1 Graph Search (Abstract) with Data Structures

Figure 1: $R$ is the known region, all the nodes that we reached already. $F$ is the frontier.

procedure Graph-Search(G, A):
    Input: G directed graph, in adjacency list format
    A node
    Output: list $R$ of reachable nodes from $A$

$R = [A]$
$F = [A]$
while $F$ is not empty:
    pick $u$ in $F$
    for each $v$ in $N(u)$:
        if $v$ not in $R$:
            add $v$ to $F$
            add $v$ to $R$
    delete $u$ from $F$

return $R$

Note: Pick $u$ in $F$ can only occur once. After selected in $R$ and is thus never added again due to if clause filtering.

We saw a few choices for managing the frontier. We can use either a stack (DFS) or a queue (BFS). For a stack we have: Top, Push, Pop all are $O(1)$. For a queue we have: Head, Enqueue, Dequeue, all are $O(1)$.

Let’s explore DFS specifically:

procedure function DFS(G, A):
    Input: G directed graph in adjacency list format
A node

Output:

for each node v:
InR(v) = False
InR(A) = True

initialize stack
push(A)

while Top ≠ nil:
    u = pop
    for each v in N(u):
        if InR(v) == False:
            InR(v) = True
            push(v)

return nodes v that have InR(v) == True

Showing graph traversal,
Order of traversal is: A, B, D, F, E, H, C. Note that G is not reachable from A. We try to get as far away as possible from the start node. When we get stuck, we back up and try to find another path.

When use DFS and not BFS?

• SCCs in a directed graph
• Linearizing via use of pre/post numbers

How do we change the above to BFS? Simply replace all stack operations with the corresponding queue operations. Top becomes Head, Push become Enqueue, and Pop becomes Dequeue. Ordering for performing BFS: A, B, C, D, E, F, H. The node G is still not reachable.

Implied in BFS: We’re not only finding paths to each reachable node. We’re finding shortest paths to each reachable node! Often we may be looking only for a node that is close by. Then BFS offers a benefit. We don’t want to explore irrelevant parts of the graph.
1.1 Time analysis of BFS

Let $n = |V|$ and $m = |E|$, the number of nodes and edges respectively.

How many times do we perform the outer loop of BFS? In the worst case each node is explored, that is, at most $n$ iterations.

How long does the inner loop require? The number of edges for a given node is upper bounded by the number of nodes in the graph. So, the inner loop requires $n$ steps in the worst case.

Inside the for loop is just constant time.

But where might this not be tight? Suppose we have $t$ iterations of the outer loop. Then what we really care about is the sum of the time $T_i$ of the inner loop iterations: $T_1, T_2, \ldots, T_t$.

When we are being lazy, we do the following:

$$\sum_{i=1}^{t} T_i \leq t \cdot \max T_i$$

Sometimes this upper bound is tight. This may be the case when, for example, there is not much variation in each $T_i$ value. For BFS, can we give an expression that is tighter?

In our case, the outer loop is $n$ iterations. This happens once for each node $u \in V$. As argued above, the inner loop for each $u$ requires $\deg(u) + 1$ steps in the worst case. Our sum becomes:

$$\sum_{u \in V} \deg(u) + 1 = \sum_{u \in V} \deg(u) + \sum_{u \in V} 1 = m + n$$

This is true since we count each edge exactly once. So, each edge contributes just 1 to the sum of degrees. In conclusion, we can restate the complexity of BFS as $O(m + n)$. It’s possible $m = n^2$ (dense graphs) and so our origial analysis is consistent. However, the multivariable analysis captures the sparse graph case as well and so is preferred.

What changes when we do this time analysis for DFS vs. BFS? Nothing!

1.2 Unweighted Path Lengths with BFS

Now, when we visit a node, we also need to keep track of its distance from the starting node. We can use an array of dists with initial distances set to infinity and the distance to $A$ set to 1. Now, the distance to $v$ can be stated as the distance (num. edges) to $u + 1$.

2 Shortest Paths in Weighted Graphs

With weights, we must consider more than just the number of edges used to reach our destination. For example, routing delays or road lengths. In a more general shortest-paths problems we might thus also have edge weights. Different edges may have different weights and we consider positive weights for now.

Our goal is to find the path from $ZA$ to each node $v \in V$ that minimizes $\sum_{e \in E} w(e)$. Can we use our previous algorithm (BFS) to handle weights? Yes, we can modify our graph so that the weighted case becomes the unweighted case.

Transformation:
• An edge of length \( w \) becomes \( w \) edges of length 1.

• If we do this consistently for all edges then finding the shortest weighted path is the same problem as finding the shortest unweighted path via BFS.

But what if the weights are very large numbers? Would we really want to add this many intermediate nodes to the graph? Probably not!

2.1 Implicit BFS of a Weighted Graph

After applying the transformation above, what is the first time we hit a real node in the graph when using BFS? When we reach the node which is closest to our starting node \( A \).

So, we have a set of nodes \( R \) that we know are reachable at a certain time point. Which edges are we starting to explore at a given time step? These edges now comprise the frontier \( F \) of our exploration rather than a set of nodes.

For an edge \( e = (u, v) \in F \) then \( u \in R \) and \( v \notin R \). Let \( d(u) \) be the shortest-path distance. This is only defined for nodes \( u \in R \). Then we started to explore \( e \) at \( d(u) \), the step when we added \( u \) to \( R \).

Via the edge \( e \) we will reach \( v \) at \( d(u) + w(e) \). When we get to \( v \) at this time, if \( v \notin R \) then \( d(v) \) will become \( d(u) + w(e) \). If instead \( v \in R \) we do not change \( d(v) \).

Consider the possible distances to reachable nodes as alarm clocks. We set alarms for a node \( v \) when we add \( v \) to \( R \). The alarm “time” is the distance \( d(v) \). These are nodes for which we know the shortest path distances.

Then we “wake-up” for a node when the earliest possible (shortest path) alarm goes off. We do not re-add nodes to the set of alarms after we have set a distance for the node.

Let \( a(u) \) be the current shortest path known to \( u \). When we move from the unknown side of the graph to the known side of the graph, we move the node with the earliest set alarm \( a(u) \). When we move it, we adjust the alarm times for all neighbors of \( u \).

The above is known as Dijkstra’s algorithm.