Sources

Diestel's Graph theory: https://diestel-graph-theory.com
Erickson's Algorithms: https://jeffe.cs.illinois.edu/teaching/algorithms/

Introduction to graph algorithms

You may find it useful to read Chapters 0-4 of Erickson. We will start with a few basic examples of graph problems and algorithms as a warm-up. Some graph preliminaries:
A graph is a pair $G(V,E)$ where V is the set of *vertices* and $E \subseteq \binom{V}{2}$ or $E \subseteq V^2$ the set of (possibly directed) *edges*. For any vertex $v \in V(G)$ we let $N(v) = \{w \mid vw \in E(G)\}$ be the *neighborhood* of v, and $d(v) = |N(v)|$ the *degree* of v. Let $\Delta(G)$ and $\delta(G)$ be the maximum, respectively minimum, degree over all vertices of a graph G. A *subgraph* of G is a graph H such that $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. Note that we cannot take any subset of $V(G)$ and $E(G)$, as this is not necessarily a graph.

Paths and cycles

A *path* is a graph of the form $P(V,E)$ with $V = \{v_0, \dots, v_k\}$ and $E = \{v_0v_1, \dots, v_{k-1}v_k\}$. A *cycle* is a graph of the form $C(V,E)$ with $V = \{v_0, \dots, v_k\}$ and $E = \{v_0v_1, \dots, v_{k-1}v_k, v_1v_k\}$. We say that a graph is *connected* if there exists a path in the graph between any pair of vertices. We let $d(v,w)$ for $v, w \in V(G)$ denote the *distance* between vertices v and w in G, i.e. the length of a shortest v,w-path in G. If no such path exists, we let $d(v,w) = \infty$. In class, we will prove the following Proposition.

Proposition 1. *Every graph G contains a path of length at least $\delta(G)$ and a cycle of length at least $\delta(G)+1$.* D. 1.3.1

Trees

A *tree* is an acyclic connected graph. The following Theorem summarizes a set of equivalent definitions of trees, which are each useful in different contexts.

Theorem 2. *The following are equivalent for a graph T:* D. 1.5.1

(i) T is a tree;

(ii) any two vertices are linked by a unique *path in T;*

(iii) T is minimally connected ($T - e$ is disconnected for every $e \in E(T)$);

(iv) T is maximally acyclic ($T + e$ has a cycle for every $e \in \overline{E(T)}$).

(v) T is a connected graph on n vertices and $n-1$ edges, for $n \in \mathbb{N}$.

We say that a subgraph H of G is *spanning* if $V(H) = V(G)$, and if $T \subseteq G$ is both spanning and a tree it is called (naturally) a *spanning tree*. Spanning trees will prove very useful as they provide a sort of skeleton of G: a minimal set of edges that can reach all vertices. We will shortly see that all connected graphs have spanning trees.

Breadth-first search and depth-first search

Searching a connected graph may be done for various purposes, such as constructing a spanning tree, checking or keeping track of all vertices, finding connected components in a graph, etc...
Given a root vertex v of the graph, both a BFS and DFS systematically search through the graph. Each newly discovered vertex is added to a set, where it waits to have its neighbors explored in turn. The difference is in the order in which discovered vertices are explored. A BFS adds vertices to a queue (first in, first out), while a DFS adds them to a stack (last in, first out). We write the pseudocode for the BFS. Replacing the queue with a stack turns this into DFS.

Algorithm 1 Breadth-First-Search

```
1:  for $w \in V$ do
2:      $Marked[w] \leftarrow False$
3:  end for
4:  $Q :=$ a queue data structure
5:  $Q.enqueue\ v$
6:  while $Q \neq \emptyset$ do
7:      $u \leftarrow Q.dequeue$
8:      $Marked[u] \leftarrow True$
9:      for $w \in N(u)$ do
10:         if not $Marked[w]$ then
11:             $Q$.enqueue $w$
12:         end if
13:     end for
14: end while
```

Here is an example of a proof of correctness of the BFS algorithm.

Proposition 3. *If a graph G is connected, then the BFS marks all vertices.*

Proof. Suppose that this is not the case, and there are still unmarked vertices after the BFS algorithm has terminated. Suppose that w is an unmarked vertex. The root vertex v enqueued at the start and is therefore marked in the the first while loop. Since, G is connected, it contains a v, w-path. Since this path starts with a marked vertex and ends with an unmarked vertex, it must contain a marked vertex that is followed by an unmarked vertex. Call these vertices s and t. Since s is marked and t is in $N(s)$, t must have been enqueued in line 11. However if it was enqueued the while loop could not have ended without dequeueing and marking t, and therefore we have reached a contradiction. □

Exercise 1. *Suppose that we create a spanning tree T using a BFS starting at some vertex v in a graph G. For example, let v be the root, and when a vertex w is marked in line 11 of the algorith, we record its predecessor u and add the edge wu to T. Show that the distance $d(v, w)$ in T is equal to the distance $d(v, w)$ in G, for any $w \in V$. Note that distances between other pairs (two vertices not involving v) is not necessarily preserved.*

Exercise 2. (Erickson 5.8) *Let G be a connected graph, and let T be a depth-first spanning tree of G rooted at some node v. Prove that if T is also a breadth-first spanning tree of G rooted at v, then $G = T$.*

Bipartite graphs

A *bipartite graph* is a graph $G(V, E)$ if there exists a bipartition $V = V_1 \cup V_2$ (with $V_1 \cap V_2 = \emptyset$) such that every edge has one endpoint in V_1 and one endpoint in V_2. We discuss the following equivalent definition and its proof in class, as well as an algorithm that determines whether a graph is bipartite.

Proposition 4. *A graph is bipartite if and only if it contains no odd cycles.* D. 1.6.1,

Exercise 3. *Let $V = \{0, 1\}^d$. In other words, V is the set of all binary strings of length d. The d-dimensional hypercube Q_d is the graph on V such that two vertices in V share an edge if and only if the strings differ in exactly one bit. Show that the hypercube Q_d is a bipartite graph, for $d = 1, 2, \ldots$*

Weighted graphs

We now turn out attention to (edge-)weighted graphs. For many applications it is practical to consider only positive weights on edges (and consider a 0 weight a non-edge), although there may be times where negative weights are useful. For now we will consider only positive weights.

Dijkstra's algorithm

As we discovered in the previous lecture, for any root $v \in V(G)$ the BFS search algorithm finds the distance (and instances of shortest paths) between v and any $w \in V(G)$. That is, as long as G is unweighted and every edges is considered to have weight 1. Suppose that edges instead do not have the same length (or weight or cost). Now, we change the meaning of a shortest path to mean the path of least total weight, and our algorithm needs to be a little smarter. The classic version of this solution is called Dijkstra's Algorithm, after Edsger W. Dijkstra who proposed it in the 50s.

Algorithm 2 Dijkstra's Algorithm

```
 1: for $w \in V$ do
 2:     $dist[w] \leftarrow \infty$
 3: end for
 4: $dist[v] \leftarrow 0$
 5: $S \leftarrow \emptyset$
 6: add $v$ to $S$
 7: while $S \neq \emptyset$ do
 8:     let $u$ be vertex of minimum $dist$
 9:     $S.remove(u)$
10:     for $w \in N(u)$ do
11:         $d(w) = dist(u) + w(uw)$
12:         if $d(w) < dist(w)$ then
13:             $dist(w) \leftarrow d(w)$
14:             $prev(w) = u$
15:         end if
16:     end for
17: end while
```

In class, we will discuss a proof of the correctness of Dijkstra's algorithm.

Minimum Spanning Tree: Kruskal's and Prim's algorithm

In unweighted connected graphs, all spanning trees have the same number of edges. Similarly to the previous section, we can now ask: if a graph is edge-weighted, can we find a spanning tree of minimum total weight? When discussing algorithms for finding spanning trees in the unweighted case, two options were brought up: one along the lines of a BFS or DFS, and one greedy algorithm that either deletes edges that lie on cycles or starts with an empty spanning subgraph and adds edged one by one as long as they do not create a cycle. It turns out that for the weighted case, this greedy approach also works. This is called Kruskal's algorithm, after Joseph Kruskal, who proposed it in the 50s.

Kruskal's algorithm is simple: start with an empty T in a weighted, connected graph G, while there are edges in G that can be added to T without creating a cycle, add the edge of least weight among them. (This is easily adapted to find minimum spanning forests in possibly disconnected graphs.) We will prove the correctness of this algorithm in class.

Alternatively, we can try a greedy version along the lines of BFS/DFS/Dijkstra. Prim's algorithm was (re)discovered by Robert C. Prim and Edsger W. Dijkstra in the 50s, although originally attributed to Vojtěch Jarník in the 30s. It starts with a root vertex v as the initial tree T, and then adds the edge of lowest weight that connects a vertex from T to a vertex not in T, until T spans all vertices.

Exercise 4. *Suppose instead that we try Kruskal in the reverse direction. Start with $T = G$, and while there are edges in T in cycles, delete such an edge of greatest weight. Does this also result in a minimum spanning tree?*

Exercise 5. *Do Dijkstra's and/or Kruskal's algorithm work if we allow negative weights?*