1 Searching Graphs using DFS

The title might seem a bit strange - precisely what could we mean by “searching” a graph? We will discuss a particular method for exploring graphs called depth first search. We recently learned how graphs could be compactly represented on a computer using arrays. What kinds of questions might we want to ask about graphs now?

One of the most elementary computations is the following – suppose the graph represents a map, and we would like to go from node A to node B – how can we accomplish that? We are also interested in actually finding the shortest route to go from node A to node B. We could assume that in some way the length of edges in the graph have been encoded, so for example we could have a map of cities on the east coast with distances marked, and try to compute a shortest route to go from one place to another. In fact, when you use Google maps, this happens in some way. We first focus on the simpler problem first of simply finding one route from A to B.

Our algorithm is itself rather simple - but we do need some mechanism for keeping track of which nodes have been visited already. Our search algorithm will start from one specific vertex p and output a list of all the nodes that are reachable from vertex p.

Along with the graph representation, this time we also define an array called V. The entry V[i] will be either true or false. The entry is false initially, and when we visit a node i, we change V[i] to true. (You could think of V for “visited”.)

When we first reach a node that is not yet visited, we begin to explore all of its unvisited neighbors in turn. The important thing to note is that the moment we find an unvisited neighbor we mark it visited and then immediately move to that node to explore its unvisited neighbors. Eventually when all edges coming out of a node v have been explored, we return to the node from where we got to v in the first place (unless v is the starting node, in which case the procedure terminates).

Consider the graph in Fig. 1. Suppose we start at node A; we first mark it visited. If node A has neighbors B, C, D and E. We move to B, mark it visited, and then from there to C, and mark C visited. Suppose now that C has no unvisited neighbors, then we simply return to where we came from – thus we would find ourselves back at node B, and then since there is an edge from B to D, we will move to D, mark it visited and since it has no unvisited neighbors we move back to where we came from, namely B. Now B has no unvisited neighbors so we go back to where we came from – namely A. Since A has an unvisited neighbor E, we move to E, mark it visited and since E has an unvisited neighbor F, we mark F as visited and move to F. However F has no unvisited neighbors, so we return to E, and since E has no more unvisited neighbors we return to A and the search terminates.

---

Department of Computer Science, University of Maryland, College Park, MD 20742. E-mail: samir@cs.umd.edu.
The actual ruby code is shown below and prints the set of nodes that are visited when we first call dfs on some node of the graph.

A graph is said to be *connected* if there is a route that connects every pair of nodes. So if we call dfs from any node in the graph, all the remaining N-1 nodes are visited. If the graph is not connected then a dfs search from any vertex cannot visit all the nodes of the graph.

While it looks as if this was perhaps a trivial problem and we did not really do too much besides list the set of nodes that are visited while running the search from an initial starting node, in fact, it turns out that depth first search (dfs) is actually a rather powerful tool in the analysis of graphs and extremely widely used for testing graphs for specific properties, along with computing certain pieces of information that cannot be computed locally.

```ruby
# DFS
N=9
G=Array.new
N.times{|j| G[j]=Array.new }
G[0]<<1<<2
G[1]<<0<<2<<6
G[2]<<0<<1<<3<<4
G[3]<<2<<4
G[4]<<2<<3<<5
G[5]<<4
G[6]<<1
G[7]<<8
G[8]<<7

#V is the visited array of size N and G is the graph
#You will have to create the representation for a graph G
V=Array.new
N.times{|j|V[j]=false}

def dfs(i)
  V[i]=true
  G[i].length.times{|k|
    if not (V[G[i][k]])
      print G[i][k]
      dfs(G[i][k])
    end
  }
end
end
```
if (ARGV.length == 1)
    p = Integer(ARGV[0])
end

print("The nodes reachable from #{p} are ")
dfs(p)
print "\n"

Application: One interesting application is the problem of indexing the entire Web. How does Google store all of the web pages in the World Wide Web (WWW)? One way they can do this is by doing what is called a “web crawl”. This web crawl might start from one web page (think of this as a node in a giant graph), say www.umd.edu, from there they could follow links to other web pages and discover them? Again some mechanism is needed for recording which web pages we have already visited. If you think about this the process is the really the same as that of “searching/exploring” a graph. Except that they have to do this incrementally and if I create a new web page (say one for CMSC 198K) and then add a link to it from my web page, and from www.cs.umd.edu, somehow Google “discovers” this new web page and links to it. That is because they probably repeatedly scan (perhaps once a day, or once a week) the web page www.cs.umd.edu to see what new links have been added.

How would we design an experiment to see how frequently Google scans the web to learn about new web pages?

2 Data Structures for managing data

A stack can be thought of as a pile of books. The last book added is on top of the stack. At every step we may add a new book, or remove a book from the stack. The main point is that this is a LIFO (last in first out) structure. If we add object X, and then object Y. Now if we wish to remove an object, we will first remove Y and then remove X.

Let S denote a stack. The functions below can be used to push a new element i onto the stack, or to pop an element from the stack. The pop function returns the popped element. Notice that we set the top element to nil as well.

S=Array.new

def push(S,i)
    S << i
end

def pop(S)
    #Assume S is not empty
    i=S[S.length-1]
    S[S.length-1]=nil
    return i
end

For computing shortest paths we use a structure that will allow us to do some specific operations. We call this construct a queue. This is similar to a stack. In a stack we just
add and remove objects from one end (the top); here we add things at one end, but we remove
them from the other end. A stack is what is called a LIFO structure (last in, first out). A queue
in contrast, is a FIFO structure (first in, first out).

The three most useful operations are emptyQ, AddQ(i) and RemoveQ. We will discuss how to
implement a queue later on.

3 Shortest Paths (BFS)
The goal is to compute the shortest path to every other node in the graph. We assume that the
input is an unweighted graph and we would like to find the shortest route from a node A to node
B. In fact, even though we only wish to compute a shortest route to one particular node in the
depth-first search by adding the starting node A to the queue. At each step we remove a node from the
queue and then immediately mark all of its unvisited neighbors as visited, and add them to the
queue. Consider the graph we ran dfs on earlier. There we would mark A as visited, and then
add all the nodes B, C, D and E to the queue after marking them as visited. We would then
remove node B from the queue. However since both of B’s neighbors have been visited already,
we do not add any more nodes to the Queue when we process node B. We next process C and
D, and again no new nodes are discovered. Now the queue contains only node E. However when
E is removed and processed we add F to the queue, and then finally F is processed.

#V is the visited array and G is the graph of N nodes
V=Array.new; N.times{|j|V[j]=false}

def bfs(i)
 V[i]=true
 AddQ(i)
 while not (emptyQ) do
  v=RemoveQ
  G[v].length.times{|j|
   if not(V[G[v][j]])
    AddQ(G[v][j]); V[G[v][j]] = true
   end
  }
 end
end

if (ARGV.length == 1)
  p = Integer(ARGV[0])
end

print("The nodes reachable from #{p} are ")
bfs(p)