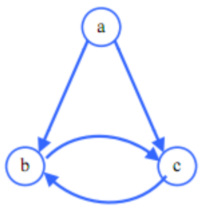
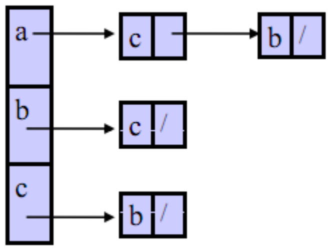
Graph Representation

The two main graph representations we use when talking about graph problems are the **adjacency list** and the **adjacency matrix**. It's important to understand the tradeoffs between the two representations. Let $G = (V, E)$ be our graph where V is the set of vertices and E is the set of edges where each edge is represented as a tuple of vertices.

Adjacency List

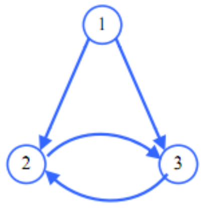
An adjacency list is a list of lists. Each list corresponds to a vertex u and contains a list of edges (u, v) that originate from u. Thus, an adjacency list takes up $\Theta(V + E)$ space.

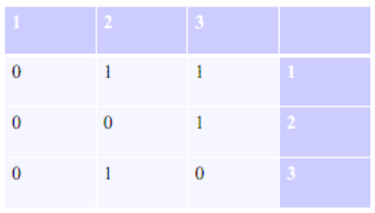




Adjacency Matrix

An adjacency matrix is a $|V| \times |V|$ matrix of bits where element (i, j) is 1 if and only if the edge (v_i, v_j) is in E. Thus an adjacency matrix takes up $\Theta(|V|^2)$ storage (note that the constant factor here is small since each entry in the matrix is just a bit).





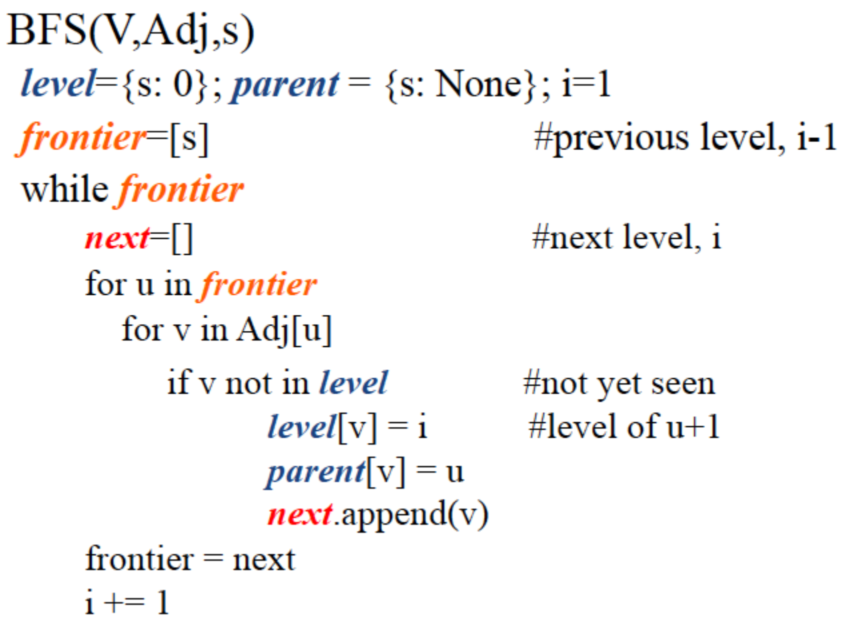
1	2	3	
0	1	1	1
0	0	1	2
0	1	0	3

Comparison

The worst case storage of an adjacency list is when the graph is **dense**, i.e. $E = \Theta(V^2)$. This gives us the same space complexity as the adjacency matrix representation. The $\Theta(V + E)$ space complexity for the general case is usually more desirable, however. Furthermore, adjacency lists give you the set of adjacent vertices to a given vertex quicker than an adjacency matrix $O(\text{neighbors})$ for the former vs $O(V)$ for the latter. In the algorithms we've seen in class, finding the neighbors of a vertex has been essential.

BFS

BFS (breadth first search) is an algorithm to find the shortest paths from a given vertex in an unweighted graph. It takes $\Theta(V + E)$ time.



```
BFS(V,Adj,s)
 level={s: 0}; parent = {s: None}; i=1
 frontier=[s]                        #previous level, i-1
 while frontier
     next=[]                         #next level, i
     for u in frontier
        for v in Adj[u]
           if v not in level         #not yet seen
                 level[v] = i        #level of u+1
                 parent[v] = u
                 next.append(v)
     frontier = next
     i += 1
```

DFS

DFS (depth first search) is an algorithm that explores an unweighted graph. DFS is useful for many other algorithms, including finding strongly connected components, topological sort, detecting cycles. DFS does not necessarily find shortest paths. It also runs in $\Theta(V + E)$ time.

- ***parent*** = {s: None}
- call ***DFS-visit*** (V, Adj, s)

```
def DFS-visit (V, Adj, u)
    for v in Adj[u]
        if v not in parent              #not yet seen
            parent[v] = u
            DFS-visit (V, Adj, v)           #recurse!
```

Edge Classification

We classify the edges in the resulting DFS tree as one of the following four types:

1. **Tree edge** - an edge that is traversed during the search.
2. **Back edge** - an edge (u, v) that goes from a node u to an ancestor of it in the DFS tree.
3. **Forward edge** - an edge (u, v) that goes from a node u to a descendant of it in the DFS tree.
4. **Cross edge** - any other edge in the original graph not classified as one of the above three types.

Selected Past Test Questions

You are at an airport in a foreign city and would like to choose a hotel that has the maximum number of shortest paths from the airport (so that you reduce the risk of getting lost). Suppose you are given a city map with unit distance between each pair of directly connected locations. Design an $O(V+E)$-time algorithm that finds the number of shortest paths between the airport (the source vertex s) and the hotel (the target vertex t).

If a topological sort exists for the vertices in a directed graph, then a DFS on the graph will produce no back edges.

Other Important Topics

We did not have time to cover all possible topics regarding Graphs/BFS/DFS at the review session. You should also review anything else in the lecture/recitation notes. For example:

- Beginning/Finishing times for DFS
- Topological sort
- BFS queue vs DFS stack
- Rubik's cube graph
- Proofs of correctness and runtime
- DAG's
- Connected components