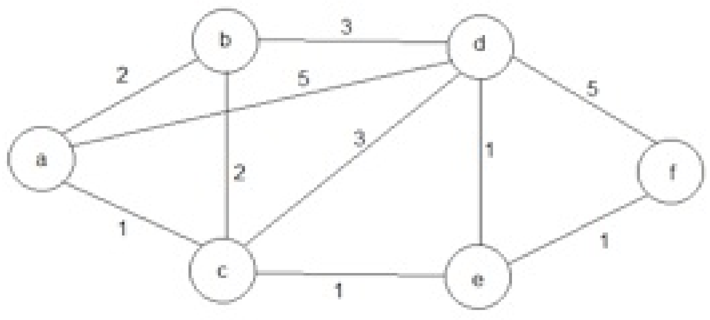
Lab 12 part 6: Finding shortest paths – Dijkstra algorithm

We have used the DFS and BFS algorithms to search a graph. In particular both algorithms find all vertices (also the paths) for which there is a path from a source (origin) vertex. The DFS algorithm uses a stack while the BFS algorithm uses a queue.

The BFS algorithm gives us the shortest paths when applied to unweighted graphs. However, we cannot use it for getting the shortest paths in weighted graphs, where the edges have non-negative costs (weights) associated with them. Dijkstra's algorithm gives us the shortest paths from a given source vertex to all other vertices in weighted graphs.

We will explain the algorithm using the following example:

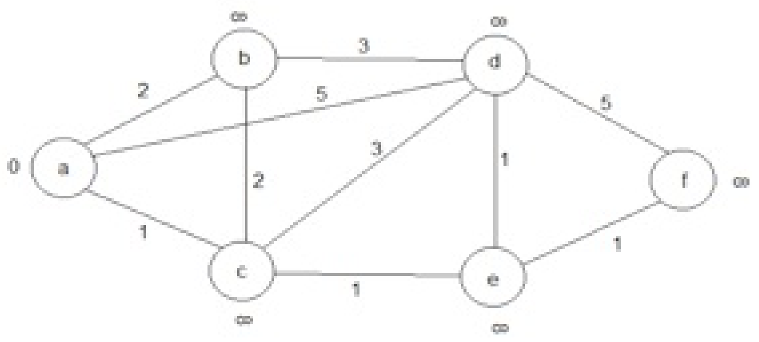


The source vertex is `a`, and the purpose of the algorithm is to find shortest paths from `a` to all other vertices: `b`, `c`, `d`, `e`, and `f`.

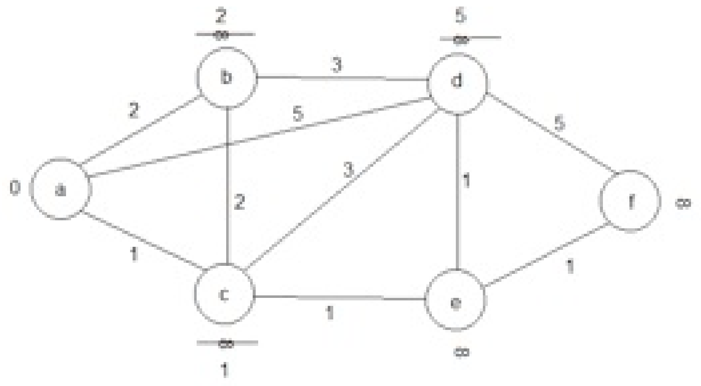
The algorithm keeps track of two things:

- the distance (cost) of each node from the source node,
- and the collection of nodes called `unvisited` whose distance from the source node has not been finalized. This collection will be stored in a Priority Queue.

It starts by assigning `0` as the distance for the source vertex and infinity (`math.inf` or `float('inf')`) for all other vertices. Moreover, all vertices are marked as unvisited (added to the priority queue). This is how the above graph will look like after this step:

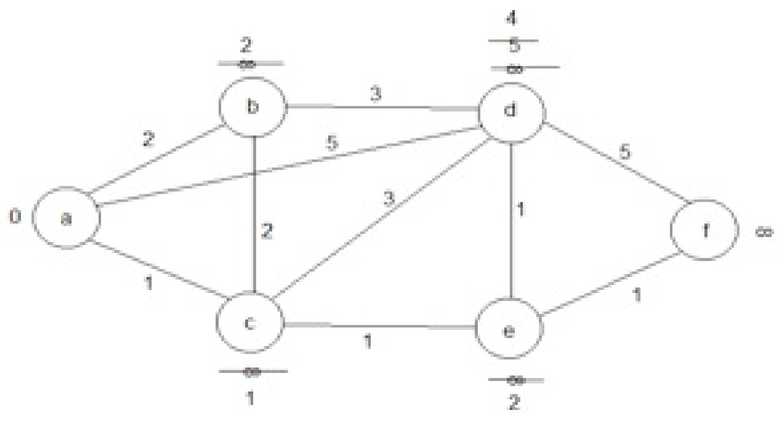


Now the main loop will run as long as we have unvisited vertices. Out of the unvisited vertices we pick the one with the minimum value of its distance (cost). In our case it will be `a`. Now we consider the neighbors of `a`: `b`, `c`, and `d`. The cost of `b` is infinity which means the distance to get to this vertex is infinity, but we can get there shorter: The distance of `a` is `0` + the weight of the edge `(a,b)`, which is `2`; their sum is `0+2 = 2`, less than infinity. So we update the cost of `b` to be `2` instead of infinity. We do the same for `c` and `d` and the new graph will look like:



Notice that we can write somewhere (a dictionary for building the search tree) that we get to `b` from `a`. This will help us build the actual paths later.

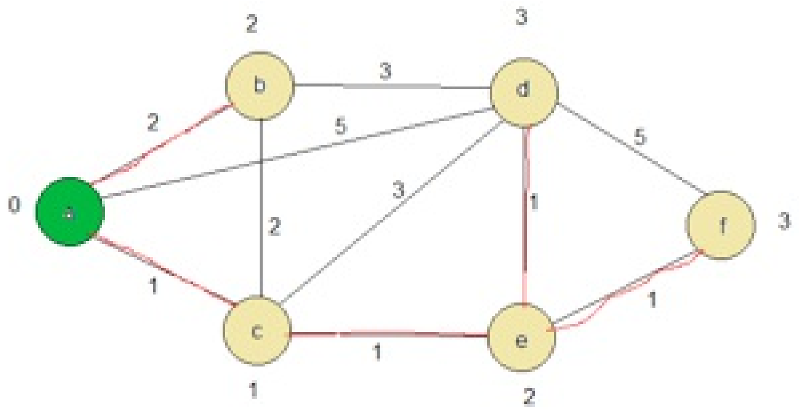
Now we go to the next iteration and pick the next minimum vertex. In this case it will be `c`. The neighbors of `c` are `a`, `b`, `d`, and `e`. To get to `a` via `c` it will cost us `1+1=2`, which is more than what `a` has (which is `0`) so we do not update `a`. To get to `b` via `c` will cost us `3` (`1` that `c` has plus the weight of the edge `(c,b)` which is `2`) which is more than what `b` has, so we do not update `b` either. To get to `d` via `c`, we need the cost of `c` which is `1` + the cost of edge `(c,d)` which is `3`; `1+3 = 4` which is less than what `d` has (`5`). In this case we update the cost of `d` to be `4` and update the data structure so that `c` precedes `d`. Finally, we update the cost for `e`. Here is our new graph:



Now we go to the third iteration and pick the unvisited vertex with smallest cost, which is `e` or `b`; let’s pick `e`. Here we have an interesting point. The path we've found so far to get to `d` has cost `4`: we start from `a`, then `c`, then `d`: `0 + 1 + 3 = 4`. But you can see that we get a shorter path to `d` if we pass through `e`. Let’s see how the algorithms finds it.

So we have picked `e` as the next minimum vertex. One of the neighbors is `d`. To get to `d` via `e` we need the cost of `e` + the weight of the edge `(e,d)`. Together it will cost `3` which is less than `4` of `d`. In this case we update the cost of `d` to be `3` and update that the previous vertex of `d` to be `e` (not `c` as before).

Now you can continue this way until iterating over all vertices. The final graph will be this one (where the red lines are the shortest paths from the green a to all the other vertices in yellow):



In this part of the lab, your job is to implement this algorithm:

```
def Dijkstra(G, sourceNode):
    pass
```

You can test it on your own graphs, or on the Actor graph from the previous part of the lab. Putting it together with your work in the previous part, here is how it will look:

```python
mapAtoM = getMapAtoM()
G = createActorGraph(mapAtoM)

paths = Dijkstra(G, 'Bacon, Kevin')
path1 = getPath(G, paths, 'Weaver, Jason')
for x in path1[::-1]:
  print (x)

path2 = getPath(G, paths, 'Costner, Kevin')
for x in path2[::-1]:
  print (x)

path3 = getPath(G, paths, 'Pesci, Joe')
for x in path3[::-1]:
  print (x)
```