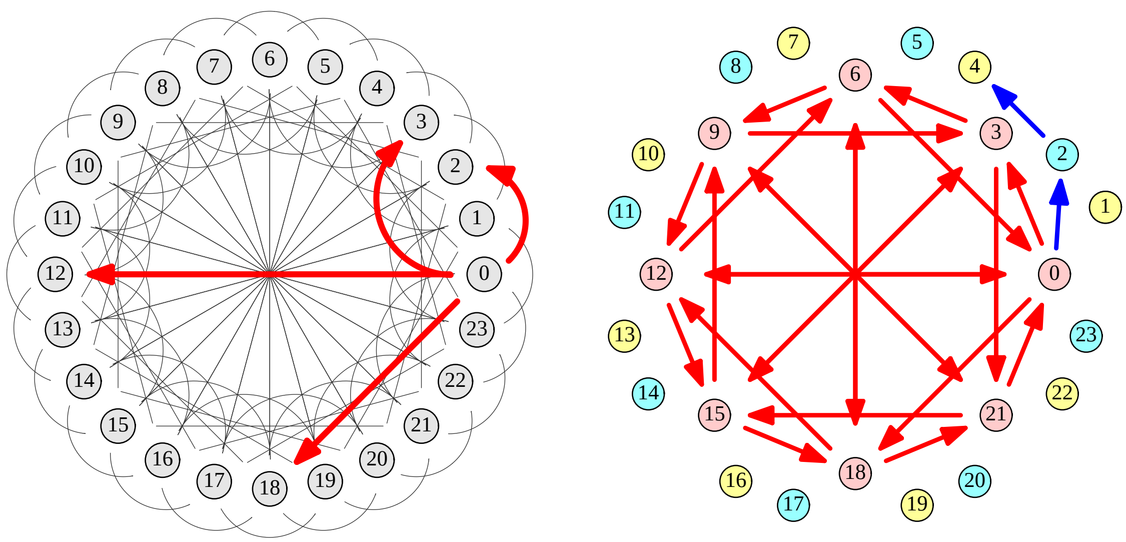

Figure 14: The graph at the left is $G = \mathcal{G}(\{24, 36, 42, 27, 50\})$ (directions and weights omitted), with the generating set of the circulant in red. On the right, the vertices of G have differing colors for each of the isomorphic components $\{C_0, C_1, C_2\}$ of $\mathcal{G}(A_4)$. The red edges at the right show C_0, which is isomorphic to $\mathcal{G}(\{8, 12, 14, 9\})$. The blue edges are the minimum-weight path from vertex 0 connecting all components of $\mathcal{G}(A_4)$.

Figure 14 illustrates the situation for the basis $A = \{24, 36, 42, 27, 50\}$ and $j = 4$; thus $A_4 = \{24, 36, 42, 27\}$ and $d_4 = 3$. In the full graph $G = \mathcal{G}(A)$ on the left, the highlighted red outbound edges from 0 are of weights $50 \equiv 2 \pmod{24}$, $27 \equiv 3 \pmod{24}$, $36 \equiv 12 \pmod{24}$, and $42 \equiv 18 \pmod{24}$. Every other vertex in G has outbound edges with the same weights. Lemma 2 is illustrated on the right: The red edges show C_0, the equivalence class of 0, which contains only edges whose weight is in $A_4 \backslash \{a_1\} = \{36, 42, 27\}$. The graph C_0 is isomorphic to $\mathcal{G}(\{8, 12, 14, 9\})$, where each of the edge weights in C_0 is $d_4 = 3$ times the corresponding edge weights $\{12, 14, 9\}$ in $\mathcal{G}(\{8, 12, 14, 9\})$. The vertices in the isomorphic graphs C_1 and C_2 are shown in differing colors, illustrating that vertex u of G is in C_v if and only if $u \equiv v \pmod{3}$.

Theorem 1 is also visible: all components of $\mathcal{G}(A_4)$ can be connected by the blue path of weight $100 = D(K) = D(\mathcal{G}(\{3, 50\}))$. Thus, $f(A) \leq d_4 f(\bar{A}_4) + D(K) = 3f(8, 12, 14, 9) + 100 = 3 \cdot 19 + 100 = 157$. In fact, $f(A) = 157$ in this example, so the upper bound is exact.

A Polynomial Time Upper Bound Algorithm

Corollaries 1, 2, and 3 are very powerful upper bound tools. Being able to use a reduced basis to bound $f(A)$ by $d_j f(\bar{A}_j) + f(d_j, a_{j+1}, \ldots, a_n) + d_j$ allows us to use exact methods in an upper bound estimate even when the original basis is far beyond practical limits for computing the exact value of $f(A)$.

For $j = 2$, if $d_2 = \gcd(A_2) = 1$, then $f(a_1, a_2) = a_1 a_2 - a_2 - a_1$ returns the exact value of $f(A_2)$ virtually instantaneously. Adding elements to A_2 can only reduce the set of nonrepresentable integers, so $f(A) \leq f(A_2)$. However, there may be no 2-element subset of A for which $d_2 = 1$ (e.g., $A = \{30, 42, 70, 105\}$), so the $f(a_1, a_2)$ formula cannot be applied directly to generate an upper bound for an arbitrary basis. But since $\gcd(\bar{A}_2) = 1$, we can always use the 2-element formula on every reduced pair to obtain the exact value of $d_2 f(\bar{A}_2)$ in each case.

Similarly, for $j = 3$, if $d_3 = \gcd(A_3) = 1$, Greenberg's algorithm returns the exact value of $f(A_3)$ in quadratic time, and $f(A) \leq f(A_3)$. As noted, there may be no triple from A for which $d_3 = 1$, so Greenberg's algorithm cannot be applied directly to generate an upper bound. But since $\gcd(\bar{A}_3) = 1$, we can always use Greenberg's algorithm on every reduced triple to obtain the exact value of $d_3 f(\bar{A}_3)$.

We can compute $D(K)$ by using the methods of our minimum-weight path algorithms for the Frobenius number, except in the rare case when d_j is large, say $d_j > 10^4$. When that happens, we can use Corollary 3 to replace $f(d_j, a_{j+1}, \ldots, a_n) + d_j$ by $(d_g - 1)\min(a_g, a_{\max})$.

Taking the minimum of the foregoing computations across all 2- and 3-element subsets of A leads to an upper bound algorithm that will always run in time that is $O(N^5)$, where N is the length of the input A. The N^5 arises from the $O(n^3)$ triples with each one requiring $O(N^2)$ operations. The use of A path algorithm would normally require exponential time, but because we have restricted it to graphs with at most 10000 vertices, it is actually $O(N)$ in the worst case if Dijkstra or Round Robin is used. We use the notation $U_{2,3}$ for the upper bound obtained in this way; this notation is suggestive since if one had a way of dealing with 4-tuples or larger subsets, the bound could be correspondingly improved.

Here is a formal description of our upper bound algorithm. Note that if $d = 1$ in Step 2 or 3 then the term $f(\{d\} \cup (A \backslash B)) + d$ will be $-1 + 1 = 0$.

FROBENIUS UPPER BOUND BY SUBSETS

Input. A set A of positive integers $a_1, a_2, \ldots, a_n$ with $\gcd(a_1, \ldots, a_n) = 1$.

Output. An upper bound $U_{2,3}$ for the Frobenius number: $f(A) \le U_{2,3}$.

Step 1. Set $d_{\max} = 10^4$.

Step 2. For each 2-element subset $B = \{a_i, a_j\}$ of A:
If $d = \gcd(B) \le d_{\max}$, use Sylvester's formula and a minimum-weight path algorithm to compute $U = df(B/d) + f(\{d\} \cup (A \backslash B)) + d$;
Otherwise, set $b_{\max} = \max(A \backslash B)$, $b_g = \min\{c \in A \backslash B : \gcd(c, B) = 1\}$, and $b = \min(b_{\max}, b_g)$; then use Sylvester's formula to compute $U = df(B/d) + (d-1)b$;

Step 3. For each 3-element subset $B = \{a_i, a_j, a_k\}$ of A:
If $d = \gcd(B) \le d_{\max}$, use Greenberg's algorithm and a minimum-weight path algorithm to compute $U = df(B/d) + f(\{d\} \cup (A \backslash B)) + d$;
Otherwise, set $b_{\max} = \max(A \backslash B)$, $b_g = \min\{c \in A \backslash B : \gcd(c, B) = 1\}$, and $b = \min(b_{\max}, b_g)$; then use Greenberg's algorithm to compute $U = df(B/d) + (d-1)b$;

Step 4. Return the minimum of the $\binom{n}{2} + \binom{n}{3}$ numbers U

One might think that the bound of step 3, using triples, would surely be better than the one of step 2, using pairs. But it can happen that no triple has a common divisor, while pairs do, and this can lead to cases where the pairs bound is lower than the triples bound. This occurs for $A = \{6, 7, 8, 9\}$: the pairs bound arising from the pair $\{6, 9\}$ is 11, but the four triples yield bounds of 17, 17, 19, 20.

Comparisons to Other Upper Bounds

Using the notation of the preceding subsection, Brauer [Bra42, Bra54] proved the upper bound $f(a_1, a_2, \ldots, a_n) \le U_{\text{Brauer}}(A) = \sum_{i=1}^{n-1} a_{i+1} d_i / d_{i+1} - \sum_{i=1}^{n} a_i$. He noted that this bound "may change by varying the numbering of the a_i," so it should be taken across all permutations of A (and is therefore not a polynomial-time algorithm).

We can rewrite Brauer's formula as $f(A) \le (\frac{d_1}{d_2} - 1)a_2 - a_1 + \sum_{i=3}^{n} (\frac{d_i}{d_{i+1}} - 1)a_i$. Since $a_1 = d_1$, Sylvester's formula $f(a_1, a_2) = a_1 a_2 - a_2 - a_1$ tells us that

$$d_2 f(\bar{A}_2) = d_2 f(\frac{a_1}{d_2}, \frac{a_2}{d_2}) = (\frac{a_1 a_2}{d_2}) - a_1 - a_2 = (\frac{d_1}{d_2} - 1)a_2 - a_1.$$

The remaining summand in Brauer's formula is therefore equivalent to constructing some path from the edges in $A \backslash A_2$ in order to connect all of the components C_v in $\mathcal{G}(A_2)$. By Theorem 1, our approach finds the shortest such connecting path. Brauer's method sometimes finds the shortest path, but the example $A = \{100, 229, 425, 586, 635, 689\}$ shows that this is not always the case: Brauer's upper bound $U_{\text{Brauer}}(A)$ returns $f(A) \le 4631$ across all permutations of A, while our shortest path method limited just to pairs A_2 returns $f(A) \le 3142$. Brauer's upper bound is never better than the bound of Corollary 1.

We can prove that $U_{2,3}(A)$ is always at least as good as several of the other upper bound formulas based on two or three elements of A. Each of the other upper bounds below is based on the assumption that A is in ascending order. For example, the Lewin [Lew72] bound for $n \geq 3$ is $f(a_1, a_2, \ldots, a_n) \leq \lfloor (a_n - 2)^2 / 2 \rfloor = U_{\text{Lewin}}(A)$, but $U_{2,3}(A) \leq f(a_1, a_2, a_n) \leq U_{\text{Lewin}}(a_1, a_2, a_n) = U_{\text{Lewin}}(A)$. Similarly, Vitek's [Vit75] bound is $f(a_1, a_2, \ldots, a_n) \leq \lfloor (a_2 - 1)(a_n - 2)/2 \rfloor - 1 = U_{\text{Vitek}}$, but $U_{2,3}(A) \leq f(a_1, a_2, a_n) \leq U_{\text{Vitek}}(a_1, a_2, a_n) = U_{\text{Vitek}}(A)$.

The Beck–Diaz–Robins [BDS02] upper bound is the following:

$$f(A) \leq (\sqrt{a_1 a_2 a_3 (a_1 + a_2 + a_3)} - a_1 - a_2 - a_3)/2 = U_{\text{BDR}}(A).$$

It is useful for its simplicity, but since it is for triples only it is not of practical value since Greenberg's algorithm gives the exact triple answer so quickly.

A definitive comparison to upper bounds varying with n is more problematic. For example, the Erdős and Graham [EG72] bound is $f(a_1, a_{2,\ldots}, a_n) \leq 2a_{n-1}\lfloor a_n/n \rfloor - a_n = U_{\text{EG}}(A)$; while Selmer [Sel77] gives $f(a_1, a_2, \ldots, a_n) \leq 2a_n \lfloor a_1/n \rfloor - a_1 = U_{\text{Selmer}}(A)$. In practice, however, these bounds are generally far less accurate than $U_{2,3}(A)$. Bounds based on least common multiples [RC96, BB01] are even less accurate.

The comparative data in Table 4, which shows the ratios of the bound to the correct answer, show that our algorithm often leads to an improvement in accuracy by orders of magnitude compared to the other published upper bounds.

For very large numbers the improvement given by $U_{2,3}$ can be described as follows, where we use ρ for the ratio of an upper bound to the exact value. For reasons explained in §5, $U_{2,3}$ is usually near $a^{3/2}$ while the other bounds are close to a^2. Since $(\max A)^2$ is an upper bound (Schur, see Cor. 4), in typical cases for large a the other bounds do not give a lot of information. Of course, for such large numbers we do not know the exact value of the Frobenius number, but the next section contains a conjecture that, if true, would say that the expected value of $f(A)$ is, generally, near $a^{n/(n-1)}$. So when n is small, say $n = 4$, $U_{2,3}$ will be about $a^{3/2}$ while the true value is near $a^{4/3}$, and therefore $\rho_{2,3}$ will be about $a^{1/6}$ while the ratio for the other bounds will be about $a^{2/3}$. As n increases, the true value on average gets close to just a, a, indicating that $U_{2,3}$, while a good improvement over the other bounds, will still be far away from $f(A)$. On the other hand, the performance of $U_{2,3}$ when $n = 4$ and a is not so large is quite good: in 100 trials with $a = 50000$ and $n = 4$, $\rho_{2,3}$ was between 1.02 and 5.38 with an average of 2.94. For the best of the rest, the ratio was between 63 and 541 with an average of 338.

To conclude, we show a random googol-sized example with $n = 4$:
$A \approx \{1, 2.67, 7.97, 8.89\} \cdot 10^{100}$. Then the best of the last four bounds in Table 4 is $4.4 \cdot 10^{200}$ while $U_{2,3}(A) = 9.7 \cdot 10^{150}$. As explained in the next section, the likely value of $f(A)$ is about $4 \cdot 10^{134}$.

Narrow Domains Narrow the Frobenius Number

When the interval spanned by the basis is small, say $[a, a + \sqrt{a}]$ or $[a, a + \log a]$ the character of the Frobenius problem changes. A lower bound of Vizvári [Viz87] turns out

Problem	$U_{2,3}$	U_{BDR}	U_{EG}	U_{Vitek}	U_{Selmer}
1	1	3.23	23	5.83	4.67
2	1.001	6.61	17	15	3.33
3	1.001	7.5	17	12	4.16
4	1	3.23	20	5.83	3.33
5	1.002	8.94	20	20	5.
6	1.053	41	101	88	22
7	1.059	26	339	115	30
8	1.072	51	119	92	29
9	1.05	44	47	51	22
10	1.109	33	117	68	20
11	1.025	43	67	86	17
12	1.011	22	85	67	19
13	1.066	47	149	136	20
14	1.003	27	190	91	25
15	1.011	35	42	51	13
16	10	743	5096	3117	524
17	5.22	1122	4289	2535	461
18	9.37	731	4375	2570	546
19	7.4	552	5516	1449	481
20	9.07	995	4401	3728	522
21	13	1206	3891	2443	931
22	5.74	523	4557	2065	523
23	10	766	5202	2491	771
24	8.99	934	5852	3216	848
25	14	1286	4094	2979	943

Table 4: The ratio of various upper bounds to the actual Frobenius number for the 25 problems of our benchmark. The first 15 problems were specially constructed and so are not typical. For the ten random examples, the best of the other upper bounds is always at least 50 times greater than the $U_{2,3}$ upper bound.

to be very good in such cases and is quite close to the upper bound $U_{2,3}$. Indeed, these two bounds typically determine several digits of $f(A)$. The Vizvári bound is the following, assuming the basis is in increasing order: $f(a_1, \ldots, a_n) \geq K(a_1^2 - a_1) - 1$, where K is the minimum value of q/r in the representations of a_i $(2 \leq i \leq n)$ as $qa_1 + r$, with $0 \leq r < a_1$.

Here are some examples that show the interval that contains the Frobenius number; the lower end is Vizvári's bound, the upper end is $U_{2,3}$.

$$f(\{10000000000, 10000008870, 10000057783, 10000072907\}) \in [1.37 \cdot 10^{15}, 2.04 \cdot 10^{15}]$$
$$f(\{10000000, 10000024, 10000053, 10000072\}) \in [1.3888 \cdot 10^{12}, 1.3896 \cdot 10^{12}]$$

Finally, for $A = 10^{100} + \{0, 947, 1167, 1757, 1908, 9012\}$, the two bounds determine the first 93 digits of $f(A)$. Thus we can say with certainty that

$$\begin{aligned} f(A) &= 1.10963160230803373280071016422547714158899245450510430537061695 \\ &\qquad 517088326675543719485130936529\ldots \cdot 10^{196}. \end{aligned}$$

These cases seem more restricted than those typically discussed in the literature, but it is nevertheless striking that one can give several digits of the Frobenius number with fairly simple computations.

5. A Frobenius Growth Model

Fast algorithms for $f(A)$ can be used to generate data that can lead to estimates on the average size of the Frobenius number. We describe such experiments here and show how they led us to a conjecture about the expected size of the Frobenius number. We first switch the context slightly for simplicity. The "positive Frobenius number" $g(A)$ is the largest integer that is not a sum of basis elements using only positive coefficients. It is easy to see that $g(A) = f(A) + \Sigma A$. Approximations are more succinctly stated for this variation. For typical large inputs the ΣA term is negligible.

We first review what is known when $n = 3$. In that case, the very fast Greenberg algorithm allows us to compute exact Frobenius numbers when the inputs are gigantic. Davison [Dav84] proved the lower bound $g(a, b, c) \geq \sqrt{3}\sqrt{abc}$ and proved that the constant $\sqrt{3}$ is sharp. Computations show that the general form of this bound seems to capture the asymptotic average behavior of g. As always, we use a spread of 10 in generating our random bases; changing this upwards has little impact on the results presented in this section.

Figure 15 (left) shows the ratio $\log g(a, b, c)/\log(abc)$ as a goes up to 10^{200}, with 10 trials for each a-value. There is surely no doubt that this value converges to $1/2$, indicating that the $\sqrt{abc}$ term in Davison's bound is correct, asymptotically on average. For an extreme case, this ratio was 0.50018 for a basis $\{10^{1000}, b, 10^{1001} - 1\}$, with b random. In Figure 15 (right; 20 trials for each power of 10 from 10^{10} to 10^{200}) the ratio $g(a, b, c)/\sqrt{3abc}$ is plotted. The mean for these 1820 data points was 1.43. The sum of the basis elements is negligible, so it seems fair to say that the average value of $f(a, b, c)$ is asymptotic to $c\sqrt{3abc}$, where c is near 1.45 (see Figure 16 for a more detailed look at the mean in an experiment to 10^{500}); the existence of such a constant was conjectured by Davison. But it seems unlikely that a constant multiple of the Davison bound will provide an upper bound. For example, if $A = \{10^6, 2 \cdot 10^6, 2 \cdot 10^6 + 1\}$ then the Davison bound is too low by a factor of 577. And increasing the 6 in this example to 100 or more yields much larger ratios.

The fact that $f(a, b, c)$ is generally near $\sqrt{abc}$ explains why $U_{2,3}(A)$, which is based on the Frobenius number of triples from A, will be near $a^{3/2}$ when the entries of A have roughly the same size.

There have been some extensions of Davison's bound to larger n (which always denotes the number of basis elements). The most noteworthy is that of Killingbergtrø [Kil00] who

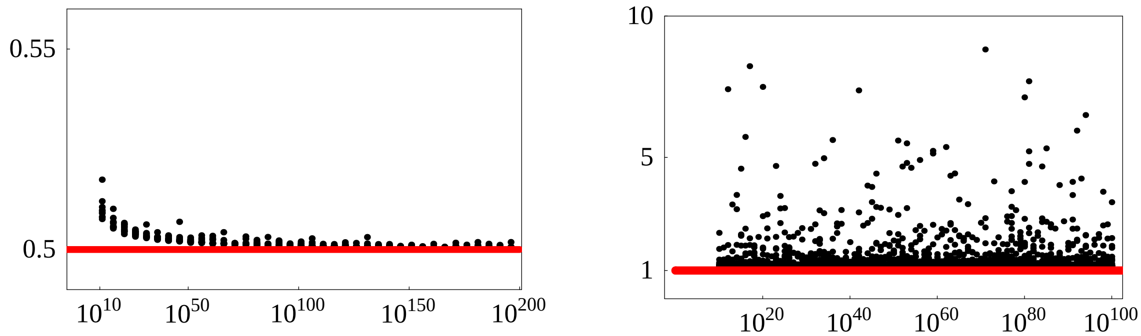

Figure 15: Left: The ratio $\log g(a, b, c)/\log(abc)$ with 10 trials for each value of a. Right: The ratio $g(a, b, c)/\sqrt{3abc}$, with 20 trials for each value of a, and the values of a running through 10^i as i goes from 10 to 200.

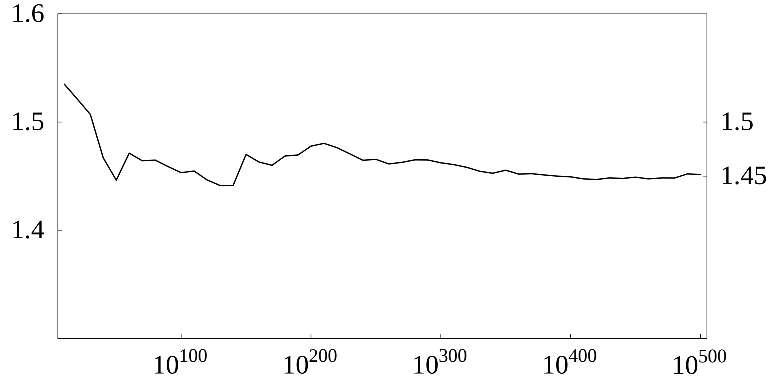

Figure 16: The cumulative means of $g(a, b, c)/\sqrt{3abc}$ for an experiment with a going from 10^{10} to 10^{500} and 150 trials for each a. The mean for each set of trials was computed, and then the cumulative means of these are plotted.

proved that $g(A) \geq ((n-1)!\,\Pi A)^{1/(n-1)}$. This bound appears to predict the correct order of magnitude of $g(A)$, but is unlikely to be sharp since it does not specialize to the right answer when $n = 3$ (it has $\sqrt{2}$ where $\sqrt{3}$ is wanted). But it is suggestive and led us to the following formula, which does specialize properly,

$$L(A) = \left(\frac{1}{2}n!\,\Pi A\right)^{\frac{1}{n-1}}$$

While numerical evidence (we checked over one billion 4-tuples) supported the conjecture that $L(A)$ is a lower bound on $g(A)$, that assertion is false. David Einstein communicated the following counterexample which he was led to by his knowledge of a paper [DF04] on a closely related subject. The $k = 7$ case in Table 8.1 of that paper leads to $A = \{84, 84n + 2, 84n + 9, 84n + 35\}$. When n is 62, $g(A)/L(A) = 0.99998$.

So we see that $L(A)$ turns out to be a good estimator of the average value of $g(A)$, but a word of warning: it is useful only when a is large relative to n. Roughly, a should

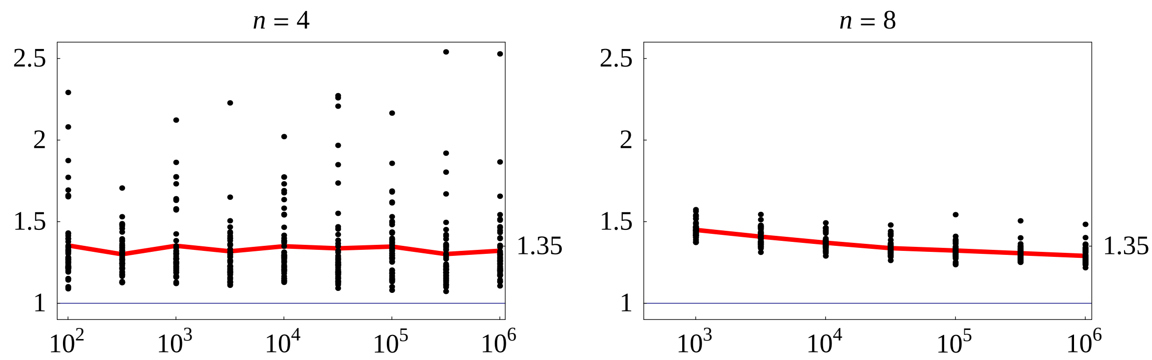

Figure 17: The ratio of the positive Frobenius number to $L(A)$ for n equal to 4 and 8, and with a going up to a million. The means (red lines) provide evidence that the expected value of $g(A)$ is asymptotic to a small constant multiple of $L(A)$.

be greater than about $n^{n-1}/n!$, for otherwise $L(A) < \Sigma A$ which is a useless estimate (switching to f, this would give a negative lower bound). So if $n = 20$, a should be at least one million. Figure 17 shows the result of an experiment with $n = 4$ and 50 trials for values of a running up to one million. The red line connects the mean of each set of trials. The other graph is similar, but with $n = 8$. The results support the claim that $L(A)$, whose form differs from Killingbergtrø's bound only in the leading constant, is a good predictor of the expected value of $g(A)$. When $n = 4$, the ratio $g(A)/L(A)$ was never greater than 3.5 and had a mean of 1.38 and a minimum of 1.07. The data for $n = 8$ are similar, but with a smaller mean at the end. Thus it appears that for each n the expected value of $g(A)$ is asymptotic to $c_n L(A)$ where $c_3 \sim 1.5$, c_4 is near 1.35, and $c_8 < 1.35$. Whether these constants change much as n grows, or even decrease to 1 in all cases is impossible to say without many more experiments, which would require faster algorithms for computing $f(A)$. But it is intriguing to think that the constants have 1 as the limit, for that would mean that, for very large n and much larger a, the expected value of $g(A)$ is very well estimated by $L(A)$.

References

[AL02] K. Aardal and A. K. Lenstra, Hard equality constrained integer knapsacks. In W. J. Cook and A. S. Schulz, eds., *Integer Programming and Combinatorial Optimization 2002, Lecture Notes in Computer Science 2337*, Springer-Verlag, Berlin/Heidelberg (2002) 350–366.

[BDR02] M. Beck, R. Diaz, and S. Robins, The Frobenius problem, rational polytopes, and Fourier–Dedekind sums, *J. Number Theory*, 96 (2002) 1–21.

[BEZ03] M. Beck, D. Einstein, S. Zacks, Some experimental results on the Frobenius problem, *Experimental Mathematics* 12 (2003) 263-269.

[BL04] S. Böcker and Z. Lipták, The money changing problem revisited: Computing the Frobenius number in time $O(ka_1)$, Computing and Combinatorics Conference (COCOON) Kunming, China (2005).

[Ber93] D. P. Bertsekas, A simple and fast label correcting algorithm for shortest paths, *Networks* 23 (1993) 703–709.

[Bra42] A. Brauer, On a problem of partitions, *Amer. J. of Math.* 64 (1942) 299–312.

[Bra54] A. Brauer and B. M. Seelbinder. On a problem of partitions II, *Amer. Journal of Math.* 76 (1954) 343–346.

[Bra62] A. Brauer and J. E. Shockley, On a problem of Frobenius, *Journal für Reine und Angewandte Mathematik* 211:3/4 (1962) 215–220.

[BB01] V. E. Brimkov and R. P. Barneva, Gradient elements of the knapsack polytope, *Calcolo* 38:1 (2001) 49–66.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, second ed., MIT Press, Cambridge, Mass., 2001.

[CUWW97] G. Cornuéjols, R. Urbaniak, R. Weismantel, and L. A. Woolsey, Decomposition of integer programs and of generating sets. In R. E. Burkard and G. J. Woeginger, eds., *Algorithms — ESA '97, Lecture Notes in Computer Science 1284*, Springer-Verlag, Berlin/Heidelberg (1997) 350–366.

[Dav94] J. L. Davison, On the linear diophantine problem of Frobenius, *J. Number Theory* 48 (1994) 353–363.

[DF04] R. Dougherty and V. Faber, The degree-diameter problem for several varieties of Cayley graphs I: The Abelian case, *SIAM J. Disc. Math.* 17 (2004) 478–519.

[ELSW∞] D. Einstein, D. Lichtblau, A. Strzebonski, S. Wagon, Frobenius numbers by lattice point enumeration, in preparation.

[EG72] P. Erdős and R. L. Graham, On a linear diophantine problem of Frobenius, *Acta Arithmetica* 21 (1972) 399–408.

[Gre88] H. Greenberg, Solution to a linear Diophantine equation for nonnegative integers, *J. Algorithms* 9 (1988) 343–353

[Gre99] H. Greenberg, The linear Diophantine equation in nonnegative variables, *Mathematica in Education and Research* 8 (1999) 72–74.

[Joh60] S. M. Johnson, A linear Diophantine problem, *Can. J. Math.* 12 (1960) 390–398.

[Kil00] H. G. Killingbergtrø, Betjening av figur i Frobenius' problem (Using figures in Frobenius's problem), (Norwegian) *Normat* 2 (2000) 75–82.

[Kan89] R. Kannan, Lattice translates of a polytope and the Frobenius problem, *Combinatorica* 12(2) (1992) 161–177.

[Kra88] H. Krawczyk and A. Paz, The diophantine problem of Frobenius: a close bound, *Discrete Applied Mathematics* 23 (1989) 289–291.

[Lew72] M. Lewin, A bound for a solution of a linear Diophantine problem, *J. London Math. Soc.* 6 (1972) 61–69.

[Nij79] A. Nijenhuis, A minimal-path algorithm for the "money changing problem", *Amer. Math. Monthly* 86 (1979) 832–838.

[NW72] A. Nijenhuis and H. Wilf, Representation of integers by linear forms in nonnegative integers, *J. Number Theory* 4 (1972) 98–106.

[Nos85] K. Noshita. A theorem on the expected complexity of Dijkstra's shortest path algorithm. *J. Algorithms* 6 (1985) 400–408.

[Owe03] R. W. Owens, An algorithm to solve the Frobenius problem, *Mathematics Magazine* 76:4 (2003) 264–275.

[RC96] M. Raczunas and P. Chrząstowski-Wachtel, A diophantine problem of Frobenius in terms of the least common multiple, *Discrete Mathematics* 150 (1996) 347–357.

[Ram96] J. Ramírez Alfonsín, Complexity of the Frobenius problem, *Combinatorica* 16 (1996) 143–147.

[Ram∞] J. Ramírez Alfonsín, The Diophantine Frobenius problem, 199 pp, Oxford Univ. Press (to appear).

[Sel77] E. S. Selmer, On the linear diophantine problem of Frobenius, *Journal für Reine und Angewandte Mathematik* 293/294:1 (1977) 1–17.

[Sha02] J. Shallit, The computational complexity of the local postage stamp problem, *SIGACT News* 33:1 (2002) 90–94.

[Syl84] J. J. Sylvester, Mathematical questions, with their solutions. *Educational Times* 41 (1884) 21.

[Vit75] Y. Vitek, Bounds for a linear Diophantine problem of Frobenius, *J. London Math. Soc.* 10 (1975) 79–85.

[Viz87] Y. Vizvári, An application of Gomory cuts in number theory, *Periodica Math. Hung.* 18 (1987) 213–228.