



8.9. Implementing Breadth First Search

With the graph constructed we can now turn our attention to the algorithm we will use to find the shortest solution to the word ladder problem. The graph algorithm we are going to use is called the “breadth first search” algorithm. **Breadth first search (BFS)** is one of the easiest algorithms for searching a graph. It also serves as a prototype for several other important graph algorithms that we will study later.

Given a graph G and a starting vertex s , a breadth first search proceeds by exploring edges in the graph to find all the vertices in G for which there is a path from s . The remarkable thing about a breadth first search is that it finds *all* the vertices that are a distance k from s before it finds *any* vertices that are a distance $k + 1$. One good way to visualize what the breadth first search algorithm does is to imagine that it is building a tree, one level of the tree at a time. A breadth first search adds all children of the starting vertex before it begins to discover any of the grandchildren.

To keep track of its progress, BFS colors each of the vertices white, gray, or black. All the vertices are initialized to white when they are constructed. A white vertex is an undiscovered vertex. When a vertex is initially discovered it is colored gray, and when BFS has completely explored a vertex it is colored black. This means that once a vertex is colored black, it has no white vertices adjacent to it. A gray node, on the other hand, may have some white vertices adjacent to it, indicating that there are still additional vertices to explore.

The breadth first search algorithm shown in Listing 2 below uses the adjacency list graph representation we developed earlier. In addition it uses a `Queue`, a crucial point as we will see, to decide which vertex to explore next.

In addition the BFS algorithm uses an extended version of the `Vertex` class. This new vertex class adds three new instance variables: `distance`, `predecessor`, and `color`. Each of these instance variables also has the appropriate getter and setter methods. The code for this expanded `Vertex` class is included in the `pythonds` package, but we will not show it to you here as there is nothing new to learn by seeing the additional instance variables.

BFS begins at the starting vertex `s` and colors `start` gray to show that it is currently being explored. Two other values, the distance and the predecessor, are initialized to 0 and `None` respectively for the starting vertex. Finally, `start` is placed on a `queue`. The next step is to begin to systematically explore vertices at the front of the queue. We explore each new node at the front of the queue by iterating over its adjacency list. As each node on the adjacency list is examined its color is checked. If it is white, the vertex is unexplored, and four things happen:

1. The new, unexplored vertex `nbr`, is colored gray.
2. The predecessor of `nbr` is set to the current node `currentVert`
3. The distance to `nbr` is set to the distance to `currentVert` + 1
4. `nbr` is added to the end of a queue. Adding `nbr` to the end of the queue effectively schedules this node for further exploration, but not until all the other vertices on the adjacency list of `currentVert` have been explored.

Listing 2

(BuildingtheWordLadderGraph.html)

(BreadthFirstSearch.html)

```

from pythonds.graphs import Graph, Vertex
from pythonds.basic import Queue

def bfs(g, start):
    start.setDistance(0)
    start.setPred(None)
    vertQueue = Queue()
    vertQueue.enqueue(start)
    while (vertQueue.size() > 0):
        currentVert = vertQueue.dequeue()
        for nbr in currentVert.getConnections():
            if (nbr.getColor() == 'white'):
                nbr.setColor('gray')
                nbr.setDistance(currentVert.getDistance() + 1)
                nbr.setPred(currentVert)
                vertQueue.enqueue(nbr)
        currentVert.setColor('black')

```

Let's look at how the `bfs` function would construct the breadth first tree corresponding to the graph in Figure 1 (BuildingtheWordLadderGraph.html#fig-wordladder). Starting from `fool` we take all nodes that are adjacent to `fool` and add them to the tree. The adjacent nodes include `pool`, `foil`, `foul`, and `cool`. Each of these nodes are added to the queue of new nodes to expand. Figure 3 shows the state of the in-progress tree along with the queue after this step.

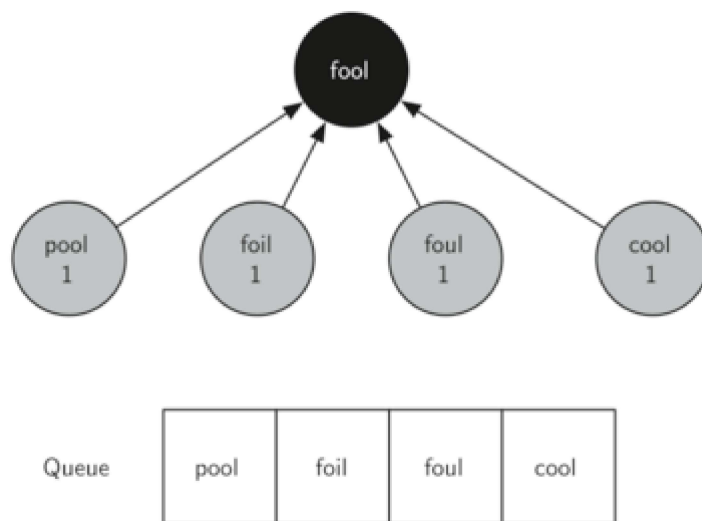


Figure 3: The First Step in the Breadth First Search

In the next step `bfs` removes the next node (`pool`) from the front of the queue and repeats the process for all of its adjacent nodes. However, when `bfs` examines the node `cool`, it finds that the color of `cool` has already been changed to gray. This indicates that there is a shorter path to `cool` and that `cool` is already on the queue for further expansion. The only new node added to the queue while examining `pool` is `poll`. The new state of the tree and queue is shown in Figure 4.

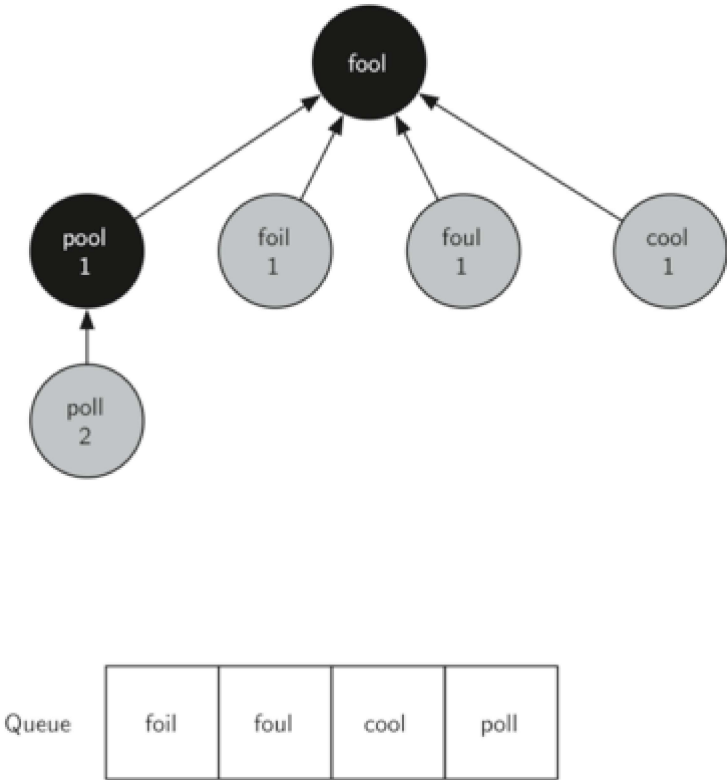


Figure 4: The Second Step in the Breadth First Search

The next vertex on the queue is foil. The only new node that foil can add to the tree is fail. As bfs continues to process the queue, neither of the next two nodes add anything new to the queue or the tree. Figure 5 shows the tree and the queue after expanding all the vertices on the second level of the tree.

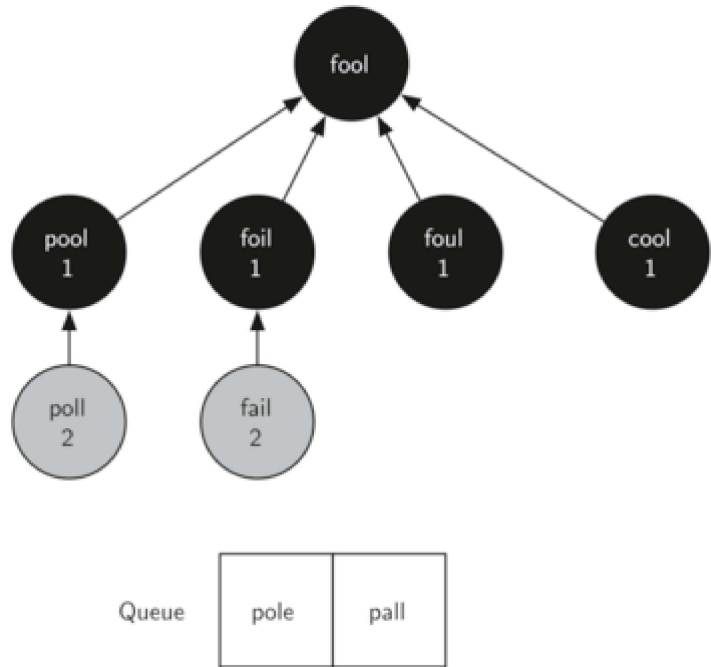


Figure 5: Breadth First Search Tree After Completing One Level

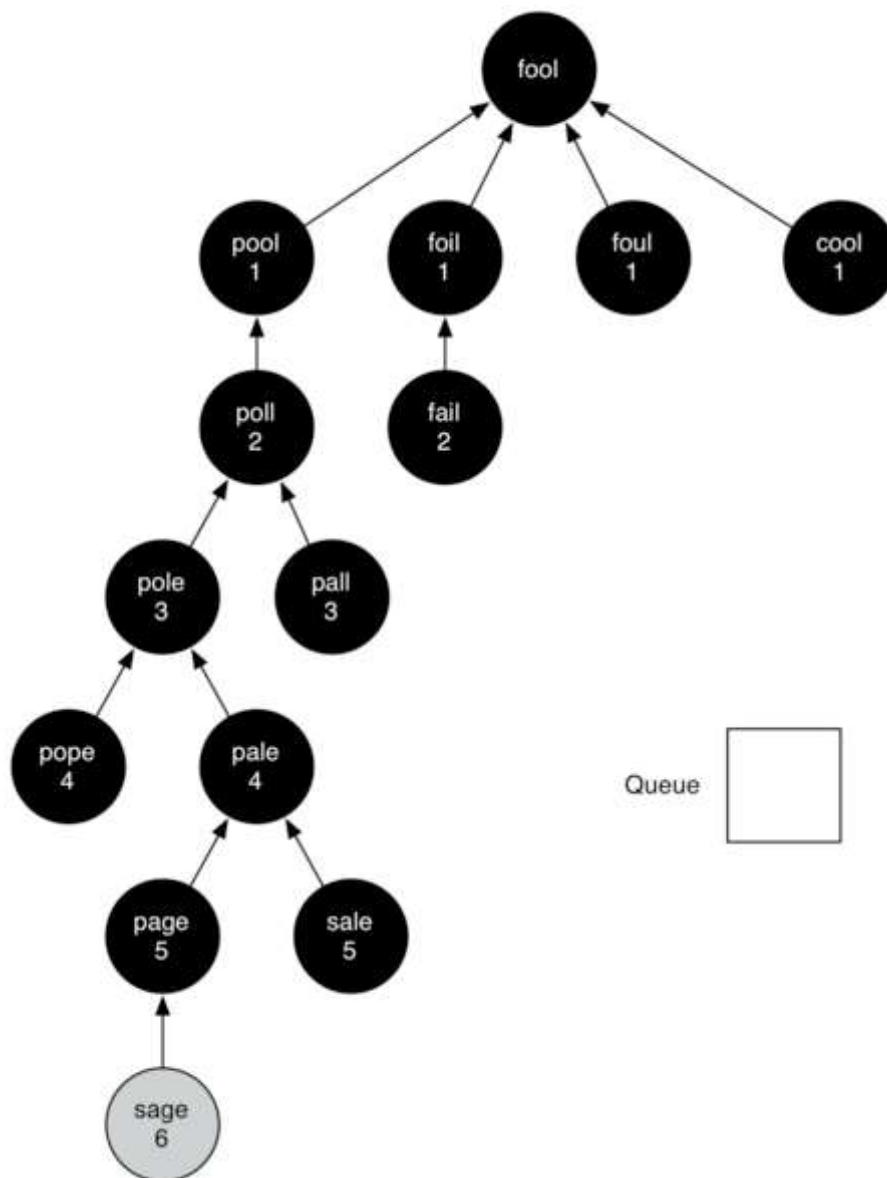


Figure 6: Final Breadth First Search Tree

You should continue to work through the algorithm on your own so that you are comfortable with how it works. Figure 6 shows the final breadth first search tree after all the vertices in Figure 3 (BuildingtheWordLadderGraph.html#fig-wordladder) have been expanded. The amazing thing about the breadth first search solution is that we have not only solved the FOOL–SAGE problem we started out with, but we have solved many other problems along the way. We can start at any vertex in the breadth first search tree and follow the predecessor arrows back to the root to find the shortest word ladder from any word back to fool. The function below (Listing 3) shows how to follow the predecessor links to print out the word ladder.

Listing 3

```


def traverse(y):
    x = y
    while (x.getPred()):
        print(x.getId())
        x = x.getPred()
    print(x.getId())

traverse(g.getVertex('sage'))

```

user not logged in

 (BuildingtheWordLadderGraph.html)

(BreadthFirstSear)