References

Chapter 4: Graph (Section 4.2)

Chapter 22: Elementary Graph Algorithms
Unweighted Graphs

**Note**
Many code snippets here use C++ 11 features. Compile with the flag
`-std=c++11` if using g++.

Throughout, $n = \#\text{ vertices}$, $m = \#\text{ edges}$.
Unweighted Graphs

Note
Many code snippets here use C++ 11 features. Compile with the flag -std=c++11 if using g++.

Throughout, \( n = \# \) vertices, \( m = \# \) edges.

Adjacency List Representation of a Graph

```cpp
// without c++11 you may need to add a space between >>
typedef vector<vector<int>> graph;
...
graph g(n); // create a graph with n vertices
g[u].push_back(v); // add v as a neighbour of u
```

For undirected graphs, just add both directions of an edge \((u, v)\). Requires \( \Theta(n + m) \) space.
Depth-First Search

Find all vertices reachable from vertex $v$.

```
// the vertices that are reached in the search
vector<bool> reached(n, false);
graph g;

void dfs(int u) {
    if (!reached[u]) {
        reached[u] = true;
        for (auto w : g[u]) dfs(w);
    }
}
...
dfs(v);
```
Depth-First Search

Find all vertices reachable from vertex \( v \).

```c++
// the vertices that are reached in the search
vector<bool> reached(n, false);

graph g;

void dfs(int u) {
    if (!reached[u]) {
        reached[u] = true;
        for (auto w : g[u]) dfs(w);
    }
}
```

If we record the vertex that discovered \( u \), we can reconstruct paths.

Runs in \( O(n + m) \) time.
Depth-First Search

Example from CLRS (page 542, Figure 22.4)
Applications of DFS: Topological Sorting

Order the vertices so all edges point left-to-right.

\[ v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4 \rightarrow v_5 \rightarrow v_6 \rightarrow v_7 \rightarrow v_8 \]
Applications of DFS: Topological Sorting

Order the vertices so all edges point left-to-right.

Impossible to do if there is a cycle. Otherwise, the following works.
Applications of DFS: Topological Sorting

Order the vertices so all edges point left-to-right.

Impossible to do if there is a cycle. Otherwise, the following works.

- Begin a DFS. Just before returning from a recursive call (i.e. just after the for loop) push_back the vertex $u$ to the end of a vector.
Applications of DFS: **Topological Sorting**

Order the vertices so all edges point left-to-right.

Impossible to do if there is a cycle. Otherwise, the following works.

- Begin a DFS. Just before returning from a recursive call (i.e. just after the for loop) push_back the vertex $u$ to the end of a vector.
- Repeat, starting with an unvisited vertex each time, until all vertices are visited.
vector<int> order;  // initially empty

void topo_sort(int u) {
    if (!reached[u]) {
        reached[u] = true;
        for (auto w : g[u]) topo_sort(w);
        order.push_back(u);
    }
}

...  

for (int u = 0; u < n; u++)
    if (!reached[u])
        topo_sort(u);
reverse(order.begin(), order.end());  // #include <algorithm>
If $u$ is ordered after $w$ for some edge $(u, w)$, it must be that the recursive call with $w$ was on the call stack when $u$ was being processed. (Why?)
If $u$ is ordered after $w$ for some edge $(u, w)$, it must be that the recursive call with $w$ was on the call stack when $u$ was being processed. (Why?)

If $w$ is on the call stack when $u$ is being processed, there is a path from $w$ to $u$. Completing this path with the edge $(u, w)$ yields a cycle.
If \( u \) is ordered after \( w \) for some edge \((u, w)\), it must be that the recursive call with \( w \) was on the call stack when \( u \) was being processed. (Why?)

If \( w \) is on the call stack when \( u \) is being processed, there is a path from \( w \) to \( u \). Completing this path with the edge \((u, w)\) yields a cycle.

Thus

If the graph has no cycles, this will topologically sort all vertices.
Articulation Points & Bridges

An **articulation point** in an undirected, connected graph is a vertex whose removal leaves a disconnected graph.

A **bridge** is an edge whose removal leaves a disconnected graph.
Articulation Points & Bridges

An articulation point in an undirected, connected graph is a vertex whose removal leaves a disconnected graph.

A bridge is an edge whose removal leaves a disconnected graph.

Can find all bridges and articulation points in $O(n + m)$ time via DFS.
A bridge will always be a **tree edge** in a DFS (actually, in any spanning tree).

**Picture**: no edge of a descendent of $u$ in the search reached a non-descendent. So the parent edge of $u$ is a bridge.
Run a DFS, record the order the vertices were discovered.

Return the **earliest** discovery time of any vertex adjacent to a descendant of \( u \). This indicates if some descendant is adjacent to a non-descendant.

```cpp
vector<int> found(n, -1); // discovery time
int cnt = 0;

int bridges(int u, int p) {
    if (found[u] != -1) return found[u];
    int mn = found[u] = cnt++;
    // record u’s discovery time
    for (auto w : g[u])
        mn = min(mn, bridges(w, u));
    if (mn == found[u] && p != -2)
        // (p, u) is a bridge, process it how you want
        return mn;
    return mn;
}
...
bridges(0, -2); // start the search from any vertex
```
Other DFS Applications

- Find all articulation points in a graph (good exercise).
- Find the strongly connected components of a directed graph.
- Compute pre/post order traversals of a tree.
- Simple code for augmenting a bipartite matching (later lecture).

All of these can be implemented to run in $O(n + m)$ time.
Breadth-First Search

A breadth-first search will explore the vertices in increasing order of their shortest path distance from the start vertex.

- Load up the start vertex in a queue \( q \).
- While \( q \) is not empty, extract the front vertex and add all of its unvisited neighbours to the back of \( q \).
queue<int> q; // #include <queue>
vector<int> prev(n, -1);

q.push(v); // v is the start vertex in the search
prev[v] = -2; // signals "root of search"

while (!q.empty()) {
    int curr = q.front();
    q.pop();
    for (auto succ : g[curr])
        if (prev[succ] == -1) {
            prev[succ] = curr;
            q.push(succ);
        }
}

Now prev[u] for \( u \neq v \) is the vertex prior to \( u \) on a shortest \( v - u \) path.

Also runs in \( O(n + m) \) time.
A thick arrow from $u$ to $w$ indicates $\text{prev}[w] = u$.

The unique path using thick arrows from the start vertex (dark) to any vertex is a shortest path in the graph.
A thick arrow from $u$ to $w$ indicates $\text{prev}[w] = u$.

The unique path using thick arrows from the start vertex (dark) to any vertex is a shortest path in the graph.

Though we illustrated with an undirected graph, the same algorithm also finds shortest paths in directed graphs.
To Come...

Next week
Algorithms for weighted graphs.
• Dijkstra’s algorithm for shortest paths.
• Floyd-Warshall for all-pairs shortest paths.
• Bellmand-Ford: handling negative weight cycles.
• Minimum Spanning Trees: Kruskal’s Algorithm

Later in the course
• Bipartite matching: unweighted and weighted.
• Network flow: max-flow/min-cut.