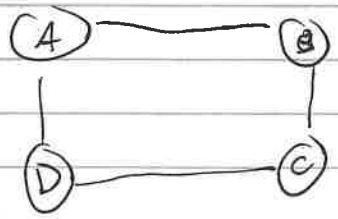
BFS and Dijkstra's Algorithm

DFS is one way for us to search a graph. By using the stack, we can quickly explore deeply into a graph. It is not particularly good at finding short paths. Here, we mean on an unweighted graph by the number of edges.

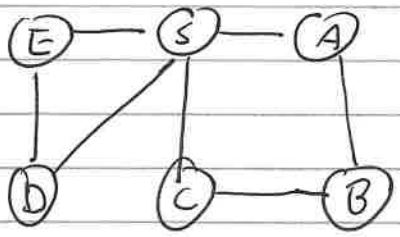


Although D is only distance 1 from A, DFS takes the route A-B-C-D giving a path of length 3.

We need a graph traversal algorithm which prioritizes the closest nodes first. This is called Breadth-First-Search, or BFS.

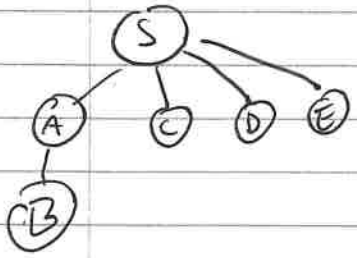
```
def bfs(G,s)
  for each u∈V: dist(u)=∞
  dist(s)=0
  Q=[s]
  while Q≠∅:
    u=eject(Q)
    for each (u,v)∈E
      if dist(v)=∞:
        inject v into Q
        dist(v)=dist(u)+1
```

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 $O(|V|+|E|)$. Takes the same time as DFS but just in a different order.

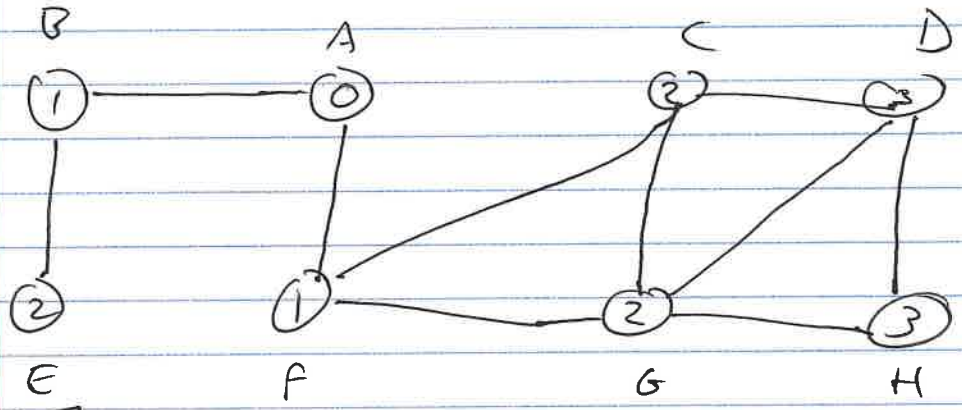


[S]
[ACDE]
[CDEB]
[DEB]

[EB]
[B],[B]

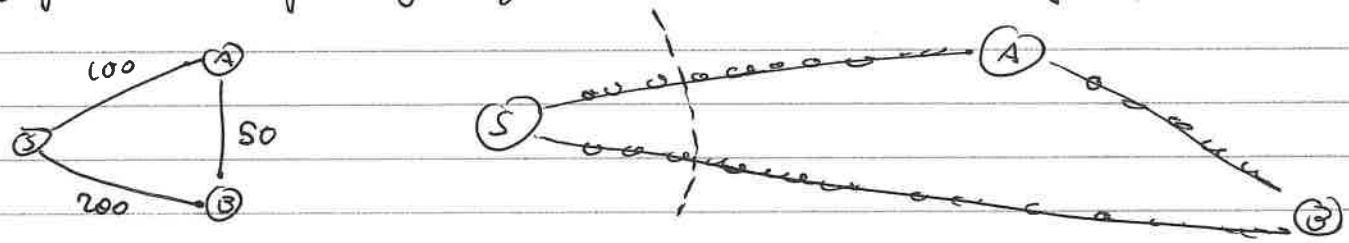


In the spirit of computing shortest paths, we are only concerned with nodes reachable from our start. The queue structure no more than two levels at a time have been fully visited. You cannot see grandchildren before seeing children.



[A]
[BF]
[FE]
[ECG]
[CG]
[GD]
[DH]

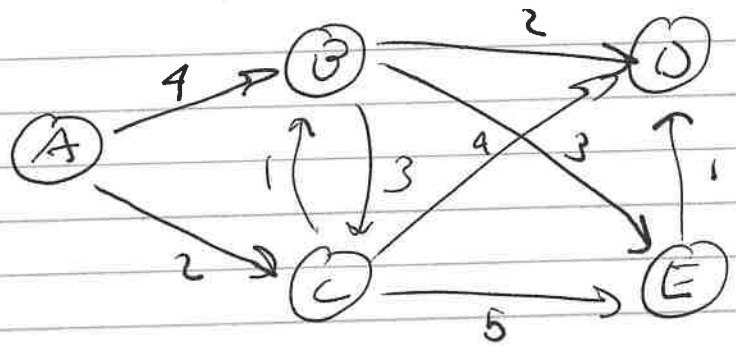
Consider a weighted graph now, with positive weights. We can compute shortest paths again by transforming it into an unweighted graph



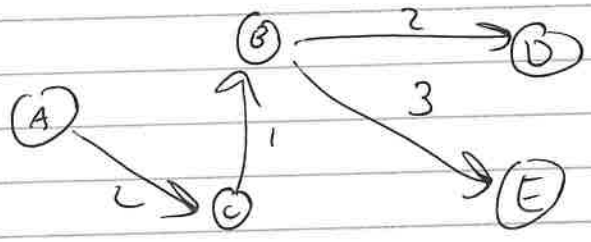
the problem now is it takes 100 steps to see that A is closer to S than B after one edge. why not just directly compare 100 with 200 then choose to BFS on A next? This is the heart of Dijkstra's algorithm. Instead of a simple queue, we use a priority queue with the following operations:

insert	into queue
decrease key	decreases value of some key
delete min	returns element w/ smallest key
make-queue	builds the queue quickly.

```
def dijkstra(G, l, s):
    for each u ∈ 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).
```



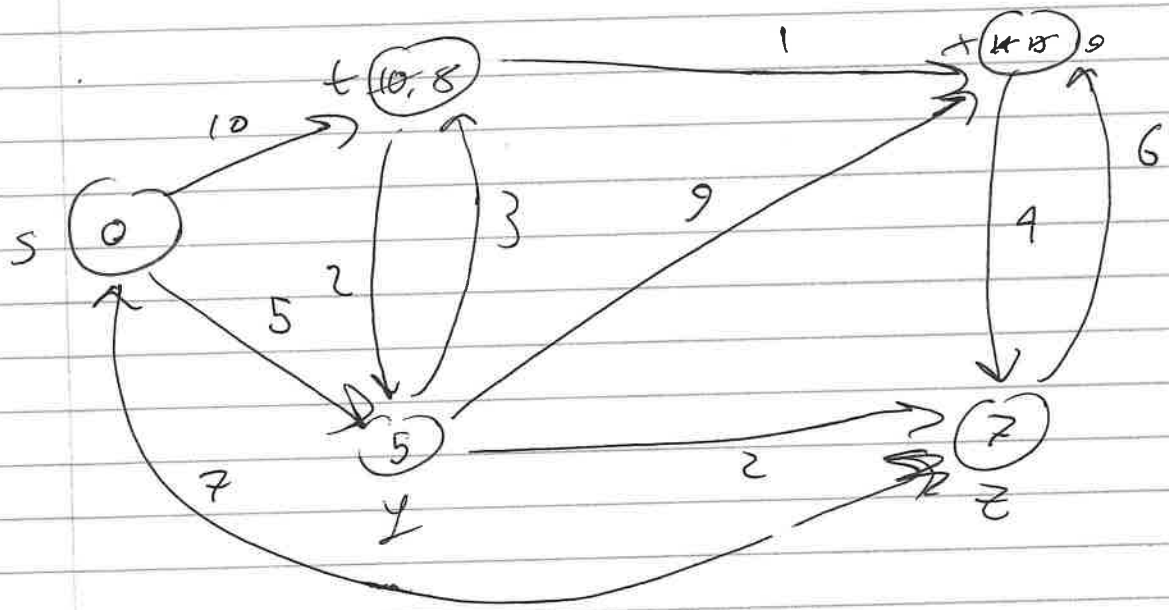
	A	C	B
A:	0		
B:	∞,	4	3
C	∞,	2	
D	∞	6	5
E	∞	7	6



Dijkstra's is just BFS, but its run time depends on the PQ implementation.
There are $|V|$ delete min operations and $|V|+|E|$ insertions/decrease keys.

	delete min	insert	Dijkstra
Array	$O(V)$	$O(1)$	$O(V^2)$
binary heap	$O(\log V)$	$O(\log V)$	$O((V+E)\log V)$
d-ary heap	$(d \log V / \log d)$	$O(\log V / \log d)$	$O((Vd+E)\frac{\log V}{\log d})$

If G is dense, $E \approx V^2$, then array is best. binary heap becomes better around $E \approx V^2/\log V$



"The shortest path s-t is less than the shortest path s-u-t ∀u."