Massachusetts Institute of Technology | Lecture 12
6.042J/18.062J: Mathematics for Computer Science | March 19, 1997
Professor David Karger

Notes 12

1 Partial Orders

We are covering a series of mathematical structures:
natural numbers and integers,
sets,
relations (including equivalence relations),
functions.

Next kind of structure: partial order.
A particular type of binary relation.

Important in computer science.
Applications to task scheduling,
Database concurrency control,
Logical time in distributed computing

1.1 Definitions

1.1.1 Partial Order Definition

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., $\forall a \neq b, aRb \Rightarrow \neg bRa$. Means NEVER symmetric!

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

Examples of partial orders:

- $A = \mathbb{N}, R = \leq$, easy to check reflexive, transitive, antisymmetric
- $A = \mathbb{N}, R = \geq$, same.
- $A = \mathbb{N}, R = <$, NOT because not reflexive
- $A = \mathbb{N}, R = |$ (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: $S \subseteq S' \wedge S' \subseteq S \Rightarrow S = S'$.
- $A =$ "set of all computers in the world", $R =$ "is (directly or indirectly) connected to".
  NOT a p.o. because it is not true that $aRb \wedge bRa \Rightarrow a = b$.
  In fact, it is symmetric, transitive. Equivalence relation.
- $A =$ "set of all propositions", $R = \Rightarrow$, NOT because not antisymmetric. Not symmetric either, so not equivalence relation.
- $A =$ some set of tasks, $R =$ "must be done before", according to some task specification.
  Antisymmetry: Does $aRb \wedge bRa \Rightarrow a = b$?
  Hopefully, because otherwise a, b can never be done! So we will only deal with task specifications with this property.
  Transitive.
  BUT not reflexive. If we want to treat this as a partial order, we must make a small modification of R to "either $=$ or must be done before".

1.1.2 Reflexive and Transitive Closures

The construction we just did is called "taking the reflexive closure".
In general, if we have a binary relation R that is antisymmetric and transitive (but not necessarily reflexive), we can turn it into a partial order simply by adding in all pairs of the form (a, a).
Adding in these pairs does not mess up the antisymmetry or transitivity properties.
(Check this.)

A related construction is "taking the transitive closure".
It starts with a relation R that is antisymmetric and reflexive, but not necessarily transitive.
However, it has no "cycles", i.e., no instances where $(a_1, a_2) \in R$,
$(a_2, a_3) \in R$,
... $(a_{k-1}, a_k) \in R$, and $(a_k, a_1) \in R$,
where each pair consists of distinct elements, and where $k \geq 2$.

E.g., "S is a subset of S', with either the same number of elements (so $S = S'$) or else exactly one fewer".
This is reflexive, antisymmetric, not transitive.
But no cycles.

Then add in all the transitively-determined pairs.
What are these?
Well, if (a, b) and (b, c) are in R, then must add in (a, c).
But not enough just to do this for the original pairs.
Must repeat, e.g., if $(c, d) \in R$, must now add in (a, d), etc.
In general, must add in any pair (a_1, a_k) such that all the pairs (a_1, a_2), (a_2, a_3), ..., (a_{k-1}, a_k) are in R.
Claim: this is enough to make the result transitive.
(For any two pairs assembled in this way that match up, the transitive combination is also included.)
Moreover, adding all these pairs doesn't mess up reflexivity (obviously) or antisymmetry (because the original relation had no cycles).
So the result is a partial order.

E.g., in the subset example above, transitive closure adds in all subset pairs without any restriction on the number of elements in the set difference.

Could also start with not-necessarily reflexive, but antisymmetric and cycle-free, and take the "reflexive transitive closure" to turn it into a p.o.
Just do both of the above constructions.

E.g., "S is a subset of S', with S missing exactly one element from S'."

1.1.3 No Cycles

In taking the transitive closure, we had to start with a relation without cycles.
In fact, it turns out that *every* partial order is cycle-free:

Theorem 1.2 A partial order has no cycles of length ≥ 2, i.e., no sequences of pairs of the form (a_1, a_2), (a_2, a_3), $...(a_{k-1}, a_k)$, and (a_k, a_1), $k \geq 2$, all in R, each pair consisting of distinct elements.

Proof. By contradiction.
Assume that such a cycle exists in R. Then by transitivity (maybe applied several times), $(a_1, a_k) \in R$. Since also $(a_k, a_1) \in R$, antisymmetry implies that $a_1 = a_k$, which is a contradiction. ∎

1.1.4 Posets and Hasse Diagrams

The reflexivity, antisymmetry and transitivity properties are abstract properties that generally describe "ordering" relationships.
So we often write an ordering-style symbol like $\preceq$ instead of just a letter like R, for a p.o. relation.

We 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 $a \preceq b$.

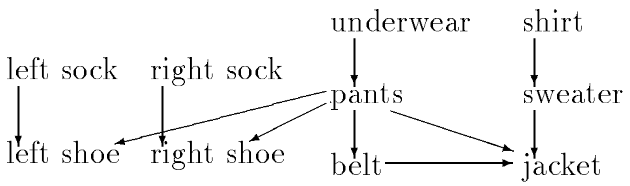
But this could be misleading – note that $\geq$ is a p.o. on natural numbers, as well as $\leq$. If we use the $\preceq$ symbol for $\geq$, things look really funny.
(In cases like this, better to use R.)
A partial order is always defined on some set A. The set together with the partial order is called a "poset":

Definition 1.3 A set A together with a partial order $\preceq$ is called a **poset** $(A, \preceq)$.

Representing posets:

Can usually be nicely drawn as a graph (Informally, dots and arrows .)
Draw an arrow from a to b if aRb. (Similar to what we've already done with relations.)

E.g.: Draw a poset that describes the order in which one would put on clothes.
The set is of clothes, and the partial order specifies what should be put on before what.



Notice that some arrows are left out. E.g., underwear precedes belt.
But all those omitted follow from transitivity.
Loops (which express reflexive relationships) also aren't shown.

Definition 1.4 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.

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

Notice that the graph above has directed paths, that is, sequences of nodes $a_1 \cdots a_n$ where $(\forall i < n) a_i R a_{i+1}$.
However, it does not have any directed cycles, a directed path where $a_1 = a_n$. Good thing, because otherwise we could get stuck trying to get dressed.
As proved above, this is true of all posets.

1.1.5 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.

E.g., 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 1.5 a and b are **incomparable** if neither $a \preceq b$ nor $b \preceq a$.

Definition 1.6 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 1.7 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 $a_2 \leq b_2$.
  Do some examples here. $(2, 37) \preceq (2, 38) \preceq (3, 2)$.
  "Lexicographic" because like dictionary.

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

This has been the basic material.
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, do some for HW.

1.2 Topological Sorting

Sometimes when we have a partial order,
e.g., of tasks to be performed,
we want to obtain a consistent total order,
e.g., 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 1.8 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 $\forall i < j$, either $a_i \preceq a_j$ or a_i and a_j are incomparable in $(A, \preceq)$.

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 1.9 Every finite poset has a topological sort.

We'll prove this by induction.
On the number of elements in the domain of the poset.
The basic idea is to pick a "smallest" element to start and then proceed recursively.

Definition 1.10 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 1.11 Every finite poset $(A, \preceq)$ has a minimal element.

Proof. By contradiction.
Assume no minimal elements. Then $(\forall a \in A)(\exists b)[b \prec a]$. (Recall $b \prec a$ means $b \preceq a$ and $b \neq a$.

Start from any element of a and construct a sequence of successively smaller elements, by using the fact above.
At each step, can get another element preceding the most recent.
Can't repeat any of the previous ones in the chain, since that would create a cycle (and we know a poset has no cycles).
So just keep going forever, creating an infinite chain of distinct elements.

But this is nonsense, because A is a finite set.
Contradiction, so there must be some minimal element.

More formally, based on assumption, can give a recursive definition for a sequence a_k. Set a_1 to be some element of A, and define a_{k+1} to be some element $\prec a_k$ (such an element always exists by assumption). But now suppose A has n elements. Then consider the values $a_1, \ldots, a_{n+1}$ of our sequence. By pigeonholing (number 1 to $n = 1$ are pigeons, values in A are holes) there must be $j \neq k$ with $a_j = a_k$. But this means we have a cycle on $a_j \ldots a_k$. We've already argued this can't happen in a poset unless all of $a_j, a_{j=1}, \ldots, a_k$ are equal. But this violates the requirements that $a_j \neq a_{j=1}$. So we have a contradiction.

A different way to prove this is by induction. You'll do this for homework (actually, the pigeonhole argument is also using induction; it's hiding in the proof of the pigeonhole principle). ■

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.
$P(n)$ = 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 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 def of topological sort – u can't be in the wrong order w.r.t. any of the later elements.)

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

E.g.: Dressing example:
Construct ordering by picking one at a time.
In each case, 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:
$\{1\}$, $\{2\}$, $\{1,2\}$
$\{3\}$, $\{1,3\}$, $\{2,3\}$, $\{1,2,3\}$,
$\{4\}$, $\{1,4\}$, $\{2,4\}$, $\{3,4\}$,
$\{1,2,4\}$, $\{1,3,4\}$, $\{2,3,4\}$, $\{1,2,3,4\}$

1.3 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, 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?
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.

On clothes example, we could do all the minimal elements first (Lsock, Rsock, underwear, shirt), remove them and repeat.
Need lots of hands, or maybe dressing servants.
(So do pants and sweater next, and then Lshoe, Rshoe, 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 1.12 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 1.13 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 1.14 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 1.15 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.

Corollary 1.16 If t is the length of the longest chain in a poset $(A, \preceq)$ then A can be partitioned into t antichains.

Definition 1.17 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

1.4 Dilworth's Theorem

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

1.4.1 The Basics

Theorem 1.18 (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.
(Can assume exactly t, since could have some empty.)
So there is an antichain with $\geq \frac{n}{t}$ elements.
As needed. ∎

For instance:

Corollary 1.19 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.

1.4.2 Increasing and Decreasing Sequences

A mathematical example:
In any sequence of n different numbers, there is either an increasing subsequence of length $> \sqrt{n}$ or a decreasing subsequence of length $\geq \sqrt{n}$.

E.g., 6 4 7 9 1 2 5 3 8 has the decreasing sequence 6 4 1 and the increasing sequence 1 2 3 8. That is, it has both.

We can get this using Dilworth's theorem; the trick is to define the appropriate poset.

The domain is the set of values in the sequence.

For the ordering, define $a \preceq b$ if either $a = b$, or else ($a < b$ and a comes before b in the sequence).

Check this is reflexive (stated explicitly), transitive, and antisymmetric.

A chain corresponds to a sequence that increases in value and moves to the right, that is, an increasing sequence.

But what does an antichain correspond to?

Lemma 1.20 If a, b are incomparable and $a > b$ then a precedes b in the sequence.

Proof. By contradiction.
Assume b precedes a.
Then $b < a$ and b precedes a, so $b \preceq a$ in the partial order.
This contradicts our assumption that a and b are incomparable. ∎

Extend this to more than 2 elements:

Lemma 1.21 If $a_1, a_2, \cdots, a_t$ are incomparable and $a_1 > a_2 > \cdots > a_t$ then $a_1, a_2, \cdots a_t$ form a decreasing subsequence.

Proof. $(\forall i)$, the fact that $a_i > a_{i+1}$ implies that a_i precedes a_{i+1} in the sequence, by the previous lemma. ∎

So given an antichain, arrange the elements so that they are in decreasing order and the "left of" relation follows, giving a decreasing subsequence.

Dilworth's theorem implies that there is either a chain of size $> \sqrt{n}$ or an antichain of size $\geq \sqrt{n}$.
By the analysis just above, this yields either an increasing sequence of length $> \sqrt{n}$ or a decreasing subsequence of length $\geq \sqrt{n}$.

1.4.3 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 db is totally sorted, can compare with middle elements, then middle of halves, etc.
Binary search.
Approximately log n comparisons needed.

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

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.

1.5 Logical Time in Distributed Systems

One last use of p.o.
No use of powerful theorems here – just showing an important place where a p.o. 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 p.o. 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 p.o.:
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 p.o.: when one step precedes another in the p.o., it must have a smaller logical time.

Behaves like real time: Formally, can reorder the steps in logical time order, still consistent with the p.o.
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.