Contents

1 Graph Theory **2**

1.1 Basic Definitions for Graphs . . . 2
1.2 Special Types of Graphs . . . 3
1.3 Matchings . . . 5
1.3.1 Definition . . . 5
1.3.2 Matchings in Bipartite Graphs . . . 5
1.4 Hall's Theorem . . . 7
1.5 Graph Coloring . . . 10
1.5.1 Scheduling Final Exams . . . 10
1.6 Connectivity and Cycles . . . 12
1.6.1 Walks, paths, and connected components . . . 12
1.7 Directed Graphs . . . 13
1.8 Acyclic Graphs . . . 14
1.8.1 Forests and Trees . . . 14
1.8.2 Spanning Trees . . . 15
1.8.3 Directed acyclic graphs (DAGs) . . . 18
1.9 Connectivity in Directed Graphs . . . 19
1.9.1 Structure Of Directed Graphs . . . 21
1.10 Searching in a Graph . . . 22
1.10.1 Depth-First Search . . . 22
1.10.2 Breadth-First Search . . . 25

Chapter 1

Graph Theory

1.1 Basic Definitions for Graphs

A graph $G = (V, E)$ is an ordered pair of sets, where V denotes the set of vertices, and E the set of edges. For our purposes, V is a finite non-empty set, and $E \subseteq V^{(2)}$, where $V^{(2)}$ is the set of 2-element subsets of V. It is common notation to let $|V| = n$ and $|E| = m$.

Example 1: See Figure 1.1.
$V = \{1, 2, 3\}$
$V^{(2)} = \{\{1, 2\}, \{1, 3\}, \{2, 3\}\}$
$E = \{\{1, 2\}, \{1, 3\}\}$

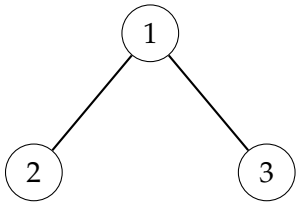

Figure 1.1: Graph in Example 1

We say that the two vertices an edge connect are the "endpoints" of the edge. The edge is said to be incident on its two endpoints. Graphs defined as above are said to be undirected, and can be used to represent symmetric relationships. For example an undirected graph could represent the relations 'is a friend of' or 'was born in the same month as'.

Definition 1. *The* degree *of a vertex $v \in V$, denoted $d(v)$, is the number of edges incident on v.*

We have the following simple lemma characterizing the relationship between the total sum of degrees in a graph and the number of edges.

Lemma 1. *The sum of degrees of vertices in a graph is twice the number of edges, i.e.,*

$$\sum_{v \in V} d(v) = 2m.$$

Proof. We will prove Lemma 1 by induction. We will induct on the number of edges in the graph. For non-negative m, we will prove the lemma holds for any graph with m edges.
Base Case: If a graph $G = (V, E)$ has no edges ($E = \emptyset$), then every vertex has degree 0. Thus, $\sum_{v \in V} d(v) = 0 = 2(0)$.
Inductive Case: For our inductive hypothesis, we assume the claim holds for any graph with less than m edges. Then, we want to prove the claim for an arbitrary graph $G = (V, E)$ where $|E| = m$.

Fix an arbitrary edge of G and remove it. Let this edge be $e = \{a, b\}$. Let $G' = (V, E')$ where $E' = E \setminus \{e\}$. Then, G' is a graph with $m - 1$ edges. Let $d'(v)$ be the degree of $v \in V$ in G'. By our inductive hypothesis,

$$\sum_{v \in V} d'(v) = 2(m-1).$$

For all $v \in V \setminus \{a, b\}$, $d'(v) = d(v)$. Additionally, $d(a) = 1 + d'(a)$ and $d(b) = 1 + d'(b)$ since a, b are endpoints of edge e, and so are incident on one more edge in G than in G'. Thus,

$$\sum_{v \in V} d(v) = 2 + \sum_{v \in V} d'(v) = 2 + 2(m-1) = 2m.$$

Therefore, for any graph $G = (V, E)$ the theorem holds. □

We now introduce the concept of subgraphs. This is similar to the idea of subsets, but for graphs instead of sets.

Definition 2. *A* subgraph $G' = (V', E')$ *of graph* $G = (V, E)$ *satisfies the property that* $V' \subseteq V$ *and* $E' \subseteq E$.

See Figure 1.2 and Figure 1.3 for an example of a subgraph.

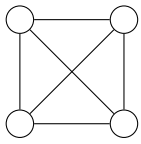

Figure 1.2: A graph

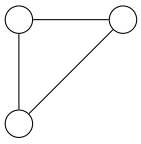

Figure 1.3: A subgraph of the graph in Figure 1.2.

1.2 Special Types of Graphs

Many categories of graphs arise frequently in both practice and theory, and have special names.

- **Empty Graph**: The empty graph is a graph with no edges, i.e. $E = \emptyset$. See Figure 1.4 for an example.
- **Complete Graph**: The complete graph is a graph with all possible edges, i.e. $E = V^{(2)}$. See Figure 1.5 for an example.
- **Cycle Graph**: A cycle graph must have at least three vertices. Suppose graph G had vertices $V = \{1, 2, \ldots, n\}$. Then G is a cycle graph if $E = \{\{1, 2\}, \{2, 3\}, \ldots \{n-1, n\}, \{n, 1\}\}$. Notice that $|E| = n$. See Figure 1.6 for an example.

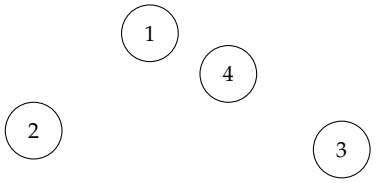

Figure 1.4: An empty graph with 4 vertices.

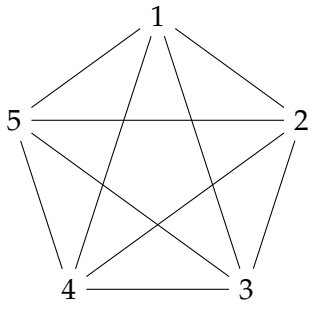

Figure 1.5: A complete graph with 5 vertices

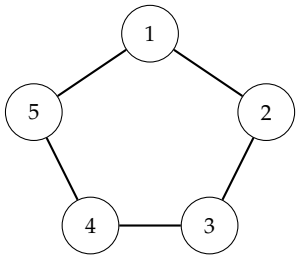

Figure 1.6: An cycle graph with 5 vertices.

Another graph that arises frequently in practice are bipartite graphs. For example, suppose we have a group of mentors and mentees. We can model this situation using a bipartite graph. Edges represent the willingness of a pair to have a mentor/mentee relationship. There will be no edges within the set of mentors or within the set of mentees, since a mentor does not need to be mentored and a mentee cannot be a mentor. We have the following property of bipartite graphs.

Definition 3. *A* bipartite *graph is a graph $G = (V, E)$ in which V can be partitioned into two sets A, B such that $E \subseteq \{\{x, y\} : x \in A, y \in B\}$.*

Lemma 2. *For a bipartite graph $G = (V, E)$ where $V = A \cup B$ and $|E| = m$,*

$$\sum_{v \in A} d(v) = \sum_{v \in B} d(v) = m$$

Proof. Let $G = (V, E)$ be an arbitrary bipartite graph where $V = A \cup B$. Each vertex is either in A or B, but not both since A and B are disjoint. Thus,

$$\sum_{v \in V} d(v) = \sum_{v \in A} d(v) + \sum_{v \in B} d(v) = 2m,$$

where the second equality holds by Lemma 1. Every edge has one endpoint in A and one endpoint in B, so $\sum_{v \in A} d(v) = \sum_{v \in B} d(v)$. Thus, $2m = 2\sum_{v \in A} d(v)$. This implies

$$\sum_{v \in A} d(v) = \sum_{v \in B} d(v) = m.$$ □

Lemma 3. *A graph is bipartite if and only if it does not have any odd cycle.*

Proof. We first prove the forward direction. Let X, Y denote the bipartition of the vertices, and let C be any cycle in the graph (if one exists). Let x be a vertex in X on C, and suppose we follow C starting at x. In each step, we alternate between a vertex in X and a vertex in Y. Therefore, to return to x, we must traverse an even number of steps, so C is an even cycle.

Now we prove the other direction by proving the contrapositive: if a graph is not bipartite, then there exists an odd cycle. Let us partition the graph in the following way: starting with any vertex v, put v into a set X. Put all neighbors of v into set Y. Then put all neighbors of the vertices in Y, which haven't been put into set X, into X. Keep adding new vertices into set X and Y until all vertices have been added. At some point, there must exist a vertex with one neighbor in X and another in Y, for otherwise the graph is bipartite. This vertex, along with the starting vertex v and the paths leaving v to the two neighbors, is part of an odd cycle. □

1.3 Matchings

1.3.1 Definition

Definition 4. *A* matching *is a subset of edges such that no two edges share a vertex.*

Example : In Figure 1.7, $\{a, i\}$ is a matching, but $\{b, e\}$ is not a matching.

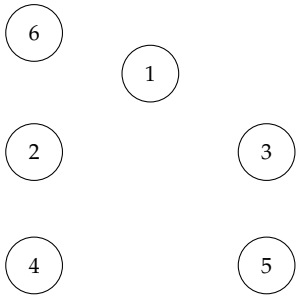

Figure 1.7: Matching Example

Lemma 4. *In a graph with n vertices, a matching cannot have size larger than* $\lfloor n/2 \rfloor$.

Proof. Adding one edge into the matching would remove two vertices along with all edges incident to them from entering the matching. Such process can only repeat $\lfloor n/2 \rfloor$ times because we only have n vertices. □

Note that not every graph has a matching of size $\lfloor n/2 \rfloor$. Consider a star graph where there is a vertex with degree $n - 1$ and all other vertices with degree 1. The maximum matching of this graph is of size 1.

1.3.2 Matchings in Bipartite Graphs

Mathchings are particularly interesting in bipartite graphs. Each semester the department needs to consider which instructor is going to teach which class in the next semester. Each node in X represents a class and each node in Y represents an instructor. Several instructors are qualified to teach each class, and each instructor is qualified to teach several classes. Now suppose a class is only taught by one instructor, and one instructor only taught one class. The graph is clearly a bipartite graph, and the department is looking for a matching that pairs up class and its instructor.

Definition 5. *A* perfect matching *is a bipartite graph* $G = (V, E)$ *such that for every vertex* $v \in V$, $d(v) = 1$.

Figure 1.9, 1.10 show two perfect matchings of the bipartite graph in Figure 1.20. In the case of matching mentors to mentees, we are asking whether there exists a perfect matching which is a subgraph of the original bipartite graph. A perfect matching may not always exist, see Figure 1.11. For a perfect matching to exist in graph $G = (V = A \cup B, E)$, it must be that $|A| = |B|$. This is a necessary condition, but not sufficient. A famous result known as Hall's Marriage Theorem gives necessary and sufficient conditions for the existence of a perfect matching in a bipartite graph. To state this theorem precisely, we need to introduce the notion of an A-perfect matching.

Definition 6. *An* A-perfect matching *is a bipartite graph* $G = (V, E)$ *with* $V = A \cup B$ *and* $E \subseteq \{(u, v) : u \in A, v \in B\}$ *such that for every* $u \in A$, $d(u) = 1$, *and for every* $v \in B$, $d(v) \leq 1$.

We also need the definition of the neighborhood of A set. This is simply the set of vertices that are connected to the vertices of the set by edges in the graph.

Definition 7. *In a bipartite graph* $G = (V, E)$ *with* $V = A \cup B$ *and* $E \subseteq \{(u, v) : u \in A, v \in B\}$, *the* neighborhood *of a set of vertices* $S \subseteq A$ *is defined as* $N(S) = \{v \in B : \exists u \in S. \{u, v\} \in E\}$.

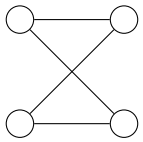

Figure 1.8: A bipartite graph.

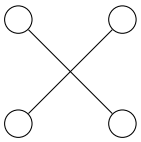

Figure 1.9: One perfect matching.

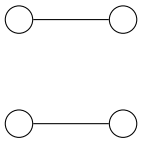

Figure 1.10: Another perfect matching.

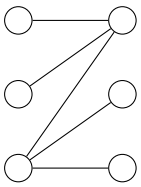

Figure 1.11: Bipartite graph with no perfect matching.

Theorem 5 (Hall's Marriage Theorem (1935))**.** *A bipartite graph* $G = (A \cup B, E)$ *has an* A*-perfect matching if and only if for any subset* S *of* A $(S \subseteq A)$, *the total number of vertices in the neighborhood of* S *is at least the number of vertices in* S, *i.e.,*

$$\forall S \subseteq A.\ |N(S)| \geq |S|.$$

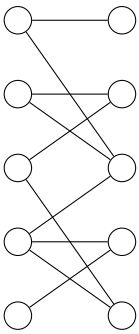

Figure 1.12: A bipartite graph.

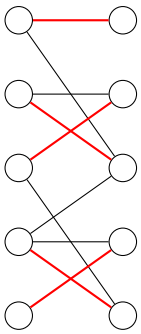

Figure 1.13: A perfect matching of the graph in Figure 1.16 highlighted in red.

1.4 Hall's Theorem

In this section, we re-state and prove Hall's theorem. Recall that in a bipartite graph $G = (A \cup B, E)$, an A-perfect matching is a subset of E that matches every vertex of A to exactly one vertex of B, and doesn't match any vertex of B more than once.

Theorem 6 (Hall 1935)**.** *A bipartite graph $G = (A \cup B, E)$ has an A-perfect matching if and only if the following condition holds:*

$$\forall S \subseteq A. |N(S)| \geq |S|,$$

where $N(S) = \{v \in B : \exists u \in S. \{u, v\} \in E.\}$.

Remark 1: If $|A| = |B|$, then a matching is A-perfect if and only if it is perfect. So, in this case, Hall's theorem tells us when a *perfect* matching exists. On the other hand, if $|A| \neq |B|$, then G cannot contain a perfect matching because every edge of a matching pairs one vertex of A with exactly one vertex of B. However, if $|A| < |B|$, then G may still contain an A-perfect matching.

Remark 2: Theorem 6 is of the form $P \leftrightarrow Q$, where P is the proposition "G has an A-perfect matching" and Q is known as *Hall's condition*. In general, $P \rightarrow Q$ states that Q is a *necessary* condition for P. (To see why, consider the contrapositive $\neg Q \rightarrow \neg P$.) Furthermore, $Q \rightarrow P$ states that Q is a *sufficient* condition for P. Thus, Hall's theorem states that Hall's condition is a necessary and sufficient condition for a bipartite graph to have an A-perfect matching.

Proof. We now begin the proof of Theorem 6.

Hall's condition is necessary: Assume that G has an A-perfect matching, which we denote by M. Let S be an arbitrary subset of A. Since M is an A-perfect matching, M matches every vertex of S to exactly one vertex of B, and no vertex of B is matched more than once. So if we restrict G to the edges in M, the vertices of S each have a distinct neighbor in $N(S)$. Since $N(S)$ is defined using *all* the edges of G and M is only a *subset* of E, this implies $|N(S)| \geq |S|$.

Hall's condition is sufficient: We will construct an A-perfect matching M by proceeding with induction on $|A|$, assuming G satisfies Hall's condition.

Base case: $|A| = 1$**.** Let a denote the sole vertex of A. Hall's condition tells us $|N(\{a\})| \geq 1$, which means a has at least one neighbor. We can set $M = \{\{a, b\}\}$ where b is any neighbor of a. Then, M is an A-perfect matching: every vertex of A is matched, and no vertex of B is matched more than once.

Inductive hypothesis (IH): Assume that for all k such that $1 \leq k \leq |A| - 1$, any bipartite graph $H = (C \cup D, F)$ satisfying $|C| = k$ has a C-perfect matching if and only if H satisfies Hall's condition on C, i.e., $\forall S \subseteq C. |N(S)| \geq |S|$.

Inductive step: We will now construct an A-perfect matching M in G, starting with $M = \emptyset$. Note that when G satisfies Hall's condition, there are two possible cases: the inequality is strict for every S that is a strict subset of A (i.e., $\forall S \subset A. |N(S)| > |S|$), or there exists at least one $S \subset A$ such that $|N(S)| = |S|$.

(i) In the first case, since $|N(S)|$ and $|S|$ are integers, we can assume

$$\forall S \subset A. |N(S)| \geq |S| + 1. \tag{1.1}$$

We claim that the following procedure returns a perfect matching:

1. Let u be an arbitrary vertex of A, and add *any* edge $e = \{u, v\}$ of E to M.
2. Remove u, v, and all edges incident to u or v from G to construct graph G'.
3. By the IH, G' has a matching M' that is $(A \setminus \{u\})$-perfect.
4. Add the edges of M' to M, and return M.

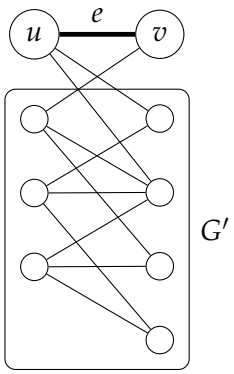

Figure 1.14: The original graph G with an arbitrary edge $\{u, v\}$ added to M (bold). The graph G' comprises the remaining vertices.

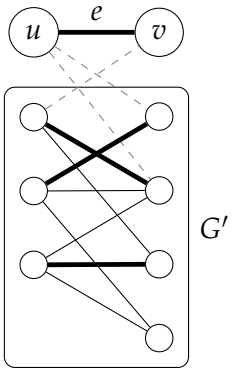

Figure 1.15: Applying the IH on G' yields a matching M' that is $(A \setminus \{u\})$-perfect. The edges of the final matching M are in bold.

For this procedure to be correct, we have to show several properties. First, to show that Step 1 is valid, we need to establish that the degree of any vertex $u \in A$ is at least 1. If this does not hold, then $N(\{u\}) = 0$, while $|\{u\}| = 1$, thereby violating Hall's condition for $S = \{u\}$.

Next, we show that G' satisfies Hall's condition so that Step 3 is valid. Let S' be any subset of $A \setminus \{u\}$, and let $N'(S')$ denote its neighbors in G' (so $N'(S') \subseteq B \setminus \{v\}$). Notice that $|N(S')| - 1 \geq |S'|$ because of (1.1). Since we only removed one vertex of B to construct G', $|N'(S')| \geq |N(S')| - 1$. Taken together, these inequalities imply $|N'(S')| \geq |S'|$, as desired.

Now we must show that $M = M' \cup \{\{u, v\}\}$ is an A-perfect matching. Since M' matches every vertex of $A \setminus \{u\}$ and e matches u, every vertex of A is indeed matched by M. Furthermore, vertex v is matched once because G' excludes v, and the vertices of $B \setminus \{v\}$ are matched at most once because M' is a matching. Thus, M is an A-perfect matching.

(ii) In the second case, we assume there exists $S \subset A$ such that $|N(S)| = |S|$. We claim that the following procedure returns a perfect matching:

1. Partition A into S and $\overline{S} = A \setminus S$ and B into $N(S)$ and $\overline{N(S)} = B \setminus N(S)$.
2. Let $G_1 = (S \cup N(S), E_1)$ where E_1 denotes the edges of G among $S \cup N(S)$. By the IH, G_1 has a matching M_1 that is S-perfect.
3. Let $G_2 = (\overline{S} \cup \overline{N(S)}, E_2)$, where E_2 denotes the edges of G among $\overline{S} \cup \overline{N(S)}$. By the IH, G_2 has a matching M_2 that is $\overline{S}$-perfect.
4. Let $M = M_1 \cup M_2$, and return M.

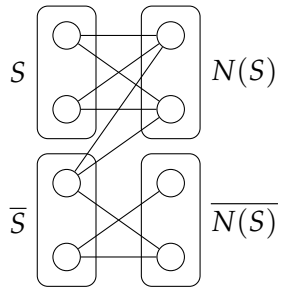

Figure 1.16: Partitioning A and B according to S and $N(S)$; notice that $|N(S)| = |S|$.

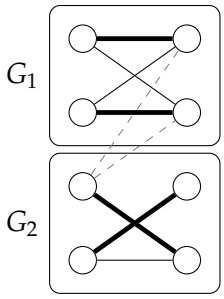

Figure 1.17: Applying the IH on G_1 and G_2 and returning the union of the two matchings.

As in the previous case, we must prove that G_1 and G_2 satisfy Hall's condition so that Step 2 and Step 3 are valid. The graph G_1 is easier to handle: observe that there are no edges from S to $\overline{N(S)}$. Thus, for any subset T of S, its neighborhood in G_1 is exactly the same as its neighborhood in G. Since G satisfies Hall's condition, this implies G_1 also satisfies Hall's condition.

Showing that G_2 also satisfies Hall's condition is slightly trickier. For contradiction, suppose G_2 violates Hall's condition. This means there exists $X \subseteq \overline{S}$ such that $|Y| < |X|$, where Y denotes the neighborhood of X in G_2, i.e., $Y = N(X) \cap \overline{N(S)}$. Now consider the set $S \cup X$. This is a subset of A, so since G satisfies Hall's condition, we know that

$$|N(S \cup X)| \geq |S \cup X|.$$

Furthermore, since S and X are disjoint, $|S \cup X| = |S| + |X|$. Also, notice that the only neighbors of $S \cup X$ contained in $\overline{N(S)}$ are the neighbors of X, i.e., $N(S \cup X) = N(S) \cup Y$. Finally, since $N(S)$ and Y are disjoint, we know that $|N(S) \cup Y| = |N(S)| + |Y|$. Putting this all together, we get

$$\begin{aligned}
|N(S \cup X)| &= |N(S) \cup Y| && \text{(definitions of } S, X, Y\text{)} \\
&= |N(S)| + |Y| && (N(S) \text{ and } Y \text{ are disjoint}) \\
&= |S| + |Y| && \text{(defining property of } S\text{)} \\
&< |S| + |X| && \text{(defining property of } X\text{)} \\
&= |S \cup X|. && (S \text{ and } X \text{ are disjoint})
\end{aligned}$$

Thus, the set $S \cup X$ violates Hall's condition in the original graph G because $|N(S \cup X)| < |S \cup X|$. This concludes the proof that G_2 satisfies Hall's condition.

Now that we know G_1 and G_2 satisfy Hall's condition, we must show that $M = M_1 \cup M_2$ is an A-perfect matching. Since M_1 is S-perfect and M_2 is $\overline{S}$-perfect, we know that M matches every vertex of $S \cup \overline{S} = A$. Furthermore, no edges of M_1 and M_2 share an endpoint because M_1 and M_2 were obtained from two disjoint graphs G_1 and G_2. Thus, no vertex of B is matched twice by M, so M is an A-perfect matching. □

1.5 Graph Coloring

1.5.1 Scheduling Final Exams

Every semester, the registrar must schedule final exams in a way so that no student must take two exams at the same time. We can begin modeling this situation by constructing a graph $G = (V, E)$. In this case, the vertex set V corresponds to the set of all classes (one vertex per class). The set E is defined as follows: there is an edge between u and v if there exists at least one student taking both class u and class v. The goal is to assign each vertex to a time slot so that the two endpoints of every edge are assigned to different slots.

The simplest solution is to assign each final exam to a unique time slot. If there are n classes, this results in n time slots, but this may lead to an extremely long final exam period. So we want a solution that is not only feasible, but also minimizes the number of total time slots.

One way to visualize the notion of "assigning a time slot" is to consider each time slot as a color. In other words, given G, we must color every vertex of G so that the two endpoints of every edge receive different colors. Fig. 1.18 gives an example of coloring a graph with 5 vertices using 3 colors, and Fig. 1.19 colors the same graph using only two colors.

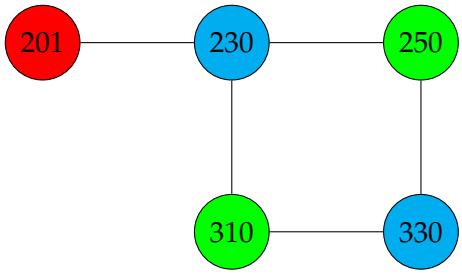

Figure 1.18: A graph G with 5 vertices corresponding to 5 classes. Each of the three colors represents a distinct time slot.

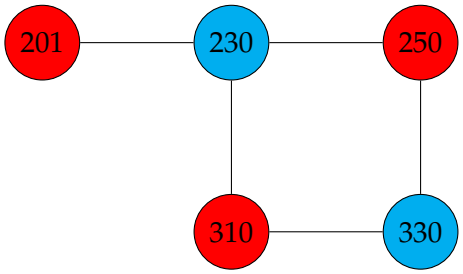

Figure 1.19: The same graph from Fig. 1.18, colored using only two colors.

Definition 8. *A* valid coloring *is a function $f : V \to C$, where C is the set of colors, such that $\forall (u, v) \in V, f(u) \neq f(v)$.*

Definition 9. *Let k be a positive integer. A graph is* k-colorable *if it is possible to assign each vertex to one of k colors such that the two endpoints of every edge are assigned different colors.*

The graph shown in Fig. 1.19 is 2-colorable, since every edge has a red endpoint and a blue endpoint. Notice that Fig. 1.18 shows that the same graph is 3-colorable—in general, if a graph is k-colorable, then it is also ℓ-colorable for any $\ell \geq k$. We will now prove a simple observation regarding graphs that are 2-colorable.

Lemma 7. *Let G be a graph. Then G is 2-colorable if and only if G is bipartite.*

Proof. Let G be a 2-colorable graph, which means we can color every vertex either red or blue, and no edge will have both endpoints colored the same color. Let A denote the subset of vertices colored red, and let B denote the subset of vertices colored blue. Since all vertices of A are red, there are no edges within A, and similarly for B. This implies that every edge has one endpoint in A and the other in B, which means G is bipartite.

Conversely, suppose G is bipartite, that is, we can partition the vertices into two subsets V_1, V_2 every edge has one endpoint in V_1 and the other in V_2. Then coloring every vertex of V_1 red and every vertex of V_2 blue yields a valid coloring, so G is 2-colorable. □

Thus, Observation 7 tells us that the graph in Fig. 1.19 is bipartite. Indeed, by observing Fig. 1.20, it becomes even clearer that this graph is bipartite.

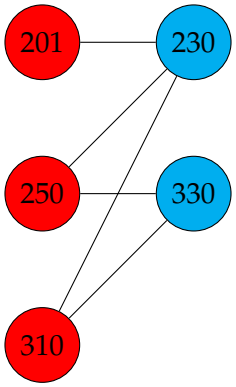

Figure 1.20: The same graph and coloring from Fig. 1.19, with the vertices both colored and rearranged to further illustrate that it's bipartite.

Coloring Planar Graphs: Finally, we restrict our attention to coloring planar graphs (defined later). Consider a map of the countries of Europe, and suppose we want to color the countries such that any two countries that share a border are colored different colors. For example, France and Germany should be assigned different colors, but France and Sweden can be colored the same color.

We can represent this problem as a graph coloring problem: the graph contains a vertex for every country, and an edge exists between two vertices if their corresponding countries share a border. Thus, coloring the regions in the map corresponds to coloring the vertices of the graph, and no two adjacent regions (vertices) should be colored the same color.

A graph is *planar* if it can be drawn on a plane (e.g., on a sheet of paper) such that no two edges cross each other. For example, the graph in Fig. 1.20 is planar, and the drawing in Fig. 1.19 illustrates this. It's intuitively clear that every map can be represented as a planar graph. The theorem below states that when coloring a map, or any planar graph, we only need 4 colors.

Theorem 8 (Four Color Theorem). *If G is a planar graph, then G is 4-colorable.*

The proof of Theorem 8 is beyond the scope of this course, so we omit it. We will note, however, that the theorem was originally conjectured in 1852. Multiple proofs were proposed, but all were shown to be flawed until 1976, when Appel and Haken announced a computer-assisted proof. Their proof involved a large number of cases that are too tedious to check by hand. Since their proof was published, the last 40 years has seen several more "verifiable" proofs of this theorem, where the number of graphs that need to be checked manually has significantly reduced over time.

1.6 Connectivity and Cycles

Now we will study a different property of graphs known as connectivity, and we will begin with some basic definitions. Throughout this section, we let $G = (V, E)$ be a graph.

1.6.1 Walks, paths, and connected components

We now formalize the notion of traversing edges in a graph. Intuitively, we can think of the following definitions as capturing a time-dependent process of moving along the edges of a graph.

Definition 10. *A* walk *in G is a sequence of vertices such that there exists an edge between every two consecutive vertices in the sequence.*

Definition 11. *A* path *in G is a walk that does not repeat any vertices.*

Remark: If a walk does not repeat vertices, then it also does not repeat edges. To see this, suppose the edge $\{u, v\}$ is traversed more than once in a walk. Then u and v each appear at least twice, so the walk also repeats vertices.

Furthermore, we can show the following: if there is a walk from s to t, then there is a path from s to t. Let w be a walk from s to t; we will eliminate repeated vertices in w to create a path. If u is a vertex that appears multiple times in w, then consider removing all vertices between the first and last appearance of u. The resulting sequence is still a walk, and this process can be repeated until there are no repeated vertices.

With these definitions, we can formalize the definition of connectivity in graphs. Intuitively, a graph is connected if it contains one "piece," but as we shall see, this can be captured through the use of an appropriately defined equivalence relation on V.

Definition 12. *Two vertices u and v are* connected *if there exists a path that starts at u and ends at v.*

Consider the relation R on V defined as follows: for all $u, v \in V$, the pair (u, v) is in R if and only if u and v are connected. It is straightforward to verify that R is an equivalence relation:

- Reflexive: every vertex u is connected to itself—simply take walk that starts at u and contains no edges. Thus, $(u, u) \in R$ for every $u \in V$.
- Symmetric: if there's a path from u to v, then that same sequence of vertices in reverse gives a path from v to u. Thus, if $(u, v) \in R$, then $(v, u) \in R$.
- Transitive: if there's a path p_1 from u to v and a path p_2 from v to w, then we can join these two paths at v to create a walk from u to w. As we discussed earlier, this walk can be used to obtain a path from u to w. This implies R is transitive.

Now recall that every equivalence relation induces a partition on the underlying set, and the blocks of the partition are the equivalence classes of the relation. Let us consider the partition that R induces on V: let u be any vertex, and let $[u] = \{v : (u, v) \in R\}$ be the equivalence class of u. Then $[u]$ contains all the vertices reachable from u; this is known as the *connected component* (or simply *component*) of u. In general, each equivalence class is known as a connected component (or component), and together, they are known as the connected components (or components) of G.

Definition 13. *A graph is* connected *if every pair of vertices is connected.*

Thus, an equivalent definition of G being connected is that G contains exactly one connected component. In this case, the partition induced by R only has one block, which is equal to V.

1.7 Directed Graphs

So far, we have worked with simple undirected graphs. Now we will introduce a new type of graph: directed graphs, also called digraphs. As usual, $G = (V, E)$ and V is a finite, non-empty set of vertices. Now, our edges will have directions. So, if $(u, v) \in E$, the edge is directed from u to v. We will assume that our graphs have no self-loops, so edges of the form (u, u) will not be included in E. Thus, $E \subseteq V \times V \setminus \{(v, v) : v \in V\}$. As usual, we let $|E| = m$ and $|V| = n$. See Figure 1.21 for an example of a directed graph.

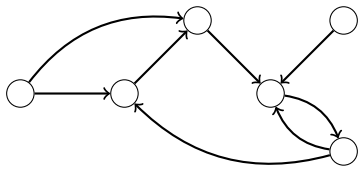

Figure 1.21: A directed graph.

Definition 14. *The* in-degree *of a vertex, denoted $d_{in}(v)$, is the number of incoming edges incident on a vertex:*

$$d_{in}(v) = |\{(u, v) \in E : u \in V\}|.$$

Definition 15. *The* out-degree *of a vertex, denoted $d_{out}(v)$, is the number of outgoing edges incident on a vertex:*

$$d_{out}(v) = |\{(v, u) \in E : u \in V\}|.$$

Lemma 9.

$$\sum_{v \in V} d_{in}(v) = \sum_{v \in V} d_{out}(v) = m$$

Proof. Intuitively, an edge (u, v) contributes one to the in-degree of vertex v and the out-degree of vertex u. We will prove this formally by induction on the number of edges.

Base case: If $m = 0$, then every vertex has in-degree and out-degree 0 and the lemma holds.

Inductive hypothesis: Let m be an arbitrary positive integer. For any graph $G' = (V', E')$ with $m' < m$ edges, assume $\sum_{v \in V'} d_{in}(v) = \sum_{v \in V'} d_{out}(v) = m'$.

Inductive step: Remove an arbitrary edge $e = (u, v)$ of G and remove it, let $G' = (V, E')$ denote the remaining graph with $m - 1$ edges, and let $d'(v)$ be the degree of $v \in V$ in G'. By our inductive hypothesis,

$$\sum_{v \in V} d'_{in}(v) = \sum_{v \in V} d'_{out}(v) = m - 1.$$

For all $v \in V \setminus \{u, v\}$, $d'_{out}(v) = d_{out}(v)$ and $d'_{in} = d_{in}(v)$. Additionally, the in-degree of u did not change, since we removed an outgoing edge from u. The out-degree of u decreased by 1: $d_{out}(u) = 1 + d'_{out}(u)$. Likewise, we removed an incoming edge from v, so $d_{in}(v) = 1 + d'_{in}(v)$. The out-degree of v did not change. Thus, we have

$$\sum_{v \in V} d_{out}(v) = 1 + \sum_{v \in V} d'_{out}(v) = 1 + m - 1 = m,$$

and similarly, we have

$$\sum_{v \in V} d_{in}(v) = 1 + \sum_{v \in V} d'_{in}(v) = 1 + m - 1 = m.$$

Thus, $\sum_{v \in V} d_{in}(v) = \sum_{v \in V} d_{out}(v) = m$, as desired. □

Paths and walks are defined identically in undirected and directed graphs. A *walk* in a digraph is a sequence of vertices such that there exists a directed edge from each vertex to the next vertex on the walk. A *path* is a walk that does not repeat vertices. Figure 1.22 shows an example of a directed path.

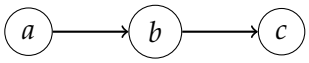

Figure 1.22: A valid directed path from a to c.

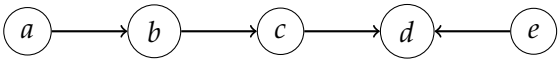

Figure 1.23: An example where there is no path from a to e.

Observe that paths may only go in one direction in directed graphs (a path may exist from u to v, but not from v to u). In Figure 1.23, see that there is no path from a to e.

1.8 Acyclic Graphs

In this section, we discuss general concepts related ot acyclic graphs, i.e., graphs that do not contain any cycles. We first consider the undirected case, and then the situation when the graph is directed.

1.8.1 Forests and Trees

Definition 16. *A graph is* acyclic *if it does not contain a cycle. Acyclic graphs are also called* forests.

Theorem 10. *An acyclic graph containing n vertices and c connected components has $m = n - c$ edges.*

Proof. We will induct on m.
Base Case: If $m = 0$, then each vertex is one component and thus $n = c$ which implies $0 = n - c$.
Inductive Hypothesis: Let m be an arbitrary positive integer. For any acyclic graph G' with $m' < m$ edges and c' connected components, the number of edges of G' is $m' = n - c'$.
Inductive Step: Let G be an arbitrary n-vertex graph with m edges. Remove an edge $e = (u, v)$ from G to create a graph G' on n vertices, $m - 1$ edges, and c' components. Since G is acyclic, removing e increases the number of connected components by 1, so $c = c' - 1$. By the inductive hypothesis, $m' = m - 1 = n - c'$. Thus, $m = n - c' + 1 = n - c$. □

Definition 17. *A* tree *is a connected acyclic graph.*

Lemma 11. *Any tree on n vertices has $n - 1$ edges.*

Proof. Since a tree has a single connected component, Theorem 10 implies $m = n - 1$. □

Lemma 12. *Any connected graph on n vertices contains $m \geq n - 1$ edges.*

Proof. For contradiction, suppose G is a connected graph with fewer than $n - 1$ edges. By Lemma 11, G is not a tree, so G must contain a cycle (but it is connected). It is possible to remove an edge from each cycle to create a tree, so the number of edges is at least $n - 1$, contradicting our initial assumption. □

Lemma 13. *Any graph on n vertices containing at least n edges has at least one cycle.*

Proof. Theorem 10 states that an acyclic graph satisfies has $m = n - c$. Since $c \geq 1$, an acyclic graph satisfies $m \leq n - 1$. Therefore, if this property is not satisfied (i.e., $m \geq n$), then the graph is not acyclic. □

The next theorem states that there are (at least) three equivalent definitions of tree: it suffices to pick any two "defining" properties, because together they imply the third.

Theorem 14. *Let G be a graph containing n vertices and m edges, and consider the following three properties.*

1. *G is connected.*
2. *G is acyclic.*
3. *$m = n - 1$.*

Then any two of these properties together imply the third.

Proof. $(\mathbf{1}) \wedge (\mathbf{2}) \rightarrow (\mathbf{3})$: This is Lemma 11.

$(\mathbf{1}) \wedge (\mathbf{3}) \rightarrow (\mathbf{2})$: For contradiction, suppose G is connected, satisfies $m = n - 1$, and contains a cycle. Removing an edge from this cycle yields a connected graph G' satisfying $m' = m - 1 = n - 2$. This contradicts Lemma 12.

$(\mathbf{2}) \wedge (\mathbf{3}) \rightarrow (\mathbf{1})$: Again, for contradiction, suppose G is acyclic, $m = n - 1$, and G is not connected. Consider any two connected components of G denoted by X and Y, and add an edge between any $u \in X$ and $v \in Y$. This yields an acyclic graph G' containing n vertices, n edges, and $c - 1 \geq 1$ components, contradicting Theorem 10. □

1.8.2 Spanning Trees

Definition 18. *A* spanning tree *of a connected graph $G = (V, E)$ is a subgraph of G whose vertex set is V and edges form a tree.*

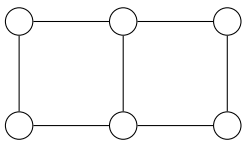

Figure 1.24: A graph G.

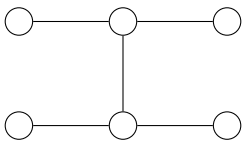

Figure 1.25: A spanning tree of G in Figure 1.24.

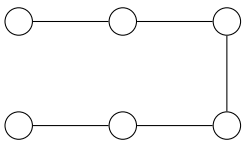

Figure 1.26: Another spanning tree of G in Figure 1.24.

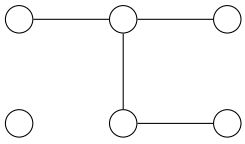

Figure 1.27: This subgraph does not connect all vertices of G in Figure 1.24.

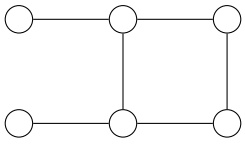

Figure 1.28: This subgraph of G in Figure 1.24 is not a tree.

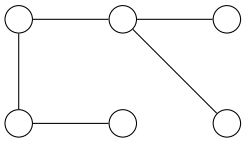

Figure 1.29: Not a valid subgraph of G in Figure 1.24.

Suppose we want to find a spanning tree of graph G in Figure 1.24. Both Figure 1.25 and Figure 1.26 are valid spanning trees of G. The definition specifies that a spanning tree must connect all vertices of G, the subgraph in Figure 1.27 does not satisfy this condition. The subgraph in Figure 1.28 is not a tree, as it contains a cycle. The graph in Figure 1.29 uses an edge that was not originally in E, so it is not a valid subgraph. All three conditions of Definition 18 must be satisfied to have a valid spanning tree.

Observe that if the graph is a tree, there is one unique spanning tree and it is the graph itself. Spanning trees are especially important for graphs whose edges are assigned numerical weights.

Definition 19. *A* weighted graph $G = (V, E)$ *is one in which each edge is associated with a real number* $w(e)$*, called its weight.*

Definition 20. *The* weight *of a graph* $G = (V, E)$ *is the sum of the weights of all its edges:*

$$w(G) = \sum_{e \in E} w(e).$$

Given a weighted graph, consider the task of finding a spanning tree with the minimum weight, also known as a *minimum spanning tree* (MST). To do this, we first consider coloring the graph and observing the edges.

Definition 21. *A* black-white coloring *of a graph G is a partition of the vertices into two sets. All vertices in one set are colored white, and all vertices in the other set are colored black. A* gray edge *of a black-white coloring is an edge with different colored endpoints.*

We have the following properties of an MST:

1. **Cycle Property:** In any cycle, the maximum weight edge will not appear in the MST.
2. **Cut Property:** If the MST is unique, for any black-white coloring of G, the MST will contain the minimum weight gray edge. In the case that the edge weights are not distinct, for any black-white coloring of G, the minimum weight gray edge will be contained in some MST.

The second property is somewhat surprising. There are 2^n different black-white colorings of a graph G (for each vertex, there are 2 choices for its color), but an MST only has $n - 1$ edges. The property implies that for many of the colorings, the minimum-weight gray edge is the same edge (because there aren't many edges in the MST, relative to the potential number of gray edges).

Both properties immediately yield methods for finding minimum spanning trees. Consider the cycle property: find a cycle, and remove the maximum weight edge from the cycle (this edge is not in the MST.) Repeat this until no cycles are left; the remaining edges form the MST.

For the cut property, there exist multiple well-known algorithms that use this fact. Here is one such method for incrementally building the MST:

1. Pick an arbitrary vertex, color it black. Color all other vertices white.
2. Find the minimum weight gray edge and add it to the tree we are building.
3. Color the other vertex incident on the chosen edge black. Recolor edges appropriately.
4. Repeat Steps 2 and 3 until all vertices are black.

See Figure 1.30 and Figure 1.31 for an example of this process. We start with one black vertex and pick the minimum weight gray edge (subsequently colored in blue), resulting in two black vertices. We continue this process until all vertices are black, and the blue edges make up the MST.

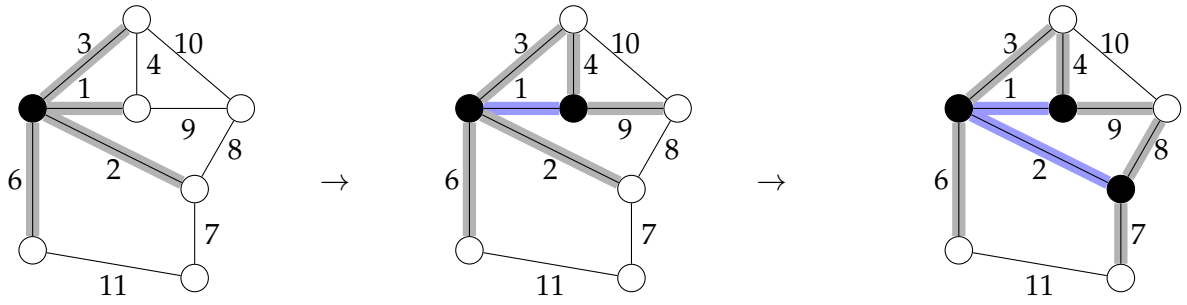

Figure 1.30: Using a black-white coloring to find a MST.

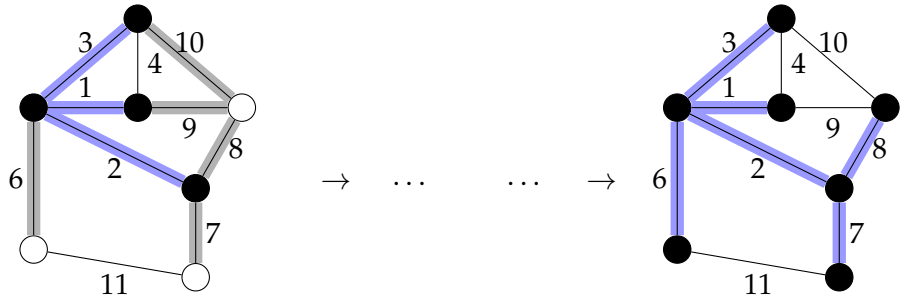

Figure 1.31: Resulting MST (blue).

Now let us prove these properties.

Proof of Cycle Property. Let T^* denote an MST of G and for contradiction, suppose the maximum weight edge $e^* = (u, v)$ of some cycle C is in T^*. Consider the forest $T^* - \{e^*\}$ containing two trees. Now consider the path on C from u to v not containing e^*: this path contains an edge f whose endpoints cross the trees of T, and $w(f) < w(e^*)$.

Now let $T = T^* - \{e^*\} + \{f\}$, that is, replace the edge e^* in T^* with this edge f. Notice that T is connected and has $n - 1$ edges, so T is a spanning tree of G. Furthermore,

$$w(T) = w(T^*) - w(e^*) + w(f) < w(T^*),$$

contradicting our assumption that T^* is an MST. □

Proof of Cut Property. For simplicity, we assume all edge weights are distinct. Let T^* be the MST of a graph G, and let $e^* = \{u, v\}$ be the minimum weight gray edge of a black-white coloring of G. For contradiction, suppose e^* is not in T^*.

Consider the graph $T^* + \{e^*\}$, which contains one cycle (see Figure 1.32). Since e^* is gray edge, u and v have different colors. Thus, the path from u to v in T^* must contain some gray edge f such that $w(e^*) < w(f)$ (see Figure 1.33) .

Now let $T = T^* + \{e^*\} - \{f\}$; note that T is a spanning tree whose weight is

$$w(T) = w(T^*) + w(e^*) - w(f) < \textbf{\textit{weight}}(T^*),$$

contradicting the assumption that T^* is an MST. □

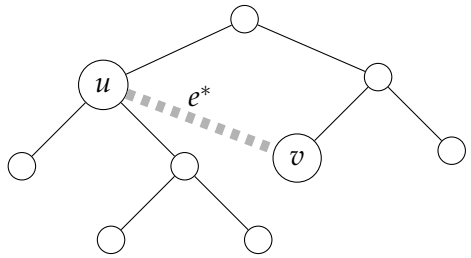

Figure 1.32: Tree T and min weight gray edge $e^* = (u, v)$ dashed.

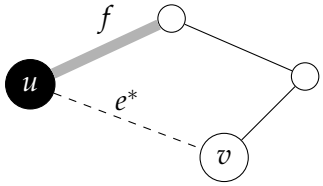

Figure 1.33: Cycle containing e^*.

1.8.3 Directed acyclic graphs (DAGs)

In this section, we consider directed acyclic graphs (i.e., directed graphs that do not contain any *directed* cycles), also known as DAGs.

Definition 22. *A vertex with in-degree 0 is called* source*. A vertex with out-degree 0 is called* sink*.*

Theorem 15. *Every DAG contains at least one source and at least one sink.*

Proof. Consider a longest path in a DAG. The starting vertex is a source because if it were not, then the path could be extended to an even longer path. Similarly, the ending vertex is a sink. □

Topological sorting

Definition 23. *A* topological sort *of a directed graph is a linear ordering of its vertices such that for every directed edge (u, v) from u to v, u comes before v in the ordering.*

A common application for topological sort is to decide the order of tasks. For example, in the morning, you need to get up before you shower, dress, eat, and leave. Furthermore, you must shower before you dress, dress before you leave, and eat before you leave. To determine a valid sequence of events, we can model each task as a vertex, and edges represent the constraints. In general, a system might contain many tasks with various constraints; a DAG can help us find the correct order to perform the tasks.

Theorem 16. *A topological sort is possible if and only if the graph is acyclic.*

Proof. We first prove the forward direction by contradiction. Suppose G has a topological sort and contains some cycle $(v_0, v_1), (v_1, v_2), ..., (v_k, v_0)$. Then in the topological sort, v_0 must be before v_1, v_1 must be before v_2, and so on, but then we conclude v_k must be before v_0, and such an arrangement cannot be satisfied by the topological sort.

Now we prove if G is acyclic, then it has a topological sort. Let v_1 be any source of G: we place v_1 first in the ordering and remove it from G (as well as all edges incident to v_1). The resulting

graph is acyclic, so we can repeat this process until all vertices have been removed. We must show that this process is indeed a topological sort.

Consider any edge $(x, y) \in E$. When we added y to the order, this edge must have been already deleted (because y is a source). The only way to delete this edge is by removing x from the graph. Therefore, x was removed before y was added, so x appears before y in the final order. □

Note that when constructing a topological ordering, in each step there can be multiple sources, so a DAG can have multiple topological orders.

1.9 Connectivity in Directed Graphs

Definition 24. *Vertices u and v are* weakly connected *if they are connected when we replace the directed edges by undirected edges.*

The idea of weakly connected leads to weakly connected components. If we replace all directed edges to undirected edges, we get an undirected graph. Each component in the original graph corresponding to a connected component in the undirected graph is a weakly connected component.

Definition 25. *Vertices u and v are* strongly connected *in a directed graph if there is a path from u to v* and *a path from v to u.*

The relation defined by strong connectivity is an equivalence relation on the vertices of a directed graph. We verify the three necessary properties.

- *Reflexivity:* For any vertex v, v is (vacuously) strongly connected to itself.
- *Symmetry:* This follows from the symmetry of the definition of strong connectivity. If s and t are strongly connected, then there is a path from s to t and a path from t to s, so t is also strongly connected to s.
- *Transitivity:* Suppose vertex a is strongly connected to vertex b and vertex b is strongly connected to vertex c. This implies there exists a path from a to b and a path from b to c. We can combine these two paths to obtain a walk from a to c. This implies there is a path from a to c. Similarly, there exists a path from c to b and a path from b to a. Combining these paths, we obtain a walk from c to a. This implies there is a path from c to a. Thus, a and c are strongly connected, and the relation is transitive.

Definition 26. Strongly connected components*(Sccs) are the equivalence classes of the equivalence relation strong connectivity on the vertices of a directed graph.*

Consider a directed cycle, such as the one shown in Figure 1.34. The vertices in a cycle are strongly connected. Thus, a directed cycle has a single strongly connected component.

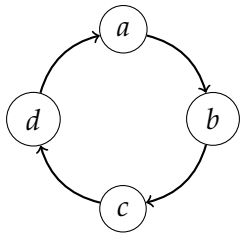

Figure 1.34: A directed cycle.

Now, consider a directed graph that does not have any cycle. Such a digraph is called a *directed acyclic graph* (DAG). The strongly connected components in a DAG are the singleton vertices. We will prove this with the following claim.

Claim 1. *In a DAG, no two vertices can be strongly connected.*

Before we prove this Claim, we will prove a useful lemma.

Figure 1.35: Base Case of Proof of Lemma 3.

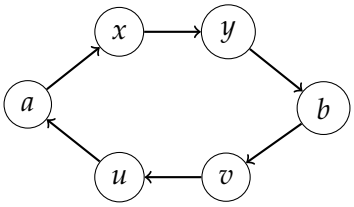

Figure 1.36: Disjoint paths from a to b and b to a.

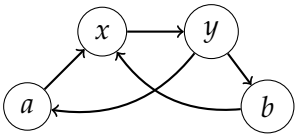

Figure 1.37: Non-disjoint paths from a to b and b to a.

Lemma 17. *Suppose $G = (V, E)$ is a directed graph. If there is a path from vertex a to vertex b and a path from b to a, then there is a cycle containing a in G.*

Proof. Suppose we have directed graph $G = (V, E)$ in which there is a path from vertex a to vertex b and a path from b to a. Let P be the path from a to b and Q be the path from b to a. We will prove Lemma 17 by induction on the sum of the lengths of the paths: $\ell = |P| + |Q|$.

Base case: The smallest possible distinct paths from a to b and b to a are two paths consisting of one edge each. In this case, $(a, b), (b, a) \in E$ and these two edges form a cycle.

Inductive hypothesis: Assume the lemma holds for paths P', Q' such that $|P'| + |Q'| < \ell$.

Inductive step: Suppose we have a graph containing paths P, Q where $\ell = |P| + |Q|$. If P and Q are disjoint, then $P + Q$ is a cycle containing a. Otherwise, paths P and Q share a vertex. Let this vertex be x. Consider the subset of path P from a to x, let this be P'. Similarly, let Q' be the subset of Q starting at x and ending at a. $|P'| + |Q'| < \ell$ since $|P'| < |P|$ and $|Q'| < |Q|$. By the inductive hypothesis, there exists a cycle containing a. Thus, in all cases the Lemma holds. □

Now the proof of Claim 1 is straightforward.

Proof of Claim 1. We will prove the claim by contradiction. Assume two vertices in a DAG are strongly connected, let these be vertices u and v. This implies there is a path from u to v and a path from v to u. By Lemma 17, there is a cycle containing u. This contradicts that the graph is a DAG. Thus, no two vertices are strongly connected in a DAG. □

1.9.1 Structure Of Directed Graphs

Strongly connected components and DAGs are useful for describing the structure of any directed graph. Consider 'contracting' each strongly connected component of a graph into a single vertex, and add an edge from one component to another component if there is an edge from some vertex in the first component to some vertex in the second. The result of this procedure is another directed graph, and it is acyclic. Intuitively, if there was a cycle containing multiple strongly connected components then they could be merged into a single strongly connected component. See Figure 1.38 for an example. Thus, any directed graph is a DAG on its strongly connected components. We will state and prove this theorem formally.

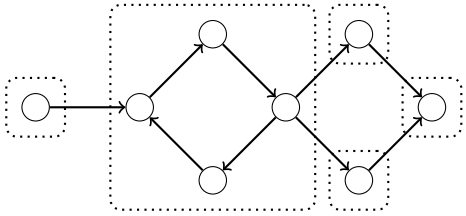

Figure 1.38: A directed graph and its strongly connected components (each in a dashed box).

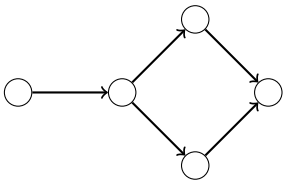

Figure 1.39: The component graph of the graph in Figure 1.38.

Definition 27. *Suppose a directed graph $G = (V, E)$ has strongly connected components $C_1, C_2, \ldots, C_k$. The* component graph *of G is $G^{SCC} = (V^{SCC}, E^{SCC})$ where $V^{SCC} = \{1, \ldots, k\}$ and $E^{SCC} = \{(i, j) : \exists (u, v) \in E \text{ s.t. } u \in C_i,\ v \in C_j\}$.*

Theorem 18. *Every component graph is a DAG.*

Proof. We will prove this theorem by contradiction. Suppose there exists a component graph containing a cycle. Let $G = (V, E)$ be the underlying directed graph, and G^{SCC} be the component graph of G. Suppose $C = c_1, \ldots, c_k$ are vertices in G^{SCC} forming a cycle. Let $C_1, \ldots C_k$ be the corresponding strongly connected components of G.

Since C is a cycle, there must be an edge between components C_i and C_{i+1} in the cycle (for $i = 1, \ldots, k-1$. Let such an edge be $(out_i, in_{i+1}) \in E$ for $i \in 1, \ldots, k-1$. Intuitively, out_i is the vertex in C_i with an outgoing edge to C_{i+1}. Similarly, in_i is the incoming edge form the previous component in the cycle. Also, there is an edge from C_k to C_1, so $(out_k, in_1) \in E$. Within any component, there is a path from in_i to out_i, because all vertices in this component are strongly connected. Thus, for any two components in a cycle, there is a path from the outgoing vertex of the first component to the incoming vertex of the second component.

Suppose C_i, C_j are two components in this cycle. Let $u, v \in G$ be vertices such that $u \in C_i$ and $v \in C_j$. u, v are in different strongly connected components in G, so there cannot be paths both from u to v and from v to u in G.

Since c_i and c_j are vertices in a cycle, there exists a path from some the outgoing vertex of C_i to the incoming vertex of C_j by the previous argument. C_i is a strongly connected component, so there must exist a path from u to out_i. Similarly, C_j is a strongly connected component, so there must exist a path from in_j to v. By concatenating these three paths, we have found a path in G from u to v.

We can similarly find a path from v to u. Since c_i and c_j are in a cycle, there exists a path from some vertex $out_j \in C_j$ to some vertex in $in_j \in C_i$ by the earlier argument. Since C_j is a strongly connected component, there exists a path from v to c_j. Similarly, since C_i is a strongly connected component, there exists a path from c_i to u. By concatenating these three paths, we have found a path in G from v to u.

We have showed there is a path from u to v and from v to u in G. However, u and v were assumed to be in different strongly connected components. This is a contradiction. □

1.10 Searching in a Graph

The language of graphs allows us to formally model a rich set of real-world problems. For example, as we saw in Lecture 15, the problem of graph coloring allows us to color regions on a map, assign time slots to final exams, and solve many other questions that naturally arise.

Another important concept in graphs is the notion of *searching*. Given a directed graph, there are many natural search-related questions that one might ask: Is the graph acyclic? Does there exist a path from vertex s to vertex t? If so, what is the length of the shortest path from s to t? All of these questions can be addressed by the topics we will discuss in this lecture.

These search questions also arise naturally in the real world. For example, if the vertices of the graph represent courses and the edges represent prerequisite requirements (edge (u, v) exists iff course u is a prerequisite for course v), then the graph must be acyclic if every student is expected to take every course. If the vertices represent locations on a map and the edges represent roads (edge (u, v) represents a road between location u and location v), then the shortest path from one vertex to another gives the fastest way to travel from one location to another.

In this lecture, we will learn about Depth-First Search and Breadth-First Search. These two algorithms solve the problems mentioned above (and many others), and they form the foundations of many algorithms in computer science. Throughout this lecture, we will assume that $G = (V, E)$ is a directed graph with n vertices and m edges. Note that these algorithms can be used for undirected graphs as well: an undirected edge $\{u, v\}$ is equivalent to two directed edges (u, v) and (v, u).

1.10.1 Depth-First Search

Depth-first search (DFS) is an algorithm that allows us to systematically explore every vertex of a directed graph. The strategy used by DFS is the following: start at any vertex s and explore one path leading away from s until nothing new can be discovered. Along this path, the search may encounter many "forks" in the road; DFS arbitrarily picks a path that leads somewhere new until it gets stuck. At this point, DFS backtracks until the last choice of path was made, and chooses another path to completely explore. This procedure is repeated until all vertices have been visited.

Note that it is sometimes necessary to restart the procedure starting at a completely new vertex. For example, this is necessary if the first vertex chosen has no outgoing edges, or the graph contains multiple components that are not connected by any edge.

While DFS explores the graph, it also maintains a global time variable τ initialized to $\tau = 1$. The algorithm uses this τ variable to mark each vertex u with the first time it is seen, as well as the time at which DFS backtracks away from u. These times are known as the *pre-* and *post*-values of u, which we denote by $pre(u)$ and $post(u)$, respectively. Every time a vertex is marked, the value of τ

increases by one. Thus, we have $pre(u) < post(u)$ for every $u \in V$, and the *pre*- and *post*-values are exactly between 1 and $2n$ (inclusive).

Algorithm 2 provides the formal pseudocode for a complete run of DFS, while Algorithm 1 is a recursive helper function. Note that Algorithm 1 alone is not enough because it is possible that some vertices cannot be reached from s. In this case, as mentioned above, DFS must restart the procedure starting at a new vertex.

Algorithm 1 explore(G, s)

Input: A directed graph $G = (V, E)$; a vertex $s \in V$.
Output: Two positive integers $pre(u)$, $post(u)$ for every vertex u reachable from s.
Note: The variable τ is global, initialized to 1. The default value of visited() is False.

```
1: visited(s) = True
2: pre(s) = τ
3: τ = τ + 1
4: for all (s, v) ∈ E do
5:     if visited(v) == False then
6:         explore(G, v)
7: post(s) = τ
8: τ = τ + 1
```

Algorithm 2 Depth-First-Search(G)

Input: A directed graph $G = (V, E)$.
Output: Two positive integers $pre(u)$, $post(u)$ for every vertex $u \in V$.

```
1: for all v ∈ V do
2:     visited(v) = False
3: for all u ∈ V do
4:     if visited(u) == False then
5:         explore(u)
```

Note that in explore(G, s), the algorithm can visit the neighbors of s in any arbitrary order. Furthermore, Depth-First-Search(G) can visit the vertices $u \in V$ in arbitrary order. Thus, the *pre*- and *post*-values are not necessarily unique.

We now illustrate the first few steps of DFS on the graph shown in Fig. 1.40, with ties broken by alphabetical order. We begin at vertex a, so $pre(a) = 1$ and there are three options: b, e, d. Since b appears first alphabetically, we explore b which sets $pre(b) = 2$. From b we explore c, so $pre(c) = 3$. The only edge leading away from c is (c, a), but a has already been visited, so our exploration of c terminates and we set $post(c) = 4$. We then backtrack to b, and since (b, c) is the only outgoing edge, we set $post(b) = 5$. We then backtrack to a, where the exploration continues with the edge (a, d). The values in Table 1.2 give the full results of DFS.

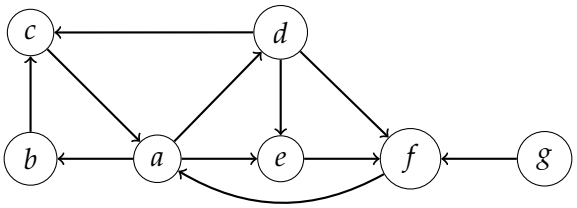

Figure 1.40: A directed graph with 7 vertices and 9 edges. In this lecture, we will be illustrate DFS and BFS on this graph.

	a	b	c	d	e	f	g
$pre()$	1	2	3	6	7	8	13
$post()$	12	5	4	11	10	9	14

Table 1.1: The full results of running DFS on the graph in Fig. 1.40, with all ties in the algorithm broken by alphabetical order.

DFS Tree

Whenever we run DFS on a graph $G = (V, E)$, we create a structure known as a *DFS tree*. In general, a "DFS tree" may actually be a set of multiple rooted trees, but we still refer to the entire structure as a "tree." The vertex set of a DFS tree is V, and the edges are the ones that pass the condition in line 5 of explore(G, s). In other words, the edges of a DFS tree are the ones that the DFS used to continue down an exploration path. Note that if s is the first vertex DFS explores, and s can reach every vertex in V, then in this case the DFS tree is actually a tree.

Now observe that the edges of a DFS tree indeed form a forest on the vertex set V. This is for the following reason: (u, v) is a tree edge if explore(G, v) is called while explore(G, u) is being executed. Furthermore, each vertex is explored exactly once because as soon as u is explored, we set visited(u) to be True. Thus, a cycle cannot appear in a DFS tree because its existence would imply that a vertex was explored more than once.

Based off a DFS tree, we can partition the set of edges in the graph. Every edge $(u, v) \in E$ can be categorized as exactly one of the following:

1. *Tree edge*: DFS explores v while in the process of exploring u.
2. *Forward edge*: u is a non-parent ancestor of v in the DFS tree.
3. *Back edge*: v is an ancestor of u in the DFS tree.
4. *Cross edge*: u and v have no ancestor-descendent relationship in the DFS tree.

Fig. 1.41 gives the partition of the graph in Fig. 1.40. Notice that each back (red) edge forms a cycle when added to the set of tree (green) edges. Furthermore, the forward (blue) edges point from ancestor to descendent, and the cross (dashed) edges connect nodes with no ancestor/descendent relationship.

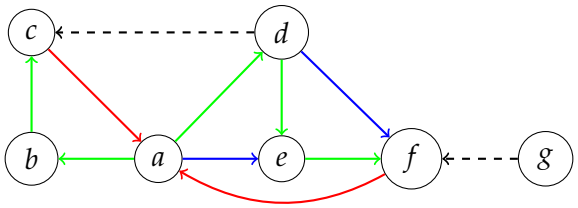

Figure 1.41: The DFS tree corresponding to a DFS run on Fig. 1.40, with ties broken in alphabetical order. Tree edges are green, forward edges are blue, back edges are red, and cross edges are dashed.

1.10.2 Breadth-First Search

Although DFS can help us determine whether or not a vertex t is reachable from a vertex s, the result may not be the shortest path. For example, in the graph in Fig. 1.40, DFS discovered a path from a to d that uses two edges: (a, e) and (e, d). However, the shortest path from a to d only uses one edge: (a, d). To find the shortest path between two vertices s and t, we can use an algorithm known as breath-first search (BFS). Before we describe the algorithm, we first formalize the notion of distance in graphs.

Definition 28. *Let $G = (V, E)$ be a directed graph, and let s and t be two vertices of G. The* distance *between s and t, denoted by $dist(s, t)$, is the length of the shortest path from s to t. If t is not reachable from s, then we say the distance from s to t is infinite, i.e., $dist(s, t) = \infty$.*

The strategy used by BFS is the following: start at some vertex s and explore the remaining vertices one "layer" at a time, with each layer being farther from s than the previous. The algorithm fully explores each "layer" before moving onto the next, and hence it is called *breadth*-first search. Algorithm 3 gives the formal pseudocode for BFS.

Algorithm 3 Breadth-First-Search(G, s)

Input: A directed graph $G = (V, E)$; a vertex $s \in V$.
Output: An integer value $d(s, u)$ (possibly ∞) for every vertex $u \in V$.
Note: The default value of $d(\cdot, \cdot)$ is ∞, and Q is initially an empty queue.

```
1: d(s,s) = 0
2: Push(Q,s)
3: while Q is not empty do
4:     u = Pop(Q)
5:     for all (u,v) ∈ E do
6:         if d(s,v) = ∞ then
7:             Push(Q,v)
8:             d(s,v) = d(s,u) + 1
```

Notice that BFS uses a queue to maintain the "layer-by-layer" approach. The vertices closer to s are added to Q before vertices farther from s, and the queue allows us to process all of the closer vertices first before processing the farther ones.

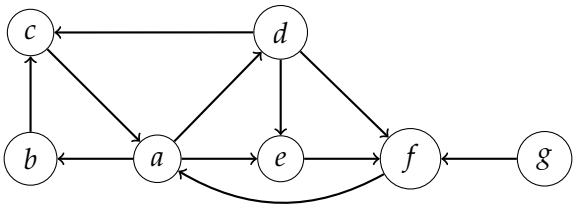

Figure 1.42: The graph from Fig. 1.40, drawn here for convenience.

	a	b	c	d	e	f	g
$d(a, \cdot)$	0	1	2	1	1	2	∞

Table 1.2: The full results of running BFS on the graph in Fig. 1.42, with all ties in the algorithm broken by alphabetical order.

We now step through the BFS algorithm on the graph in Fig. 1.42 a few times to better understand the process. We start with $s = a$, so the only element of Q is a. Thus, the first element popped is a, and none of the three neighbors b, d, e have been visited, so they are pushed onto Q in that order and their distances are set to $0 + 1 = 1$. The next element popped is b, and (b, c) is the only outgoing edge, so c is pushed onto Q. Later, after we've processed d and e, we'll pop c and set $d(s, c) = d(s, b) + 1 = 1 + 1 = 2$. This process continues until f is popped since f has no outgoing edges. Note that g is never pushed onto the queue, so $d(a, g)$ remains at ∞. We can verify that g is not reachable from a by visually inspecting the graph.

BFS Tree

Recall that a run of DFS creates a structure known as a DFS tree, which is technically a forest since it may contain multiple rooted trees. Similarly, a run of BFS creates a tree structure known as a *BFS tree*. If we strictly follow Algorithm 3, then the resulting BFS tree contains exactly one tree, which is rooted at s and contains the vertices reachable from s. However, we can extend the BFS procedure to create a forest by restarting the algorithm at an unvisited vertex as long as such a vertex exists. This idea is captured in Lines 3-5 of Algorithm 2—Algorithm 3 can be modified similarly.

An edge (u, v) is in the BFS tree if, when u was popped from Q, vertex v satisfied $d(v) = \infty$. In other words, if vertex v was unvisited after u was popped from the queue, then u is the parent of v in the BFS tree. Thus, in both a DFS and a BFS "tree," a tree rooted at r contains the vertices reachable from r, and the edges of the tree are the ones used by the algorithm to reach those vertices.

Furthermore, BFS categorizes every edge of the original graph as tree, cross, or back. (The definitions are identical to the ones given for a DFS tree.) Fig. 1.43 gives the partition of edges from Fig. 1.42 according to the BFS that starts at a and breaks ties alphabetically. This run of BFS produced 5 tree edges, 4 cross edges, and 2 back edges. Notice that there are no forward edges—in general, BFS never categorizes any edge as a forward edge, and we will now prove this.

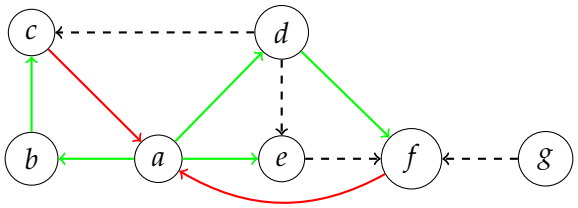

Figure 1.43: The BFS tree corresponding to a run of BFS on Fig. 1.42, with ties broken in alphabetical order. Tree edges are green, back edges are red, and cross edges are dashed.

Theorem 19. *A run of BFS will never categorize any edge as a forward edge.*

Proof. For contradiction, assume (u, v) is a forward edge. This means u is a non-parent ancestor of v, so there exists a path from u to v in the BFS tree with at least 2 edges. Let x be the parent of v in the tree. The existence of the path tells us that u entered Q before v, so when u was popped, v must have either been in the queue or would be immediately added. However, (x, v) being a tree edge implies v was added when x was popped, contradicting our conclusion that v was added when u was popped (or earlier). □

Finally, consider a tree with root r of the BFS tree. The edges of this tree form the shortest path from r to every vertex reachable from r. In other words, the final value of $d(s, u)$ in Algorithm 3 is indeed equal to $dist(s, u)$. For example, in Fig. 1.43, we can see that the green path from a to each vertex (except g, which a cannot reach) is a shortest path. This property follows from the "layer-by-layer" approach taken by BFS, though we will not rigorously prove it here. For this reason, a BFS tree is sometimes known as a *shortest-path tree*. In general, the shortest paths obtained via BFS are the ones that begin at the roots of the trees of the BFS tree.

Application: Shortest Paths

We've seen that the BFS tree rooted at s gives a shortest path from s to every vertex rooted at s. Now we will generalize this notion to edges with lengths: suppose each edge e has a positive integer *length* value. (We can think of the previous cases as all lengths being 1.) For example, consider the graph in Fig. 1.44. Running BFS on this graph might lead us to conclude that the shortest path from a to c is (a, c) (which has length 3), when in reality it is (a, b, c) (length 2).

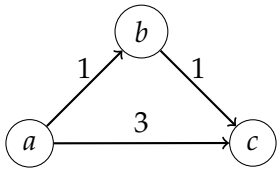

Figure 1.44: A directed graph with edge lengths. The shortest path from a to c is (a, b, c), and this path has length $1 + 1 = 2$.

However, we can fix this problem by subdividing each edge according to its length, so that the new edges each have length 1. In Fig. 1.45, we see that by adding two new vertices to the graph in Fig. 1.44, we can create a new graph whose edge lengths are all 1. Of course, this technique

easily generalizes to any graph with positive integer edge lengths. An edge with length ℓ can be subdivided by placing $\ell - 1$ vertices on the edge, and all new edges are assigned length 1. By running BFS on this new graph starting at vertex s, we can obtain the distances from s to all other vertices in the original graph.

But, this technique has two major shortcomings: it does not work if the edge lengths are anything other than positive integers, and even in that case, the process is very expensive in terms of running time because it introduces a large number of non-existent vertices to model a long edge. These shortcomings can be overcome by more sophisticated algorithms for finding shortest paths that are beyond the scope of our current discussion.

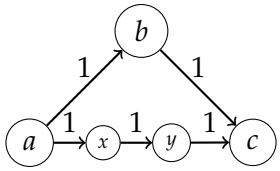

Figure 1.45: The graph from Fig. 1.44 with the edge (a, c) subdivided, so that all edges of this new graph have length 1.

Optimal substructure: One property we would like to note about shortest paths is known as the optimal substructure property. This property is quite intuitive and easy to understand, but given its importance in algorithm design, we shall state the property here. Suppose $p = (s, u_1, u2, \ldots, u_k, t)$ is a shortest path from vertex s to vertex t. Now consider two vertices x, y on this path, so p is of the form $(s, \ldots, x, \ldots, y, \ldots, t)$. The optimal substructure property states that the path from x to y that follows p is the shortest path from x to y. The reasoning is quite intuitive: if there were a shorter path from x to y, then we could use that path to create a shorter path from s to t.