Massachusetts Institute of Technology
6.042J/18.062J: Mathematics for Computer Science
Professors Nancy Lynch and Shlomi Dolev

Lecture 6
September 26, 2000

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 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.

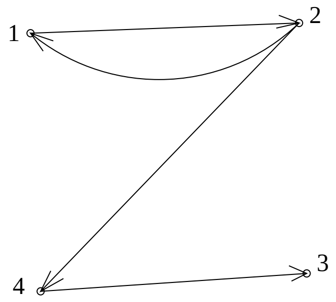

Figure 1: *This is a picture of a directed graph or digraph.*

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, $w(u, v)$ might be the distance between airports u and v, or the flying time between them, or even the airfare.

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

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

In a digraph, a *path* from a vertex u to a vertex v is a sequence of vertices $u = u_0, u_1, \ldots, u_k = v$ such that each consecutive pair (u_i, u_{i+1}) form an edge of the digraph. We allow a path of length 0, which is just a single vertex u_0. The *length* of the path is k (the number of edges). A path may contain the same edge or vertex multiple times.

This is a little different from Rosen's definition: He insists that paths have at least two vertices (one edge). Our definition is consistent, for example, with the Cormen, Leiserson, Rivest textbook used in 6.046.

Examples

A path is *simple* if no vertex appears more than once, with the possible exception that the first vertex may be the same as the last. In other words, $u_0, u_1, u_2, \ldots, u_{k-1}$ are all distinct vertices, and $u_1, u_2, \ldots, u_k$ are all distinct vertices, but $u_0 = u_k$ is permitted.

Examples

Again, this definition is different from the one used in Rosen: he defines "simple" to mean "no repeated edges".

Theorem: If $|V| = n$, then any simple path has length at most $n - 1$.

Theorem: If $|V| = n$, and there is a path from u to v in G, then there is a simple path from u to v (which must be of length at most $n - 1$).

Proof. Consider the shortest path from u to v (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 $u_0, \ldots, u_i, \ldots, u_j, \ldots, u_k$ where $u_i = u_j$. But then $u_0, \ldots, u_i, u_{j+1}, \ldots, u_k$ is a shorter path, a contradiction. ∎

Examples

Circuit (also known as a *cycle*) is a path that begins and ends at the same vertex. (Usually, the degenerate case of a one-node, length 0 path is not considered to be a circuit, but we won't exclude it—will have to take care of boundary cases in proofs, though.)

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

Now choose any edge in H that is incident on u (it doesn't matter if it's ingoing or outgoing to u. 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, u. 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. Examples of digraphs that are unions of strongly connected components, others that aren't (have multiple sccs but with one-way connections from some components to others).

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 (u, v) is present, edge (v, u) is also present. In an irreflexive relation, no edge (u, u) 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 u to v, there is another edge from v to u.

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

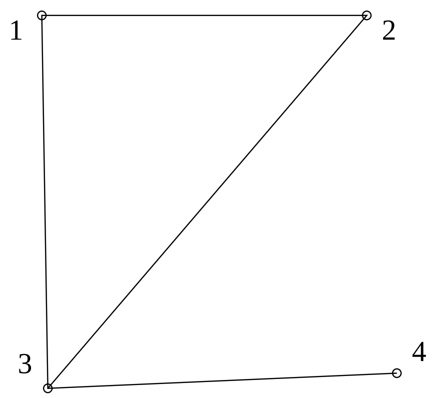

Figure 2: *This is a picture of a graph $G = (V, E)$. There are four vertices and four edges. The set of vertices V is $\{1, 2, 3, 4\}$. The set of edges E is $\{(1, 2), (1, 3), (2, 3), (3, 4)\}$. Vertex 1 is adjacent to vertex 2, but is not adjacent to vertex 4.*

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.

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.

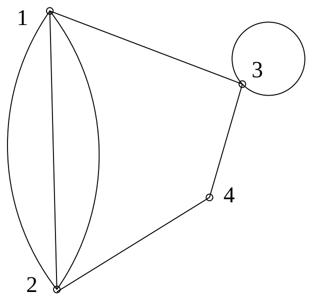

Figure 3: *This is a picture of a multigraph with a self-loop. In particular, there are three edges connecting vertices 1 and 2 and there is a self-loop on vertex 3.*

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 $w(u, v)$ is the entry at row u and column v.

3.3 Standard Types of Graphs

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

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).

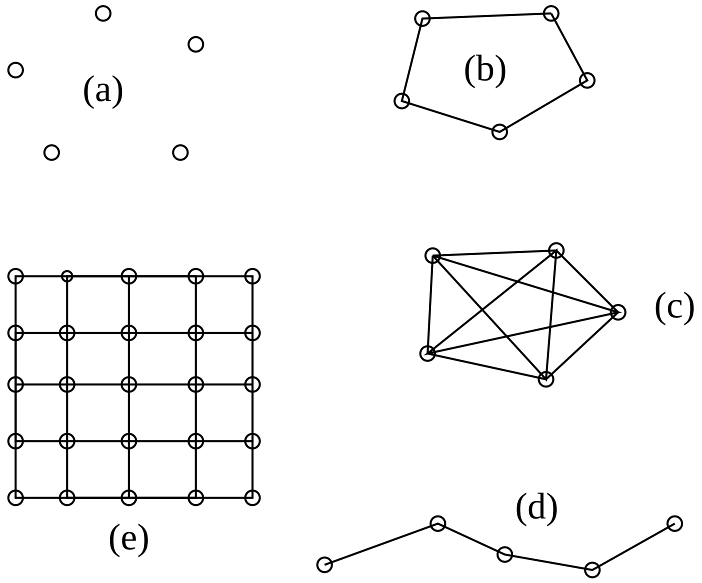

Figure 4: *The types of graph shown here are so common that they have names. (a) The empty graph on five vertices, E_5. An empty graph has no edges at all. (b) The cycle on five vertices, C_5. (c) The complete graph on five vertices, K_5. A complete graph has an edge between every pair of vertices. (d)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

3.4.2 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.

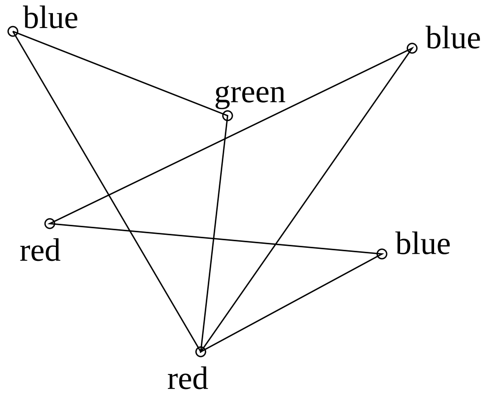

Figure 5: *This graph represents the exam scheduling problem. Each vertex stands for a course. An edge between two vertices indicates that a student is taking both courses and therefore the exams can not be scheduled at the same time. Each exam time slot is associated with a color. A schedule that creates 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 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:

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.
■

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

Definitions similar to those for digraphs.

In a graph, a *path* from a vertex u to a vertex v is a sequence of vertices $u = u_0, u_1, u_2, \ldots, u_k = v$, where each (u_i, u_{i+1}) is an edge of the graph. A path may contain the same edge or vertex multiple times.

In a *simple path*, no vertex appears more than once. However, as a special case, the initial and final vertices may be the same. In other words, $u_0, u_1, u_2, \ldots, u_{k-1}$ are all distinct vertices, and $u_1, u_2, \ldots, u_k$ are all distinct vertices, but $u_0 = u_k$ is permitted.

Because we are working with undirected graphs, 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*, or *cycle* is a path from a vertex back to itself, i.e. a path $u = u_0, u_1, u_2, \ldots, u_k = v$ where $u = u_0 = u_k = v$.

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. Like the analogous proof for the digraph case. ■

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.

Theorem: In any undirected graph, connectedness is an equivalence relation.

Proof It is reflexive because we allow 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 each of which can reach all the others, and none of which can 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.

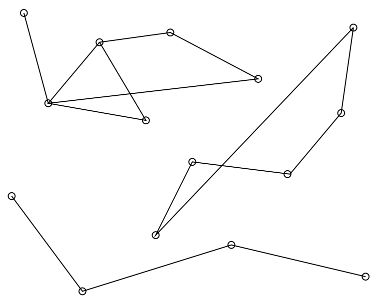

Figure 6: *This is a picture of a graph with 3 connected components.*

Examples