Partially Ordered Sets; Digraphs and Graphs

Topics for today:

- Partially ordered sets (continued)
- Digraphs
- Graphs
- Graph coloring
- Graph connectivity

Reading: Rosen Section 6.6; Chapter 7

1 Partially Ordered Sets

This section contains some more applications and properties for partially ordered sets.

1.1 Parallel Task Scheduling

When elements of a poset are tasks that need to be done and the partial order represents precedence constraints, topological sorting provides us with a legal way to execute tasks sequentially, i.e., without violating any precedence constraints.
But what if we have the ability to execute more than one task at the same time?
For example, say tasks are programs, partial order indicates data dependence, and we have a parallel machine with lots of processors instead of a sequential machine with only one.
How should we schedule the tasks?
Our goal should be to minimize the total *time* to complete all the tasks.
(For simplicity, say all the tasks take the same amount of time and all the processors are identical.)
This leads to the question:
Given a finite poset of tasks, how long does it take to do them all, in an optimal parallel schedule?
Assume that all tasks take 1 unit of time and we have an unlimited number of processors.

Use partial order concepts to analyze this problem.

Example: Clothes (from last lecture)
We could do all the minimal elements first (left sock, right sock, underwear, shirt), remove them and repeat.
(Need lots of hands, or maybe servants.)
Do pants and sweater next, and then left shoe, right shoe, and belt, and finally jacket.

Can't do any better, because the sequence shirt, pants, belt, jacket must be done in that order.
A sequence like this is called a chain.

Definition 1.1 A **chain** in a poset is a sequence of elements of the domain, each of which is smaller than the next in the partial order ($\preceq$ and $\neq$).

Clearly, parallel time $\geq$ length of any chain.
This is true, because if we used less time, then by pigeonhole two tasks in the chain would have to be done at the same time, but by definition of chain this violates precedence constraints.

A longest chain is also known as a *critical path*.

So we need at least t steps, where t is the length of the longest chain.
Fortunately, it is always possible to use only t:

Theorem 1.2 Given any finite poset $(A, \preceq)$ for which the longest chain has length t, it is possible to partition A into t subsets, $A_1, A_2, \cdots, A_t$ such that $(\forall i \in [1,t])(\forall a \in A_i)(\forall b \preceq a, b \neq a)[b \in A_1 \cup A_2 \cup \cdots \cup A_{i-1}]$.

That is, can divide up the tasks into t "groups" so that for each group i, all tasks that have to precede tasks in i are in smaller-numbered groups.

Corollary 1.3 For $(A, \preceq)$ and t as above, it is possible to schedule all tasks in t steps.

Proof. $\forall i$, schedule all elements of A_i at time i.
This satisfies the precedence requirements, because all tasks that must precede a task are scheduled at preceding times.

■

Corollary 1.4 Parallel time = length of longest chain

So it remains to prove the partition theorem:

Proof. Use the rule:
Put each a in set A_i, where i is the length of the longest chain ending at a.

See this works:
It gives just t sets, because the longest path is length t.

Need to show $(\forall i)(\forall a \in A_i)$
$(\forall b \preceq a, b \neq q)[b \in A_1 \cup A_2 \cup \cdots \cup A_{i-1}]$.

Proof by contradiction:
If $b \notin A_1 \cup A_2 \cup \cdots \cup A_{i-1}$ then there is a path of length $> i-1$ ending in b,
which means there is a path of length $> i$ ending in a,
which means $a \notin A_i$.
This is a contradiction. ■

So with an unlimited number of processors, the time to complete all the tasks is the length of the longest chain.

It turns out that this theorem is good for more than parallel scheduling. It is usually stated as follows.

Definition 1.5 An **antichain** is a set of incomparable elements.

Corollary 1.6 If t is the length of the longest chain in a poset $(A, \preceq)$ then A can be partitioned into t antichains.

This is a corollary because we showed before how to partition the tasks into sets that could be done at the same time, which means that these were incomparable. (If any two comparable then one would have to precede other and so couldn't be done at same time.)

Review:

- **chain**: all elements comparable (total order)
- **antichain**: all elements incomparable

1.2 Dilworth's Theorem

We can use this theorem to prove a famous result about posets:

1.2.1 The Basics

Theorem 1.7 (Dilworth) $\forall t$, every poset with n elements must have a chain of size $> t$ or an antichain of size $\geq \frac{n}{t}$.

Ex: Dressing poset. $n = 10$. Try $t = 3$. Has chain of length 4.
Try $t = 4$. Has no chain of length 5, but has antichain of size $4 \geq \frac{10}{4}$.

Proof. Assume there is no chain of length $> t$.
So, longest chain has length $\leq t$.
Then by previous result, the n elements can be partitioned into $\leq t$ antichains.
(Cannot assume exactly t, since could have some empty.)
So there is an antichain with $\geq \frac{n}{t}$ elements.
As needed. ∎

For instance:

Corollary 1.8 Every poset with n elements has a chain of length $> \sqrt{n}$ or an antichain of size $\geq \sqrt{n}$.

Set $t = \sqrt{n}$ and this is immediate.

Posets arise in all sorts of contexts, and when they do Dilworth's theorem can have interesting implications.

1.2.2 Number of Comparisons for Database Setup and Searching

Consider a "database" of records with (distinct) numerical keys.
Suppose when records are inserted, some comparisons of these keys occur, and the database maintains a record of these comparisons.

E.g., at one extreme, database could be kept totally sorted.
E.g., totally unsorted.

Each comparison costs something – the time to perform it.

Now suppose that after this setup, a search is to be performed on the database to determine if a record with a particular key k appears anywhere in the database.
The search is supposed to be based on comparisons only.
Then the worst-case number of comparisons it takes depends on the database partial order.

E.g., if the database is totally sorted, we can compare with middle elements, then middle of halves, etc.
Binary search.
Approximately $\log n$ comparisons needed.

E.g., if database totally unsorted, then in the worst case, have to compare with every element.
This is because each comparison with an element in the database gives no information about whether the sought element might be any of the others in the database.

In fact, if there is an antichain of size k, then in the worst case, the new key could be any of the elements of the antichain.
So in the worse case, have to compare with all k, since each one gives no information about the others.

See what Dilworth's theorem says about the database partial order.
For any t, either have a chain of length $> t$, or an antichain of size $\geq \frac{n}{t}$.
Setting up a chain of length $> t$ requires at least t comparisons.
The antichain requires at least $\frac{n}{t}$ comparisons.
So this says that we have to pay the cost sometime – tradeoff – either we do $> t$ comparisons while setting up the db, or at least $\frac{n}{t}$ in the worst case to search the db.

More refined analysis is possible.
That's because reliably setting up a long chain, say length t, in the database actually requires *more* than t comparisons.
Because in the worst case, the comparisons won't come out in the most desirable way, setting up a linear chain.
Ex: Compare a and b, find out $a > b$. Then compare b and c. Would be nice if we found out $b > c$.
But we might not – might get $c > b$. Then we don't have a chain of length 3, but still an antichain of size 2. To get a chain, need an additional comparison.

Return to this in analysis of algorithms.

1.3 Logical Time in Distributed Systems

One last use of partially ordered sets.
No use of powerful theorems here – just showing an important place where a poset is used as a mathematical model for a real computational situation.

A distributed computation consists of a sequence of "steps" occurring at each of a collection of n "processes".
In addition to performing local steps, the processes can send messages to each other.
So, have *send* steps and *receive* steps.

The processes are mostly working independently.
The only interaction is by means of the messages.
So, the only dependence among steps (feeding data from one to another, using results to make decisions, etc.) is by:

1. Successive steps of same process.
2. Send vs. receive of same message.

Taking reflexive transitive closure leads to partial order describing the dependence among the computation steps.

Basic theorem: Any topological sort of this partial order describes a possible sequence in which all the distributed steps could occur.
(No changes in what happens based on reordering incomparable steps.)

Example of what one might do with this partial order:
Assign "logical time" to every step.
Instead of an actual real time.
The time is just any assignment of real numbers to the steps that is consistent with the partial order: when one step precedes another in the partial order, it must have a smaller logical time.

Behaves like real time: Formally, can reorder the steps in logical time order, still consistent with the partial order.
Then the system is behaving as if the logical time were actually real time.

Can be maintained by simple distributed algorithm.
(Piggybacks time on message, adjusts local times in response to what is heard in some messages.)

Although logical time isn't the same as actual real time, for many purposes, it can be used as an adequate substitute.
Logical time can be used, e.g., to timestamp files to keep updated copies consistent.

This notion of logical time was invented by Lamport. This year, the original (late 1970s) paper containing this idea won an award for the most influential paper ever, in distributed computing theory.

2 Digraphs

2.1 Basic definitions and notation

We have actually already defined digraphs:
A digraph (directed graph) is just the same as a binary relation on a finite set.
People don't usually think of it that way, though; when people refer to a digraph, they are generally thinking of a particular representation of a binary relation in terms of dots and arrows.

Figure 1 depicts a directed graph.

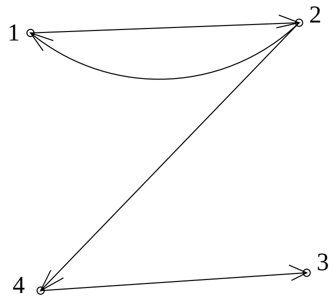

Figure 1: *This is a picture of a directed graph or digraph.*

Recall that a digraph allows self-loops.

Notation: $G = (V, E)$, where V is the set of *vertices* and E is the set of *directed edges*. Vertices are sometimes called *nodes*. Edges are sometimes called *arcs*.

A little different from the usual relation notation: For a relation R on a finite set A, we would write $G = (V, E)$, where $V = A$, the underlying set, and $E = R$, the relation, that is the set of ordered pairs (x, y) such that xRy.

The graphical notation motivates the study of other properties, especially, properties involving paths and connectivity (see below).

Can also represent digraph by adjacency matrix.

Examples of digraphs we have already seen:

1. Tournament digraphs (define as special case of digraphs).
2. Precedence digraphs (e.g., for dressing, or prerequisites).
3. The digraphs we studied when we related matrix multiplication to path composition.

Some (other) applications:

Computer network A computer network can be modeled nicely as a digraph. In this instance, the set of vertices V represents the set of computers in the network. An edge (u, v) represents a direct communication link from the computer corresponding to u to the computer corresponding to v.

Airline Connections Here the vertices are airports and edges are flight paths. We could indicate the direction that planes fly along each flight path by using a directed graph. We could use weights to convey even more information. For example, $w(i, j)$ might be the distance between airports i and j, or the flying time between them, or even the air fare.

Precedence Constraints Suppose you have a set of jobs to complete, but some must be completed before others are begun. (For example, Attila the Hun advised that you should always pillage *before* you burn.) Here the vertices are jobs to be done. Directed edges indicate constraints; there is a directed edge from job u to job v if job u must be done before job v is begun.

Program Flowchart Each vertex represents a step of computation. Directed edges between vertices indicate control flow.

Variants of digraph definition:
Could add *weights* to the edges, representing some kind of cost. E.g., time for airline trip. Could also use other kinds of *labels*, representing kinds of relationships between vertices.

2.2 More definitions

If (x, y) is an edge of G (element of the edge set E of G), then we call x the *initial vertex* and y the *terminal vertex* of the edge.

The *in-degree* of x is defined to be the number of edges with x as terminal vertex.
The *out-degree* of x is defined to be the number of edges with x as initial vertex.

Examples

Theorem: Sum of the indegrees of all nodes in a graph = sum of outdegrees of all nodes in the graph = number of edges

Proof: Induction on number of edges.

2.3 Paths and connectivity

2.3.1 Paths, simple paths, and circuits

Path in a digraph: Same as path in the corresponding binary relation. Sequence of nodes $a_0, a_1, ..., a_n$ such that each consecutive pair (a_i, a_{i+1}) form an edge of the digraph. Allow a path of length 0, which is just a single vertex a_0.

Examples

Path is *simple* if it does not repeat any vertices, except possibly that the first vertex may be the same as the last.

Examples

This definition is different from the one used in Rosen: He says no repeated edges. And he insists that paths have to have at least two vertices (one edge). This definition is consistent, for example, with CLR, the algorithms textbook (6.046).

Theorem: If $|V| = n$, and there is a path from x to y in G, then there is a path from x to y of length at most $n - 1$.

Examples

Circuit (a.k.a. *cycle*) is a path that begins and ends at the same vertex.

Examples

2.3.2 Paths and relational operations

Main results, (mainly review from relations, restated in digraph notation): Let $G = (V, E)$.

- There is a path of length k from x to y in G if and only if $(x, y) \in E^k$. (E is the relation corresponding to the edges.)
- There is a path from x to y in G if and only if (x, y) is in the reflexive transitive closure of E. (In this case, we say that x is *connected to* y in G.)

Recall/review how the adjacency matrix representation can be used to calculate the needed products and take the needed closures.

Can also use this representation to calculate the number of paths between all pairs of vertices. Do Rosen p. 472, Theorem 2 and its proof?

Examples

2.3.3 Euler circuits

Euler circuit in digraph G: A circuit in G that traverses each edge of G exactly once.

Examples

Theorem: Suppose G is a weakly connected digraph (define). Then G has an Euler circuit if and only if the in-degree and out-degree of each vertex are equal.

Proof: Can leave this for HW or tutorial, or do in class.

[[[This theorem is derived from Rosen Section 7.5, exercise 22 and 34. Proof needs working out. Ideas:

"only if": (sketch) For any vertex x, appears some number k of times in the circuit. Each time, has one edge in and one edge out. No repeats, and exhausts all the edges.

"if": Assume G is a weakly connected digraph such that each node's degrees are equal. Give a procedure. It seems to me that basically the same argument that is used for undirected graphs also works here. ??? A little more detail:

If edge set is empty, then weak connectivity implies that there can be only one node. That node by itself constitutes a trivial (0 edge) Euler circuit. So assume that the edge set is nonempty.

In this case, we can show that there exists some circuit: Show this by starting with any edge, and tracing from node to node until we get back to the node we started with (argument as in the argument for the undirected case).

Now choose a maximum length circuit with no repeating edges, in G; call it C. If C includes all the edges, we're done. So assume that it doesn't.

Let H be the subgraph of G consisting of all the edges of G that aren't in C, and all the nodes of G that are incident on these edges. Claim that some node in H is also in C. For if not, G is not weakly connected. Check me here... Choose such a node, x.

Now choose any edge in H that is incident on x (it doesn't matter if it's ingoing or outgoing to x. Starting with that edge, construct a cycle in H, following the same procedure as above. Eventually, it has to get back to its starting point.

Now we have two cycles with disjoint edge sets and a common vertex, x. Can splice them together to get a larger cycle than C. Contradicts choice of C as maximum.]]]

[[[If we do this in lecture, then we could ask about Euler PATHS for HW.]]]

2.3.4 Strong connectivity

Digraph G is *strongly connected* provided that every vertex is connected to every other vertex, that is, there is a path from every vertex to every other vertex.

Examples of digraphs that are and aren't strongly connected.

3 Undirected Graphs

Some relations, in particular, those that are symmetric but not reflexive, have nice representations using a different kind of graph, called an *undirected graph*, or just a *graph*.

3.1 Simple graphs

Start with a symmetric, irreflexive relation. In a symmetric relation, if edge (x, y) is present, edge (y, x) is also present. In an irreflexive relation, no edge (x, x) is present.

A digraph representing such a relation has the following properties:

1. No self-loops.
2. Edges come in pairs: wherever there is an edge from x to y, there is another edge from y to x.

Because edges come in pairs, it simplifies matters to think of these two edges as one "undirected edge". An undirected graph is defined formally to be a structure that consists of vertices and undirected edges:

Definition: An undirected *graph*, also known as just a *graph*, is a pair of sets (V, E). The elements of V are called *vertices*. The elements of E are called *edges*. Each edge is an unordered pair of distinct vertices.

Graphs can be nicely represented with a diagram of dots and lines (not arrows), as shown in Figure 2

As noted in the definition, each edge $(u, v) \in E$ is a pair of distinct vertices $u, v \in V$. Because it's an unordered pair, we really shouldn't be writing it using Cartesian pair notation (because order matters in that notation). Really picky mathematicians denote such an unordered edge by the unordered *set* $\{u, v\}$. But this rapidly fills up your paper with hard to read $\{\}$ symbols. So most people are a little casual here and use the ordered notation even though it's an unordered pair.

Edge (u, v) is said to be *incident* to vertices u and v. Vertices u and v are said to be *adjacent* or *neighbors*. Phrases like, "an edge joins u and v" and "the edge between u and v" are common.

Example: Computer networks are often modeled nicely as undirected graphs, because most communication links are bidirectional. The set of vertices V represents the set of computers in the network. There is an edge (u, v) if there is a direct communication link between the computers corresponding to u and v.

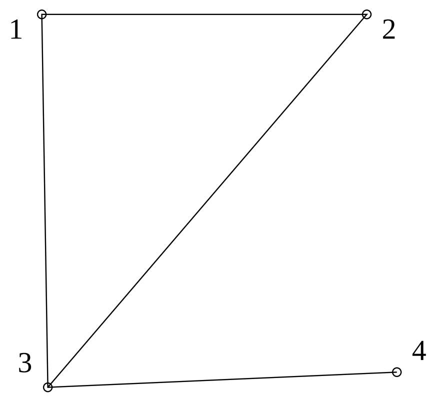

Figure 2: *This is a picture of a graph $G = (V, E)$. There are four vertices and four edges. The set of vertices V is $\{1, 2, 3, 4\}$. The set of edges E is $\{(1, 2), (1, 3), (2, 3), (3, 4)\}$. Vertex 1 is adjacent to vertex 2, but is not adjacent to vertex 4.*

3.2 Not-So-Simple Graphs

The definition in the preceding section only describes *simple undirected graphs*. There are many ways to complicate matters.

3.2.1 Multigraphs

In a simple graph, there are either zero or one edges joining a pair of vertices. In a *multigraph*, multiple edges are permitted between the same pair of vertices.

3.2.2 Pseudographs

In a simple graph or multigraph, each edge is a pair of distinct vertices. A *self-loop* is an edge that connects a vertex to itself. A *pseudograph* allows multiple edges and self-loops. Figure 3 depicts a pseudograph.

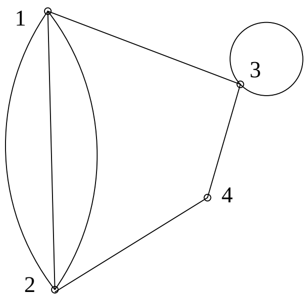

Figure 3: *This is a picture of a multigraph with a self-loop. In particular, there are three edges connecting vertices 1 and 2 and there is a self-loop on vertex 3.*

3.2.3 Weighted Graphs

Sometimes it is useful to associate a number with each edge in a graph. These numbers are usually called *weights*. The weight of an edge (u, v) is often denoted $w(u, v)$. (If there is no edge between vertices u and v, then often $w(u, v)$ is taken to be a default value, like infinity or zero.)

Recall that an ordinary graph can be represented nicely by an *adjacency matrix*. Because the underlying relation for an undirected graph is symmetric, such a matrix is symmetric around the main diagonal. Also, because the underlying relation is irreflexive, the main diagonal consists of all 0s.

Some variants of simple graphs can also be described well by matrices. For example, a weighted graph is naturally represented by a matrix where $w(i, j)$ is the entry at row i and column j.

3.3 Standard Types of Graphs

Some graphs appear so frequently that they have names. The most important examples are shown in Figure 4.

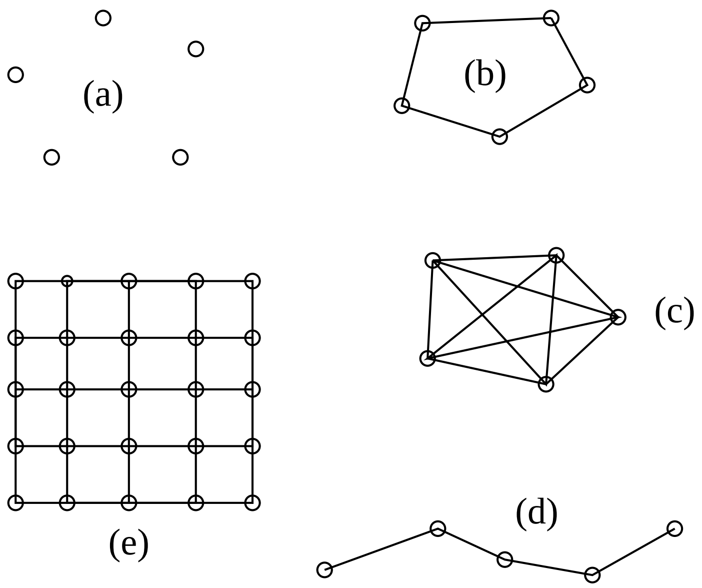

Figure 4: *The types of graph shown here are so common that they have names. (a) The empty graph on five vertices, E_5. An empty graph has no edges at all. (b) The cycle on five vertices, C_5. (c) The complete graph on five vertices, K_5. A complete graph has an edge between every pair of vertices. (d) A five-vertex line graph. (e) A 5×5 2-dimensional mesh.*

3.4 More definitons

3.4.1 Subgraphs

A *subgraph* of a graph $G = (V, E)$ is a graph $G' = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq E$. For example, a triangle is a subgraph of K_4 (the complete graph on four vertices), but not of C_4 (a cycle with four vertices).

3.4.2 Vertex degree

The *degree* of a vertex is the number of incident edges. The degree of vertex v is often denoted $d(v)$. For example, referring to Figure 4, every vertex in an empty graph has degree 0, but every vertex in a cycle has degree 2. A look at the other examples suggests the following theorem:

Theorem 3.1 In a graph with m edges, the sum of degrees of the vertices is $2m$.

Proof. By induction on the number of edges. 0 edges means the sum is 0 as claimed. Consider an m-edge graph. Take away one of the edges. The remaining graph has degree sum $2(m-1)$ (by induction). Putting back the edge increments two degrees by one, so the degree sum becomes m. ■

Corollary 3.2 In every graph, there are an even number of vertices of odd degree.

Proof. If we had an odd number of odd degrees, the degree sum would be odd. But it is even. ■

4 Graph Coloring

Time to discuss final exams. The MIT Schedules Office needs to assign a time slot for each final. This is not easy, because some students are taking several classes with finals, and a student can take only one test during a particular time slot. The Schedules Office wants to avoid all conflicts, but wants to make the exam period as short as possible.

This scheduling problem can be represented by a graph. Let each vertex represent a course. Put an edge between two vertices if there is some student that is taking both courses. Identify each possible time slot with a color. For example, Monday 9-12 is red, Monday 1-4 is blue, Tuesday 9-12 is green, etc. The problem of assigning time slots to exams is then equivalent to the problem of assigning colors to vertices.

If a student is taking two courses, then there is an edge between two vertices. If these vertices are the same color, then there is a conflict because the student has to take exams for the two courses at the same time. Therefore, the problem is to color each vertex of the graph so that no adjacent vertices have the same color and so that the number of colors used is as small as possible. An example is shown in Figure 5.

In general, the minimum number of colors needed to color the vertices of a graph G so that no two adjacent vertices are the same is called the *chromatic number* and is written $\chi(G)$[1]. For example, if G is the 3-cycle or triangle graph, then $\chi(G) = 3$.

In general, finding the chromatic number of a graph is very hard. However, we can at least put some upper bounds on the chromatic number. For example, the chromatic number of an n-vertex graph is certainly at most n, since every vertex could be assigned a different color.

We can do better, however:

Theorem 4.1 Any graph in which the maximum vertex degree is Δ can be colored with at most $\Delta + 1$ colors.

This theorem implies, for example, that a graph with thousands of vertices, each of degree 3, requires at most 4 colors. The proof is surprisingly easy:

[1]This is the Greek letter "chi" pronounced "key" by the ancient Greeks and "kie" by mathematicians and frats.

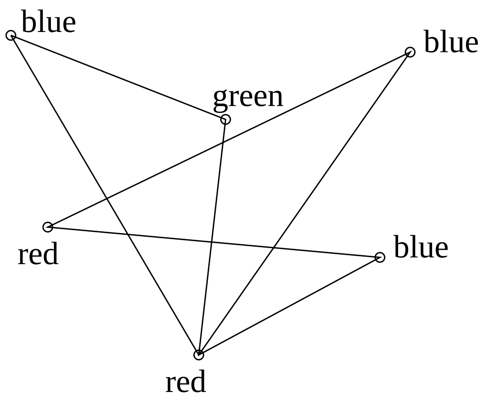

Figure 5: *This graph represents the exam scheduling problem. Each vertex stands for a course. An edge between two vertices indicates that a student is taking both courses and therefore the exams can not be scheduled at the same time. Each exam time slot is associated with a color. A schedule that creates no conflicts corresponds to a coloring of the vertices such that no adjacent vertices receive the same color.*

Proof. The proof is by induction. Let $P(n)$ be the proposition that every graph with n vertices, all with degree at most Δ, can be colored using at most $\Delta + 1$ colors.

For the base case, $P(1)$, we consider a single vertex with degree zero. In this case, $\Delta = 0$ and the algorithm requires $1 \leq \Delta + 1$ colors.

In the inductive step, assume $P(n)$ to prove $P(n+1)$. Let G' be the graph obtained from G by removing a vertex v and incident edges. No vertex in G' has degree greater than Δ, since removing a vertex and incident edges can not increase the degree of any other vertex. By induction, we can color G' with at most $\Delta + 1$ colors.

Now we need to "put back" the vertex v we removed and color it. But v has at most Δ neighbors in G. Among them, these other vertices can "use up" at most Δ colors. Thus, one of our $\Delta + 1$ candidate colors is still available to color the replaced vertex without violating the coloring rules. ■

5 Connectivity

Is a graph all in one piece or composed of several pieces? To walk from one vertex to another, how many edges must one cross? Are there many different routes or just one? These are questions about connectivity.

5.1 Basic Notions of Connectivity

In a graph, a *path* from a vertex u to a vertex v is a sequence of edges $(u, x_1), (x_1, x_2), \ldots, (x_k, v)$. A path may contain the same edge or vertex multiple times.

In a *simple path*, no vertex is crossed more than once. However, as a special case, the initial and final vertices may be the same. In other words, $u, x_1, x_2, \ldots, x_k$ are all distinct vertices, and $x_1, x_2, \ldots, x_k, v$ are all distinct vertices, but $u = v$ is permitted.

There is a path from u to v iff there is a path from v to u; one path is obtained from the other by reversing the sequence of edges. The same statement holds for simple paths.

A *circuit* is a path from a vertex back to itself, i.e. a sequence of edges $(u, x_1), (x_1, x_2), \ldots, (x_k, u)$. A *cycle* is a simple path from a vertex back to itself. Thus, a path is to a circuit as a simple path is to a cycle; both of the former may contain the same edge or vertex multiple times, but this is ruled-out in the latter.

We say two vertices are *connected* if there is a path between them.

Lemma 5.1 There is a path between two vertices iff there is a simple path between them.

So we could equally say connected means there is a simple path between them.

Proof. Consider the shortest path between the two vertices (this uses the well ordering principle, so is really induction). Suppose it is not a simple path—so some vertex is repeated. So the path has the form $a_0, \ldots, a_i, \ldots, a_j, \ldots, a_k$ where $a_i = a_j$. But then $a_0, \ldots, a_, a_{j+1}, \ldots, a_k$ is a shorter path, a contradiction. ■

A graph is *connected* if there is a path between every pair of distinct vertices. For example, referring back to Figure 4, the empty graph is disconnected, but all others shown are connected.

Connectedness is an equivalence relation. It is clearly reflexive (if we allow for length 0 paths). It is symmetric because we can reverse paths in an undirected graph. It is transitive because we can concatenate paths.

Like any equivalence relation, connectedness induces a partition of the vertices into equivalence classes. Each equivalence class is called a *connected component.* A component is a set of vertices that can all reach each other, but can't reach any other vertices.

We can also say that a connected component of a graph is a subgraph that is connected and maximal. This is, if any more vertices are added to the subgraph, then it is no longer connected. For example, every connected graph has exactly one connected component. The empty graph on n vertices has n connected components. A graph with three connected components is shown in Figure 6.

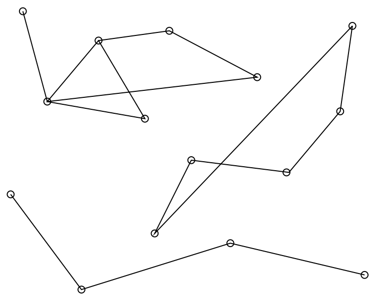

Figure 6: *This is a picture of a graph with 3 connected components.*

5.2 Cuts

Another way to look at connectivity is to see what prevents it.

Definition 5.2 An *empty cut* of a graph G is a partition of the vertices V into two nonempty sets W and $V - W$ such that no edge connects V and $V - W$ (that is, no edge has one endpoint in each set).

Lemma 5.3 A graph is connected iff it has no empty cuts.

Proof. Suppose G has an empty cut $(W, V - W)$. Let $w \in W$ and $v \in V - W$. We show by contradiction that there is no path from v to w, which means G is not connected. So suppose there is a path $v = p_0, \ldots, p_k = w$ from v to w. Consider the smallest j such that $p_j \in W$. (This is using the well ordering principle: the set of j such that $p_j \in W$ is nonempty since k is in the set, so it has a smallest element j). Note $j \neq 0$, so p_{j-1} is a vertex on the path that is not in W. So the edge (p_{j-1}, p_j) has one endpoint in W and the other in $V - W$, contradicting the claim that $(W, V - W)$ is an empty cut.

For the other direction suppose G is disconnected; we construct an empty cut. By assumption there is a pair of vertices v and w such that there is no path from v to w. Let W be the set of vertices connected to w (by a path). Then $(W, V - W)$ is an empty cut. For suppose the cut is not empty. Then there is an edge (x, y) with $x \in W$ and $y \in V - W$. But there is a path from w to x. Adding the edge (x, y) gives a path from w to y, so $y \in W$. This is a contradiction. ■

A False Theorem about Connectivity

If a graph is connected, then every vertex must be connected to some other vertex. Is the converse of this statement true? If every vertex is connected to some other vertex, then is the graph connected? The answer is no. In fact, the graph with three connected components showing in Figure 6 is a counterexample. So what is wrong with the following proof?

False Theorem 1 If every vertex in a graph is adjacent to another vertex, then the graph is connected.

Nothing helps a false proof like a good picture; see Figure 7.

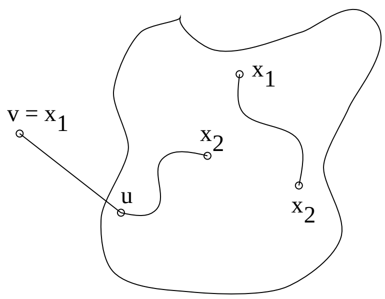

Figure 7: *This picture accompanies the false proof. Two situations are depicted. In one, vertices x_1 and x_2 both lie among the first n vertices, and so there is a connecting path by induction. In the second, $v = x_1$ and x_2 is among the first n vertices. In this case there is a connecting path because there is an edge from v to u and a path from u to x_2 by induction.*

Proof. The proof is by induction. Let $P(n)$ be the predicate that if every vertex in an n-vertex graph is adjacent to another vertex, then the graph is connected. In the base case, $P(1)$ is trivially true because there only one vertex.

In the inductive step, we assume $P(n)$ to prove $P(n+1)$. Start with an n-vertex graph; this is connected by induction. Now add a vertex v. By assumption v is connected to one of the first n vertices; call that one u.

Now we must show that for every pair of distinct vertices x_1 and x_2, there is a path between them. If both x_1 and x_2 are among the first n vertices, then there is a path by induction. Otherwise, one of the vertices is v (say x_1) and the other is among the first n vertices. These are connected by the path from x_1 to u to x_2 as shown in the figure. ■

The error is in the early statement "Start with an n-vertex graph; this is connected by induction." The induction hypothesis does not say that every n-vertex graph is connected, but only, "*if every vertex in an n-vertex graph is adjacent to another vertex,* then the graph is connected".

The core problem here is called "buildup error" and is a common source of mistakes in inductive proofs.

The name buildup error arises from the intuition that usually drives the error: that in order to create an instance of size $n+1$, you take an instance of size n and extend it by one. If you are careless, you can forget some of the possible ways a size $n+1$ problem can arise: you fail to "build" all the size $n+1$ instances, so your proof doesn't cover all the instances.

More formally, buildup error occurs when the induction hypothesis $P(n)$ is itself an implication of the form $A(n) \Rightarrow B(n)$. In this instance, $A(n)$ is the statement, "every vertex in an n-vertex graph is adjacent to another vertex" and $B(n)$ is the statement, "the graph is connected".

In the inductive step of a correct proof, we assume $A(n) \Rightarrow B(n)$ in order to prove $A(n+1) \Rightarrow B(n+1)$. In an erroneous proof, we assume $B(n)$ in order to prove $A(n+1) \Rightarrow B(n+1)$. This is precisely what happened; we assumed, "the graph is connected" rather than, "if every vertex in an n-vertex graph is adjacent to another vertex, then the graph is connected".

A True Theorem about Connectivity

If a graph has too few edges, then there can not be a path between every pair of vertices. More generally, a graph with a very small number of edges ought to have many connected components. (Remember, "many connected components" does not mean "very connected"; rather, it means "broken into many pieces"!) The following theorem generalizes these observations.

Theorem 5.4 A graph with k edges and n vertices has at least $n - k$ connected components.

For example, a graph with 0 edges and n vertices has at least $n - 0 = n$ connected components. A graph with 100 vertices and 35 edges must have 100 - 35 = 65 connected components.

The theorem is proven by induction on k. This may seem odd, because for $k > n$, the theorem says that the number of connected components is greater than some negative number! This is useless, but certainly true, so there is no harm.

Proof. The proof is by induction.

Let $P(k)$ be the predicate that an n-vertex graph with k edges has at least $n - k$ connected components. In the base case, $P(0)$ holds because an n-vertex graph with 0 edges does have at least $n - 0 = n$ components.

In the inductive step, assume that an n-vertex graph with k edges has at least $n - k$ connected components to prove that an n-vertex graph with $k + 1$ edges has at least $n - k - 1$ connected components. Let G be a graph with n vertices and $k+1$ edges. Remove one edge e. The resulting graph has n vertices and k edges. By induction there are at least $n - k$ connected components.

There are two possibilities when we replace the edge e. First, if e is incident to two vertices in the same connected component, then the number of connected components is unchanged at $n - k$. Otherwise, if e is incident to vertices in two different connected components, then the two become a single connected component. This reduces the number of connected components by one to $n-k-1$. In both cases, the number of connected components is at least $n - k - 1$ as claimed. ■

If a graph has fewer than $n - 1$ edges, then the theorem implies it has more than one connected component. Therefore, a graph must have at least $n - 1$ edges to be connected. In fact, there are graphs with $n - 1$ edges and a single connected component; the line graph of Figure 4 is an example.

Is this a complete proof? Not quite. We claimed that adding on edge reduces the number of connected components by one. This is obvious by picture, but how do we prove it formally?

Lemma 5.5 If an edge is added that connects two distinct components of a graph, the number of components is reduced by one.

Proof. Suppose the added edge is (v, w), and let $[v]$ and $[w]$ be the connected components containing v and w respectively. We claim that upon adding edge (v, w), we merge $[v]$ and $[w]$ into a single component and affect no other components.

To see that no other component is changed, suppose for contradiction that there is some vertex q *not* in the component of v or w for which adding (v, w) changes its component–that is, the set of vertices it can reach. There must be a new path from q to some other vertex r. The only way this path can be "new" is if it uses edge (v, w). So bother v and w are on the path from q to r. Suppose without loss of generality that v appears first on the path. Then there is a path from q to v that does not use edge (v, w). This path existed before we added (v, w), meaning that q is in the same component as v. This contradicts our initial assumption. ■

Corollary 5.6 Deleting a graph edge increases the number of components by at most one.

Proof. Suppose deleting edge e increased the number by more than one. We have just seen that putting

$$e$$

back can decrease the number of components by at most one. So deleting e and putting it back would give a net gain in the number of components! This is a contradiction. ■

6 Trees

Important special type of graph for CS applications.

We have just seen that $n - 1$ edges are required for connectivity. They are sometime sufficient (a line).

Let's explore the smallest connected graphs.

6.1 Definitions

What is a good notion of smallest? No wasted edges.

Definition 6.1 A *tree* is a connected graph with no cycles.

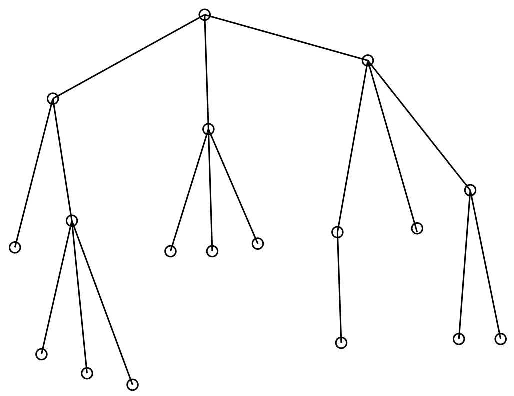

Figure 8: *This tree has 11 "leaves", which are defined as vertices of degree at most one. The remaining 7 vertices are called "internal nodes".*

The vertices in a tree can be classified into two categories. Vertices of degree at most one are called *leaves*, and vertices of degree greater than one are called *internal nodes*.

Trees are usually drawn as in Figure 8 with the leaves on the bottom. Keep this convention in mind; otherwise, phrases like "all the vertices *below*..." will be confusing. (The English mathematician Littlewood once remarked that he found such directional terms particularly bothersome, since he habitually read mathematics reclined on his back!)

Trees arise in many problems. For example, the file structure in a computer system can be naturally represented by a tree. In this case, each internal node corresponds to a directory, and each leaf corresponds to a file. If one directory contains another, there there is an edge between the associated internal nodes. If a directory contains a file, then there there is an edge between the internal node and a leaf.

Trees can actually be defined several different ways, as the following lemma shows).

Lemma 6.2 The following properties of a graph G are equivalent (each implies the others):

1. G is a connected graph with no cycles
2. G is a connected graph that is disconnected by the removal of any edge, ie, a "minimal" connected graph.
3. G is a graph with a unique simple path between every pair of vertices

Since the definitions are equivalent, any one of them could have been taken as the definition of a tree.

Theorems asserting that several mathematical statements are equivalent are quite common. One way to prove such a theorem would be to consider every pair of statements in turn and prove them equivalent. That is, we would prove $1 \Leftrightarrow 2$, $2 \Leftrightarrow 3$, and $3 \Leftrightarrow 1$.

This is overkill. A simpler approach is to show $1 \Rightarrow 2$, $2 \Rightarrow 3$, and $3 \Rightarrow 1$. These three implications are sufficient to prove every pair of statements equivalent. For example, $2 \Leftrightarrow 3$ follows because we prove $2 \Rightarrow 3$ directly, and prove $3 \Rightarrow 2$ with the two steps $3 \Rightarrow 1 \Rightarrow 2$.

Lemma 6.3 ($1 \implies 2$) A connected graph with no cycles is disconnected by the removal of any edge.

Proof. By contradiction. Suppose we can delete edge (u, v) without disconnecting the graph. This means there is a path from u to v that doesn't use (u, v), and thus a simple path. Adding edge (u, v) to a simple path that doesn't use (u, v) creates a cycle. This contradicts the claim of no cycles. ∎

Lemma 6.4 ($3 \implies 1$) A graph with a unique simple path between any pair of nodes is connected and has no cycles.

Proof. It clearly meets the definition of connectivity. To see there are no cycles we argue by contradiction. Suppose we have a (simple) cycle $v_1, \ldots, v_k, v_1$. Then there are 2 simple paths from v_1 to v_k, namely $v_1, \ldots, v_k$ and the edge (v_1, v_k). ∎

Lemma 6.5 ($2 \implies 3$) A connected graph that is disconnected by the removal of any edge has a unique simple path between any pair of vertices.

Proof. Suppose there are two different simple u-v paths. Call one P_1: $u = x_0, x_1, \ldots, x_k = v$. Call the other P_2: $u = y_0, \ldots, y_l = v$. Let i be the smallest such that $x_i \neq y_i$ (why does this have to exist?). Claim: we can remove (y_{i-1}, y_i) without disconnecting G. To see this, consider a pair of vertices z and w. There's a simple path between them at the start. It remains to show there's still a path (doesn't need to be simple) after removing (y_{i-1}, y_i). We use proof by cases.

Case 1: (y_{i-1}, y_i) wasn't on the path. Then the path remains after removing (y_{i-1}, y_i).

Case 2: (y_{i-1}, y_i) was on the path. WLOG, y_{i-1} is encountered first. So the path looked like $z, \ldots, y_{i-1}, y_i, \ldots, w$. So there's a different path: go from z to y_{i-1} on the first part of z-w path, then from y_{i-1} to u on the first part of P_2, then from u to v on P_1, then from v to y_i on the second part of P_2, then from y_i to w on the second part of the z-w path. ∎

6.2 Spanning Trees and Forests

We've just argued that the "minimal" (poset, under edge containment) connected graphs are trees. It turns out that every connected graph actually contains one:

Definition 6.6 A *spanning tree* of a graph is a subset of the graph edges that forms a tree on all the graph nodes.

Notice that the notion "tree" is an intrinsic concept (cares only about the nodes adjacent to the edges of the tree), while "spanning tree" is extrinsic (cares about whether you have all the graph's nodes).

Lemma 6.7 Every connected graph contains a spanning tree.

Proof. Well ordering. Consider the counterexample with the fewest edges. If you cannot delete any edge and stay connected, it is a tree (3rd definition of tree). So you can delete an edge and get a smaller counterexample. Contradiction. ■

Some graphs aren't connected; they don't have spanning trees. Instead, they have spanning *forests*:

Definition 6.8 A *forest* is any acyclic graph.

Lemma 6.9 Each connected component of a forest is a tree.

Proof. Each connected component of a forest is connected and acyclic. ■

Definition 6.10 A *spanning forest* for a graph G is a forest that connects every pair of vertices that were connected in G,

People take some liberties with "tree:" they call a set of edges a tree if it connects its vertices and is acyclic. So we talk about a forest as being "a collection of trees" even though each tree is only on vertices of a connected component, not on all graph vertices.

In particular, a spanning forest G is just a collection of trees, one spanning each connected component of G.

6.3 Counting Tree Edges

Definition 6.11 A *leaf* is a node of degree 1.

Lemma 6.12 Any tree with more than one vertex has a leaf.

Proof. Consider a longest simple path $v_1, \ldots, v_k$ (why does such exist?). I claim v_1 is a leaf. Proof: suppose not. Then it has a second neighbor x. Is $x = v_i$ for some i? No, that would give a cycle. Thus $x, v_1, \ldots, v_k$ is a longer simple path, a contradiction. ■

The following is "obvious," but I'll write down a proof for those who want to read it.

Lemma 6.13 If we remove a leaf and its edge from a tree, we get a tree.

Proof. Suppose we remove leaf v and its one incident edge (u, v). We need to show the remaining graph is still connected (it is clearly still acyclic). Consider the unique simple path between two other vertices x and y. Can (u, v) be on this path? No, because (u, v) is the only edge incident on v, so we would have to traverse it twice to enter and leave v. This would make a non-simple path.

Since (u, v) is not on the path from x to y, there is still a path after we delete (u, v). This is true for all pairs x and y, so the remaining graph is connected. ■

Lemma 6.14 Any tree on n vertices has $n-1$ edges.

Proof. (Trivial base case). Inductive step: Start with an arbitrary tree with $n+1$ nodes. Remove one node and its incident edge. By the previous lemma, the result is a tree (with n nodes). By the IH, this tree has $n-1$ edges. So the original tree has n edges. ■

Lemma 6.15 Any n vertex connected graph G with $n-1$ edges is a tree.

Proof. We proved earlier that to be connected (have one connected component) a graph must have at least $n-1$ edges. So if we remove any edge from G, it becomes disconnected. This is one of the definitions of a tree. ■

7 Euler Tours

The Koenigsberg bridge problem: can we cross every bridge in Koenigsberg exactly once, without repeats, returning to our starting point?

```
          |    |           |     /
      ----|----|---   -----|----/
     /    |    |   \ /     |
----<             --X--
     \    |    |   / \     |
      ----|----|--/   \----|----\
          |    |           |     \
```

This problem has a graph representation.

Euler, 1736, proved NO. So we call such tours "Euler Tours."

Theorem 7.1 If undirected G has Euler tour, then G is connected and every vertex has even degree.

Proof. Direct every edge according to tour. We enter a vertex as many times as we leave it. So every vertex must have indegree equal to outdegree. So total even. ■

100 years later, Hierholzer proved the converse (trouble with coming second—no one remembers your name!)

Theorem 7.2 If G is connected and has even degree for every node, then it has an Euler tour.

Lemma 7.3 If G has even degree and at least one edge, then it has a circuit.

Proof.

If connected, handshaking proves n edges, so not a tree, so cycle. But this also applies to unconnected: consider a component that contains at least one edge and apply the above. (There must be such a component because every edge of the graph must be in one such component). ■

Proof. (Of main result) By contradiction.

Idea: Let C be the longest circuit without any repeated edges. Delete it. Delete all isolated vertices. Since C is not an Euler tour, some vertices remain. Some vertex u that remains is on C. Why? Because G is connected. Take any remaining vertex v, it has a path to a vertex in C. The first time we reach an edge of C, we stop. This gives u. Now, by the lemma, find a circuit containing u. Attach to C, giving a bigger circuit. (A contradiction.) ■

8 Hamiltonian Paths

A slightly different question: is there a tour that traverses every *vertex* exactly once?

Such a cycle is called a *Hamiltonian Cycle.*

In a small variant, a simple path that traverses every vertex exactly once is a *Hamiltonian Path.*

Small change, but suddenly, instead of answer being trivial, becomes very complicated. It is believed that no computer program can answer this question efficiently (it is NP-complete).