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. Also, generally 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 vertex (s in the algorithm below).

```
Procedure BFS (G(V,E), s ∈ V)
    graph G(V,E)
    array[|V|] of integers dist
    queue q;
    dist[s] := 0
    inject(q,s)
    placed(s) := 1
    while size(q) > 0
        v := pop(q)
        previsit(v)
        for (v,w) ∈ E
            if placed(w) = 0 then
              inject(q,w)
              placed(w) := 1
              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? A simple induction proof suffices. It is certainly true if the distance is zero (this happens only at s). And, if it is true for $\text{dist}(v) = d$, then it can be easily shown to be true for values of dist equal to $d+1$ —any vertex that receives this value has an edge from a vertex with

dist d, and from no vertex with lower value of dist. Notice that vertices not reachable from s will not be visited or labeled.

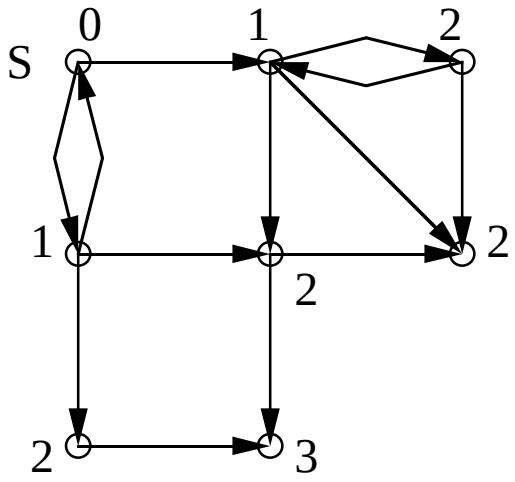

Figure 4.1: BFS of a directed graph

BFS runs, of course, in linear time $O(|E|)$, under the assumption that $|E| \geq |V|$. 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 length(v,w), and we wish to find the shortest paths from s to all vertices reachable from it?[1] BFS offers a possible solution. We can subdivide each edge (u,v) into length(u,v) edges, by inserting length$(u,v)-1$ "dummy" nodes, and then apply DFS 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 an data structure that keeps a set of objects, where each object has an associated value. The

[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.

operations a heap H implements include the following:

deletemin(H)	return the object with the smallest value
insert(x,y,H)	insert a new object x/value y pair in the structure
change(x,y,H)	if y is smaller than x's current value, change the value of object x to y

We will not distinguish between insert and change, since for our purposes, they are essentially equivalent; changing the value of a vertex will be like re-inserting it. [2]

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, dist$[v]$, will eventually contain the true distance of v from s. The other array, 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 dist$[v]$ will contain a conservative over-estimate of the true shortest distance of v from s.* Of course dist$[s]$ is initialized to its true value 0, and all other dist's are initialized to ∞, which is a remarkably conservative overestimate. The algorithm is known as Djikstra's algorithm, named after the inventor.

Algorithm Djikstra ($G = (V,E,\text{length})$; $s \in V$)
    v,w: vertices
    dist: array$[V]$ of integer
    prev: array$[V]$ of vertices
    H: priority heap of V
    $H := \{s : 0\}$
    for $v \in V$ do
        dist$[v] := \infty$, prev$[v]$:=nil

[2] In all heap implementations we assume that we have an array of pointers that gives, 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. Furthermore, it makes changes and inserts essentially equivalent operations.

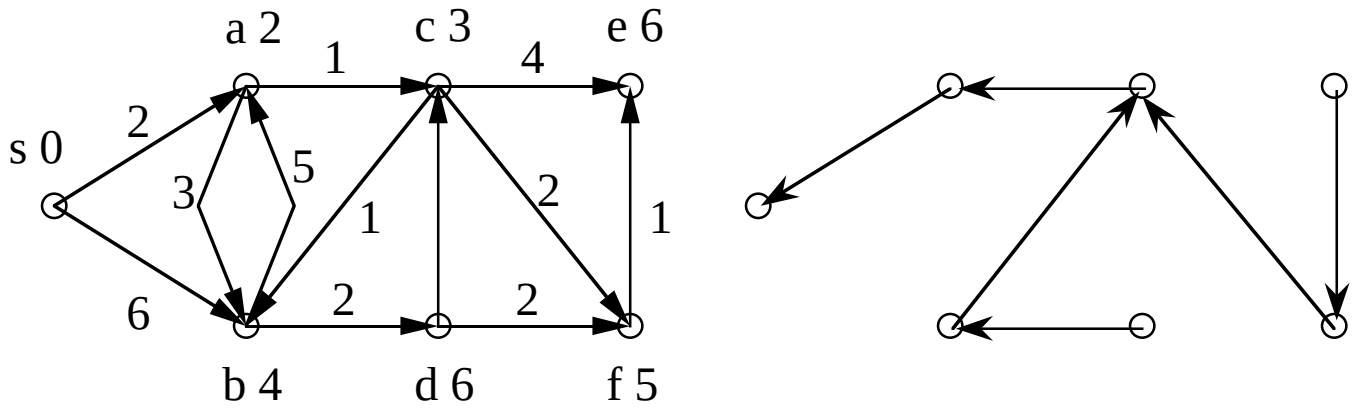

Figure 4.2: Shortest paths

```
    rof
    dist[s] := 0
    while H ≠ ∅
        v := deletemin(h)
        for (v,w) ∈ E
            if dist[w] > dist[v]+ length(v,w)
               dist[w] := dist[v] + length(v,w), prev[w] := v, insert(w,dist[w],H)
            fi
        rof
    end while end shortest paths 1
```

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 prev.

What is the running time of this algorithm? The algorithm involves $|E|$ insert operations and $|V|$ deletemin operations on H, 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	$\|V\|\times$deletemin$+\|E\|\times$insert
linked list	$O(\|V\|)$	$O(1)$	$O(\|V\|^2)$
binary heap	$O(\log \|V\|)$	$O(\log \|V\|)$	$O(\|E\| \log \|V\|)$
d-ary heap	$O(\frac{d \log \|V\|}{\log d})$	$O(\frac{\log \|V\|}{\log d})$	$O((\|V\| \cdot d + \|E\|)\frac{\log \|V\|}{\log d}$
Fibonacci heap	$O(\log \|V\|)$	$O(1)$ amortized	$O(\|V\| \log \|V\| + \|E\|)$

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!) Since the depth of any such tree with $|V|$ nodes is $\frac{\log |V|}{\log d}$, it is easy to see that inserts take this amount of time. Deletemins take d times that, because deletemins go down the tree, and must look at the children of all vertices visited.

The complexity of this algorithm is a function of d. We must choose d to minimize it. A natural choice is $d = \frac{|E|}{|V|}$, which is the the average degree! (Note that this is the natural choice because it equalizes the two terms of $|E| + |V| \cdot d$. Alternatively, the "exact" value can be found using calculus.) This yields an algorithm that is good for both sparse and dense graphs. For dense graphs, its running time is $O(|V|^2)$. For graphs with $|E| = O(|V|)$, it is $|V| \log |V|$. Finally, for graphs with intermediate density, such as $|E| = |V|^{1+\delta}$, where δ is the *density* of the graph, the algorithm is *linear!*

The fastest known implementation of Djikstra'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.

Single-Source Shortest Paths: General Lengths

Our argument of correctness of our shortest path algorithm was based on the "time metaphor:" the most imminent prospective event (arrival of an explorer) must take place, exactly because it is the most imminent. This however would not work if we had *negative edges*. (Imagine explorers being able to arrive before they left!) If the length of edge (a,b) in Figure 2 were -1, the shortest path from s to b would have value 1, not 4, and our simple algorithm fails. Obviously, with negative lengths we need more involved algorithms, which repeatedly update the values of dist.

We can describe a general paradigm for constructing shortest path algorithms with arbitrary edge weights. The algorithms use arrays dist and prev, and again we maintain the invariant that dist is always a conservative overestimate of the true distance from s. (Again, dist is initialized to ∞ for all nodes, except for s for which it is 0).

The algorithms maintain dist so that it is always a conservative overestimate; it will only update the a value when a suitable path is discovered to show that the overestimate can be lowered. That is, suppose we find a neighbor w of v, with $\text{dist}[v] > \text{dist}[w] + \text{length}(w,v)$. Then we have found an actual path that shows the distance estimate is too conservative. We therefore repeatedly apply the following update rule.

```
procedure update ((w,v))
    edge (w,v)
    if dist[v] > dist[w]+ length(w,v) then
      dist[v] := dist[w] + length(w,v), prev[v] := w
```

A crucial observation is that this procedure is *safe*, in that it never invalidates our "invariant" that dist is a conservative overestimate.

The key idea is to consider how these updates along edges should occur. In Djikstra's algorithm, the edges are updated according to the time order of the imaginary explorers. But this only works with positive edge lengths.

A second crucial observation concerns how many updates we have to do. Let $a \neq s$ be a node, and consider the shortest path from s to a, say $s, v_1, v_2, \ldots, v_k = a$ for some k between 1 and $n-1$. If we perform update first on (s, v_1), later on (v_1, v_2), and so on, and finally on (v_{k-1}, a), then we are sure that $\text{dist}(a)$ contains the true distance from s to a, and that the true shortest path is encoded in prev. (**Exercise:** Prove this, by induction.) We must thus find a sequence of updates that guarantee that these edges are updated in this order. We don't care if these or other edges are updated several times in between, since all we need is to have a sequence of updates that contains this particular subsequence. There is a very easy way to guarantee this: update all edges $|V|-1$ times in a row!

```
Algorithm Shortest Paths 2 (G = (V,E,length); s ∈ V)
    v,w: vertices
    dist: array[V] of integer
    prev: array[V] of vertices
    i: integer
    for v ∈ V do
        dist[v] := ∞, prev[v] :=nil
    rof
    dist[s] := 0
    for i = 1...n-1
        for (w,v) ∈ E update(w,v)
end shortest paths 2
```

This algorithm solves the general single-source shortest path problem in $O(|V| \cdot |E|)$ time.

Negative Cycles

In fact, there is a further problem that negative edges can cause. Suppose the length of edge (b,a) in Figure 2 were changed to -5. The the graph would have a *negative cycle* (from *a* to *b* and back). On such graphs, it does not make sense to even *ask* the shortest path question. What is the shortest path from *s* to *c* in the modified graph? The one that goes directly from *s* to *a* to *c* (cost: 3), or the one that goes from *s* to *a* to *b* to *a* to *c* (cost: 1), or the one that takes the cycle twice (cost: -1)? And so on.

The shortest path problem is ill-posed in graphs with negative cycles. It makes no sense and deserves no answer. Our algorithm in the previous section works only in the absence of negative cycles. (Where did we assume no negative cycles in our correctness argument? Answer: When we asserted that a shortest path from *s* to *a* exists!) But it would be useful if our algorithm were able to *detect* whether there is a negative cycle in the graph, and thus to report reliably on the meaningfulness of the shortest path answers it provides.

This is easily done. After the $|V|-1$ rounds of updates of all edges, do a last update. If any changes occur during this last round of updates, there is a negative cycle. This must be true, because if there were no negative cycles, $|V|-1$ rounds of updates would have been sufficient to find the shortest paths.

Shortest Paths on DAG's

There are two subclasses of weighted graphs that automatically exclude the possibility of negative cycles: graphs with non-negative weights and DAG's. We have already seen that there is a fast algorithm when the weights are non-negative. Here we will give a *linear* algorithm for single-source shortest paths in DAG's.

Our algorithm is based on the same principle as our algorithm for negative weights. We are trying to find a sequence of updates, such that all shortest paths are its subsequences. But in a DAG we know that all shortest paths from *s* must go in the topological order of the DAG. All we have to do then is first topologically sort the DAG using a DFS, and then visit all edges coming out of nodes in the topological order. This algorithm solves the general single-source shortest path problem for DAG's in $O(m)$ time.