Notes 7a

These notes contain material covered in tutorial as well as half a lecture.

1 Partial Orders

Partial orders are a particular type of binary relation. They are very important in computer science. Applications to task scheduling, database concurrency control, logical time in distributed computing.

1.1 Definitions for Partial Orders

Definition 1.1 A binary relation $R \subseteq A \times A$ is a **partial order** if it is reflexive, transitive, and antisymmetric.

Quick review:

reflexive: aRa

transitive: $aRb \wedge bRc \Rightarrow aRc$

symmetric: $aRb \Rightarrow bRa$

antisymmetric: $aRb \wedge bRa \Rightarrow a = b$. i.e., $\forall a \neq b, aRb \Rightarrow \neg bRa$. This means *never* symmetric!

For comparison, recall that a relation that is reflexive, transitive, and symmetric is an equivalence relation.

The reflexivity, antisymmetry and transitivity properties are abstract properties that generally describe "ordering" relationships such as "less than or equal to." So we often write an ordering-style symbol like $\preceq$ instead of just a letter like R, for a p.o. relation. This lets us use notation similar to for $\leq$. For example, we write $a \prec b$ if $a \preceq b$ and $a \neq b$. Similarly, we write $b \succeq a$ as equivalent to $a \preceq b$.

Note this could be misleading – note that $\geq$ is a p.o. on natural numbers, as well as $\leq$. If we use the $\preceq$ symbol for $\geq$, things look really funny. In cases like this, better to use R.

A partial order is always defined on some set A. The set together with the partial order is called a "poset":

Definition 1.2 A set A together with a partial order $\preceq$ is called a **poset** $(A, \preceq)$.

Examples of posets:

- $A = \mathbb{N}, R = \leq$, easy to check reflexive, transitive, antisymmetric

- $A = \mathbb{N}, R = \geq$, same.
- $A = \mathbb{N}, R = <$, NOT because not reflexive
- $A = \mathbb{N}, R = \mid$ (divides), easy to check reflexive, transitive, antisymmetric
- $A = P(\mathbb{N}), R = \subseteq$, check reflexive: $S \subseteq S$, transitive: $S \subseteq S' \wedge S' \subseteq S'' \Rightarrow S \subseteq S''$, antisymmetric: $S \subseteq S' \wedge S' \subseteq S \Rightarrow S = S'$.

The following are *not* posets:

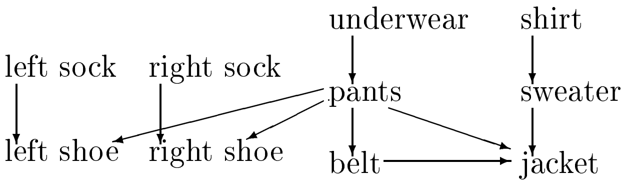
- $A =$ "set of all computers in the world", $R =$ "is (directly or indirectly) connected to". This is not a p.o. because it is not true that $aRb \wedge bRa \Rightarrow a = b$. In fact, it is symmetric, transitive. Equivalence relation.
- $A =$ "the set of all propositions", $R = \Rightarrow$, is not antisymmetric. Not symmetric either, so not equivalence relation.

1.2 Directed Acyclic Graph Definition

A common source of partial orders in computer science is in "task graphs". You have a set of tasks A, and a relation R in which aRb means "b cannot be done until a is finished". Implicitly, "if all the things that point at b are done, I can do b."

This can be nicely drawn as a graph. We draw an arrow from a to b if aRb.

For example, below is a graphs that describes the order in which one would put on clothes. The set is of clothes, and the edges say what should be put on before what.



The "depends on" graph imposes an ordering on tasks. But is it a partial order? No, because it isn't reflexive or transitive. But there is a natural extension: the reflexive transitive closure *is* a partial order. It gives the relation "must be done before."

Wait, we know the reflexive transitive closure is reflexive and transitive, but is it antisymmetric? What if I add a relation edge from belt to underwear? In that case my dependency graph stops making sense: there is no way to get dressed!

What goes wrong? A cyclic dependency.

Definition 1.3 A *cycle* is a walk that ends where it started (i.e., the last vertex equals the first).

Definition 1.4 A *directed acyclic graph (DAG)* is a directed graph with no cycles.

Theorem 1.5 Any partial order $\preceq$, with the reflexive "loops" $a \preceq a$ removed, is a DAG.

Proof. By contradiction. Suppose the resulting relation had a cycle $a_1, \ldots, a_k, a_1$ where $a_k \neq a_1$. By transitivity (and induction on the length of the path) we have $a_1 \preceq a_k$ and $a_k \preceq a_1$. Thus (since $\preceq$ is a partial order) $a_k = a_1$. This means the final step in our path is a loop on a_1. But we removed the loops so this step does not exist, a contradiction. ∎

Theorem 1.6 The transitive reflexive closure of any DAG is a partial order.

That is, given a DAG, the relation "there is a path of length 0 or greater from a to b in the DAG" is a partial order.

Proof. The given relation is reflexive by definition, and transitive because, as we saw last time, we can "concatenate" paths from a to b and from b to c to get an a-c path. So we just have to prove antisymmetry. So suppose there is a path from a to b and a path from b to a but $a \neq b$. Then concatenating these paths gives a path from a to a that contains a second vertex $b \neq a$—that is, a cycle. This contradicts the claim that we started with a DAG. ∎

1.3 Hasse Diagrams

If we draw the graph of a poset, we may get *lots* of edges (consider the $\leq$ order on some natural numbers). But a lot of those edges are "implicit" from the transitivity of the order. We can get a much less messy picture by leaving out these arrows.

Consider the clothing example. Under the transitive closure, underwear precedes belt. But we don't need to see that edge to know it is there. Similarly loops (which express reflexive relationships) aren't shown.

Definition 1.7 A **Hasse diagram** for a poset $(A, \preceq)$ is a graph:

- whose vertices are the elements of A,
- whose edges (arrows) represent pairs (a, b), where $a \leq b$ but $a \neq b$,
- that contains no "transitive edges",
- for which all pairs in $\preceq$ are obtainable by transitivity from edges in the diagram.

It's a "transitive disclosure" of the poset.

To convert the clothing diagram above to an "official form" Hasse diagram, we must remove the pants $\rightarrow$ jacket arrow.

1.4 Partial vs. Total Orders

A partial order is called *partial* because it is not necessary that an ordering exist between every pair of elements in the set. E.g., Lshoe and Rshoe have no prescribed ordering between them above. In the "is a subset of" partial order, for two sets, it's not necessary that either be a subset of the other.

When there is no prescribed order between two elements we say that they are "incomparable".

Definition 1.8 a and b are **incomparable** if neither $a \preceq b$ nor $b \preceq a$.

Definition 1.9 a and b are **comparable** if $a \preceq b$ or $b \preceq a$.

E.g., for subsets of $\mathbb{N}$, $\{1, 2, 3\}$ and $\{2, 3, 4\}$ are incomparable. However, a partial order need not have incomparable elements. As a special case, we can have a p.o. in which there is a specified order between every pair of elements. Such partial orders are called "total orders."

Definition 1.10 A poset $(S, \preceq)$ is **totally ordered** if $(\forall a, b \in S)[a \preceq b \vee b \preceq a]$.

Examples:

- $S = \mathbb{N}, \preceq = \leq$
- $S = \mathbb{N}, \preceq = |$, NOT because neither $3|5$ nor $5|3$
- $S = P(\mathbb{N}), \preceq = \subseteq$, NOT because neither $\{3\} \subseteq \{5\}$ nor $\{5\} \subseteq \{3\}$
- Lexicographic order on pairs of natural numbers, defined by: $(a_1, a_2) \preceq (b_1, b_2)$ if and only if either $a_1 < b_1$, or else $a_1 = b_1$ and $a_2 \leq b_2$. This is called "lexicographic" because it's like the dictionary order.

The Hasse diagram of a total order looks like a line.

This has been the basic material. The Book has a bit more: minimal, maximal elements (nothing smaller, nothing larger)
lower bounds, upper bounds (for a subset – $\leq$ everything in the subset, or $\geq$)
lub, glb,
lattices.
Read these parts.

1.5 Topological Sorting

Sometimes when we have a partial order, e.g., of tasks to be performed, we want to obtain a consistent total order. That is, an order in which to perform all the tasks, one at a time, so as not to conflict with the precedence requirements.

The task of finding an ordering that is consistent with a partial order is known as "topological sorting". I think because the sort is based only on the shape (topology) of the poset, and not on the actual values.

Definition 1.11 A **topological sort** of a finite poset $(A, \preceq)$ is a total ordering of all the elements of A, $a_1, a_2, \cdots, a_n$ in such a way that $\forall i < j$, $a_i \not\succ a_j$. That is, either $a_i \preceq a_j$ or a_i and a_j are incomparable.

For example, underwear, shirt, Lsock, Rsock, pants, sweater, Lshoe, Rshoe, belt, jacket is a topological sort of how to dress.

One of the nice facts about posets is that such an ordering always exists and is even easy to find:

Theorem 1.12 Every finite poset has a topological sort.

The basic idea to prove this theorem is to pick off a "first" element and then proceed inductively.

Definition 1.13 A **minimal** element a in a poset $(A, \preceq)$ is one for which $(\forall b \in A)[a \preceq b \vee$ a and b are incomparable]. Equivalently, it is an element a for which no $b \prec a$.

Notice that there can be more than one minimal element. There are 4 in the clothes example: Lsock, Rsock, underwear, shirt.

Lemma 1.14 Every finite poset $(A, \preceq)$ has a minimal element.

We'll do two proofs of this theorem.

Proof. Suppose no element of A is minimal. So every element has one preceding it. We derive a contradiction. Take some element $a \in A$ and define the following sequence recursively. $a_0 = a$, and for $i > 0$, a_i is some element that precedes a_{i-1}.

This definition is a bit odd because it is nondeterministic. But it doesn't matter which preceding element you pick at each step. The key is that it defines some a_i for every i.

Now note that the sequence $a_{n+1}, \ldots, a_0$ is a walk of length $n+1$. As we saw before, this means some element is repeated: $a_i = a_j$ for some $i < j$. But this means $a_i, \ldots, a_j$ is a cycle, which is impossible in a poset. So we have a contradiction. ■

If you didn't like that "repeated elements" argument, here is a clean proof by induction. The basic idea is to pick a "smallest" element to start and then proceed recursively.

If you didn't like that "repeated elements" argument, here is a clean proof by induction.

Proof. By induction on the number of elements in the poset. The base case is clear. Suppose it is true for all size-n posets. Consider any size $n+1$ poset. Delete some element a from the poset. This leaves a poset on n elements (you can verify this). So the smaller poset has a minimal element m.

Now consider two cases. First suppose $a \not\preceq m$. The m is a minimal element of the whole poset. Now suppose $a \preceq m$. Then I claim a is a minimal element of the whole poset.

To prove this claim we argue by contradiction. If a is not minimal then some $b \preceq a$. But then, but transitivity, since $a \preceq m$, we have $b \preceq m$. But this would prevent m from being minimal in the poset with a removed. This contradicts our finding of m. ■

Note that an infinite poset might have no minimal element. Consider $\mathbb{Z}$.

Now we can prove the existence of the topological sort.

Proof. By (ordinary) induction. Let $P(n)$ be "any poset with n elements has a topological sort". Base case ($P(1)$): for a poset with one element the one element makes a topological sort.

Inductive step: Assume ($P(n)$). Consider a poset of $n+1$ elements. By the previous lemma it has a minimal element u. Consider the same relation on $A - \{u\}$. It is easy to check that it is still reflexive, transitive, and anti-symmetric. So by the inductive hypothesis, $A - \{u\}$ has a topological sort. We put u first and now have a topological sort on the whole thing. (According to the definition of topological sort – u can't be in the wrong order w.r.t. any of the later elements.)

Thus every finite poset has a topological sort, by induction. ■

Example 1.15 Consider the dressing example. Construct an ordering by picking one item at a time. At each step, look at the remaining p.o. for remaining elements. Lsock, shirt, sweater, Rsock, underwear, pants, Lshoe, belt, jacket, Rshoe

E.g., subsets of $\{1, 2, 3, 4\}$
$\emptyset$ is the unique minimal element, then have choices, e.g., do:
$\{1\}$, $\{2\}$, $\{1, 2\}$
$\{3\}$, $\{1, 3\}$, $\{2, 3\}$, $\{1, 2, 3\}$,
$\{4\}$, $\{1, 4\}$, $\{2, 4\}$, $\{3, 4\}$,
$\{1, 2, 4\}$, $\{1, 3, 4\}$, $\{2, 3, 4\}$, $\{1, 2, 3, 4\}$

I'm not sure if infinite posets always have topological sorts—I think this might be equivalent to the well ordering property, which is equivalent to the axiom of choice, a hotly debated mathematical axiom.

1.6 Parallel Task Scheduling

When elements of a poset are tasks that need to be done and the partial order is 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, let's say all the tasks take the same amount of time and all the processors are identical.)

So, given a finite poset of tasks, how long does it take to do them all, in an optimal parallel schedule? We can use partial order concepts to analyze this problem.

On the clothes example, we could do all the minimal elements first (Lsock, Rsock, underwear, shirt), remove them and repeat. (We'd need lots of hands, or maybe dressing servants). We can do pants and sweater next, and then Lshoe, Rshoe, and belt, and finally jacket.

We 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 1.16 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 $\neq$).

That is, a sequence is a simple path in the partial order.

Clearly, the parallel time $\geq$ length of any chain. For if we used less time, then two tasks in the chain would have to be done at the same time. But by definition of chains 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 1.17 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, we 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 1.18 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 1.19 parallel time = length of longest chain

So it remains to prove the partition theorem:

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

Let's see why this works. It gives just t sets, because the longest path is length t. We need to show $(\forall i)(\forall a \in A_i)$
$(\forall b \preceq a, b \neq a)[b \in A_1 \cup A_2 \cup \cdots \cup A_{i-1}]$.

The proof is 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. This 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.

Corollary 1.20 If t is the length of the longest chain in a poset $(A, \preceq)$ then A can be partitioned into t antichains.

Definition 1.21 An **antichain** is a set of incomparable elements.

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

2 Graphs

Just like DAGs are a nice way to think about partial orders, graphs in general are a nice way to think about relations. They give a picture of the relation, which can be much more revealing than a list of tuples. They motivate the study of other properties.

2.1 Definitions

We'll start with a bunch of definitions. Many are inherited from the definition of relations. Others are slightly modified for convenience. Yet others are new.

2.2 Simple Graphs

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 a pair of distinct vertices.

We are going to concentrate on *undirected* graphs. These correspond to symmetric relations. If edge (u, v) is present, edge (v, u) is also present. Therefore, it simplifies matters to think of these two edges as one and the same. This faintly violates the notion of a Cartesian pair (where order matters) and really picky mathematicians denote such an undirected edge by the unordered *set* $\{u, v\}$, but this rapidly fills up your paper with hard to read $\{\}$ symbols.

Graphs are also sometimes called *networks.* Vertices are also sometimes called *nodes.* Edges are sometimes called *arcs.* Graphs can be nicely represented with a diagram of dots and lines as shown in Figure 1

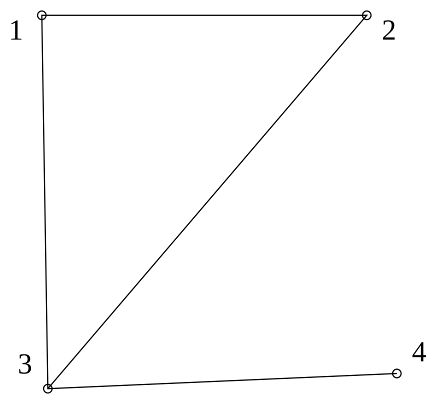


Figure 1: *This is a picture of a graph $G = (V, E)$. There are four vertices and four edges. The set of vertices V is $\{1, 2, 3, 4\}$. The set of edges E is $\{(1, 2), (1, 3), (2, 3), (3, 4)\}$. Vertex 1 is adjacent to vertex 2, but is not adjacent to vertex 4.*

As noted in the definition, each edge $(u, v) \in E$ is a pair of distinct vertices $u, v \in V$. 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.

A computer network is can be modeled nicely as a graph. In this instance, 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.

2.3 Not-So-Simple Graphs

There are actually many variants on the definition of a graph. The definition in the preceding section really only describes *simple undirected graphs*. There are many ways to complicate matters.

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.

Self-Loops

In a simple graph, each edge is a pair of distinct vertices. A *self-loop* is an edge that connects a vertex to itself. Figure 2 depicts a multigraph with self-loops.

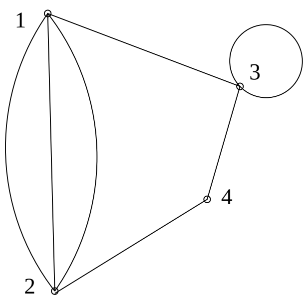


Figure 2: *This is a picture of a multigraph with a self-loop. In particular, there are three edges connecting vertices 1 and 2 and there is a self-loop on vertex 3.*

Directed Graphs

In a *directed graph* or *digraph*, edges are regarded not as lines, but as arrows. (The technical difference is that in a simple graph each edge is an unordered pair of vertices, whereas in a directed graph each edge is an ordered pair.) Figure 3 depicts a directed graph.

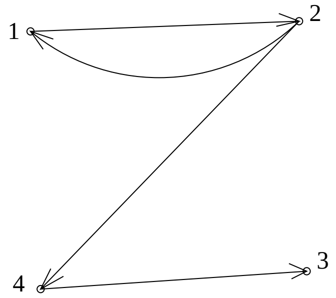


Figure 3: *This is a picture of a directed graph or digraph.*

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 infinity or zero.)

2.4 Graphs in the Real World

There are many real-world phenomena other than computer networks that can be described nicely by graphs. Here are some examples:

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, Atilla the Hun advised that you should always to 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.

2.5 Standard Graphs

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

2.6 Adjacency Matrices

We defined a graph in terms of a set of vertices and a set of edges. In a computer it is often represented by an *adjacency matrix*—another name for the matrix representation of the underlying relation.

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

2.7 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).

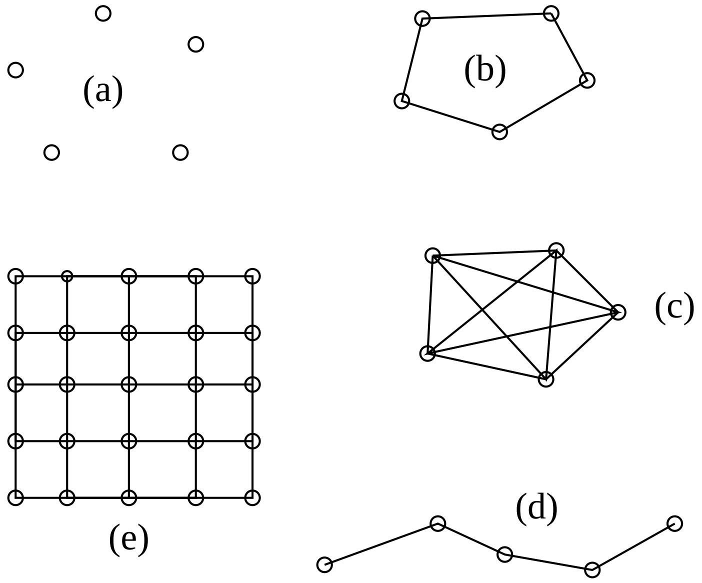


Figure 4: *The types of graph shown here are so common that they have names. (a) The empty graph on five vertices, E_5. An empty graph has no edges at all. (b) The cycle on five vertices, C_5. (c) The complete graph on five vertices, K_5. A complete graph has an edge between every pair of vertices. (d)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

2.8 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 2.1 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 2.2 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. ∎

3 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.

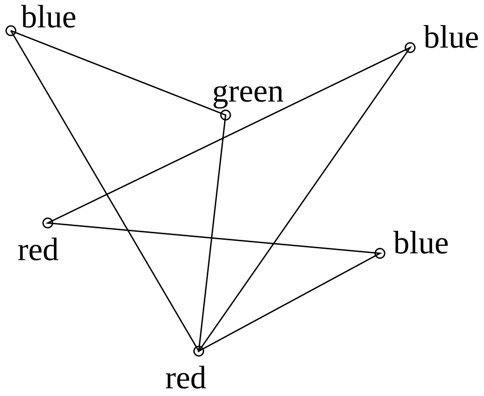


Figure 5: *This graph represents the exam scheduling problem. Each vertex stands for a course. An edge between two vertices indicates that a student is taking both courses and therefore the exams can not be scheduled at the same time. Each exam time slot is associated with a color. A schedule that creates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

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 3.1 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:

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.

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

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.
■

4 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.

4.1 Basic Notions of Connectivity

In a graph, a *walk* 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 walk may contain the same edge or vertex multiple times.

In a *simple path*, no vertex or 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 walk from u to v iff there is a walk 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 walk 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 walk between them.

Lemma 4.1 There is a walk 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 walk 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 walk, 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.

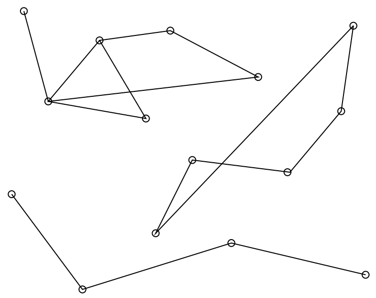


Figure 6: *This is a picture of a graph with 3 connected components.*

4.2 Cuts

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

Definition 4.2 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 4.3 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.

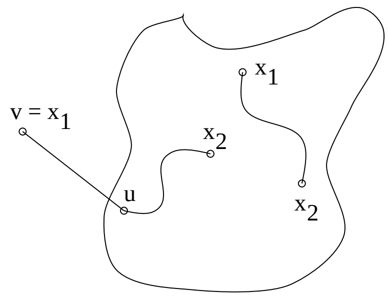


Figure 7: *This picture accompanies the false proof. Two situations are depicted. In one, vertices x_1 and x_2 both lie among the first n vertices, and so there is a connecting path by induction. In the second, $v = x_1$ and x_2 is among the first n vertices. In this case there is a connecting path because 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 4.4 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 4.5 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 4.6 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. ∎

5 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.

5.1 Definitions

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

Definition 5.1 A *tree* is a connected graph with no cycles.

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).

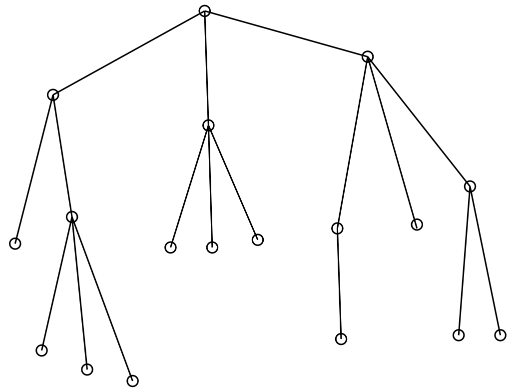


Figure 8: *This tree has 11 "leaves", which are defined as vertices of degree at most one. The remaining 7 vertices are called "internal nodes".*

Lemma 5.2 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 5.3 ($1 \implies 2$) 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 5.4 ($3 \implies 1$) 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 5.5 ($2 \implies 3$) 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. ■

5.2 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 5.6 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 5.7 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 5.8 A *forest* is any acyclic graph.

Lemma 5.9 Each connected component of a forest is a tree.

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

Definition 5.10 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.