Shortest paths

In the last two lectures we examined how to search for all vertices `w` that can be reached by a (directed) path from a given vertex `v`. (This starting vertex is sometimes called the source vertex.) We had a field `visited` for each vertex, which was initialized to `false`. If this vertex is reached during the search, the field `visited` is assigned the value `true`.

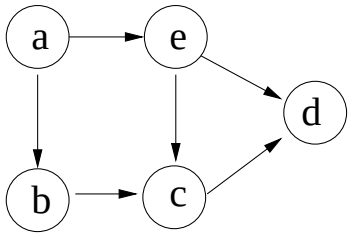
Suppose we would like to know *more explicitly* what is the shortest path to each vertex that can be reached from the source vertex. How can we modify the breadth first search algorithm we saw last lecture to achieve this?

Let's define a field `pathlength` for each vertex (e.g. `w.pathlength`) which holds our best current estimate for the shortest path from source `v_s` to vertex `w`. We initialize it to `INF` i.e. infinity, which is the distance if there is no path. We also define a field `w.prev` which keeps track of the vertex coming before `w` on the currently estimated shortest path. Here is the new algorithm:

```
shortestPath(v_s){
   for each vertex w
      w.pathlength = INF
   v_s.pathlength = 0
   make empty queue q
   q.enqueue(v_s)
   while (!q.empty){
     v = q.dequeue()
     for each w in adjList(v){
       if w.pathlength == INF{
         q.enqueue(w)
         w.pathlength = v.pathlength + 1
         w.prev = v
       }
     }
   }
}
```

This is basically the same as the `bfs` algorithm from last lecture. The main difference is that we are now keeping track of the path length and the previous node visited along the path.

Example



```
ADJACENCY LIST              QUEUE (v,v.pathlength,v.prev) snapshots

    a -  b,e                  (a,0,NULL)
    b -  c                    (b,1,a), (e,1,a)
    c -  d                    (e,1,a), (c,2,b)
    d -                       (c,2,b), (d,2,e)
    e -  c,d                  (d,2,e)
```

Correctness (only sketched in class – NOT on exam)

How can we be sure that this gives us the shortest path to each vertex ? The proof is by induction. We want to show that the algorithm gives the shortest path for all vertices whose shortest path is of length less than or equal to $k = |V|$.

The base case $k = 0$ is obvious (the path from source to itself has length 0). The induction hypothesis is that the algorithm correctly identifies the vertices whose shortest paths have pathlength less than or equal to k. The induction step is to show that it must therefore identify correctly any vertex whose shortness pathlength is $k + 1$.

Consider a vertex `w'` whose shortest path length from the source `v_s` is $k + 1$. There must exist at least one vertex `v'` which has an edge `(v',w')` and whose shortest shortest path from the source is of length k. (Note `v'` cannot have a shortest path of length less than k, since then `w` would have shortest path of length less than $k + 1$.) By the induction hypothesis, the algorithm correctly identifies the path length of any such vertex `v'`. (Note, there may be more than one such `v'`.)

Consider the first of these `v'` vertices to be removed from the queue. When it is removed, `v'.pathlength` is k. During that pass through the `while` loop, the vertex `w` will be examined since `w` is in `adjList(v')`. By the assumptions above, it is the first time `w'` will be examined, so `w'.pathlength` will be `INF` before it is examined. And since `(v',w')` is an edge, `w'.pathlength` will be assigned $k+1$, and `w'` will be enqueued. Once a vertex is enqueued, its computed pathlength is fixed. (And every node eventually gets removed from the queue.) So we are done.

Weighted shortest path (Dijkstra's algorithm)

For many graphs, there is a positive weight associated with each edge. In a transportation network, this could be the distance or time or cost taken to travel along that edge. We associate a `length(v,w)` to the edge `(v,w)`. If there is no edge `(v,w)` in the graph, then we can consider the length of such an edge to be infinite. Our problem is to compute the path of shortest length (or least total cost) from a given source node `v_s` to every other node, that is, the path `(v_s,..., w)` such that the *sum of the lengths* of the edge in this path is minimized.

Notice that this problem is generalization of the previous problem. There, we were addressing the case where the length of each edge was 1.

The algorithm below is called Dijkstra's algorithm[1] It is similar to the algorithm above, except that we use a priority queue rather than just a queue. The priority of a vertex is the current estimate of the `pathlength` of the shorted weighted path. The algorithm removes the vertex `v` with highest priority (lowest pathlength). One can show that this gives the shortest `pathlength v_s` to vertices `v` remaining in the priority queue (proof deferred to COMP 251).

[1] after Edsger Dijkstra, who discovered this method in the 1950's (advice: pronounce his name "Dike-stra").

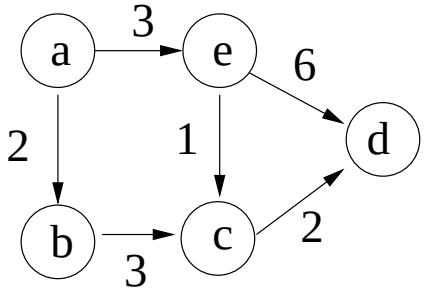
```
weightedShortestPath(G,v_s){
  for every vertex w in V{
    w.prev = null
    w.pathlength = INF
    q.enqueue(w)                        // NEW
  }
  v_s.pathlength = 0
  while (!q.empty)
    v = q.dequeue()                    //  removeMin i.e. priority queue

    for every w in adjList(v){
      if w in queue{
        tmp = v.pathlength + length(v,w)
        if (tmp < w.pathlength){
          w.pathlength = tmp
          adjust w's position in queue  //  e.g.  upHeap   O(log |V|)
          w.prev = v
        }
      }
    }
  }
```

Example

Let's work through an example. The adjacency list and edge weights are as follows:

```
a -  (b,2), (e,3)
b -  (c,3)
c -  (d,2)
d -
e -  (c,1), (d,6)
```

and we take `a` to be the source vertex. We draw the graph:



After each pass through the *while* loop, we have the following:

a	b	c	d	e	so remove ...
INF	INF	INF	INF	INF	
(0,null)	INF	INF	INF	INF	a
	(2,a)	INF	INF	(3,a)	b
		(5,b)	INF	(3,a)	e
		(4,e)	(9,e)		c
			(6,c)		d

Run time

How long does it take to run this Dijkstra's algorithm ? The `for` loop runs $|V|$ times, so the big O is at least $O(|V|)$. How does the runtime depend on $|E|$? Each vertex `w` is removed from the priority queue at some time. When that happens, all edges in `adjList(w)` are examined. In the worst case, the `pathlength` field is updated each time, and the priority queue needs to be adjusted. If a heap is used, then this adjustment takes $O(\lg |V|)$ time. Since it can happen once per edge, this implies that the algorithm runs in time $O(|V| + |E| \lg |V|)$.

Are we surprised that it doesn't run in $O(|V|+|E|)$, like the *unweighted* shortest path algorithm?[2] No. The new problem is more general, since it allows arbitrary positive weights for the edges, not constant weights. Since there is more information to consider in the solution of the general problem, we should not be surprised it takes longer to solve.

[2]To see where that O() comes from, note that the algorithms are basically the same, but the unweighted algorithm uses a regular queue, not a priority queue.