Floyd-Warshall Again

Given a graph with nodes N_0, N_1, N_2, ..., N_n and knowledge of which are directly connected by arcs, and the lengths of those arcs, find the shortest path between N_i and N_j for all i and j, all at the same time.

We'll treat the traditional sequence of matrices as a function instead this time: `SP(a, i, j)` is the length of the shortest path between N_i and N_j, with the parameter `a` allowing us to reach the solution step-by-step. When `a` is small we are hardly allowed to do anything, as `a` grows we get more and more options and when `a` reaches its maximum there will be no restrictions so we'll have a complete solution (assuming that we get the step-by-step design right).

The minimum sensible set of options would be to use only what we already know. If we already know something, it will require no effort. What we already know is the input data, the map itself, represented by `SP(0, i, j)`. It is the length of the direct connection (arc or edge) between nodes N_i and N_j. If there is no direct connection between N_i and N_j we still need the function to return something, so we'll use the badly named constant `INFINITY` from `#include <cmath>` but abbreviate it to `inf`.

To recap, the first effortless step is

`SP(0, i, j)` = length of arc directly connecting nodes N_i and N_j if there is one, or `inf` if there isn't.

`SP(0, i, j)` is a given, the map, part of the problem statement, we do not have to calculate `SP(0, i, j)` in any way.

The enormous number of possibilities needs to be tamed, so that a sensible sequence of steps can be found. Gradual increments to what we already know is generally a good idea to try. That is what increasing the parameter a signifies.

What we already know is the map itself, `SP(0, i, j)` for all i and j. Consider the first parameter, `a` = 0 here, to represent a set of possibilities for way-points (nodes that can be visited on the way from i to j). 0 represents the empty set { }, there are no nodes we can visit on the way, so only direct connections can be used.

`a` = 1 represents a set with only one element. We should hope that it will make no difference what order we consider the nodes in, but starting with 0 and increasing by one makes the notation much clearer. That means that `a` = 1 represents the set of nodes { N_0 }, `a` = 2 represents the set of nodes { N_0, N_1 }, `a` = 3 represents the set of nodes { N_0, N_1, N_2 }, and so on. To be perfectly clear, the case of `a` = 3 means that we are trying to get from N_i to N_j and may stop at any, all, or none of N_0, N_1, N_2 along the way. In other words we would be working out `SP(3, i, j)`.

Start at the beginning and introduce node N_0:

This is the first node we will allow ourselves to visit on the way from N_i to N_j, and (by definition) the values of `SP(1, i, j)` will be the shortest path length from N_i to N_j, if we are allowed to visit N_0 on the way. We don't have to visit N_0, but we can if we want to.

Of course we have no way of knowing at this stage whether or not we should want to stop at N_0 on the way, so we have to consider both options, visit N_0 or don't visit N_0.

Clearly when i = 0 or j = 0, N_0 is already one of the end-points, so being allowed to visit N_0 on the way can't possibly make any difference. Only when i and j are both non-zero do we have to make the choice about stopping at N_0.

Option 1, don't visit N_0 on the way:

N_0 is the only new stopping place allowed when `a` (`SP`'s first parameter) is increased to 1, so if we choose not to visit N_0, we already have the answer, the shortest path with no stops on the way, `SP(0, i, j)`.

Option 2, do visit N_0 on the way:

Remember we don't have to visit N_0, this is just what happens if choose to. To get from N_i to N_j in the shortest distance, stopping at N_0 on the way, we must travel first from N_i to N_0, then from N_0 the rest of the way to N_j. In those two sub-journeys it should be obvious that visiting N_0 again could only make the trip longer.

That means that the trip from N_i to N_0 will not stop at N_0 along the way, and the trip from N_0 to N_j will also not stop at N_0 on the way. Plus we are working on `SP(1, i, j)` meaning that no other node is allowable either. So the shortest distance from N_i to N_0 not stopping at N_0 or above will already be `SP(0, i, 0)` and the rest of the trip will be `SP(0, 0, j)`, giving a total length of `SP(0, i, 0) + SP(0, 0, j)`.

Those are the only two options. `SP(1, i, j)` must be either

`SP(0, i, j)`

or `SP(0, i, 0) + SP(0, i, j)`.

As we want the shortest distance we must take the least of those two values. That gives the recursive solution

`SP(1, i, j) = min(SP(0, i, j), SP(0, i, 0) + SP(0, 0, j))` for all i and j.

Now we move on to a = 2. `SP(2, i, j)` is the length of the shortest path between N_i and N_j given that we are allowed to stop at N_0 and N_1 (either or both or neither) along the way. Whether or not we stop at N_1 has no bearing on whether or not we stop at N_0, they are totally separate decisions.

Again there are only two options. Either we do stop an N_1 on the way, or we don't. Clearly if i or j are 1, stopping at N_1 again on the way will only lengthen the journey, so it can be ignored.

But specifically ignoring i = 1 and j = 1 would complicate the programming. It is also totally unnecessary because a journey that re-visits N_1 would be longer than one that doesn't, and we always take the minimum so the results when i = 1 or j = 1 would never be used.

Option 1, don't stop at N_1 on the way:

We are still allowed to stop at N_0 (but don't have to), which means that the best path from i to j (able to stop at N_0 but nowhere else) is going to be the already known `SP(1, i, j)`, by definition.

Option 1, do stop at N_1 on the way:

As well as stopping at N_1, we are still allowed to stop at N_0 (but don't have to), that is a totally separate decision. We have to get from N_i to N_1, and then from N_1 to N_j. Those two trips will not include N_1 again, but may include N_0, so they have already been programmed: `SP(1, i, 1)` and `SP(1, 1, j)`.

Those are again the only two options, so the recursion is the same as before, just with parameter `a` being one bigger:

```
SP(2, i, j) = min(SP(1, i, j), SP(1, i, 0) + SP(1, 0, j))
```

for all i and j.

It keeps on the same way. When we're calculating (just for example) `SP(7, i, j)` we are allowed to stop at any or all of N_0, N_1, N_2, N_3, N_4, or N_5 on the way, so the possibilities if we don't stop at the newly allowed N_6 are already known as `SP(6, ...)`. If we don't stop at N_6 the shortest path is `SP(6, i, j)`. If we do stop at N_6 on the way from N_i to N_j it will be `SP(6, i, 6)` + `SP(6, 6, j)`.

The final function needs to have the initial data (the inputs to the problem: knowledge of the lengths of all direct arcs) available, it doesn't matter how that information is represented, so let's just assume they are in a two dimensional array `AM` (for Adjacency Matrix) such that `AM[i][j]` = `inf` if there is no arc from N_i to N_j, or the length of that arc if there is one. That gives us

```
double SP(int a, int i, int j)
{ if (a == 0)
    return AM[i][j];
  else
  { double option1 = SP(a - 1, i, j);
    double option2 = SP(a - 1, i, a - 1) +
                     SP(a - 1, a - 1, j);
    return min(option1, option2); } }
```

Remember that `inf` is respected by the arithmetic hardware, if `x` is a proper number, then

```
inf + x = inf,
x + inf = inf,
inf + inf = inf,
min(inf, x) = x,
min(x, inf) = x,
min(inf, inf) = inf.
```

That means we just don't have to take `inf` into consideration at all.

The recursive solution will be exceptionally slow because there will be a lot of recursive calls to `SP` with exactly the same parameters. Consider a graph with 10 nodes. Each call to SP with `a` = 10 involves three recursive calls with `a` = 9, and each of those involves three more recursive calls with `a` = 8. It only stops when `a` = 0 so there will be 3^{10} = 59,049 calls.

On the other hand `a`, `i`, and `j` are all restricted to the range 0 to 9 (actually 10 for `a`) so there are only 1,000 possible different function calls.

Memoisation - recording all previous calls and their results will help a lot:

```
double oldresults[big][big][big] = { all -1 };

double SP(int a, int i, int j)
{ if (oldresults[a][i][j] != -1)
    return oldresults[a][i][j];
  if (a == 0)
    return AM[i][j];
  else
  { double option1 = SP(a - 1, i, j);
    double option2 = SP(a - 1, i, a - 1) +
                     SP(a - 1, a - 1, j);
    double answer =  min(option1, option2);
    oldresults[a][i][j] = answer;
    return answer; } }
```

Dynamic programming simplifies and speeds the computation by completing the `oldresults` array (now just called `results` array, as there is nothing old about them any more) with all possible answers before any questions get asked. The recursive solution makes the pattern clear: if we only compute `oldresults[a][...][...]` after all the `oldresults[a - 1][...][...]` have already been computed, we will have everything we need:

```
for (int a = 0; a < big; a += 1)
  for (int i = 0; i < big; i += 1)
    for (int j = 0; j < big; j += 1)
    { if (a == 0)
        answer = AM[i][j];
      else
      { double option1 = results[a - 1][i][j];
        double option2 = results[a - 1][i][a - 1] +
                         results[a - 1][a - 1][j];
        answer =  min(option1, option2); }
      results[a][i][j] = answer; }
```