1 Lecture Notes, Fundamental Algorithms: Breadth First Search

Breadth First Search, BFS for short, is a method for exploring the vertices of a graph, whether directed or undirected.

Let $G = (V, E)$ be a graph. A BFS of G starts from a chosen vertex s, and explores all vertices reachable from s. In a search on an undirected graph this will identify the vertices in s's connected component. In order to identify all the connected components, the search is continued from an as yet unexplored vertex, iterating until all vertices have been reached.

One can view a BFS from s as exploring the vertices in the order of their distance from s, measured in number of edges. So first the vertices that are neighbors of s are explored, the vertices at distance 1. Next, the neighbors of the vertices at distance 1 are explored, excluding those vertices that have already been explored; these are the vertices at distance 2. And so on.

To implement this, we use an initially empty queue Q. Recall that a queue supports two operations: EnQueue(v, Q) which adds item v to the rear of Q, and DeQueue(Q), which removes and returns the item at the front of Q. In the standard implementations of a queue both these operations run in time $O(1)$. (These implementions are based on either an array or a linked list.)

Pseudo-code for the BFS follows.

```
for v = 1 to n do Expl[v] ← False
end for

BFS(G, s)
EnQueue(s, Q); Expl[s] ← True;
while not Empty(Q) do
    u ← DeQueue(Q);
    for each edge (u, v) do
        if not Expl[v] then
            EnQueue(v, Q); Expl[v] ← True
        end if
    end for
end while
```

We can enhance this algorithm so as to compute the distance in number of edges from s to each vertex v, as follows. The additional code has been underlined.

```
for v = 1 to n do Expl[v] ← False; Dist[v] ← ∞
end for

BFS(G, s)
EnQueue(s, Q); Expl[s] ← True; Dist[s] ← 0;
while not Empty(Q) do
    u ← DeQueue(Q);
    for each edge (u, v) do
        if not Expl[v] then
            EnQueue(v, Q); Expl[v] ← True; Dist[v] ← Dist[u] + 1
        end if
    end for
end while
```

BFS proceeds in phases, $i = 0, 1, 2, \ldots$. The following invariant is maintained.

Invariant. At the start of the ith phase, the vertices distance at i from s are on the queue and

nothing else. For these vertices, and all vertices nearer to s, Expl[v] = True; for the remaining vertices, Expl[v] = False.

Lemma 1. *The Invariant remains true throughout the course of the BFS algorithm.*

Proof. We prove the claim by induction on the phase number.

Initially, the algorithm is in Phase 0, with s on the queue, and Dist[s] = 0. The invariant holds at this point.

Suppose that the invariant holds at the state of Phase i: exactly those vertices distance at i from s are on the queue Q, and for all vertices v with Dist[v] $\leq i$, Expl[v] = True. Then, in processing the vertices on Q, all their neighbors which have not yet been explored will be added to Q, with their distance from s set to $i+1$. These are precisely the vertices at distance $i+1$, for any vertex v at distance $i+1$ must be on a path with the vertex u immediately preceding v being at distance i. Thus, at the end of Phase i, or equivalently the start of Phase $i+1$, all the vertices at distance i have been removed from Q and the vertices at distance $i+1$ have been added. □

Next, we give the code for computing connected components using BFS.

```
BFSCC(G)
for v = 1 to n do Expl[v] ← False
end for
cpt-no ← 0
for v = 1 to n do
    if not Expl[v] then
        cpt-no ← cpt-no +1; Expl[v] ← True; BFS(G, v)
    end if
end for

BFS(G, s)
EnQueue(s, Q); Expl[s] ← True; Cpt[s] ← cpt-no;
while not Empty(Q) do
    u ← DeQueue(Q);
    for each edge (u, v) do
        if not Expl[v] then
            EnQueue(v, Q); Expl[v] ← True; Cpt[v] ← cpt-no
        end if
    end for
end while
```

Lemma 2. *Suppose undirected graph G has k connected components $C_1, C_2, \ldots, C_k$. Then the vertices in each component C_i are assigned the same number, and vertices in distinct components are assigned distinct numbers.*

Proof. The call to BFS(G, s) reaches all the vertices in s's component and gives all of them the same component number. Each call to BFS(G, v) is preceded by incrementing cpt-no; therefore, the vertices in successive searches receive successively larger numbers, meaning that vertices in distinct components are assigned distinct numbers. □

Lemma 3. *BFSCC runs in linear time.*

Proof. Each vertex is put on the queue exactly once and removed from the queue exactly once. On removing an edge from the queue, each of its outgoing edges is processing in $O(1)$ time. Each edge is processed two times, once from each end. So the total costs for processing the edges is $O(|E|) = O(m)$. Otherwise, there is $O(1)$ work per vertex, for a total of $O(|V|) = O(n)$ work. So the overall runtime is $O(m + n)$, i.e. linear time. □

Note that this bound also applies to a BFS on a directed graph. The only other difference is that some edges may go from a vertex v to a vertex closer to s.

2 Shortest Path Algorithms

We now consider graphs in which the edges have lengths. Typically, the input consists of a directed graph $G = (V, E)$, a source vertex s, and an edge length $\ell(e) \geq 0$ for each edge $e \in E$. The problem is to find, for each $v \in V$, the length of a shortest path from s to v.

In the adjacency list representation of a graph $\ell(e)$ is stored in the node representing edge e. In the adjacency matrix representation, it can be kept in an additional matrix L, where $L[u, v] = \ell(u, v)$.

The idea of our shortest path algorithm is to explore in all directions from s at a uniform rate. As soon as we reach a vertex v we conclude that the distance traversed is the length of the shortest path to v. One can think as having an army of ants marching away from s, and whenever the ants come to a new vertex, they split into groups marching along each outedge. The ants are assumed to all walk at the same speed, so the time to traverse an edge is proportional to its length.

At a high level, this is a BFS traversal starting from s. Indeed, if the edge lengths are all equal to one, then a BFS from s solves the single source shortest path problem.

Now suppose all the edge lengths are positive integers. Then, if we introduce intermediate vertices on each edge separated by unit distance, we could apply BFS to this larger graph in order to find the distance to every vertex from s. There are two problems with this approach: (i) it may increase the size of the graph by a factor of up to $\max_e \ell(e)$ which could be very large; (ii) the edge lengths may not be integers or even rational numbers.

Still, let's see whether we can implement the BFS approach efficiently, and for now we focus on the case that the edge lengths are positive integers. To make the exposition clearer, we introduce a series of queues, $Q_1, Q_2, Q_3, \ldots$. Q_1 will receive the vertices, including intermediate vertices, at distance 1 from s, Q_2, the vertices at distance 2, and so on. What happens when we process an edge (s, v) of length ℓ? First, an intermediate vertex on (s, v) at distance 1 from s is added to Q_1; then, in turn, a second intermediate vertex on (s, v) is added to Q_2, another one to Q_3, ..., and an $(i-1)$th intermediate vertex is added $Q_{\ell-1}$. Finally, if v has not yet been explored, v is added to Q_ℓ.

Clearly, adding the intermediate vertices to each of $Q_1, Q_2, \ldots, Q_{\ell-2}$ is completely unnecessary. And the only reason to add one to $Q_{\ell-1}$ is to decide whether to add v to Q_ℓ subsequently. A simpler approach is to add v to Q_ℓ and then remove it if v gets explored before Q_ℓ is processed.

What this implies is that the only queues we care about are those on which actual (i.e. non-intermediate) vertices are placed, and at each stage of the processing we want to know what is the next queue with an actual vertex, for that is the next queue we need to process. We then go through each vertex u on this queue, processing their outedges (u, v) to as yet unexplored vertices, and putting each far endpoint v on the appropriate queue Q_d where $d = \text{Dist}[u] + \ell(u, v)$, where $\text{Dist}[u]$ is the computed distance to u.

There is one further optimization: if v is already on a queue $Q_{d'}$, there is no point to having v on both Q_d and $Q_{d'}$: we just keep the instance on the first queue, $Q_{d''}$, where $d'' = \min\{d, d'\}$.

We introduce a new data structure called a priority queue, also denoted by Q, which will support the operations we would need for this set of queues. Q stores pairs (v, d), where v is a vertex, and d is a distance from s. We need to support the following operations on Q:

- DeleteMin(Q), which returns the pair (v, d), where v is a vertex, with the minimum d value, and removes it from Q. This allows us to identify the next vertex to process.
- ReduceKey(Q, v, d'), which updates item (v, d) to (v, d') if $d' < d$.

It is also convenient to be able to create a priority queue with an initial set of n pairs. We call this the BuildQueue operation. Our algorithm is further simplified by adding imaginary edges of length ∞ from s to each vertex v. If the final distance to a vertex v is ∞, this indicates there is no path to v from s.

This leads to the following algorithm, called Dijkstra's algorithm (pronounced "Dikestra").

```
SSSP(G, s)
for v = 1 to n do Dist[v] ← ∞
end for
Dist[s] ← 0;
BuildQueue(Q, V, Dist) (* for each v ∈ V, adds (v, Dist[v]) to *)
                       (*an initially empty priority queue Q *)
while Q not empty do
    u ← DeleteMin(Q);
    for each edge (u, v) do
        new-dist ← Dist[u] + ℓ(u, v);
        if Dist[v] > new-dist then
            Dist[v] ← new-dist; ReduceKey(v, new-dist)
        end if
    end for
end while
```

In fact, this algorithm will work for all non-negative edge lengths, and not just positive integer lengths. We proceed to justify this rigorously.

Lemma 4. *Suppose $G = (V, E)$ is a directed graph with non-negative edge lengths. Then Dijkstra's algorithm correctly computes the lengths of the shortest paths from s to all $v \in V$.*

Proof. We have already seen that BFS correctly computes the distances if all edge lengths are equal to 1. As already described, BFS processes the vertices in phases, and thereby puts them in layers. Layer 0 is vertex s, Layer 1 comprises the vertices at distance 1 from s, and more generally, Layer i comprises the vertices at distance i from s.

In fact, all we need is that every edge from Layer i to Layer $i + 1$ has the same length, $\ell_i > 0$; the correctness argument is essentially unchanged.

The invariant becomes:

Invariant. At the start of the ith phase, the vertices distance $d_i = \sum_{h=0}^{i-1} \ell_h$ from s are on the queue and nothing else. For these vertices, and all vertices nearer to s, Expl[v] = True; for the remaining vertices, Expl[v] = False.

We prove the Invariant is true at the start of the each phase by induction on the phase number.

Initially, the algorithm is in Phase 0, with s on the queue, and Dist[s] = 0. The invariant holds at this point.

Suppose that the invariant holds at the state of Phase i: exactly those vertices at distance d_i from s are on the queue, and for all vertices v with Dist[v] $\leq d_i$, Expl[v] = True. Then, in

processing the vertices on the queue, all their neighbors which have not yet been explored will be added to the queue, with their distance from s set to $d_i + \ell_i$. These are precisely the vertices at distance d_{i+1}, for any vertex v at distance d_{i+1} must be on a path with the vertex u immediately preceding v being at distance d_i. Thus, at the end of Phase i, or equivalently the start of Phase $i+1$, all the vertices at distance d_i have been removed from Q and the vertices at distance d_{i+1} have been added.

Note that this result holds for all positive values of ℓ_i, and not just positive integer values. We will explain how to handle zero-valued edge lengths later.

If we introduce intermediate vertices on the edges corresponding to the points at distances d_i for $i = 1, 2, \ldots$, we can then view the graph as having this form: all the edges from one layer to the next have the same length, and so BFS will compute the lengths of the shortest paths correctly.

To understand Dijkstra's algorithm let's rename the layers of vertices by their distance from s, i.e. L_d comprises those vertices at distance d, and Q_d will be the queue of vertices at distance d. So when processing a vertex u at distance $d = \text{Dist}[u]$ from s, for each edge (u, v) we want to add v to $Q_{d'}$ where $d' = \text{Dist}[u] + \ell(u, v)$ (thereby avoiding the processing of the intermediate vertices on (u, v)). Note that this means we have found a path from s to v of length d', namely the shortest length path to u followed by edge (u, v).

Since in the BFS we explore each vertex v the first time we encounter it, if v is on a queue $Q_{d'}$ already and we seek to put it on a second queue $Q_{d''}$, we only want to keep the instance $Q_{d'''}$, where $d''' = \min\{d', d''\}$. But this is exactly what Dijkstra's algorithm is doing.

We conclude that Dijkstra's algorithm is implementing BFS on a graph with intermediate vertices on the edges, and equal length edges between successive layers of vertices, and therefore correctly computes the lengths of the shortest paths.

Finally, how can we handle for zero length edges? Let $\epsilon > 0$ be an infinitesimal, i.e. it is much smaller than the difference between the lengths of any two unequal length paths in G. We replace each zero length edge by an edge of length ϵ, and proceed to compute shortest paths in this graph. If we find a shortest path in the new graph of length $d + k\epsilon$ for some $k \geq 0$, we know there is a path in the original graph of length d. Also, there is no path of length $d' < d$ in the original graph, for this would yield a path of length $d' + k'\epsilon$ in the modified graph, and by the definition of ϵ, $d' + k'\epsilon < d + k\epsilon$ as $d' < d$. □

We turn to analyzing the running time of Dijkstra's algorithm. We express the runtime bound in terms of the number of Priority Queue operations that we perform. We perform one BuildQueue operation, which adds $|V|$ items to the priority queue Q. We then perform $|V|$ DeleteMin operations at the start of the while loop, as no new items are added to Q. For each DeleteMin of a vertex u, we process the outedges from u, performing at most one ReduceKey operation per edge. As each vertex is deleted once from Q, this is a total of at most $|E|$ ReduceKey operations. The remaining processing takes $O(1)$ time per vertex and $O(1)$ time per edge. We summarize the bound in the following lemma.

Lemma 5. *Dijkstra's algorithm runs in time $O(|V| + |E|)$ plus the time for one BuildQueue on $|V|$ items, $|V|$ DeleteMins, and $|E|$ ReduceKeys.*

We conclude by stating some runtime bounds for various implementations of a priority queue on n items. We also include the Insert operation, which is not needed for Dijkstra's algorithm, but is a standard priority queue operation.

	BuildQueue	DeleteMin	ReduceKey	Insert
Binary heap	$O(n)$	$(O \log n)$	$(O \log n)$	$(O \log n)$
d-ary heap	$O(n)$	$O\big(d\frac{\log n}{\log d}\big)$	$O\big(\frac{\log n}{\log d}\big)$	$O\big(\frac{\log n}{\log d}\big)$
Strict Fibonacci heap	$O(n)$	$O(\log n)$	$O(1)$	$O(1)$
Relaxed heap	$O(n)$	$O(\log n)$	$O(1)$	$O(1)$

Corollary 6. *Dijkstra's algorithm runs in the following times.*

- *With a binary heap: time* $O(|V| + |E|) \log |V|) = O((n + m) \log n)$.
- *With a d-ary heap with* $d = |E|/|V|$*:* $O(|V|+|E| \log |V|/ \log(|E|/|V|)) = O(n+m \log n/ \log(m/n))$.
- *With a strict Fibonacci heap or a relaxed heap:* $O(|V| \log |V| + |E|) = O(n \log n + m)$.

Note that if $m = n^2$ the d-ary heap time becomes $O(n + m)$; if $m = n^{1+\epsilon}$, the time becomes $O((n + m)/\epsilon)$. If ϵ is not too small, this is highly efficient.

2.1 Path Recovery

Suppose one wants to be able to report shortest paths from s to v. In the same way as with dynamic programming, for each vertex v, we record the choice of the predecessor vertex of v on the current shortest path. To do this we augment Dijkstra's algorithm as follows (the additional code is underlined).

```
SSSP(G, s)
for v = 1 to n do Dist[v] ← ∞; Pred[v] ← nil
end for
Dist[s] ← 0;
BuildQueue(Q, V, Dist) (* for each v ∈ V, adds (v, Dist[v]) to *)
                       (*an initially empty priority queue Q *)
while Q not empty do
    u ← DeleteMin(Q);
    for each edge (u, v) do
        new-dist ← Dist[u] + ℓ(u, v);
        if Dist[v] > new-dist then
            Dist[v] > new-dist; ReduceKey(v, new-dist);
            Pred[v] ← u
        end if
    end for
end while
```

This adds $O(|V| + |E|)$ time to the overall runtime of Dijkstra's algorithm, so leaves its asymptotic runtime unchanged.

It is now easy to recover the path to v in reverse order; we simply follow the predecessor links until we reach s.

```
RevPath(v)
Print(v)
while Pred[v] ≠ nil do RevPath(Pred[v])
end while
```

But we would like the path in the forward direction. To obtain this, as we traverse the path in the reverse direction, we push the vertices on a stack. Once s is reached, we repeatedly pop the stack which outputs the path in the forward direction.

```
Path(v)
create empty stack S;
repeat
    Push(v, S); v ← (Pred[v])
until v = nil
while S not empty do u ← Pop(S); Print(u)
end while
```

Clearly, the runtime for printing a path is linear in the length of the path.