

All pair shortest paths

The all pair shortest path problem is the following: We are given a weighted, usually directed graph $G(V, E)$ and its cost function $c: E \rightarrow \mathbb{R}$. We want to find the shortest path from each vertex to each other vertex.

This problem might arise if we want to make a table of distances between all pairs of cities on a road map. Or we might be given the flying times, fares or distances on certain routes connecting cities, and we wish to make a table that gives the shortest time required to fly/the cheapest or shortest route from any one city to any other. We usually want to put the results in a table.

Singlesource shortest path algorithms

Certainly we can do this by using a single source shortest path algorithm that we run once for each vertex as the source. Thus we can apply Dijkstra's algorithm, if we have only positive edge weights. If we use binary heap implementation of the priority queue, the running time is $O(nm \lg n)$.

If there are also negative weighted edges (but no negative cycles), we need the Bellman-Ford algorithm. The resulting running time is $O(n^2 m)$.

But there are also other, direct possibilities to solve the problem, which I want to describe in the following pages.

Representation of the data

We want the output in tabular form: the entry in u 's row and v 's column should be the weight of a shortest path from u to v . So we represent the data in an adjacency matrix.

Our input will be an $n \times n$ matrix C representing the edge weights of a graph with n vertices:

$$c_{ij} = \begin{cases} 0 & \text{if } i=j, \\ \text{the weight of edge } ij & \text{if } i \neq j \text{ and } (i,j) \in E, \\ \infty & \text{if } i \neq j \text{ and } (i,j) \notin E. \end{cases}$$

Negative-weighted edges are allowed, but we don't want to have negative weight cycles for the moment.

We want to compute an $n \times n$ matrix D where the entry d_{ij} contains the weight of a shortest path from i to j .

Matrix multiplication

One method to compute this matrix is a procedure that is very similar to matrix multiplication on real numbers. The algorithm will look like repeated matrix multiplication.

Let $d_{ij}^{(m)}$ be the minimum weight of any path from i to j that contains at most m edges.

If $m=0$, there is a shortest path from i to j with no edges if and only if $i=j$.

$$d_{ij}^{(0)} = \begin{cases} 0 & \text{if } i=j, \\ \infty & \text{if } i \neq j. \end{cases}$$

If $m=1$, the shortest path from i to j with only one edge of course is the entry in matrix C , so

$$d_{ij}^{(1)} = c_{ij}.$$

Form > 1 ,

$$d_{ij}^{(m)} = \min(d_{ij}^{(m-1)}, \min_{1 \leq k \leq n} \{d_{ik}^{(m-1)} + c_{kj}\})$$

Since $c_{jj} = 0$ for all $j \in V$, this equals

$$(1) \quad d_{ij}^{(m)} = \min_{1 \leq k \leq n} \{d_{ik}^{(m-1)} + c_{kj}\}.$$

If there are no negative cycles, all shortest paths are simple and thus contain at most $n-1$ edges.

Hence

$$d(i, j) = d_{ij}^{(n-1)}.$$

In order to obtain the shortest path weights, we can compute a series of matrices $D^{(1)}, D^{(2)}, \dots, D^{(n-1)}$ with $D^{(m)} = (d_{ij}^{(m)})$. By looking at all possible predecessors k of j , in each step we extend the shortest paths computed so far by one more edge.

The final matrix $D^{(n-1)}$ will contain the actual shortest path weights.

The following procedure takes matrices $D^{(m-1)}$ and C and returns $D^{(m)}$. That is, it extends the shortest paths computed so far by one more edge.

EXTEND-SHORTEST-PATHS(D, C)

$n \leftarrow \text{rows}[D]$

let $D' = (d'_{ij})$ be an $n \times n$ matrix

for $i \leftarrow 1$ **to** n

do for $j \leftarrow 1$ **to** n

do $d'_{ij} \leftarrow \infty$

for $k \leftarrow 1$ **to** n

do $d'_{ij} \leftarrow \min(d'_{ij}, d_{ik} + c_{kj})$

return D'

Its running time is $O(n^3)$ due to the three nested **for** loops.

If we make a number of substitutions, we can see the relation to matrix multiplication.

Assume we want to compute $C = A \cdot B$.

Then

$$(2) \quad c_{ij} = \sum_{k=1}^n a_{ik} \cdot b_{kj}$$

If we make the following substitutions

$$d^{(m-1)} \rightarrow a$$

$$c \rightarrow b$$

$$d^{(m)} \rightarrow c$$

$$\min \rightarrow +$$

$$+ \rightarrow \cdot$$

we obtain (2) from (1).

So as I already mentioned, we compute the shortest path weights by extending the shortest paths edge by edge.

This sequence of then $n-1$ matrices $D^{(1)}, D^{(2)}, \dots, D^{(n-1)}$ will be the following:

$$\begin{aligned} D^{(1)} &= D^{(0)} \cdot C = C, \\ D^{(2)} &= D^{(1)} \cdot C = C^2, \\ D^{(3)} &= D^{(2)} \cdot C = C^3, \\ &\vdots \\ D^{(n-1)} &= D^{(n-2)} \cdot C = C^{n-1} \end{aligned}$$

The following procedure computes this sequence:

SLOW-ALL-PAIRS-SHORTEST-PATHS (C)

```

n ← rows[C]
D(1) ← C
for m ← 2 to n - 1
    do D(m) ← EXTEND-SHORTEST-PATHS(D(m-1), C)
return D(n-1)
    
```

The running time of this procedure will be $O(n^4)$ because we run EXTEND-SHORTEST-PATHS, whose running time is $O(n^3)$, n times.

Improving the running time

We can improve this running time, if we do not compute all $D^{(m)}$ matrices. We are interested only in $D^{(n-1)}$, which is $D^{(m)}$ for all integers $m \geq n-1$. So if we compute the sequence

$$\begin{aligned} D^{(1)} &= C, \\ D^{(2)} &= C^2 = C \cdot C, \\ D^{(4)} &= C^4 = C^2 \cdot C^2, \\ D^{(8)} &= C^8 = C^4 \cdot C^4, \\ &\vdots \\ D^{(2^{\lceil \lg(n-1) \rceil})} &= C^{2^{\lceil \lg(n-1) \rceil}} = C^{2^{\lceil \lg(n-1) \rceil - 1}} \cdot C^{2^{\lceil \lg(n-1) \rceil - 1}}, \end{aligned}$$

we need only $\lceil \lg(n-1) \rceil$ matrix products.

$2^{\lceil \lg(n-1) \rceil}$ is greater or equal to $n-1$, so the final product is equal to $D^{(n-1)}$.

The following procedure computes this sequence.

FASTER-ALL-PAIRS-SHORTEST-PATHS (C)

```

n ← rows[C]
D(1) ← C
m ← 1
while n - 1 > m
    do D(2m) ← EXTEND-SHORTEST-PATHS(D(m), D(m))
    m ← 2 * m
return D(m)
    
```

The running time of this procedure is $O(n^3 \lg n)$ since each of the $\lceil \lg(n-1) \rceil$ matrix products takes $O(n^3)$ time.

Floyd's algo rithm

Another possibility to compute all pair shortest paths is to use the Floyd algorithm. Again, we have a weighted directed or undirected graph, for the time we have nonnegative weight cycles. So if the graph is not directed, we have nonnegative weight edges.

This time, we consider the intermediate vertices of a shortest path from i to j . An intermediate vertex is any vertex of the path other than i and j .

The algorithm is based on the following observation:

We define V as the set $\{1, 2, \dots, n\}$. For any pair of vertices i and j , we take all paths from i to j whose intermediate vertices are all taken from a subset $\{1, 2, \dots, k\}$, and then look for the minimum path from among them, which is again simple because we have nonnegative cycles.

Now let $d_{ij}^{(k)}$ be the weight of a shortest path from i to j with intermediate vertices $\{1, 2, \dots, k\}$.

If $k=0$, we have no intermediate vertices at all, and hence $d_{ij}^{(0)} = c_{ij}$.

For $k > 0$,

$$d_{ij}^{(k)} = \begin{cases} c_{ij} & \text{if } k=0, \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{if } k \geq 1. \end{cases}$$

Here we use a relation ship between this path and the shortest paths from i to j from $\{1, 2, \dots, k-1\}$. If k is not in the shortest path from i to j with intermediate vertices in $\{1, 2, \dots, k-1\}$, all vertices of the path are in the set $\{1, 2, \dots, k-1\}$. If k is in this path, we can decompose the path into a path from i to k and a path from k to j with intermediate vertices only from $\{1, 2, \dots, k-1\}$.

So again we compute a series of matrices, this time by enlarging our set by one vertex in each step. The result will be $D^{(n)} = (d_{ij}^{(n)}) = d(i, j)$ because then all intermediate vertices are in the set $\{1, 2, \dots, n\}$, which represents the whole graph.

The following procedure computes these matrices by increasing the value of k .

FLOYD(C)

$n \leftarrow \text{rows}[C]$

$D^{(0)} \leftarrow C$

for $k \leftarrow 1$ **to** n

do for $i \leftarrow 1$ **to** n

do for $j \leftarrow 1$ **to** n

$d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$

return $D^{(n)}$

Its running time is determined by the triply nested **for** loops and amounts to $O(n^3)$

In the beginning, we said that we don't want to have negative cycles in the given graphs, but actually both algorithms, MATRIX MULTIPLICATION and FLOYD could discover negative cycles in a graph. If there are negative cycles, the shortest paths from a vertex i in this cycle to itself won't be 0 anymore, but infinitely small. If we run one of our algorithms on a graph with negative cycles, the diagonal elements in the matrix D , which should be zero, will, for each entry d_{ii} with i element of the negative cycle, contain the costs, or better profits, that occur, if we run through the negative cycle once.

Constructing the shortest paths

Sometimes we want to know, which vertices the shortest path passes. There are a variety of different methods to find this out. We can compute a predecessor matrix $P = (p_{ij})$, where p_{ij} is NIL if $i=j$ or if there is no path from i to j , otherwise p_{ij} is the predecessor of j on a shortest path

from i to j . We can do this similarly to Floyd's algorithm just as the algorithm works: we compute a sequence of matrices $P^{(0)}, P^{(1)}, \dots, P^{(n)}$ where $P = P^{(1)}$ and $p_{ij}^{(k)}$ is the predecessor of j on the shortest path from i to j so far.

$$p_{ij}^{(k)} = \begin{cases} p_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\ p_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}. \end{cases}$$

Transitive closure

Another problem, that can be solved by using an algorithm similar to Floyd's algorithm, is to find the transitive closure of a graph. This means to find out *whether* there is a path from i to j in the given graph and add an edge from i to j if it is so. The transitive closure of G will be $G^* = (V, E^*)$, where $E^* = \{(i, j) : \text{there is a path from } i \text{ to } j \text{ in } G\}$.

One way to compute the transitive closure, with running time $O(n^3)$, is to assign a weight of 1 to each edge and run the Floyd algorithm. The output will be either infinity or the number of edges on the path from i to j with the least edges.

Another way is to substitute AND and OR by min and + in the Floyd algorithm. The running time will also be $O(n^3)$, but this way can save time and space in practice, because on some computers logical operations on single-bit values execute faster than arithmetic operations on integer words of data, and the storage of Boolean values requires less space than the Floyd algorithm.

Again we compute a series of matrices $T^{(k)} = (t_{ij}^{(k)})$ in order of increasing k :

$t_{ij}^{(k)} = 1$ if there exists a path in G from i to j with intermediate vertices from 1 to k , 0 otherwise.

We construct the transitive closure by putting an edge (i, j) into E^* if $t_{ij}^{(n)} = 1$.

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{if } i \neq j \text{ and } (i, j) \notin E \\ 1 & \text{if } i = j \text{ or } (i, j) \in E \end{cases}$$

$k \geq 1$:

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \text{ OR } (t_{ik}^{(k-1)} \text{ AND } t_{kj}^{(k-1)}).$$

We convert this into the following procedure:

TRANSITIVE-CLOSURE (G)

$n \leftarrow |V[G]|$

for $i \leftarrow 1$ **to** n

do for $j \leftarrow 1$ **to** n

do if $i = j$ **or** $(i, j) \in E[G]$

then $t_{ij}^{(0)} \leftarrow 1$

else $t_{ij}^{(0)} \leftarrow 0$

for $k \leftarrow 1$ **to** n

do for $i \leftarrow 1$ **to** n

do for $j \leftarrow 1$ **to** n

do $t_{ij}^{(k)} \leftarrow t_{ij}^{(k-1)} \text{ OR } (t_{ik}^{(k-1)} \text{ AND } t_{kj}^{(k-1)})$

return $T^{(n)}$

Bibliography

- A.Aho,J.E.HopcroftandJ.D.Ullman(1987): *DataStructuresandAlgorithms*, Addison-WesleyPublishingCompany,ReadingMassachusetts
- A.Aho,J.D.Ullman(1996): *Informatik.DatenstrukturenundKonzeptederAbstraktion*, InternationalThompsonPublishing,Bonnu.a.
- T.H.Cormen;C.E.Leiserson;R.L.Rivest(1990): *IntroductionintoAlgorithms*, TheMIT Press,Cambridge,Mass.u.a.
- T.Ottmann;P.Widmeyer(1996): *AlgorithmenundDatenstrukturen*,SpektrumVerlag Heidelberg
- R.Sedgewick(1988): *Algorithms*,Addison -Wesley
- A.Brandstädt(1994): *GraphenundAlgorithmen*,Teubner,Stuttgart
- R.E.Bellman,K.L.Cooke,J.A.Lockett(1970): *Algorithms,graphsandcomputers*,Acad. Pr.,NewYorku.a.
- C.Berge(1985): *Graphs*,NorthHolland,Amsterdamu.a.
- S.Even(1979): *Graphalgorithms*,Springeru.a.,Berlinu.a.