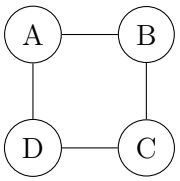
CS 3510 Algorithms 2/01/2024

Lecture 6: BFS and Dijkstra's

Lecturer: Abrahim Ladha *Scribe(s): Diksha Holla*

We want to calculate shortest paths in a graph. DFS is one way for us to search a graph. By using the stack, we can quickly explore deeply into a graph. However it is not particularly good at finding short paths. Here, we mean on an unweighted graph, the shortest path is the number of edges.



If we start at A and run DFS to find the shortest path to get to each element from A, we get that the shortest path from A to D is 3. DFS visits $A \to B \to C \to D$, and would miss the edge connecting A to D that returns a path of 1. We need a graph traversal algorithm which prioritizes the closest nodes first. This is called Breadth-First-Search or BFS. The shortest path algorithm will take on input a graph and a starting node, and it will compute the shortest paths from that node to all others.

1 BFS

BFS utilizes a queue data structure to pop and push from whereas DFS uses a stack. The queue helps us to see all the children of a node first before seeing the grandchildren. This way we can explore the closest nodes first.

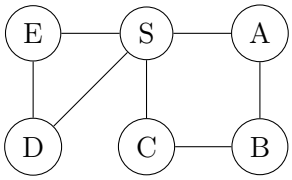
G is our graph and s is our start node

```
def bfs(G, s):
    for each u in V: dist(u) = ∞
    dist(s) = 0
    Q = [s]
    while Q ≠ ∅:
        u = eject(Q)
        for each (u,v) ∈ E:
            if dist(v) = ∞:
                inject v into Q
                dist(v) = dist(u) + 1
```

Each vertice is pushed and popped from the queue exactly once which takes O(1) time, so O(V). The adjacency lists are looped over once for each, their sum of lengths being O(E), giving us a total runtime of O(|V| + |E|). This takes the same time as DFS but in a different

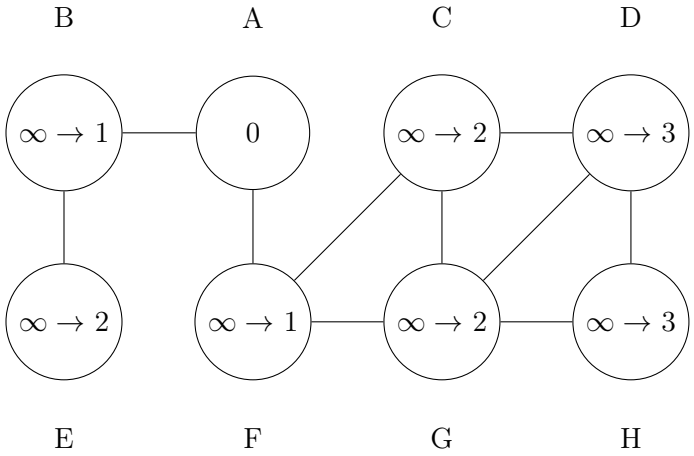
order

Example :



we start at S:
[S]
[A C D E] - pop S and add neighbors of S
[C D E B] - pop A and add neighbors of A
[D E B] - pop C and add neighbors of C
[E B] - pop d and add neighbors of D
[B] - pop E and add neighbors of E
[] - pop B and add neighbors of B, nothing is left so we end

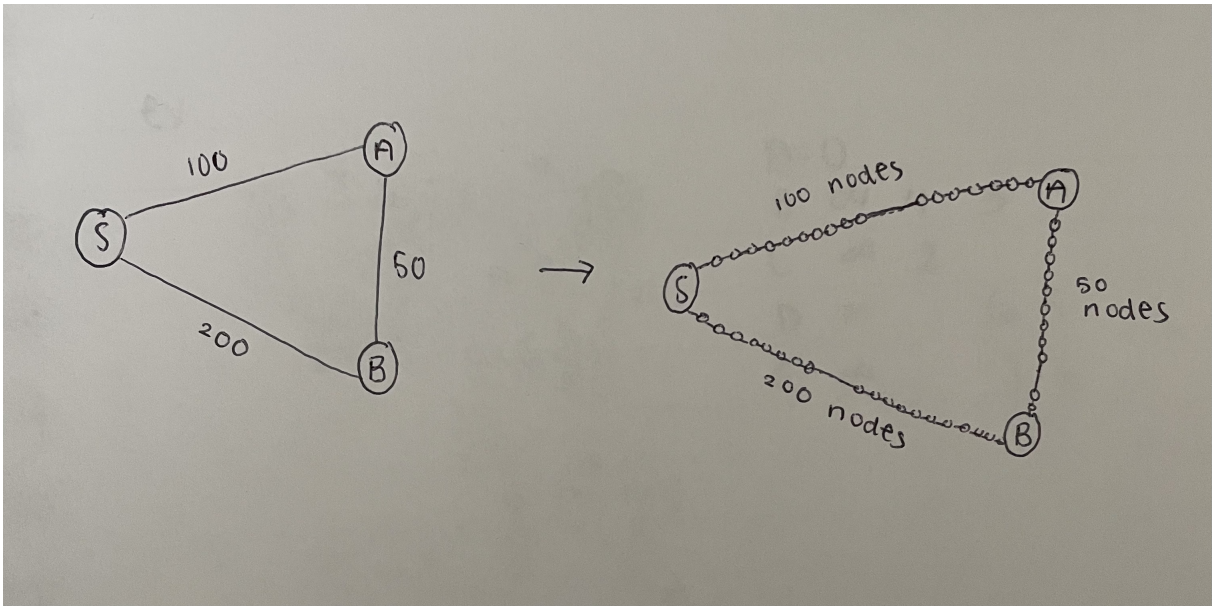
Example : Lets do an example where we add the shortest path from A to each node in the corresponding circle



we set node A to 0 and the rest of the nodes to infinity, and now we start BFS
[A]
[B F] - added B, F and set dist to A + 1
[F E] - added E and set E to B + 1
[E C G] - added C, G and set them to F + 1
[C G] - no nodes added
[G D] - added D and set to C + 1
[D H] - added H and set to G + 1
[H] - no nodes added
[] - finished

2 Dijkstra's

Consider a weighted graph now, with positive weights. We want to compute the shortest paths not counting the number of edges, but the total sum of the edges in the path. We can compute shortest paths again with BFS by transforming a weighted graph into an unweighted one.



The problem now is it takes 100 steps to see that A is closer to S than B after one edge. Why not directly compare 100 with 200 then choose to BFS on A next? This is the heart of Dijkstra's algorithm. It is simply weighted BFS. Instead of a simple queue, we use a priority queue with the following operations

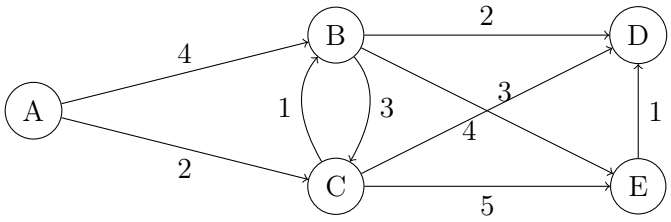
- Insert - into queue
- Decrease key - decreases value of some key
- Delete min - returns element with smallest key
- Make-queue - Constructor for the queue, faster than making an empty queue and performing V insertions.

G is our graph and s is our start node and l is the edge weights. The prev array contains back pointers to those actually store the shortest path. We want not only the distance, but the path itself.

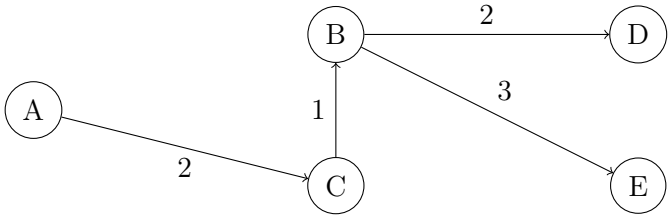
```
def dijkstras(G, l, s):
    for each u in V:
        dist(u) = ∞
```

```
        prev(u) = null
    dist(s) = 0
    H = makeQueue(V) with dist as keys
    while H ≠ ∅:
        u = deleteMin(H)
        for each (u,v) ∈ E:
            if dist(v) > dist(u) + l(u,v):
                dist(v) = dist(u) + l(u,v)
                prev(v) = u
                decreaseKey(H,v)
```

Example :



A to every node:
A: 0
B: ∞, 4, 3
C: ∞, 2
D: ∞, 6, 5
E: ∞, 7, 6
Here is the above graph with the edges used to find each nodes shortest paths

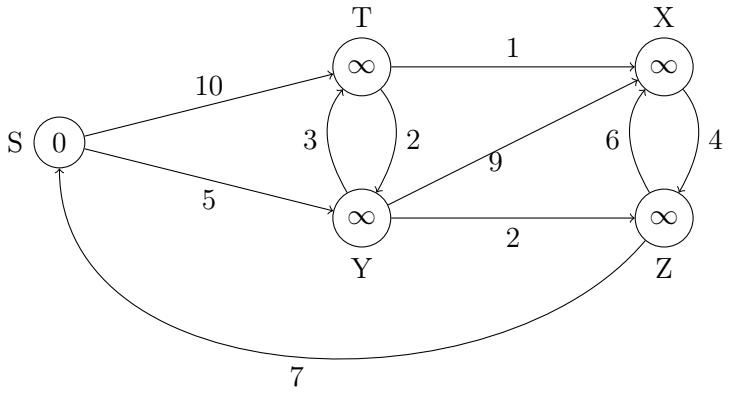


Dijkstra's is just BFS, but its runtime depends on the priority queue implementation. There are |V| deleteMin operations and |E| insert/decreaseKeys. We may ballpark a make-queue operation as |V| insert/decreaseKeys.

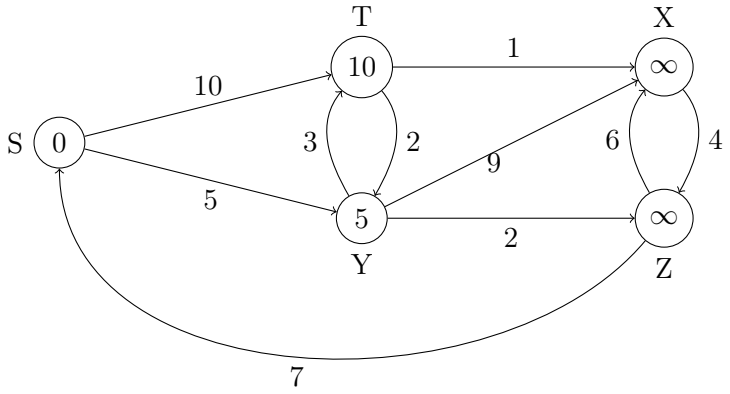
data structure	deleteMin	decreaseKey	total Dijkstra
array	O(|V|)	O(1)	O(V^2)
binary heap	O(logV)	O(logV)	O((V+E)logV)
d-ary heap	O($\frac{dlogv}{logd}$)	O($\frac{logv}{logd}$)	O((Vd + E)$\frac{logV}{logd}$)

If G is dense, E $\approx V^2$, then array is best to use. A binary heap becomes better to use at around E $\approx \frac{V^2}{logV}$

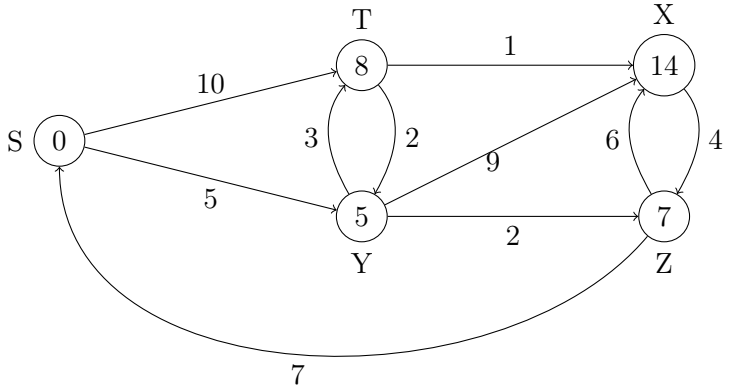
Example :



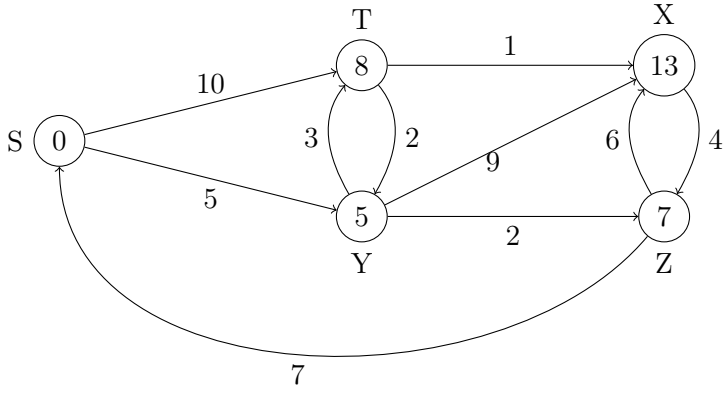
next iter: T = S + 10, Y = S + 5



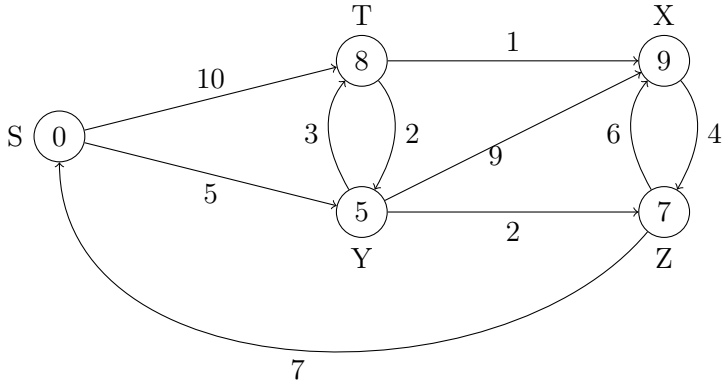
next iter: T = Y + 3, X = Y + 9, Z = Y + 2



next iter: X = Z + 6



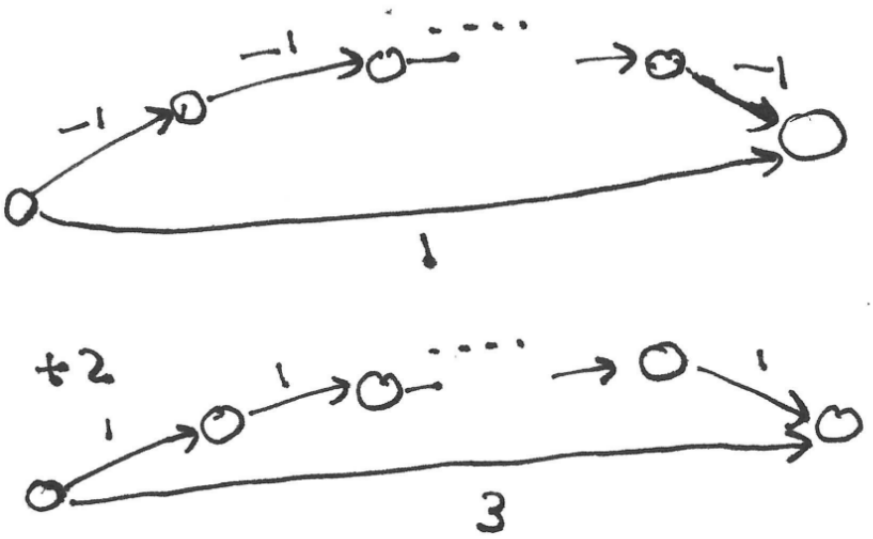
next iter: X = T + 1



Dijkstra's only works on graphs with non-negative edge weights. It relies on the following property: "The shortest path s-t is less than or equal to the shortest path s-u-t ∀u". As you perform updates, the current minimum shortest path computed for some node can only decrease as Dijkstra's continues. This is not true for negative edge weighted graphs, which makes them a very difficult, much worse object of study, even in 2023.[1]

You may think that you can simply pad each edge by a constant value, and all comparisons made in Dijkstra's should remain the same. Here is a counter example.

[1] `https://www.quantamagazine.org/finally-a-fast-algorithm-for-shortest-paths-on-negative-graphs-20230118/`



The top path is the shortest, but after adding two to each edge, the bottom path is now the shortest.