**Breadth-First Search**

A searching technique with different properties than DFS is *Breadth-First Search (BFS)*. While DFS used an implicit stack, BFS uses an explicit queue structure in determining the order in which vertices are searched. Below we show code for doing BFS from a particular start or “source” vertex $s$.

```plaintext
Procedure BFS (G(V,E), s ∈ V)
  graph G(V,E)
  array[|V|] of integers dist, initialized to ∞
  array[|V|] of booleans visited, initialized to False
  queue q;
  dist[s] := 0
  inject(q,s)
  visited(s) := True
  while size(q) > 0
    v := pop(q)
    for (v,w) ∈ E
      if not visited(w) then
        inject(q,w)
        visited(w) := True
        dist(w) = dist(v)+1
      fi
    rof
  end while
end BFS
```

Although BFS does not have the same subtle properties of DFS, it does provide useful information. BFS visits vertices in order of increasing distance from $s$. In fact, our BFS algorithm above labels each vertex with the *distance* from $s$, or the number of edges in the shortest path from $s$ to the vertex. For example, applied to the graph in Figure 4.1, this algorithm labels the vertices (by the array dist) as shown.

Why are we sure that the array dist is the shortest-path distance from $s$? First, we prove two lemmas. Below we use the notation $δ(a,b)$ to denote the length of the shortest path from $a$ to $b$.

**Lemma 4.1** *For every* $v ∈ V$, $\text{dist}(v) ≥ δ(s,v)$. 
Proof: We prove this by induction on the number $k$ of inject operations into the queue. This is true for the base case when $k = 1$, since then the queue only contains $s$ which has its dist value set to $0 = \delta(s, v)$. For the inductive hypothesis, say its true for the first $k$ inject operations and we want to show its true when injecting some vertex $u$ for the $(k+1)$st time. We are injecting $u$ due to some edge $(v, u)$, where $v$ was injected earlier. We thus know $\delta(s, u) \leq \delta(s, v) + 1 \leq \text{dist}(v) + 1$, where the first inequality is via the triangle inequality and the second is via the inductive hypothesis. Since we set $\text{dist}(u) = \text{dist}(v) + 1$, the claim is proven.

Lemma 4.2 At all points in time, if the queue contains vertices $v_1, \ldots, v_r$ then:

1. $\forall 1 \leq i < r$, $\text{dist}(v_i) \leq \text{dist}(v_{i+1})$, and

2. $\text{dist}(v_r) \leq \text{dist}(v_1) + 1$.

Proof: This is proven by induction on the number $k$ of total queue operations (injections and pops). The base case when $k = 1$ follows since the queue has size 1 (containing only $s$). For the inductive step, there are two choices for the $(k+1)$st operation: a pop or an inject. When we pop some vertex $v$, the two conditions trivially still hold if they already did. When we inject some $w$, it’s after having just popped some previous $v_1$. Then note either the queue is empty, so inserting $w$ doesn’t violate (1) and (2), or there is some other $v_2$ still in the queue with $\text{dist}(v_2) \geq \text{dist}(v_1)$ and also $\text{dist}(v_r) \leq \text{dist}(v_1) + 1 \leq \text{dist}(v_2) + 1$. Since we set $\text{dist}(w) = \text{dist}(v_1) + 1$, both conditions still hold.

Note that the above lemma implies that vertices are visited in increasing order of distance from $s$.

Theorem 4.3 At the end of BFS, $\text{dist}(v) = \delta(s, v)$ for all vertices $v$ (or $\infty$ if $v$ is unreachable).

Proof: We focus on the case of $v$ reachable.

For the sake of contradiction, suppose this is not true for all reachable $v$. Let $v$ then be such that of all vertices with $\text{dist}(v) \neq \delta(s, v)$, $\delta(s, v)$ is minimum. Then $\text{dist}(v) > \delta(s, v)$ by Lemma 4.1. We know $v \neq s$, since $\text{dist}(s)$ is correctly set to 0. Let $s = v_0, v_1, v_2, \ldots, v_r = v$ be a shortest path from $s$ to $v$. Then we know by minimality that $v_{r-1}$ has its dist set correctly. Thus when $v_{r-1}$ is popped from the queue, it will attempt to inject $v_r = v$ since $v$ is its neighbor. If it did, $\text{dist}(v)$ would be set correctly, which we assumed didn’t happen, and thus it didn’t, meaning $v$ had already been visited. But since at this time all vertices in the queue or those that have already been visited have their dist values set to at most $\text{dist}(v_{r-1}) + 1 = \delta(s, v_{r-1}) + 1 = \delta(s, v)$, it thus cannot be the case that $\text{dist}(s, v)$ was set to anything larger, a contradiction.
BFS runs, of course, in linear time $O(m + n)$. The reason is that BFS visits each edge exactly once, and does a constant amount of work per edge.

**Single-Source Shortest Paths — Nonnegative Lengths**

What if each edge $(v,w)$ of our graph has a *length*, a positive integer denoted $\text{length}(v,w)$, and we wish to find the shortest paths from $s$ to all vertices reachable from it? BFS offers a possible solution. We can subdivide each edge $(u,v)$ into $\text{length}(u,v)$ edges, by inserting $\text{length}(u,v) - 1$ “dummy” nodes, and then apply BFS to the new graph. This algorithm solves the shortest-path problem in time $O(\sum_{(u,v) \in E} \text{length}(u,v))$. Unfortunately, this can be very large — lengths could be in the thousands or millions. So we need to find a better way.

The problem is that this BFS-based algorithm will spend most of its time visiting “dummy” vertices; only occasionally will it do something truly interesting, like visit a vertex of the original graph. What we would like to do is run this algorithm, but only do work for the “interesting” steps.

To do this, we need to generalize BFS. Instead of using a queue, we will instead use a *heap* or *priority queue* of vertices. A heap is a data structure that keeps a set of objects, where each object has an associated value. The operations a heap $H$ implements include the following:

---

1. What if we are interested only in the shortest path from $s$ to a specific node $t$? As it turns out, all algorithms known for this problem have to compute the shortest path from $s$ to all vertices reachable from it.
In all heap implementations we consider in the course, the heap will store its items explicitly. Then for the decreasekey operation, we assume that we are given a pointer to \( x \) in the heap.

Each entry in the heap will stand for a projected future “interesting event” of our extended BFS. Each entry will correspond to a vertex, and its value will be the current projected time at which we will reach the vertex. Another way to think of this is to imagine that, each time we reach a new vertex, we can send an explorer down each adjacent edge, and this explorer moves at a rate of 1 unit distance per second. With our heap, we will keep track of when each vertex is due to be reached for the first time by some explorer. Note that the projected time until we reach a vertex can decrease, because the new explorers that arise when we reach a newly explored vertex could reach a vertex first (see node b in Figure 4.2). But one thing is certain: the most imminent future scheduled arrival of an explorer must happen, because there is no other explorer who can reach any vertex faster. The heap conveniently delivers this most imminent event to us.

As in all shortest path algorithms we shall see, we maintain two arrays indexed by \( V \). The first array, \( \text{dist}[v] \), will eventually contain the true distance of \( v \) from \( s \). The other array, \( \text{prev}[v] \), will contain the last node before \( v \) in the shortest path from \( s \) to \( v \). Our algorithm maintains a useful invariant property: at all times \( \text{dist}[v] \) will contain a conservative over-estimate of the true shortest distance of \( v \) from \( s \). Of course \( \text{dist}[s] \) is initialized to its true value 0, and all other \( \text{dist} \)’s are initialized to \( \infty \), which is a remarkably conservative overestimate. The algorithm is known as Dijkstra’s algorithm, named after the inventor.

```plaintext
ithm Dijkstra \((G = (V,E,\text{length}); s \in V)\)

dist: array\([V]\) of integers, initialized each to \( \infty \)
prev: array\([V]\) of vertices, initialized each to nil
\( H := \) empty minheap
for \( v \in V \) do
    \( H.insert(v, \infty) \) // insert \( v \) with key \( \infty \)
end for
\( H.decreasekey(s,0) \)
dist[s] := 0
while \( H \neq \emptyset \)
    \((v,v.key) := H.deletemin()\)
    for \((v,w) \in E\)
        if dist[w] > dist[v] + \text{length}(v,w)
            dist[w] := dist[v] + \text{length}(v,w)
            prev[w] := v
        end if
    end for
end while
```

\( H.deletemin \) return the object with the smallest value

\( H.insert(x,y) \) insert a new object \( x/value \ y \) pair in the structure

\( H.decreasekey(x,y) \) if \( y \) is smaller than \( x \)’s current value,
change the value of object \( x \) to \( y \)
The algorithm, run on the graph in Figure 4.2, will yield the following heap contents (node: dist/priority pairs) at the beginning of the while loop:

\{s : 0\}, \{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 Figure 2, together with the shortest path tree from \(s\), the rooted tree defined by the pointers \(\text{prev}\).

What is the running time of this algorithm? The algorithm involves \(|V|\) insert operations and \(|V|\) deletemin operations on \(H\), and at most \(|E|\) decreasekey operations, and so the running time depends on the implementation of the heap \(H\). There are many ways to implement a heap. Even an unsophisticated implementation as a linked list of node/priority pairs yields an interesting time bound, \(O(|V|^2)\) (see first line of the table below). A binary heap would give \(O(|E| \log |V|)\).

Which of the two should we prefer? The answer depends on how dense or sparse our graphs are. In all graphs, \(|E|\) is between \(|V|\) and \(|V|^2\). If it is \(\Omega(|V|^2)\), then we should use the linked list version. If it is anywhere below \(\frac{|V|^2}{\log |V|}\), we should use binary heaps.
| heap implementation | deletemin | insert | decreasekey | $|V| \times (\text{deletemin} + \text{insert}) + |E| \times \text{decreasekey}$ |
|---------------------|-----------|--------|-------------|------------------------------------------------------------------|
| linked list         | $O(|V|)$  | $O(1)$ | $O(1)$      | $O(|V|^2)$                                                       |
| binary heap         | $O(\log |V|)$ | $O(\log |V|)$ | $O(\log |V|)$ | $O((|E| + |V|) \log |V|)$                                      |
| Fibonacci heap      | $O(\log |V|)$ | $O(1)$ | $O(1)$ amortized | $O(|E| + |V| \log |V|)$                                      |

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. (Here $d$ should be at least 2, however!)

The fastest known implementation of Dijkstra’s algorithm uses a data structure known as a Fibonacci heap, which we will not cover here. Note that the bounds for the insert operation for Fibonacci heaps are amortized bounds: certain operations may be expensive, but the average cost over a sequence of operations is constant.