-

Shortest Path and Closure Algorithms for Banded Matrices.

L. Allison,
T. I. Dix and
C. N. Yee.

Department of Computer Science, Monash University, Australia 3168.
Partially supported by Australian Research Grant A48830856.

Abstract. A fast algorithm is given for the all-pairs shortest paths problem for banded matrices having band-width b. It solves the negative-cycle problem and calculates all path lengths within the band in $O(nb^2)$ time and calculates all other path lengths in $O(n^2b)$ time.

Keywords: shortest path, negative cycle, closure, banded matrix, algorithm.

Preprint to Inf. Proc. Lett., 40, pp317-322, 30 Dec 1991, and TR 89/133 Dept. of Computer Science[*], Monash University, Australia, 1989.

1. Introduction.

A directed graph with n points and with weights on arcs representing distances between the points can be represented by an n×n distance matrix. The *all-pairs shortest paths* problem is to calculate the shortest distance between every ordered pair of points. The *negative-cycle problem* is to detect the presence of a cycle with a negative length. Floyd's algorithm[5] solves these problems in $O(n^3)$ time. Here an incremental algorithm is given which allows a new point to be added in $O(n^2)$ time. This is then used as the basis of an algorithm for graphs having banded matrices. (For a uniformly banded matrix with d diagonals, $\lfloor d/2 \rfloor$ either side of the main diagonal, the band-width is d.) Given a matrix with a band-width b, this latter algorithm solves the negative-cycle problem and calculates shortest paths for all pairs of points within the band in $O(nb^2)$ time. The matrix need not be uniformly banded; b is the maximum band-width. Finally it is shown how to calculate shortest paths between all other pairs of points in $O(n^2b)$ time. The shortest paths problem is an instance of a *closure* problem, as is finding connected components[8]. These algorithms can be turned to other closure tasks.

2. Floyd's Algorithm.

Floyd gives an algorithm to calculate the shortest distance between every pair of points in $O(n^3)$ time given an n×n distance matrix m. The algorithm consists of three nested loops, shown in figure 1, and operates on the matrix f which initially has

```
f[i,j] = m[i,j], the weight of arc from point i to point j and i≠j
f[i,j] = ∞ where no arc exists from point i to point j and i≠j
f[i,i] = 0
```

The outer loop increases k from 1 to n. The invariant of this loop is that f[i,j] represents the minimum distance between i and j taken over all paths that pass only through intermediate points in the range [1..k-1]. Initially k=1 and f contains only the given arcs - the direct paths - which implies the invariant in this case. When passing through a new intermediate point k is permitted and f[i,j] is being updated, it is only necessary to consider new paths passing through k once. It is therefore sufficient to consider paths from i to k and from k to j and these path lengths are known already.

Floyd's algorithm works for negative weights provided there are no negative cycles; these must be detected and some action taken when f[i,j]+f[j,i]<0. An incremental version of Floyd's algorithm is described in the next section.

3. Incremental Algorithm.

It is possible to modify Floyd's algorithm to produce an incremental algorithm (figure 2). This takes $O(n^3)$ time in total but can add a new point plus its arcs to the shortest paths matrix in $O(n^2)$ time. (A similar algorithm for the case of sparse graphs has been described by Srinivasan and Rao[7].) It forms the basis of the fast algorithm for banded matrices which is described in the next section.

At a given stage in the incremental algorithm, the submatrix f[1..k-1,1..k-1] contains all shortest paths derived from the submatrix m[1..k-1,1..k-1] (figure 3). In the next step, the set of points {1..k-1} is augmented with point k. This introduces a new point and new arcs to and from k. Any shortest path from j<k to k will only visit k as its final point. A shortest path from k to j<k must leave k by an arc and not return to k. Therefore to calculate shortest paths to or from k, it is sufficient to consider arcs to or from k plus shortest paths in f[1..k-1,1..k-1]. Such paths are computed by the first (inner) pair of nested loops in figure 2. The paths to or from k can shorten other paths between points in {1..k-1}. Any new shorter path between two points i<k and j<k visits k at most once. It is therefore sufficient to consider new paths being the sum of paths from i to k and from k to j. The second pair of nested loops in figure 2 computes such paths. Negative cycles must be detected in both nested loops. This allows f[1..k-1,1..k-1] to be extended to f[1..k,1..k] in $O(n^2)$ time. By iterating k from 3 to n, all shortest paths can be calculated in $O(n^3)$ time.

4. Window Algorithm.

The incremental algorithm of the previous section uses a growing *window*, f[1..k,1..k]. At each step a point, k, is added. If p is the first point in the window and k is the last point and if there is no arc between p and any point j≥k then it is possible to drop p (figure 4). So for a uniformly banded matrix with band-width b, the window can grow to w×w, where w=⌊b/2⌋+1, and then slide, adding and dropping a point at each step until k=n. For a non-uniformly banded matrix, the window can shrink further if there is no arc between p+1 and j, and so on. Sliding the window (figure 5) forms the basis of the window algorithm. A forward scan and a reverse scan are needed to find the shortest paths bounded by all windows. Notice there need not be an arc between every pair of points within a window.

High and low bounds for sliding windows are easily calculated by preprocessing. This takes O(nb) time for a graph represented by lists and O(n) time for a graph represented by a vector of reduced vectors. Hi[i] is the maximum index for points ahead that a single arc can reach past i either from i or from a point less than i. Lo gives minimum indices for points looking backwards.

```
Hi[i] = max (k) s.t. ∃ j≤i and k≥i and there is an arc j→k or j←k
Lo[i] = min (k) s.t. ∃ j≥i and k≤i and there is an arc j→k or j←k
```

Hi defines the window f[i..Hi[i],i..Hi[i]] above i; Lo defines the window f[Lo[i]..i,Lo[i]..i] below i. Note that this form of matrix includes those which are decomposable according to Hu's linear scheme[6,9]. For a uniformly banded matrix with band-width b, Hi[i] is the lesser of i+⌊b/2⌋ and n and Lo[i] is the greater of i-⌊b/2⌋ and 1.

The forward scan (figure 6) is a modification of the incremental algorithm. The modified invariant of the outer loop states that each previous window contains the true minimum path lengths for the points that lie within it, for paths via these points and via prior points. However distances between points dropped from a window do not embody later points and are "out of date". Formally, for j in [1..k-1] and for i in [Lo[j]..j], that is in the window below j, f[i,j] and f[j,i] are minimum path lengths via nodes in [1..min(k-1,Hi[i])], that is before i or in the window above i but less than k-1.

Shortest paths in the current window are found by augmenting points from the immediately previous window by point k, as for the incremental algorithm except that points may have been dropped. The two pairs of nested loops consider paths between k and j and between i and j respectively. The invariant of the outer loop over k is true for k=3 and for all k≤n by an inductive argument similar to that for the incremental algorithm: Firstly, any allowable path between j and k must either be known or via some point in the previous window. Secondly, any allowable path between i and j can only visit k at most once. Note, the final window when k reaches n contains the minimum path lengths between all points in the window via any point inside or outside it including all dropped points.

A reverse scan can now be made where the window slides back from k=Lo[n]-1 until it includes the first point (figure 7). The invariant of the outer loop over k states that each previous window contains the true minimum path lengths for the points that lie within it for paths via any point; this is certainly true initially. Formally, for i in [k+1..n] and j in [i..Hi[i]], f[i,j] and f[j,i] are minimum path lengths between i and j via any node. Shortest paths in the current window are found by augmenting the immediately previous window by point k. It is sufficient to update paths between k and every other point j≥k in the window to maintain the invariant. f[k,j] contains an upper bound on the distance from k to j derived only from paths via points in {1..Hi[k]} during the forward scan. This might be reduced in the light of information available in the reverse scan. Any such *new* shorter path must involve a (first) point i in [k+1..Hi[k]]. The intermediate path lengths from k to i and i to j are known. Hence only one pair of nested loops is required, examining paths between k and j via i where i and j are in [k+1..Hi[k]]. A similar argument applies to f[j,k]. Again negative cycles must be detected.

The forward and reverse scans each take $O(nb^2)$ time for maximum band-width b and therefore maximum window size $(\lfloor b/2 \rfloor+1)\times(\lfloor b/2 \rfloor+1)$.

At the end of the forward and reverse scans, if i and j are in a common window then f[i,j] is the minimum distance from i to j via any path, otherwise it is ∞ representing unknown. If these unknown distances are needed they can be calculated by working outwards from the main diagonal in $O(n^2b)$ time (figure 8). If i and j>i are not in a common window, then any path from i to j must involve a point k, i<k<j

and k in [i+1..Hi[i]]. To calculate f[i,j] it is sufficient to use all such k, adding f[i,k] and f[k,j]. The latter two matrix elements lie closer to the main diagonal than f[i,j] and are already known when f[i,j] is calculated. Similarly, any path from j to i must involve a point k, i<k<j and k in [Lo[j]..j-1].

5. Conclusion.

An algorithm has been presented which solves the all-pairs shortest paths problem and the negative cycle problem for banded matrices. It takes $O(nb^2)$ time to detect a negative cycle and to calculate path lengths within the band and takes $O(n^2b)$ time to calculate all other path lengths. Banded matrices include matrices decomposable according to Hu's linear scheme. Our algorithm has a lower order of complexity than the algorithms of Hu[6] and Yen[9].

Banded matrices arise from "long" "thin" graphs. The algorithm was motivated by the restriction site mapping problem[1,2,3] in genetic engineering. This is a combinatorial search problem and the constraints involve checking paths in certain graphs. A partial solution is rejected if a cycle with negative weight is found. The graphs in the problem naturally yield banded matrices. The final stage for filling in the outer values is not needed in that application.

If a distance matrix is not banded, there are fast heuristics and slow algorithms[4] which can be used as a preprocessing stage to try to put it in banded form for the algorithm described here. In general this is a difficult problem and this approach cannot be guaranteed to speed up the shortest paths problem for arbitrary sparse matrices.

References.

[1] L. Allison, T. I. Dix and C. N. Yee. Shortest path and closure algorithms for banded matrices. Dept. Computer Science, Monash University TR 89/133 (1989).

[2] L. Allison and C. N. Yee. Restriction site mapping is in separation theory. Computer Applications in the BioSciences 4(1) 97-101 (1988).

[3] T. I. Dix and D. H. Kieronska. Errors between sites in restriction site mapping. Computer Applications in the BioSciences 4(1) 117-123 (1988).

[4] I. S. Duff. Survey of sparse matrix research. Proc. IEEE. 65(4) 500-535 (1977).

[5] R. W. Floyd. Algorithm 97: Shortest Path. CACM 5 345 (1962).

[6] T. C. Hu and W. T. Torres. Shortcut in the decomposition algorithm for shortest paths in a network. IBM J. Research and Development 13(4) 387-390 (1969).

[7] B. Srinivasan and V. K. Rao. On the shortest path algorithms for sparse graphs. Asian Institute of Technology TR3 (1982).

[8] S. Warshall. A theorem on boolean matrices. JACM 9(1) 11-12 (1962).

[9] J. Y. Yen. On Hu's decomposition algorithm for shortest paths in a network. Operations Research 19(4) 983-985 (1971).

```
for k:=1 to n do
{for all i and j in [1..n],
f[i,j] is the min path length from i to j via nodes in [1..k-1]}
   for i:=1 to n do
      for j:=1 to n do
      begin
         f[i,j]:=min(f[i,j], plus(f[i,k],f[k,j])); {plus preserves infinity}
         {f[i,j] is the min path length from i to j via nodes in [1..k]}

         if f[i,j]+f[j,i] < 0  then
            negative path: take some action
      end;
```

Figure 1 Floyd's Algorithm O(n^3) .

```
for k:=3 to n do
{for all i and j in [1..k-1],
 f[i,j] is the min path length from i to j via nodes in [1..k-1]}
begin
   for i:=1 to k-1 do
      for j:=1 to k-1 do
      begin
         f[j,k]:=min(f[j,k], plus(f[j,i], f[i,k]));
         {f[j,k] is the min path length from j to k via nodes in [1..k]}

         f[k,j]:=min(f[k,j], plus(f[k,i], f[i,j]));
         {f[k,j] is the min path length from k to j via nodes in [1..k]}

         if f[k,j]+f[j,k] < 0 then
            negative path: take some action
      end;

   for i:=1 to k-1 do
      for j:=1 to k-1 do
      begin
         f[i,j]:=min(f[i,j], plus(f[i,k],f[k,j]));
         {f[i,j] is the min path length from i to j via nodes in [1..k]}

         if f[i,j]+f[j,i] < 0 then
            negative path: take some action
      end
   {for all i and j in [1..k],
  f[i,j] is the min path length from i to j via nodes in [1..k]}
end;
```

Figure 2 Incremental Algorithm $O(n^3)$.

```
     1                     k
    .---------------------.-.
 1  |                     | |
    |                     | |
    |                     | |
    |                     | |
    |                     | |
    |                     | |
    |                     | |
    |                     | |
    |                     | |
    .---------------------+-+
 k  |                     | |
    .---------------------.-.
```

Figure 3 Incremental Growth .

```
   p                   k
  .....................
p .                   .
  . .-------------------.
  . |                 . |
  . |                 . |
  . |                 . |
  . |                 . |
  . |                 . |
  . |                 . |
  ..|.................. |
k   |                   |
    .-------------------.
```

Figure 4 Dropping Point From Window .

Forward Scan Reverse Scan

```
      .--------------------.
      | .     |        oo  |
      |   .---|-------|    |
Lo[i] |   | . |       |    |
      |---|---.       |    |
      |   |     .-------|  |
      |   |     | .-----|--|
    i |   |     | | . | |  |
      |   ------|-|---. |  |
      |         --------.  |
Hi[i] | oo        |       .|
      .--------------------.
```

Figure 5 Sliding Windows .

```
for k:=3 to n do
{for all j in [1..k-1] and i in [Lo[j]..j]
 f[i,j] is the min path length from i to j via nodes in [1..min(k-1, Hi[i])]
 and similarly f[j,i]}
begin
   lok:=Lo[k];
   for i:=lok to k-1 do {i in [Lo[k]..k-1] and k in [i..Hi[i]]}
      for j:=lok to k-1 do
      begin
         f[j,k]:=min(f[j,k], plus(f[j,i], f[i,k]));
         {f[j,k] is the min path length from j to k via nodes in [1..i]}

         f[k,j]:=min(f[k,j], plus(f[k,i], f[i,j]));
         {f[k,j] is the min path length from k to j via nodes in [1..i]}

         if f[k,j]+f[j,k] < 0 then
            negative path: take some action
      end;

   for i:=lok to k-1 do
      for j:=lok to k-1 do
      begin
         f[i,j]:=min(f[i,j], plus(f[i,k],f[k,j]));
         {f[i,j] is the min path length from i to j via nodes in [1..k]}

         if f[i,j]+f[j,i] < 0 then
            negative path: take some action
      end;
end;
```

Figure 6 Window Algorithm, Forward Scan $O(nb^2)$.

```
for k:=Lo[n]-1 downto 1 do
{for all i in [k+1..n] and j in [i..Hi[i]]
f[i,j] is the min path length from i to j via any node
 and similarly f[j,i]}
   begin
      hik:=Hi[k];
      for i:=k+1 to hik do
         for j:=k+1 to hik do
         begin
            f[j,k]:=min(f[j,k], plus(f[j,i], f[i,k]));
            {f[j,k] is the min path length from j to k via any node
     not in [i+1..Hi[k]]}

            f[k,j]:=min(f[k,j], plus(f[k,i], f[i,j]));
            {f[k,j] is the min path length from k to j via any node
     not in [i+1..Hi[k]]}

            if f[k,j]+f[j,k] < 0 then
               negative path: take some action
         end
   end;
```

Figure 7 Window Algorithm, Reverse Scan $O(nb^2)$.

```
{bmin is minimum band-width}
for diag:= (bmin div 2)+1 to n-1 do {diagonal}
{for all i and j in [1..n] s.t. |i-j| < diag
f[i,j] is the min path length from i to j}
   for i:=1 to n-diag do
   begin
      j:=diag+i;   {eg. diag=2, i,j:  1,3  2,4  3,5 ...}

      {note,
   1. if i<j then any path from i to j must involve a node k in [i+1..Hi[i]]
   2. k in [i+1..Hi[i]] so f[i,k] is min path length from i to k,
   3. j-Hi[i] <= j-k = diag+i-k <= diag-1
     so if k<j then f[k,j] is min path length from k to j}
      for k:=i+1 to Hi[i] do
         f[i,j] := min( f[i,j], plus( f[i,k], f[k,j] ));

      {note,
   if i<j then any path from j to i must involve a node k in [Lo[j]..j-1]}
      for k:= Lo[j] to j-1 do
         f[j,i] := min( f[j,i], plus( f[j,k], f[k,i] ))
   end;
```

Figure 8 Window Algorithm, Fill Corners $O(n^2b)$.