Graph Representations

Let $G = (V, E)$ with $|V| = n$, $|E| = m$. In other words, for some graph $G$, it contains $n$ vertices and $m$ edges.

Adjacency Matrix

One way to represent $G$ is with an $n \times n$ matrix $A$ where $A[i][j] = 1$ if there is an edge from vertex $i$ to vertex $j$ and 0 otherwise. The primary advantage of this approach is that you can check whether or not there is an edge connecting two vertices in $O(1)$ time. The disadvantage, however, is that this representation takes up $O(n^2)$ space. When $n$ is large, this might become untenable.

Two things worth noting:

- If $G$ is undirected, then its adjacency matrix is symmetric. That is, flipping the matrix along its main diagonal will produce the same matrix.
- Entries along the diagonal of an adjacency matrix (technically representing the presence of edges from vertices to themselves) are 0 by convention, as our graphs are simple. Non-simple graphs have self-loops, where vertices contain edges to themselves (these will not be dealt with in this course).

Adjacency List

Another way to represent $G$ is to use an adjacency list. Each vertex $u$ is associated to a list $\text{neighbors}(v)$ which contains the nodes $v$ such that $(u, v) \in E$. The advantage of this representation is that we use less space, $O(n + m)$, which is better than $O(n^2)$ of adjacency matrices as long as $m \ll n^2$. The disadvantage, though, is that checking whether $(u, v) \in E$ takes (potentially) linear time.

Graph Traversals

We now look at two ways to traverse a graph.

BFS (Breadth First Search)

In BFS, we begin at a node $v$ (level 0) and explore the graph in “layers.” First we would explore all children of $v$ (level 1), then the children of the nodes in level 1 (these would make up level 2), etc. The key point here is that we explore all nodes at level $i$ before exploring any nodes at level $i + 1$. The output of BFS is called a BFS tree. We typically use a queue to implement this algorithm. For implementation details, see https://en.wikipedia.org/wiki/Breadth-first_search.

The running time of BFS is $O(n + m)$, because each vertex is added and removed from the queue once and, in the worst case, we need to traverse every edge to visit each node.

DFS (Depth First Search)

In DFS, we begin at a node $v$ and examine its neighbors. As soon as we encounter a neighbor that hasn’t been visited, visit it. Once we arrive at a node for which all of its neighbors have been visited, we “backtrack” until we reach a node that has still unvisited neighbors (in the form of returning from recursive visit calls). We typically use a stack. There is also a recursive method to implement this algorithm. Please see both implementation methods in the link below.
The running time analysis for DFS is similar to that of BFS, giving a running time of $O(n + m)$.

**Problem 1: Cycle Detection**

Design an algorithm to determine whether or not a graph has a cycle.

*Solution.* You can perform a BFS or DFS and just keep track of which elements have been seen. For example, you can run a DFS and store vertices you have seen in a set and just track whether or not any previously seen node is encountered again by checking if it is in the set. Since we are simply doing a BFS or DFS, this algorithm runs in linear time.

**Problem 2: More Cycle Detection!**

Design an algorithm to determine whether or not a connected graph has a cycle in $O(n)$ time.

*Solution.* Perform the same algorithm as problem 1. However, terminate early if you explore at least $n$ edges. Recall that a tree has exactly $n - 1$ edges. An additional edge would signify that two nodes are connected by two independent paths. Thus, there is a cycle in the graph. This algorithm will take $O(n)$ time to check each vertex and $O(n)$ to check each edge (since we are checking at most $n$ edges). Thus, the running time is $O(n)$.

Note that a graph with $n$ edges must have a cycle regardless of whether it’s connected. This example is just simple to prove with BFS.

**Problem 3: Shortest Path**

Design an algorithm to find the shortest path between nodes $u$ and $v$ in a connected, unweighted graph.

*Solution.* Since the graph is unweighted, we can just run BFS starting from $u$ and for each node $x$ that we visit, we just keep a pointer to its parent node (the node we visited $x$ from). When we reach $v$, we stop and find the shortest path by backtracking through the pointers that we kept (i.e. we could see that $v$’s parent was $d$, $d$’s parent was $c$, and $c$’s parent was $u$, so our path would be $u \rightarrow c \rightarrow d \rightarrow v$). We just do a BFS and backtrack no more than $O(n)$ times (the longest path in a graph is $n - 1$ edges), so this algorithm also runs in linear time.

**Problem 4: Tic-tac-toe**

Suppose we are given a graph of tic-tac-toe moves such that nodes are board states and an edge from $u$ to $v$ means that $v$ is reachable from $u$ in one move. Design an algorithm that takes in a board state and determines the best possible next move (i.e. in most cases the move that will guarantee a draw or a win, unless of course every move results in a loss).

*Solution.* Perform DFS on the input node with the following modification. Assign all leaf nodes a value of 1 if it is a winning or tie board, and 0 for a loss (leaf nodes are full boards, so the result is decided).

For any node $v$ in the graph, let $W$ be the total number of wins and ties reachable from a child of $v$, and let $L$ be the total losses reachable from a child of $v$. Let $S$ be the max of $\frac{W}{W + L}$ over all of each node’s children. This is the value of the child with the largest proportion of win and tie moves over losses.

Just before a node is popped off the stack in DFS, find which of it’s children has the maximum value of $S$. At the end of the algorithm, the root (input) node will have several children with different values of $S$. Return the child of the root with the largest $S$ value. This will represent the best next possible board state.
Problem 5: Recursive Permutation

Recursively generate all the permutations of the character sequence 'ABCD'.

Solution. The key to understanding how we can generate all permutations of a given string is to imagine the string (which is essentially a set of characters) as a complete graph where the nodes are the characters of the string. This basically reduces the permutations generating problem into a graph traversal problem: given a complete graph, visit all nodes of the graph without visiting any node twice. How many different ways are there to traverse such a graph?

It turns out, each different way of traversing this graph is one permutation of the characters in the given string!

We can use DFS to traverse this graph of characters. The important thing to keep in mind is that we must not visit a node twice in any "branch" of the depth-first tree that runs down from a node at the top of the tree to the leaf which denotes the last node in the current "branch".

The code solution is on this website: [http://exceptional-code.blogspot.com/2012/09/generating-all-permutations.html](http://exceptional-code.blogspot.com/2012/09/generating-all-permutations.html)

Beyond the Class: BFS and DFS in Artificial Intelligence

If you are curious about how BFS and DFS are used in the artificial intelligence context then please look at the link below. These slides go over the subtle pro’s and con’s of DFS vs. BFS as well as a new type of search called Iterative Deepening Search.

[http://www.seas.upenn.edu/~cis521/Lectures/uninformed-search.pdf](http://www.seas.upenn.edu/~cis521/Lectures/uninformed-search.pdf)