Notes for Lecture 6

1 Breadth-First Search

_Breadth-first search (BFS)_ is the variant of search that is guided by a _queue_, instead of DFS's stack (remember, DFS _does_ use a stack, the one implicit in its recursion). There is one stylistic difference: One does not restart BFS, because BFS only makes sense in the context of exploring the part of the graph that is reachable from a particular node (s in the algorithm below). Also, although BFS does not have the wonderful and subtle properties of DFS, it does provide useful information: Because it tries to be “fair” in its choice of the next node, it visits nodes in order of increasing distance from s. In fact, our BFS algorithm below labels each node with the shortest distance from s, that is, the number of edges in the shortest path from s to the node. The algorithm is this:

```
Algorithm BFS(G=(V,E): graph, s: node);
variables:
  v, w: nodes; Q: queue of nodes, initially {s};
  dist: array[V] of integer, initially ∞
  dist[s]:=0
while Q is not empty do
  v:= eject(Q),
  for all edges (v,w) out of v do
    { if dist[w] = ∞ then
      { inject(w,Q), dist[w]:=dist[v]+1
    }
```

For example, applied to the graph in Figure 1, this algorithm labels the nodes (by the array dist) as shown. Why are we sure that the dist[v] is the shortest-path distance of v from s? It is certainly true if dist[v] is zero (this happens only at s). And, if it is true for dist[v] = d, then it can be easily shown to be true for values of dist equal to d + 1—any node that receives this value has an edge from a node with dist d, and from no node with lower dist. Notice that nodes not reachable from s will not be visited or labeled.

![Graph example](image-url)
Breadth-first search runs, of course, in linear time $O(|V| + |E|)$. The reason is the same as with DFS: BFS visits each edge exactly once, and does a constant amount of work per edge.

2 Dijkstra’s Algorithm

Suppose each edge $(v, w)$ of our graph has a weight, a positive integer denoted $\text{weight}(v, w)$, and we wish to find the shortest from $s$ to all vertices reachable from it.\footnote{What if we are interested only in the shortest path from $s$ to a specific vertex $t$? As it turns out, all algorithms known for this problem also give us, as a free byproduct, the shortest path from $s$ to all vertices reachable from it.}

We will still use BFS, but instead of choosing which vertices to visit by a queue, which pays no attention to how far they are from $s$, we will use a heap, or priority queue, of vertices. The priority depends on our current best estimate of how far away a vertex is from $s$: we will visit the closest vertices first. These distance estimates will always overestimate the actual shortest path length from $s$ to each vertex, but we are guaranteed that the shortest distance estimate in the queue is actually the true shortest distance to that vertex, so we can correctly mark it as finished.

As in all shortest path algorithms we shall see, we maintain two arrays indexed by $V$. The first array, $\text{dist}[v]$, contains our overestimated distances for each vertex $v$, and will contain the true distance of $v$ from $s$ when the algorithm terminates. Initially, $\text{dist}[s]=0$ and the other $\text{dist}[v]=\infty$, which are sure-fire overestimates. The algorithm will repeatedly decrease each $\text{dist}[v]$ until it equals the true distance. The other array, $\text{prev}[v]$, will contain the last vertex before $v$ in the shortest path from $s$ to $v$. The pseudo-code for the algorithm is in Figure 1.

The procedure $\text{insert}(w, H)$ must be implemented carefully: if $w$ is already in $H$, we do not have to actually insert $w$, but since $w$’s priority $\text{dist}[w]$ is updated, the position of $w$ in $H$ must be updated.

The algorithm, run on the graph in Figure 2, will yield the following heap contents (vertex: dist/priority pairs) at the beginning of the while loop: \{s\}, \{a : 2, b : 6\}, \{b : 5, c : 3\}, \{b : 4, e : 7, f : 5\}, \{e : 7, f : 5, d : 6\}, \{e : 6, d : 6\}, \{e : 6\}, \{\}. The distances from $s$ are shown in the figure, together with the shortest path tree from $s$, the rooted tree defined by the pointers prev.

2.1 What is the complexity of Dijkstra’s algorithm?

The algorithm involves $|E|$ insert operations (in the following, we will call $m$ the number $|E|$) on $H$ and $|V|$ deletemin operations on $H$ (in the following, we will call $n$ the number $|V|$), and so the running time depends on the implementation of the heap $H$, so let us discuss this implementation. There are many ways to implement a heap.\footnote{In all heap implementations we assume that we have an array of pointers that give, for each vertex, its position in the heap, if any. This allows us to always have at most one copy of each vertex in the heap. Each time $\text{dist}[v]$ is decreased, the $\text{insert}(w, H)$ operation finds $w$ in the heap, changes its priority, and possibly moves it up in the heap.}
algorithm Dijkstra(G=(V, E, weight), s)
variables:
v,w: vertices (initially all unmarked)
dist: array[V] of integer
prev: array[V] of vertices
heap of vertices prioritized by dist
for all v ∈ V do { dist[v] := ∞, prev[v] := nil }
H:=\{ s \} , dist[s] := 0 , mark(s)
while H is not empty do
{ 
v := deletemin(H) , mark(v)
for each edge (v,w) out of E do 
{ 
if w unmarked and dist[w] > dist[v] + weight[v,w] then 
{
    dist[w] := dist[v] + weight[v,w]
    prev[w] := v
    insert(w,H)
}
}

Figure 1: Dijkstra's algorithm.

Shortest Paths

Figure 2: An example of a shortest paths tree, as represented with the prev[] vector.
unsophisticated one (an amorphous set, say a linked list of vertex/priority pairs) yields an interesting time bound, \( O(n^2) \) (see first line of the table below). A binary heap gives \( O(m \log n) \).

Which of the two should we use? The answer depends on how dense or sparse our graphs are. In all graphs, \( m \) is between \( n \) and \( n^2 \). If it is \( \Omega(n^2) \), then we should use the linked list version. If it is anywhere below \( \frac{n^2}{\log n} \), we should use binary heaps.

<table>
<thead>
<tr>
<th>heap implementation</th>
<th>deletemin</th>
<th>insert</th>
<th>( n \times \text{deletemin} + m \times \text{insert} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>linked list</td>
<td>( O(n) )</td>
<td>( O(1) )</td>
<td>( O(n^2) )</td>
</tr>
<tr>
<td>binary heap</td>
<td>( O(\log n) )</td>
<td>( O(\log n) )</td>
<td>( O(m \log n) )</td>
</tr>
<tr>
<td>( d )-ary heap</td>
<td>( O\left(\frac{\log n}{\log d}\right) )</td>
<td>( O\left(\frac{\log n}{\log d}\right) )</td>
<td>( O\left((n \cdot d + m) \frac{\log n}{\log d}\right) )</td>
</tr>
</tbody>
</table>

A more sophisticated data structure, the \( d \)-ary heap, performs even better. A \( d \)-ary heap is just like a binary heap, except that the fan-out of the tree is \( d \), instead of 2. In an array implementation, the child pointers of vertex \( i \) are implemented as \( d \cdot i, \ldots, d \cdot i + d - 1 \), while the parent pointer by \( \left\lfloor \frac{i}{d} \right\rfloor \). Since the depth of any such tree with \( n \) vertices is \( \frac{\log n}{\log d} \), it is easy to see that inserts take this amount of time, while deletemins \( d \) times that—because deletemins go down the tree, and must look at the children of all vertices visited.

The complexity of our algorithm is therefore a function of \( d \). We must choose \( d \) to minimize it. The right choice is about \( d = \frac{m}{n} \)—the average degree! It is easy to see that it is about the right choice because it equalizes the two terms of \( m + n \cdot d \). (To more precisely minimize the bound, differentiate with respect to \( d \), set the derivative equal to 0 and solve for \( d \).) This yields an algorithm that is good for both sparse and dense graphs. For dense graphs, its complexity is \( O(n^2) \). For graphs with \( m = O(n) \), it is \( n \log n \). Finally, for graphs with intermediate density, such as \( m = n^{1+\delta} \), where \( \delta \) is the density of the graph, the algorithm is linear!

### 2.2 Why does Dijkstra’s algorithm work?

Here is a sketch. Recall that the inner loop of the algorithm (the one that examines edge \((v, w)\) and updates \( \text{dist} \)) examines each edge in the graph exactly once, so at any point of the algorithm we can speak about the subgraph \( G' = (V, E') \) examined so far: it consists of all the nodes, and the edges that have been processed. Each pass through the inner loop adds one edge to \( E' \). We will show that the following property of \( \text{dist}[] \) is an invariant of the inner loop of the algorithm:

\[
\text{dist}[w] \text{ is the minimum distance from } s \text{ to } w \text{ in } G',
\]

if \( v_k \) is the \( k \)-th vertex marked, then \( v_1 \) through \( v_k \) are the \( k \) closest vertices to \( s \) in \( G' \), and the algorithm has found the shortest paths to them.

We prove this by induction on the number of edges \(|E'|\) in \( E' \). At the very beginning, before examining any edges, \( E' = \emptyset \) and \(|E'| = 0 \), so the correct minimum distances are \( \text{dist}[s] = 0 \) and \( \text{dist}[w] = \infty \), as initialized. And \( s \) is marked first, with the distance to it equal to zero as desired.

Now consider the next edge \((v, w)\) to get \( E'' = E' \cup (v, w) \). Adding this edge means there is a new way to get to \( w \) from \( s \) in \( E'' \): from \( s \) to \( v \) to \( w \). The shortest way to do this is \( \text{dist}[v] + \text{weight}(v, w) \) by induction. Also by induction the shortest path to get to \( w \)
not using edge \((v, w)\) is dist\([w]\). The algorithm then replaces dist\([w]\) by the minimum of its old value and possible new value. Thus dist\([w]\) is still the shortest distance to \(w\) in \(E''\). We still have to show that all the other dist\([u]\) values are correct; for this we need to use the heap property, which guarantees that \((v, w)\) is connected to the node \(v\) in the heap closest to the source \(s\). To complete the induction we have to show that finding a new, shortest path to \(w\) via \(v\) does not change the values of any other dist\([u]\]. dist\([u]\) could only change if the shortest path from \(s\) to \(u\) had previously gone through \(w\). To have built this path means that we would have had to examine edges out of \(w\) earlier, before examining edges out of \(v\). But this is impossible, since we just decided that \(v\) was closer to \(s\) than \(w\) (since the shortest path to \(w\) is via \(v\)), so the heap would not yet have marked \(w\) and examined edges out of it.