Computing Extremal and Approximate Distances in Graphs Having Unit Cost Edges

Kellogg S. Booth
Richard J. Lipton

June, 1979

CS-78-08

Computing Extremal and Approximate Distances In Graphs Having Unit Cost Edges

Kellogg S. Booth

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1

Richard J. Lipton

Computer Science Division
University of California, Berkeley
Berkeley, California 94720

ABSTRACT

Using binary search and a Strassen-like matrix multiplication algorithm we obtain efficient algorithms for computing the diameter, the radius, and other distance-related quantities associated with undirected and directed graphs having unit cost (unweighted) edges. Similar methods are used to find approximate values for the distances between all pairs of vertices, and if the graph satisfies certain regularity conditions to find the exact distances.

A preliminary version of this research was presented at the Sixteenth Annual Allerton Conference on Communication, Control, and Computing and appears in the Proceedings for that conference. This work was supported in part by the National Research Council of Canada under grant grant A4307 and by the Office of Naval Research under gran N00014-75-6-0752.

Computing Extremal and Approximate Distances In Graphs Having Unit Cost Edges

Kellogg S. Booth

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1

Richard J. Lipton

Computer Science Division
University of California, Berkeley
Berkeley, California 94720

1. Introduction

We consider the problem of computing distances in directed and undirected graphs whose edges have unit costs. We will use standard notation from graph theory [1,4]. We assume that a graph $G = (V,E)$ is connected (strongly connected if it is directed), that its vertex set is $V = \{1, 2, \cdots, n\}$, and that its edge set E has e elements. We develop algorithms for computing the diameter, radius, and other distance-related quantities associated with the graph. Our algorithms employ binary search techniques and use fast matrix multiplication routines to achieve asymptotic efficiency. Using similar methods we develop algorithms for computing approximate distances between all pairs of vertices. Finally, we discuss a class of graphs for which the exact distances can be easily computed.

The *distance* between two vertices i and j is the number of edges in a shortest path connecting them. This distance is denoted by $d(i,j)$. The *eccentricity* of a vertex is the greatest among all of its distances to other vertices in the graph. We denote the eccentricity of a vertex by

$$e(i) = \max\{d(i,j) \mid j \in V\}.$$

The *diameter* and *radius* of a graph G are the largest and smallest eccentricities of its vertices, thus defining for us the two notions

$$diameter(G) = \max\{e(i) \mid i \in V\}$$

$$radius(G) = \min\{e(i) \mid i \in V\}.$$

Vertices of maximum eccentricity are on the *periphery* of G, while those of minimum eccentricity are in the *center* of G, these two sets being defined as

$$periphery(G) = \{i \mid e(i) = diameter(G)\}$$

$$center(G) = \{ i \mid e(i) = radius(G) \}.$$

A *pair of peripheral points* consists of two vertices whose distance apart is equal to the diameter of the graph. A shortest path between a pair of peripheral points is called a *geodesic*.

For our purposes a graph will be represented by its *augmented adjacency matrix*, which is the $n \times n$ matrix $A = (a_{i,j})$ defined by

$$a_{i,j} = \begin{cases} 1, & \text{if there is an edge } \{i,j\} \in E \text{ or if } i = j \\ 0, & \text{if there is no edge } \{i,j\} \in E \text{ and } i \neq j. \end{cases}$$

Choosing the augmented adjacency matrix instead of the ordinary adjacency matrix effectively means that ones are added along the main diagonal of the matrix. This corresponds to adding loops to every vertex in the graph. This feature will be used in the algorithms which follow.

A second matrix associated with a graph is also of interest here. The *distance matrix* for G is the $n \times n$ matrix $D = (d_{i,j})$ in which $d_{i,j}$ is the distance between i and j.

For directed graphs we define similar notions and consider the obvious in- and out- generalizations of eccentricity, diameter, radius, periphery, and center. Unless otherwise noted we will state our results for undirected graphs and imply the corresponding results for directed graphs.

Floyd's shortest path algorithm computes the distance matrix of a graph in $O(n^3)$ steps [2]. Fredman [3] has an algorithm whose running time is a bit less, $O(n^3(\log\log n / \log n)^{1/3})$, but Floyd's version is the practical choice for real applications. Our algorithms will be asymptotically faster than either of these algorithms, although we will not displace Floyd's algorithm except for problems which are only of theoretical interest.

Both of these algorithms compute the distance matrix of a graph having arbitrary non-negative costs on the edges. Our graphs have unit cost edges. Although this would appear to make the problem easier, we do not see how to capitalize on the extra information inherent in the unit cost asumption. In fact, we know of no better way of finding the distance matrix for unit cost edges than to apply a general algorithm designed to handle arbitrary cost edges.

In some applications we do not need to know the entire distance matrix. Knowledge of extremal distances such as the diameter or radius may suffice. Or we might need only a list of the peripheral or central vertices. If the graph has arbitrary costs on its edges we see no shortcuts, but for unit cost edges we show how to compute these quantities in $O(M(n)\log n)$ steps, where $M(n)$ is the number of steps required to multiply two $n \times n$ matrices. If we really are interested in the entire distance matrix but not in the exact values of its entries, we show how to obtain, with relative error at most $\epsilon > 0$, approximate distances between all pairs of vertices in $O(M(n)(\log n)^2)$ steps.

Throughout our discussion we will let $M(n)$ stand for the complexity of $n \times n$ matrix multiplication and we will express all of our upper bounds in terms of this quantity. Strassen's algorithm provides an $O(n^{\log_2 7})$ upper bound [7]. Pan [6] has given a slightly better asymptotic bound. The results given here will automatically improve whenever a new upper bound is shown for matrix multiplication.

2. Counting Walks

The problem of computing a graph's diameter can be viewed as finding the smallest integer k such that every pair of vertices is connected by a path whose length does not exceed k. Similarly a graph's radius is the smallest integer k such that some vertex is connected to all other vertices by paths whose lengths do not exceed k. If we can efficiently test these two conditions for any k we can find the minimum k quickly using a binary search. Since both the diameter and the radius are bounded above by n, at most $\log n$ values of k need be tested in order to find the diameter or radius.

Directly deciding the existence of paths of length k between all pairs of vertices might be as hard as computing the entire distance matrix. It seems easier to test a slightly different but closely related condition.

A *walk* between two vertices i and j is a sequence of vertices beginning at i and ending at j having the property that successive pairs of vertices are adjacent in G. In general a walk is not a path because vertices, even edges, may be repeated. But a shortest walk is always a shortest path, and *vice versa*, since otherwise the walk or path could be made even shorter by eliminating the edges connecting two instances of a repeated vertex.

For our purposes walks have one major advantage over paths: the existence of a path of length k from i to j usually tells nothing about the existence of longer paths between i and j, but a walk of length k can always be extended to a walk of length $k+1$ simply by traversing any loop around a vertex in the walk, thus picking up the extra length. Notice that without loops there might not be a walk of length $k+1$ since in the undirected case we might be forced to repeat an edge twice (producing a walk of length $k+2$), and in the directed case there might be no way at all to extend the walk.

Using this fact plus our observation that shortest paths are the same as shortest walks we can state the following characterization for the diameter of a graph.

LEMMA 1: A graph G has diameter k iff

(1) between every pair of vertices there exists a walk of length k, and

(2) between at least one pair of vertices a walk of length $k-1$ does not exist. □

We state a similar characterization for the radius of a graph which, together with the previous lemma, will reduce the extremal distance problems to simply counting walks within a graph.

LEMMA 2: A graph G has radius k iff

(1) for some vertex there exist walks of length k to all vertices, and

(2) for all vertices there is at least one other vertex to which a walk of length $k-1$ does not exist. □

To implement an efficient test for these two conditions we cite a well-known result relating walks of length k to the kth power of the augmented adjacency matrix [1,4].

LEMMA 3: Let A be the augmented adjacency matrix for a graph G. Then $(A^k)_{i,j}$ is precisely the number of distinct walks of length k between i and j. □

We can restate our first two lemmas in terms of the augmented adjacency matrix and thus characterize all of the problems under discussion in a form convenient for our algorithms. The characterizations in which we are really interested are the following.

LEMMA 4: A graph G has diameter k iff

(1) for every i and j $(A^k)_{i,j}>0$, and

(2) for some i and j $(A^{k-1})_{i,j}=0$. □

The characterization of the radius is similar.

LEMMA 5: G has radius k iff

(1) for some i and for all j $(A^k)_{i,j}>0$, and

(2) for all i and for some j $(A^{k-1})_{i,j}=0$. □

Note that having found the diameter of G we can easily find the periphery according to the rule that

$$periphery(G) = \{ i \mid \text{for some } j,\ A_{i,j}^{k-1}=0 \}.$$

Here k is the computed diameter of G. If k is instead the computed radius of G we find the center according to the rule that

$$center(G)=\{ i \mid \text{for all } j\ (A^k)_{i,j}>0 \}.$$

All pairs of peripheral points can be calculated in a similar manner and a breadth-first search will easily find a geodesic between any such pair. These last two computations can be carried out in $O(n^2)$ operations given the appropriate matrices A^{k-1} and A^k.

3. Extremal Distances

Our algorithms are designed to utilize any fast method for multiplying $n \times n$ matrices. Our running times will have corresponding improvements each time that the upper bound on matrix multiplication is improved.

The basic idea is to precompute the power-of-two powers of the augmented adjacency matrix and to then conduct a binary search to find the k which satisfies the criterion of lemma 4 or 5. Our algorithm is asymptotically faster than Floyd's because it uses a Strassen-like matrix multiplication routine.

We construct an algorithm to find the diameter of a graph given its augmented adjacency matrix as input. The final value of k corresponds to the diameter of G as given by lemma 4. The matrix B is the $k-1$st power of the matrix A and is used to test the conditions of lemma 4. At each iteration one additional bit of accuracy is added to k, so that after $\lceil \log n \rceil$ iterations the value of k is the exact diameter of the graph.

Throughout the discussion we use the notation "$X > 0$" to mean that the matrix X has all of its entries non-zero and we use the notation "$X \not> 0$" to mean just the opposite, that the matrix X has at least one zero entry.

```
**integer procedure** DIAMETER($A$, $n$):
  **begin**
    { $A$ is the augmented adjacency matrix of a connected $n$-vertex graph $G$ }
    **if** $n \leqslant 1$ **then return** 0;
    **if** $A > 0$ **then return** 1;
    $M_0 := A$;
    **for** $p := 1$ **to** $\lceil \log n \rceil$ **step** 1 **do**
      $M_p := M_{p-1} \cdot M_{p-1}$;
    $k := 2$;
    $B := A$;
    **for** $p := \lceil \log n \rceil - 1$ **to** 0 **step** $-1$ **do**
      { $B = A^{k-1} \not> 0$, $B \cdot M_{p+1} > 0$, and $M_l = A^{2^l}$ for $0 \leqslant l \leqslant \lceil \log n \rceil$ }
      **if** $B \cdot M_p \not> 0$ **then**
        **begin**
          $k := k + 2^p$;
          $B := B \cdot M_p$
        **end**;
    { $A^{k-1} \not> 0$ and $A^k > 0$ }
    **return** $k$
  **end**
```

LEMMA 6: Let $M(n)$ be the number of operations required to multiply two $n \times n$ matrices. Algorithm DIAMETER computes the diameter of a connected n-vertex graph using $O(M(n) \log n)$ operations and $O(n^2 \log n)$ words of storage.

PROOF:

Correctness: If G has 0 or 1 vertices then trivially it has diameter 0 and the procedure obviously returns the correct value. We also notice that the only possible way that any graph can have a diameter of 1 is for the graph to be complete. Because the procedure explicitly checks for $A > 0$ this case is again handled correctly. Thus it remains only to verify that the procedure is correct when processing a graph whose diameter is at least 2.

If the graph does have diameter at least 2 both tests must fail, so the procedure continues its initialization by building a table of the power-of-two powers of the matrix A. The 2^l power is stored in the matrix M_l for $0 \leqslant l \leqslant \lceil \log n \rceil$. The algorithm never uses any higher powers of the matrix because the diameter of an n-vertex graph is at most $n-1$. The next two assignments initialize k and B. Thus whenever the program first reaches the second loop the assertion stated there is valid.

We next show that the assertion remains true each time the loop is executed if it is true before the loop is executed. Let us use B', p', and k' to denote the values of the corresponding variables after executing the entire loop and reserve B, p, and k to denote the values of the variables before executing any part of the loop. The loop consists of an if-statement, so there are two cases to consider.

In the case where $B \cdot M_p > 0$ we have no changes to the variables, except for the loop index p. Thus $B'=B$, $p'=p-1$, and $k'=k$. Substituting these values into the invariant, we obtain

$$B' = B = A^{k-1} = A^{k'-1} \not> 0,$$
$$B' \cdot M_{p'+1} = B \cdot M_p > 0.$$

Obviously none of the matrices M_l is changed by the loop. Hence the assertion remains valid.

In the case where $B \cdot M_p \not> 0$ we have a similar argument. The M_l again are unchanged, but this time $B' = B \cdot M_p$, $p' = p-1$, and $k' = k + 2^p$. Substituting into the assertion, we find that

$$B' = B \cdot M_p = A^{k-1} \cdot A^{2^p} = A^{k'-1} \not> 0,$$
$$B' \cdot M_{p'+1} = (B \cdot M_p) \cdot M_p = B \cdot M_{p+1} > 0.$$

Thus regardless of the path through the loop, we know that the values of the variables after executing the loop satisfy the assertion whenever the values of the variables before executing the loop satisfy the assertion. The assertion is thus an invariant of the program.

From this it follows that the final assertion is also an invariant because the only way in which the return statement can be reached is when the second loop terminates with $p=-1$ (we assume that the control variable p is defined after the loop exit only as a convenience in our proof). This final invariant implies partial correctness of the program. Total correctness follows from the observation that termination is guaranteed because each loop is executed at most $\lceil \log n \rceil$ times.

Timing: As we have just remarked, each loop is executed at most $\lceil \log n \rceil$ times. The primary work within each loop consists of a single matrix multiplication. All of the other operations within the loops and within the rest of the program are dominated by these matrix multiplications. Hence the total number of operations is $O(M(n) \log n)$.

Storage: The largest space requirement is for the matrices M_l. Each of these has $O(n^2)$ entries and there are $\lceil \log n \rceil + 1$ of the matrices. Thus the total storage is $O(n^2 \log n)$. □

The procedure is easily modified to find the radius of a graph simply by looking for a single row or column of the matrix which contains no zeros, instead of demanding that the entire matrix contain no zeros. We have now shown the following.

THEOREM 7: We can find the diameter, radius, periphery, center, all pairs of peripheral points, or a geodesic of an n-vertex graph in $O(M(n) \log n)$ operations.

PROOF:

Using the procedures outlined above we can find the diameter or radius in the desired time. The periphery, center, and all pairs of peripheral points can be read out of the appropriate matrices A^{k-1} and A^k using the characterizations given in section 2. This requires at most a single scan over the two matrices, which is only $O(n^2)$ additional work. Given a pair of peripheral points it is easy to find a connecting geodesic in $O(n^2)$ operations using a breadth-first search. □

If we use a Strassen-like method for matrix multiplication we improve both upon Floyd's algorithm and upon Fredman's.

It is worth noting that our approach is useless without a fast matrix multiplication scheme. If we use the naive $O(n^3)$ method our algorithms are actually worse than Floyd's algorithm, both in a practical sense and asymptotically.

The storage requirement for our algorithm is $O(n^2 \log n)$ because of the $\log n$ matrices which we precompute. We could store these using $O(n^2 \log n)$ bits because we really do not need the actual numerical values of the matrices. Rather we are only interested in the zero or nonzero state of each entry. In an implementation this savings in storage might be worth the extra overhead required to store the bits in a packed form. The asymptotic time bounds would remain unchanged.

A final comment is that our algorithm precomputes a table of the power-of-two powers of the augmented adjacency matrix. This table of matrices is the dominant storage cost in the algorithm. This cost can be substantially reduced if the matrices are recomputed each time they are needed. This increases the running time by a factor of $\log n$ but reduces storage requirement by the same factor. Whether this time-space tradeoff is optimal we do not know.

4. Approximate Distances

In the previous section we discovered that extremal distances can be computed in less time than required by current algorithms for computing the entire distance matrix. If we are interested in computing all of the distance matrix, but will settle for having our answers correct only to within a factor of two, our algorithms also apply. We are essentially asking for the integer parts of the logarithms of the distances. The power-of-two powers of the adjacency matrix computed earlier provide sufficient information to determine all distances to within a factor of two using $O(M(n)\log n)$ operations.

We can actually obtain approximate distances correct to within any constant factor if we are willing to perform a little more work. For any $\epsilon>0$ define $\phi = 1+\epsilon$. If we let

$$\ddot{d}(i,j) = \min\{\lfloor\phi^k\rfloor \mid (A^{\lfloor\phi^k\rfloor})_{i,j} > 0\}$$

then $\ddot{d}(i,j)$ is a good approximation to $d(i,j)$. This is formalized by the next lemma.

LEMMA 8: For all $i \neq j$, $\dfrac{|\ddot{d}(i,j)-d(i,j)|}{d(i,j)} < \epsilon$.

PROOF:

Since $i \neq j$ we must have $d(i,j)>0$. Choose the minimal k such that $\ddot{d}(i,j) = \lfloor\phi^k\rfloor$. By definition,

$$\lfloor\phi^{k-1}\rfloor < d(i,j) \leqslant \lfloor\phi^k\rfloor = \ddot{d}(i,j).$$

Because $d(i,j)$ is an integer we can ignore the floor function and observe that in fact $\phi^{k-1} < d(i,j)$. Thus we find that

$$\ddot{d}(i,j) = \lfloor\phi^k\rfloor \leqslant \phi^k < \phi\cdot d(i,j).$$

But $\phi = 1+\epsilon$ so when we subtract $d(i,j)$ from each side we are left with

$$\ddot{d}(i,j)-d(i,j) < \epsilon\cdot d(i,j).$$

Because $\ddot{d}(i,j)$ is no smaller than $d(i,j)$ we may conclude that in fact

$$\frac{|\ddot{d}(i,j)-d(i,j)|}{d(i,j)} < \epsilon$$

which is the desired result. □

To compute $\ddot{d}(i,j)$ for all i and j we need only compute the sequence of matrix powers $A^{\lfloor\phi^0\rfloor}, A^{\lfloor\phi^1\rfloor}, \cdots, A^{\lfloor\phi^k\rfloor}, \cdots$, stopping certainly when the nth power of A is reached and possibly much sooner. As we compute successive powers we check to see when each (i,j)-entry first becomes nonzero in one of the matrices, recording the corresponding power as $\ddot{d}(i,j)$.

THEOREM 9: Given any $\epsilon>0$ there is an $O(M(n)(\log n)^2)$ algorithm (whose constant of proportionality depends upon ϵ) which computes approximate distances $\ddot{d}(i,j)$ having a relative error no larger than ϵ.

PROOF:

We compute $\log_\phi n$ matrix powers. Each matrix can be computed using at most $2\log n$ matrix multiplications by inspecting the binary expansion of its exponent. These computations dominate the remaining work. If we note that

$\log_\phi n$ is actually $\log n / \log \phi$ then an upper bound on the total number of operations is $O(M(n)(\log n)^2)$. In terms of ϵ the constant of proportionality varies as $1/\log(1+\epsilon)$, which is $1/\epsilon$ for sufficiently small ϵ.

The number ϕ is of course not an integer. We must keep enough of the low-order (fractional) bits to ensure that the truncated value $\lfloor \phi^k \rfloor$ is correct in its integer part. If we initially compute ϕ to $\log_\phi n$ fractional bits the computation stops when ϕ^k reaches n so the cumulative error will not propagate into the integer part of our calculation.

Technically we would require a different algorithm for each n because the actual data constant which represents ϕ depends upon n, but in practice we can always choose $\epsilon = 2^{-t}$ and eliminate this problem altogether. □

The storage requirement for this algorithm is clearly $O(n^2)$ because we have only a fixed number of matrices in use at any time. It is interesting to ask whether the second factor of $\log n$ is really necessary in theorem 9. Is it possible to calculate the $\lfloor \phi^k \rfloor$-th power of A from the $\lfloor \phi^{k-1} \rfloor$-th power of A using only a fixed number of matrix multiplications? For arbitrary values of ϕ we see no better method than examining the binary expansion of $\lfloor \phi^k \rfloor$ to obtain $A^{\lfloor \phi^k \rfloor}$, even if we have available all of the powers of A previously computed. (This method would, of course, require additional storage.) Perhaps there is an easier method. We leave this as an open question.

5. All Distances

If we want to compute all distances instead of just the extremal distances our techniques do not always work. If the graph is sparse we can take advantage of its sparseness by computing distances using a breadth-first search beginning at each vertex [5]. This will yield an $O(ne)$ algorithm which for $e < n^2$ will be better than Floyd's algorithm. The problem seems to be with dense graphs.

It would seem that with many edges a dense graph would be likely to have a small diameter. If the diameter is small then we can compute the matrices A^0, A^1, A^2, ... , $A^{diameter(G)}$ to find all distances in the graph, a method similar to that outlined in theorem 9. This would require at most $O(M(n) \cdot diameter(G))$ operations. For arbitrary graphs we could then use a hybrid approach, depending upon the density of edges, and perhaps achieve a uniformly fast algorithm.

But density alone does not insure a small diameter [8]. It is easy to exhibit graphs with almost $n^2/4$ edges whose diameter is n. If we want to employ our hybrid solution we need a more uniform condition than density.

Define a graph to be *pseudo-regular of degree* ρ iff the degree of each vertex is bounded above and below by

$$\rho/2 \leqslant degree(i) \leqslant 2\rho$$

so that all vertices have about ρ neighbors. Note that all regular graphs are pseudo-regular but that the converse is not true.

LEMMA 10: If G is a pseudo-regular graph of degree ρ then $diameter(G) \leqslant 6n/\rho$.

PROOF:

Consider a geodesic of G whose vertices are $i_0, i_1, \cdots, i_{diameter(G)}$. Because a geodesic is a shortest path between its endpoints the neighbors of i_j and i_{j+3} must be distinct for every j. Thus the total number of vertices on the geodesic and adjacent to it must be at least $\lceil diameter(G)/3 \rceil \cdot \lceil \rho/2 \rceil$. This cannot exceed the number of vertices in the graph, hence we know that $diameter(G) \leqslant 6n/\rho$. □

Our hybrid algorithm will want to balance the cost of the breadth-first search for sparse psuedo-regular graphs against the cost of performing $diameter(G)$ matrix multiplications when the graph is dense. Ignoring constants and low-order terms we must solve the equation

$$n\,M(n)/\rho = n^2\rho$$

for ρ, where the left-hand side is the cost for dense graphs using our matrix technique, and the right-hand side is the cost for sparse graphs using breadth-first search. The crossover occurs when

$$\rho = (M(n)/n)^{1/2}.$$

We have shown the following result.

THEOREM 11: There is an $O((n^3M(n))^{1/2})$ algorithm for computing all distances in any pseudo-regular graph. □

This algorithm achieves the same running time as does Johnson's algorithm [5] if the input graph is sparse, but our algorithm is superior if the number of edges exceeds a threshold which depends upon the running time of the matrix multiplication algorithm. (The lower the bound for matrix multiplication, the lower the threshold and hence our hybrid algorithm becomes even more attractive.) Recall, however, that our algorithm is only designed to work well for pseudo-regular graphs whereas Johnson's algorithm works well for any graph.

It is interesting to note that if we use a naive matrix multiplication instead of Strassen's method we won't improve on Floyd's algorithm and in the unlikely event that $M(n)=O(n^2)$we would matchff the $O(n^{2.5})$ decision tree complexity of the algorithm given by Fredman [3].

6. Conclusions

We have exploited the fact that it is unnecessary to know all distances within a graph when computing various distance statistics such as the diameter and radius of a graph. These same techniques have also been used to compute approximate distances between all pairs of vertices in a graph with unit cost edges. Finally, we have achieved an improved running time for the exact all pairs problem in graphs with unit cost edges under the assumption of pseudo-regularity.

It would be nice to find other classes of graphs for which the all pairs problem can be solved efficiently. We suspect that distances are not as hard to compute for unit cost edges as for arbitrary costs and propose this as an area for further research.

7. Acknowledgement

The authors would like to thank a number of people who commented on the preliminary version of this work and who made suggestions which simplified our algorithms. In particular the use of the augmented adjacency matrix was pointed out to us at the Allerton Conference. Its use avoids a problem in our original presentation for directed graphs.

References

[1] J. A. Bondy and U. S. R. Murty, *Graph Theory With Applications*, MacMillan, London, 1976.

[2] R. W. Floyd, Algorithm 97: Shortest path, *Communications of the ACM 5:6* (June 1962) p. 345.

[3] M. L. Fredman, New bounds on the complexity of the shortest path problem, *SIAM Journal On Computing 5:1* (March 1976) pp. 83-89.

[4] F. Harary, *Graph Theory*, Addison-Wesley, Reading, Massachusetts, 1969.

[5] D. B. Johnson, Efficient algorithms for shortest paths in sparse networks, *Journal of the ACM 24:1* (January 1977) pp. 1-13.

[6] V. Pan, Strassen's algorithm is not optimal, *IEEE 19th Annual Symposium on Foundations of Computer Science* (November 1978) pp. 166-176.

[7] V. Strassen, Gaussian elimination is not optimal, *Numerishe Mathematik 13* (1969) pp. 354-356.

[8] V. G. Vizing, The number of edges in a graph of given radius, *Soviet Mathematics - Doklady 8:2* (1967) pp. 535-536.