CS512 LECTURE NOTES - LECTURE 12

1 Breadth First Search (BFS)

Given an undirected graph $G = (V, E)$ we can define the length of a path $x_0x_1 \dots x_r$ to be equal to the number of edges traversed from x_0 to x_r. Notice that given two vertices u, v there might exists several paths that go from u to v, each one of those paths might have a different distance.

The shortest path from u to v is a path of minimum length. The value of this minimum length is called the distance from u to v. So whenever we use the term *distance* we refer to the shortest path.

We now want to find the distance to every vertex $v \in V$ from a source (start) node s. In order to do this we will process the vertices starting from s, then the neighbors of s (which will be at distance 1), then the neighbors of the neighbors (at distance 2), and so on. If along the way we find a vertex that has already been assigned a distance, we ignore it and continue to the next vertex.

To achieve this effect, we will use a normal Queue with the following operations:

- `enqueue(v)`
- `dequaue()`
- `isEmpty()`

Notice that a standard implementation using linked lists can perform each one of these operations in $O(1)$ time.

We will store two attributes in each vertex:

- `v.dist`: is the distance from s to v
- `v.prev`: is the previous vertex to v on a shortest path from s to v

First we will initialize the distance to every vertex to be ∞, except for s which is 0. Then we will process them using the queue described above.

Algorithm BFS(G,s)

for each vertex $v \in V$
    $v.dist = \infty$
    $v.prev = null$
$s.dist = 0$
Q.enqueue(s)
while not Q.isEmpty()
    v =Q.dequeue()
    for each $(v, u) \in E$
        if $u.dist = \infty$
            $u.dist = v.dist + 1$
            $u.prev = v$
            Q.enqueue(u)

To show the correctness of the algorithm notice that we can prove the following property using induction:

There is a point during the execution of the algorithm where a vertex is at distance d from s if and only if it is in the queue.

The base case is just showing that s is at distance 0. Inductive hypothesis will say that there is a point where a vertex is at a distance k form s if and only if it is in the queue for a fixed k. The inductive step can be easily proven by contradiction assuming that when all vertices from level k have been processed in the queue, there is a a vertex in the queue that is at a distance less than $k+1$ from s, which would imply that one of the previous distances was con computed correctly. The details of the proof are left as an exercise.

1.1 Analysis

We should notice, as we did in the case of DFS, that the number of times that each vertex is "seen" by the algorithm is equal to 2, and since queue operations are all $O(1)$, then the total running time is $O(|V| + |E|)$.

2 Weighted graphs

2.1 Definitions

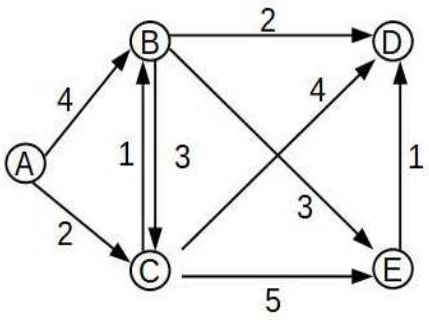
A weighted graph is a graph $G = (V, E)$ that has a weight function

$$w : E \to \mathbb{R}$$

The weight function assigns a weight to each edge. The weights can represent

- Distance
- Capacity
- Number of edges of a multigraph
- etc.

The following is an example of a directed weighted graph:



Weight function:

w(A,B)=4
w(A,C)=2
w(B,C)=3
w(B,D)=2
w(B,E)=3
w(C,B)=1
w(C,D)=4
w(C,E)=5
w(E,D)=1

The length of a path $v_0, v_1, v_2, \ldots, v_k$ is equal to the sum of the weights of its edges:

$$\text{length of path from } v_0 \text{ to } v_k = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

For example, the length of the path from A to E given by A,B,C,E = 4+3+5 = 12

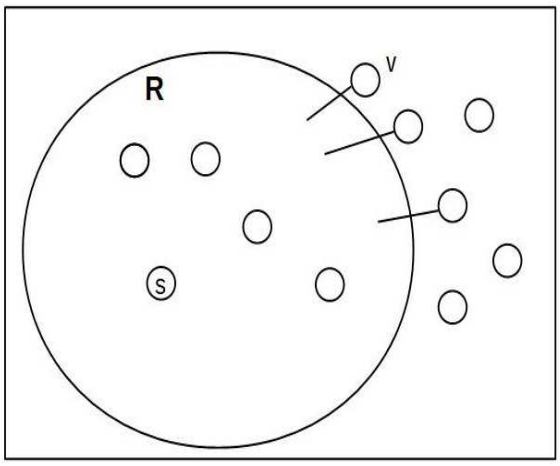
Notice that given a pair of vertices u and v there might be different paths from u to v.

For example: the path A,B,E is also a path from A to E, that has a length of 4+3=7.

2.2 Shortest paths

The shortest path from u to v is a path of minimum length among all of the possible paths from u to v.

We will give an algorithm that given a start vertex s will compute the length of all shortest paths to each one of the vertices in G.



The idea is somewhat similar to BFS:

1. start from s
2. keep finding vertices in order of increasing length from s and adding them to a set R

Assume that at some point during the execution of the algorithm we were able to find a set of vertices R that are closer to s than any vertex outside of R.

We will look for the next vertex to add to R

This next vertex (call it v) must be the closest vertex to s that is not in R.

How do we find it?

Property 1:

v is the closest vertex to s that is not in R

Assume Property 1 is true, then

- v must be adjacent to a vertex in R:

  Let $s, \ldots, x, v$ be a shortest path from s to v, where x is not in R, so we have that

  $$path(s, x) + w(x, v) = dist(s, v)$$

  Notice that $dist(s, v)$ represents the length of a shortest path from s to v and $path(s, x)$ is the length of the path from s to x.

  Clearly $dist(s, x)$, the length of a shortest path from s to x, must be less than or equal to $path(s, x)$, and since $w(x, v) > 0$ we have

  $$dist(s, x) < path(s, x) + w(x, v) = dist(s, v)$$

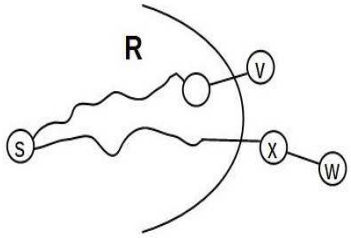
  So we have that

  $$dist(s, x) < dist(s, v)$$

  We can conclude that x would be closer form s than v and so, x would be the closest vertex to s not in R, and v would not satisfy **Property 1**.

  Because of the condition above, the shortest path from s to v must be a single-edge extension of a known shortest path in R.

- v is the single edge-extension from R with minimum length: Suppose it is not true, then there must be another vertex x which is also a one-edge extension closer from s, but then vertex v would no longer satisfy **Property 1**.



Therefore v can be found as the single-edge extension with minimum length of a known shortest path in R

2.3 Dijkstra's singles source shortest paths algorithm

We now have the idea of how to proceed, the only thing missing is an algorithm to find the vertex with smallest single-edge extension to those in R. To do this we will use a **Priority Queue** Q.

The methods of the priority queue (min-heap) are:

1. `build(A)`: Given an array of elements, build the priority queue.
2. `isEmpty()`: Returns true if the queue is empty.
3. `extractMin()`: Return the element with minimum value.
4. `decreaseKey(v)`: Decreases the value of an element (uses bubbleUp).

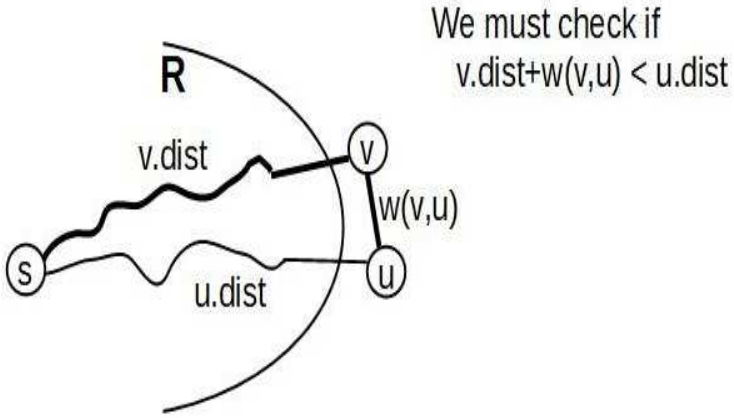
While running the algorithm we will compute:

- `v.dist` The current distance to s.
- `v.prev` The predecessor in a path from s to v. When the shortest path is found, this will help reconstruct the shortest paths tree starting from s.

Initialization:

to initialize the algorithm, the source vertex should be placed at a distance of 0, and all other vertices at a distance of ∞ (same as in BFS). The predecessors are all set to `NULL`.
When a new vertex v is added to R:

In this case we must update the distances and predecessors for each of the neighbors u of v since it might be possible that a path that goes through v is shorter than a previously seen path:



We are ready to write down Dijkstra's single source shortest paths algorithm.

```
Algorithm: shortestPaths
Input:A directed graph G=(V,E)
       A source vertex s ∈ V

begin
∀ v∈ V
    v.pred=NULL
    v.dist=∞
s.dist=0
Q.build(V)
while ! Q.isEmpty()
    begin
    v=Q.extractMin()
    for each (v,u)∈ E
       begin
       if v.dist+w(v,u)<u.dist
            begin
            u.dist=v.dist+w(v,u)
            u.pred=v
            Q.decreaseKey(u)
            end
       end
    end
end
```

2.4 Analysis

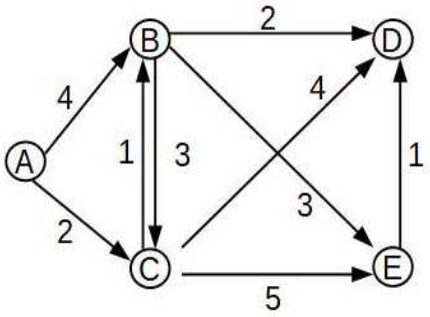
- The initialization process is done for each vertex: $O(|V|)$
- Extracting the min is done once for each vertex: $O(|V| \lg |V|)$ if using a binary heap.
- In the worst case, the condition "if v.dist+w(v,u)<u.dist" could be true twice for each edge. The decreaseKey(u) method (using bubbleUp) requires $O(\lg |V|)$ time: $O(|E| \lg |V|)$

Therefore the total running time of Dijkstra's algorithm using a binary heap is $O(|V| + |V| \lg |V| + |E| \lg |V|) = O((|V| + |E|) \lg |V|)$ in the worst case.

Notice that in the case of unweighted graphs (the weight of each edge is 1), it is better to use Breadth First Search (BFS) since it will run in $O(|V| + |E|)$.

2.5 Example:

Show the contents of the priority queue for each iteration of Dijkstra's Algorithm on the following graph:



Weight function:

w(A,B)=4
w(A,C)=2
w(B,C)=3
w(B,D)=2
w(B,E)=3
w(C,B)=1
w(C,D)=4
w(C,E)=5
w(E,D)=1

iteration	dist/pred	A	B	C	D	E
1	dist	0	∞	∞	∞	∞
	pred	null	null	null	null	null
2	dist		4	2	∞	∞
	pred		A	A	null	null
3	dist		3		6	7
	pred		C		C	C
4	dist				5	6
	pred				B	B
5	dist					6
	pred					B
6	dist					
	pred					
Final Result	dist	0	3	2	5	6
	pred	null	C	A	B	B