# The Algorithm

This is just the algorithm taught in class for minimum path cost between two nodes in a graph without negative cycles. The reason we imposed that we use this algorithm on a graph without negative cycles is that the minimum cost then might be negative infinity because you could just loop over the negative cycle infinitely many times to get decreasing costs each time, so you can’t limit the edge count to anything. However, putting an edge limit on it would still make it work – it just gets the minimum cost within the allotted k edges. The algorithm stays the same, just takes in an extra parameter k and runs for k iterations instead of n-1 iterations:

Shortest-Path(G,s,t,k)

Array M[0 ... k, V]

Define M[0,t] = 0 and M[0,v] = ∞ for all over v ∈ V

For i = 1, ..., k

For v ∈ V in any order

node w = the node that minimizes M[i-1,w] + edge\_weight(v,w)

M[i,v] = min(M[i-1,v], M[i-1,w] + edge\_weight(v,w))

Endfor

Endfor

Return M[k,s]

# Runtime Analysis

Iterating through M is O(nk) because there’s k columns for the k edges we’re allowed to traverse and n nodes. Updating each cell in M is O(n) because there’s at most n nodes we have to consider. So altogether, the complexity is

# The Algorithm

Let’s consider the first part for now. Finding the minimum cost path from s to t if G has no negative cycles is just having columns in our table M and returning because, as proved in class and stated in the textbook, “If G has no negative cycles, then there is a shortest path from s to t that is simple (i.e., does not repeat nodes), and hence has at most n − 1 edges”.

Now let’s consider detecting a negative cycle. To return the negative cycle, we need two things, to know that there IS a negative cycle, and the actual path (i.e. node sequence) – not just the weight – from each node to t, so we could return the negative cycle.

To address the former, section 6.10 of the textbook proves that if we let the algorithm above run for one more iteration (have a column ), and see if any of the cells in that column are different from their corresponding cells in the same row but prior column (column ). If anything differs, then we have a negative cycle (according to Piazza post [@101](https://piazza.com/class/jzqwd6s59yh6bm?cid=101), we don’t have to prove what’s in the textbook already).

To address the latter, we would need to store the path as well, not just the path weight. We would maintain an extra table S that stores the paths. The algorithm is as follows:

Get-Negative-Cycle(G,v)

run Shortest-Path(G,v,v)'s algorithm

Define x = the first column where M[x,v] != 0

return M[x,v]

Shortest-Path(G,s,t)

Array M[0 ... n, V]

Array S[0 ... n, V]

Define M[0,t] = 0 and M[0,v] = ∞ for all over v ∈ V

Define S[0,t] = [[t]] and S[0,v] = [[]] for all over v ∈ V

For i = 1, ..., n

For v ∈ V in any order

node w = the node that minimizes M[i-1,w] + edge\_weight(v,w)

if (M[i-1,w] + edge\_weight(v,w) < M[i-1,v)]) then

M[i,v] = M[i-1,w] + edge\_weight(v,w)

S[i,v] = v prepended to all of S[i-1,w]'s lists

else if (M[i-1,w] + edge\_weight(v,w) > M[i-1,v])

M[i,v] = M[i-1,v]

S[i,v] = S[i-1,v]

else

M[i,v] = M[i-1,v]

S[i,v] = join(S[i-1,v], v prepended to all of S[i-1,w]'s lists)

endif

Endfor

Endfor

For v ∈ V in any order

if M[n,v] != M[n-1,v] then

return Get-Negative-Cycle(G,v)

Endfor

Return S[n-1,s]

# Runtime Analysis

There are (n+1)n cells in tables M and S, so it’s to traverse them. Defining each cell takes because there’s possible nodes for . Getting a negative cycle is the same runtime as Shortest-Path because it runs the exact same algorithm. Altogether, it is

1. Our algorithm in part b stores all paths with minimum cost (each cell in table S is a list of lists of nodes), so we just need to see the length of S[n-1,t]. If it’s 1, then it’s unique, it it’s greater than 1, then it’s not unique.