20190906

Dijkstra's Algorithm

Let G be a graph with positive weights on the edges. We want to find the shortest paths (more precisely, the *least weight* paths) from some particular vertex A to all other vertices of G (note that if G is not connected, some vertices may not be reachable from A).

First question:

> Why not just compute the weight of every path from A to each of the other vertices and choose the best? (The BFI approach)

Consider a complete graph (all edges present) – the number of paths is extremely high. Examining them all would take a long time!

Fortunately Edsger Dijkstra found a better solution – his algorithm became the starting point for just about all modern routing algorithms.

Version 1 : Conceptual description

```
# Goal: Find least-weight paths from A to all other vertices
# Weight of edge (v,w) is given by Weight(v,w)

# Call a vertex a candidate if we know at least one path from A to the vertex
# Keep track of the least-weight path found to each vertex – call this the
#   estimated cost to reach that vertex
# On each iteration:
#       Choose x to be the candidate with the lowest estimated cost,
#               and set x's actual cost to be its current estimated cost
#       Update the estimated costs to the neighbours of x
```

Version 2 : Adding details to keep track of the status of the vertices, and details of updating estimates

```
# Goal: Find least-weight paths from A to all other vertices
# Weight of edge (v,w) is given by Weight(v,w)

Cost(A) = 0
Reached = {A}
for each vertex x, other than A:
    Estimate(x) = infinity
for each neighbour x of A:
    Estimate(x) = Weight(A,x)
Candidates = {all neighbours of A}

while not finished:
    Let v be a vertex in Candidates with the lowest Estimate
    Cost(v) = Estimate(v)
    add v to Reached
    remove v from Candidates
    for each neighbour y of v such that y is not in Reached:
        if Cost(v) + Weight(v,y) < Estimate(y):
            Estimate(y) = Cost(v) + Weight(v,y)
            add y to Candidates      # y may already be in Candidates
```

As we get closer to an implementation of the algorithm we have to start making decisions. For example we have to decide how to represent the sets of vertices. For this discussion I'm going to use a very simple method: each set will be represented by a Boolean vector, containing "True" for each vertex that is in the set and "False" for each vertex not in the set.

I'll use integer arrays to hold the Cost and Estimate values

Version 3 : Choosing to implement sets as Boolean arrays

```
# Goal: Find least-weight paths from A to all other vertices
# Weight of edge (v,w) is given by Weight(v,w)

Cost[A] = 0                    # Cost is an integer array
Reached[A] = True              # Reached is a Boolean array
for each other vertex x:
    Reached[x] = False
for each neighbour x of A:
    Estimate[x] = Weight(A,x)      # Estimate is an integer array
    Candidate[x] = True            # Candidate is a Boolean array
for all other vertices z:
    Estimate[z] = infinity
    Candidate[z] = False
while not finished:
    Let v be a vertex with Candidate[v] == True,  with the lowest Estimate
    Cost[v] = Estimate[v]
    Reached[v] = True
    Candidate[v] = False
    for each neighbour y of v such that Reached[y] == False:
        if Cost[v] + Weight(v,y) < Estimate[y]:
            Estimate[y] = Cost[v] + Weight(v,y)
            Candidate[y] = True
```

Essential questions:

Question 1 : Can we prove that this algorithm always finds the correct solution?

We'll discuss an informal proof here – it's not hard to transform it into a full proof – it's a good exercise.

The structure of the proof is to assume that the algorithm has not made any mistakes up to a certain point, and then show that it does not make a mistake on the next iteration. Combining this with the observation that the algorithm has not made any mistakes when Reached contains just the vertex A, we get an inductive proof that the algorithm does not make a mistake on any iteration – so its solution is correct.

Suppose that at the beginning of some iteration, the algorithm has correctly found the shortest path lengths to all the vertices in Reached.

Let x be the vertex in Candidates with the lowest Estimate value. The algorithm sets this as the actual shortest path cost for vertex x. Suppose this is incorrect – ie suppose the graph contains a path from A to x with lower cost. But this is impossible because such a path would have to cross over from Reached to Candidates at some point, and all such crossings would go through one of the other candidates ... but these paths cost at least as much as the path to x, and extending any of them through the graph to end at x would increase the cost even more ... so such a path must cost more than the current estimate for reaching x. Therefore there is no lower-cost path to x, and thus the algorithm is correct when it identifies this as the actual cost of reaching x.

As mentioned, we combine this argument with the observation that before the first iteration the algorithm has made no errors. This completes the informal proof.

Question 2 : How fast is this algorithm?

The answer to this question depends on the way we are storing information about the graph. Let's use an adjacency matrix. Because we will store the edge-weights in the matrix we will call it W. Thus W[x][y] will contain the weight of the edge joining x and y. Since the edge-weights are all positive, we can use W[x][y] = 0 to indicate that there is no edge joining x and y

Once we have made this decision we can add a few more details to the algorithm – enough to discuss its speed. Here's the new version of the algorithm:

```
# Goal: Find least-weight paths from A to all other vertices
# Weight of edge (v,w) is given by W[v][w]

Cost[A] = 0
Reached[A] = True
for each other vertex x:
    Reached[x] = False
for each neighbour x of A:
    Estimate[x] = Weight(A,x)
    Candidate[x] = True
for all other vertices z:
    Estimate[z] = infinity
    Candidate[z] = False
while not finished:
    # find the best candidate
    best_candidate_estimate = infinity
    for each vertex x:
        if Candidate[x] == True and Estimate[x] < best_candidate_estimate:
            v = x
            best_candidate_estimate = Estimate[x]
    Cost[v] = Estimate[v]
    Reached[v] = True
    Candidate[v] = False
    for each vertex y:                  # update the neighbours of v
        if W[v][y] > 0  and  Reached[y] == False:
            if Cost[v] + Weight(v,y) < Estimate[y]:
                Estimate[y] = Cost[v] + W[v][y]
                Candidate[y] = True
```

Note that there is one rather glaring short-coming of this algorithm: it doesn't actually tell us the paths! It just computes the costs. Fortunately finding the paths can be done with just a couple of minor additions. Basically all we need to know for each vertex is "what vertex comes just before this one on the least-cost path from A?" This is easy to keep track of – we just add an array called Predecessor and make sure Predecessor[x] contains the identifier of the appropriate vertex.

Now the algorithm looks like this:

```
# Goal: Find least-weight paths from A to all other vertices
# Weight of edge (v,w) is given by W[v][w]

Cost[A] = 0
Reached[A] = True
for each other vertex x:
    Reached[x] = False
for each neighbour x of A:
    Estimate[x] = Weight(A,x)
    Candidate[x] = True
for all other vertices z:
    Estimate[z] = infinity
    Candidate[z] = False
while not finished:
    # find the best candidate
    best_candidate_estimate = infinity
    for each vertex x:
        if Candidate[x] == True and Estimate[x] < best_candidate_estimate:
            v = x
            best_candidate_estimate = Estimate[x]
    Cost[v] = Estimate[v]
    Reached[v] = True
    Candidate[v] = False
    for each vertex y:                  # update the neighbours of v
        if W[v][y] > 0  and  Reached[y] == False:
            if Cost[v] + Weight(v,y) < Estimate[y]:
                Estimate[y] = Cost[v] + W[v][y]
                Candidate[y] = True
                Predecessor[y] = v
```

This is the version you will be asked to implement and test in your Week 1 lab.