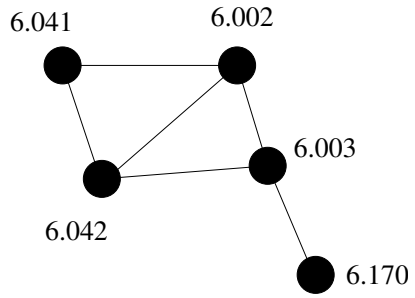
Graphs and Digraphs

1 Coloring Graphs

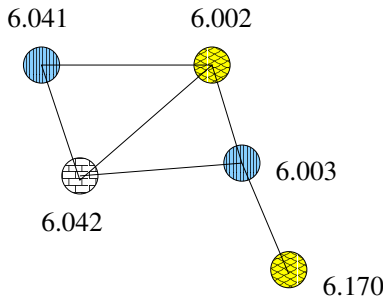
In the "Sex in America" graph In Week 5 Notes, we used edges to indicate an affinity between two nodes, but having an edge represent a *conflict* between two nodes also turns out to be really useful. For example, each term the MIT Schedules Office must assign a time slot for each final exam. This is 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. Of course, you can make such a schedule by having every exam in a different slot, but then you would need hundreds of slots for the hundreds of courses, and exam period would run all year! So, the Schedules Office would also like to keep exam period short.

The Schedules Office's problem is easy to describe as a graph. There will be a vertex for each course with a final exam, and two vertices will be adjacent exactly when some student is taking both courses. For example, suppose we need to schedule exams for 6.041, 6.042, 6.002, 6.003 and 6.170. The scheduling graph might look like this:



6.002 and 6.042 cannot have an exam at the same time since there are students in both courses, so there is an edge between their nodes. On the other hand, 6.042 and 6.170 can have an exam at the same time if they're taught at the same time (which they sometimes are), since no student can be enrolled in both (that is, no student *should* be enrolled in both when they have a timing conflict). Next, identify each time slot with a color. For example, Monday morning is red, Monday afternoon is blue, Tuesday morning is green, etc.

Assigning an exam to a time slot is now equivalent to coloring the corresponding vertex. The main constraint is that *adjacent vertices must get different colors* —otherwise, some student has two exams at the same time. Furthermore, in order to keep the exam period short, we should try to color all the vertices using as *few different colors as possible*. For our example graph, three colors suffice:



This coloring corresponds to giving one final on Monday morning (white), two Monday afternoon (blue), and two Tuesday morning (green).

Can we use fewer than three colors? No! We can't use only two colors since there is a triangle in the graph, and three vertices in a triangle must all have different colors.

This is an example of what is a called a *graph coloring problem*: given a graph G, assign colors to each node such that adjacent nodes have different colors. A color assignment with this property is called a *valid coloring* of the graph —a "coloring," for short. A graph G is ***k-colorable*** if it has a coloring that uses at most k colors. The minimum value of k for which a coloring exists is called the ***chromatic number***, $\chi(G)$, of G.

In general, trying to figure out if you can color a graph with a fixed number of colors can take a long time. It's a classic example of a problem for which no fast algorithms are known. In fact, it is easy to check if a coloring works, but it seems really hard to find it (if you figure out how, then you can get a $1 million Clay prize).

1.1 Degree-bounded Coloring

There are some simple graph properties that give useful upper bounds on colorings. For example, if we have a bound on the degrees of all the vertices in a graph, then we can easily find a coloring with only one more color than the degree bound.

Theorem 1.1. *A graph with maximum degree at most k is $(k+1)$-colorable.*

Unfortunately, if you try induction on k, it will lead to disaster. It is not that it is impossible, just that it is extremely painful and would ruin you if you tried it on an exam. Another option, especially with graphs, is to change what you are inducting on. In graphs, some good choices are n, the number of nodes, or e, the number of edges.

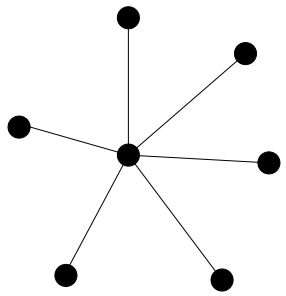
Proof. We use induction on the number of vertices in the graph, which we denote by n. Let $P(n)$ be the proposition that an n-vertex graph with maximum degree at most k is $(k+1)$-colorable.

Base case: $(n = 1)$ A 1-vertex graph has maximum degree 0 and is 1-colorable, so $P(1)$ is true.

Inductive step: Now assume that $P(n)$ is true, and let G be an $(n+1)$-vertex graph with maximum degree at most k. Remove a vertex v (and all edges incident to it), leaving an n-vertex subgraph, H. The maximum degree of H is at most k, and so H is $(k+1)$-colorable by our assumption $P(n)$. Now add back vertex v. We can assign v a color different from all its adjacent vertices, since there are at most k adjacent vertices and $k+1$ colors are available. Therefore, G is $(k+1)$-colorable. This completes the Inductive step, and the theorem follows by induction. □

Sometimes $k+1$ colors is the best you can do. For example, in the complete graph, K_n, every one of its n vertices is adjacent to all the others, so all n must be assigned different colors. Of course n colors is also enough, so $\chi(K_n) = n$. So K_{k+1} is an example where Theorem 1.1 gives the best possible bound. This means that Theorem 1.1 also gives the best possible bound for *any* graph with degree bounded by k that has K_{k+1} as a subgraph.

But sometimes $k+1$ colors is far from the best that you can do. Here's an example of an n-node star graph for $n = 7$:



In the n-node star graph, the maximum degree is $n-1$, but the star only needs 2 colors!

1.2 Why coloring?

Coloring problems come up in all sorts of applications. For example, at Akamai, a new version of software is deployed over each of 20,000 servers every few days. The updates cannot be done at the same time since the servers need to be taken down in order to deploy the software. Also, the servers cannot be handled one at a time, since it would take forever to update them all (each one takes about an hour). Moreover, certain pairs of servers cannot be taken down at the same time since they have common critical functions. This problem was eventually solved by making a 20,000 node conflict graph and coloring it with 8 colors – so only 8 waves of install are needed!

Another example comes from the need to assign frequencies to radio stations. If two stations have an overlap in their broadcast area, they can't be given the same frequency. Frequencies are precious and expensive, so you want to minimize the number handed out. This amounts to finding the minimum coloring for a graph whose vertices are the stations and whose edges are between stations with overlapping areas.

Coloring also comes up allocating registers for program variables. While a variable is in use, its value needs to be saved in a register, but registers can often be reused for different variables. But two variables need different registers if they are referenced during overlapping intervals of program execution. So register allocation is the coloring problem for a graph whose vertices are the variables; vertices are adjacent if their intervals overlap, and the colors are registers.

Finally, there's the famous map coloring problem mentioned in Week 1 Notes. The question is how many colors are needed to color a map so that adjacent territories get different colors? This is the same as the number of colors needed to color a graph that can be drawn in the plane without edges crossing. A proof that four colors are enough for the *planar* graphs was acclaimed when it was discovered about thirty years ago. Implicit in that proof was a 4-coloring procedure that takes time proportional to the number of vertices in the graph (countries in the map). On the other

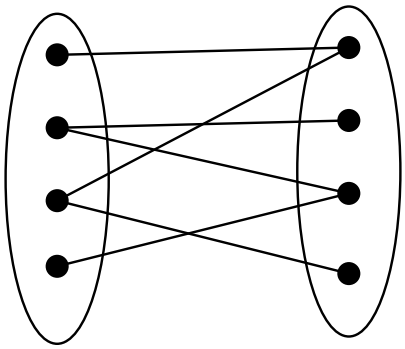
hand, it's another of those million dollar prize questions to find an efficient procedure to tell if a planar graph really *needs* four colors or if three will actually do the job. But it's always easy to tell if an *arbitrary* graph is 2-colorable, as we show next. Later we'll develop enough planar graph theory to show that planar graphs are 6-colorable.

1.3 Bipartite Graphs

There were two kinds of vertices in the "Sex in America" graph —males and females, and edges only went between the two kinds. Graphs like this come up so frequently they have earned a special name —they are called ***bipartite graphs***.

Definition 1.2. A *bipartite graph* is a graph together with a partition of its vertices into two sets, L and R, such that every edge is incident to a vertex in L and to a vertex in R.

So every bipartite graph looks something like this:



Now we can immediately see how to color a bipartite graph using only two colors: let all the L vertices be black and all the R vertices be white. Conversely, if a graph is 2-colorable, then it is bipartite with L being the vertices of one color and R the vertices of the other color. In other words,

> "bipartite" is a synonym for "2-colorable."

The following Lemma gives another useful characterization of bipartite graphs.

Theorem 1.3. *A graph is bipartite iff it has no odd-length cycle.*

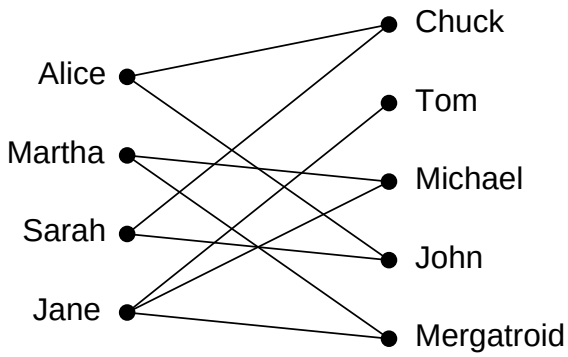
The proof of Theorem 1.3 will be on a problem set.

2 Bipartite Matchings

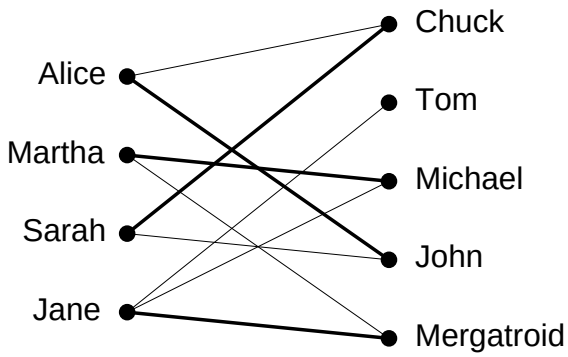
The *bipartite matching problem* resembles the stable Marriage Problem in that it concerns a set of girls and a set of at least as many boys. There are no preference lists, but each girl does have some

boys she likes and others she does not like. In the bipartite matching problem, we ask whether every girl can be paired up with a boy that she likes.

Any particular matching problem can be specified by a bipartite graph with a vertex for each girl, a vertex for each boy, and an edge between a boy and a girl iff the girl likes the boy. For example, we might obtain the following graph:



Now a *matching* will mean a way of assigning every girl to a boy so that different girls are assigned to different boys, and a girl is always assigned to a boy she likes. For example, here is one possible matching for the girls:



Hall's Matching Theorem states necessary and sufficient conditions for the existence of a matching in a bipartite graph. It turns out to be a remarkably useful mathematical tool.

2.1 The Matching Condition

We'll state and prove Hall's Theorem using girl-likes-boy terminology. Define *the set of boys liked by a given set of girls* to consist of all boys liked by at least one of those girls. For example, the set of boys liked by Martha and Jane consists of Tom, Michael, and Mergatroid.

For us to have any chance at all of matching up the girls, the following ***matching condition*** must hold:

Every subset of girls likes at least as large a set of boys.

For example, we can not find a matching if some 4 girls like only 3 boys. Hall's Theorem says that this necessary condition is actually sufficient; if the matching condition holds, then a matching exists.

Theorem 2.1. *A matching for a set of girls G with a set of boys B can be found if and only if the matching condition holds.*

Proof. First, let's suppose that a matching exists and show that the matching condition holds. Consider an arbitrary subset of girls. Each girl likes at least the boy she is matched with. Therefore, every subset of girls likes at least as large a set of boys. Thus, the matching condition holds.

Next, let's suppose that the matching condition holds and show that a matching exists. We use strong induction on $|G|$, the number of girls. If $|G| = 1$, then the matching condition implies that the lone girl likes at least one boy, and so a matching exists. Now suppose that $|G| \geq 2$. There are two possibilities:

1. Every proper subset of girls likes a *strictly larger* set of boys. In this case, we have some latitude: we pair an arbitrary girl with a boy she likes and send them both away. The matching condition still holds for the remaining boys and girls, so we can match the rest of the girls by induction.

2. Some proper subset of girls $X \subset G$ likes an *equal-size* set of boys $Y \subset B$. We match the girls in X with the boys in Y by induction and send them all away. We will show that the matching condition holds for the remaining boys and girls, and so we can match the rest of the girls by induction as well.

   To that end, consider an arbitrary subset of the remaining girls $X' \subseteq G - X$, and let Y' be the set of remaining boys that they like. We must show that $|X'| \leq |Y'|$. Originally, the combined set of girls $X \cup X'$ liked the set of boys $Y \cup Y'$. So, by the matching condition, we know:

   $$\left|X \cup X'\right| \leq \left|Y \cup Y'\right|$$

   We sent away $|X|$ girls from the set on the left (leaving X') and sent away an equal number of boys from the set on the right (leaving Y'). Therefore, it must be that $|X'| \leq |Y'|$ as claimed.

In both cases, there is a matching for the girls. The theorem follows by induction. □

The proof of this theorem gives an algorithm for finding a matching in a bipartite graph, albeit not a very efficient one. However, efficient algorithms for finding a matching in a bipartite graph do exist. Thus, if a problem can be reduced to finding a matching, the problem is essentially solved from a computational perspective.

2.2 A Formal Statement

Let's restate Hall's Theorem in abstract terms so that you'll not always be condemned to saying, "Now this group of little girls likes at least as many little boys..."

In any graph, the set $N(S)$, of *neighbors*[1] of some set, S, of vertices is the set of all vertices adjacent to any vertex in S. That is,

$$N(S) ::= \{r \mid s\text{—}r \text{ is an edge for some } s \in S\}.$$

S is called a *bottleneck* if

$$|S| > |N(S)|.$$

A *matching* in a graph is an injection on the set of vertices that only maps a vertex to an adjacent vertex.

Theorem 2.2 (Hall's Theorem)**.** *Let G be a bipartite graph with vertex partition L, R. There is total matching from L to R iff no subset of L is a bottleneck.*

2.2.1 An Easy Matching Condition

The bipartite matching condition requires that *every* subset of girls has a certain property. In general, verifying that every subset has some property, even if it's easy to check any particular subset for the property, quickly becomes overwhelming because the number of subsets of even relatively small sets is enormous —over a billion subsets for a set of size 30.

However, there is a simple property of vertex degrees in a bipartite graph that guarantees a match and is very easy to check. Namely, call a bipartite graph *degree-constrained* if vertex degrees on the left are at least as large as those on the right. More precisely,

Definition 2.3. A bipartite graph G with vertex partition L, R is *degree-constrained* if $\deg(l) \geq \deg(r)$ for every $l \in L$ and $r \in R$.

Now we can always find a matching in a degree-constrained bipartite graph.

Lemma 2.4. *Every degree-constrained bipartite graph satisifies the matching condition.*

Proof. Let S be any set of vertices in L. The number of edges incident to vertices in S is exactly the sum of the degrees of the vertices in S. Each of these edges is incident to a vertex in $N(S)$ by definition of $N(S)$. So the sum of the degrees of the vertices in $N(S)$ is at least as large as the sum for S. But since the degree of every vertex in $N(S)$ is at most as large as the degree of every vertex in S, there would have to be at least as many terms in the sum for $N(S)$ as in the sum for S. So there have to be at least as many vertices in $N(S)$ as in S, proving that S is not a bottleneck. So there are no bottlenecks, proving that the degree-constrained graph satisifies the matching condition. □

[1] An equivalent definition of $N(S)$ uses relational notation: $N(S)$ is simply SJ, where J is the adjacency relation of the graph.

3 Digraphs

A *directed graph* (*digraph* for short) is formally the same as a binary relation, R, on a set, A, but we picture the digraph geometrically by representing elements of A as points on the plane, with an arrow from the point for a to the point for b exactly when $(a, b) \in \text{graph}(R)$. The elements of A are referred to as the *vertices* of the digraph, and the pairs $(a, b) \in \text{graph}(R)$ are called its *directed edges*. We use the notation $a \to b$ as an alternative notation for the pair (a, b).

For example, the divisibility relation on $\{1, 2, \ldots, 12\}$ is represented by the digraph:

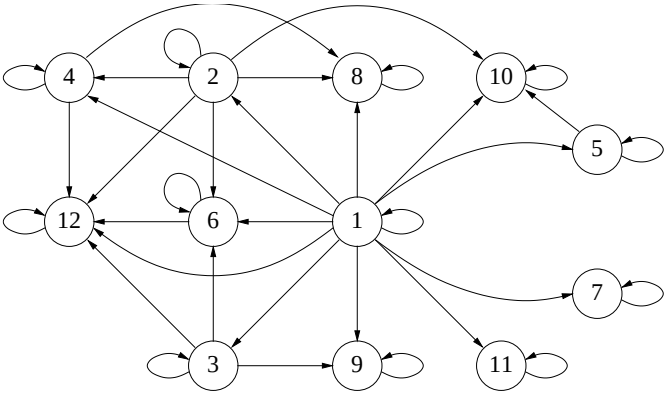


Figure 1: Divisibility Digraph on $\{1, 2, \ldots, 12\}$.

3.1 Paths in Digraphs

Pictured with points and arrows, a length k path in a digraph looks like a line that starts at a point, a_0, and traverses k arrows between successive points, $a_1, a_2, \ldots$ to end at a point, a_k. Note that k may be 0 —a single vertex counts as length zero path to itself, just as for simple graphs. The precise definitions are very similar to those for simple graphs:

Definition 3.1. A *path in a digraph* is a sequence of vertices $a_0, \ldots, a_k$ with $k \geq 0$ such that $a_i \to a_{i+1}$ is an edge for every $i \geq 0$ such that $i < k$. The path is said to *start* at a_0, to *end* at a_k, and the *length* of the path is defined to be k. The path is *simple* iff all the a_i's are different, that is, $a_i = a_j$ only if $i = j$.

Many of the relational properties have geometric descriptions in terms of digraphs. For example:

Reflexivity: All vertices have self-loops (a *self-loop* at a vertex is an arrow going from the vertex back to itself).

Irreflexivity: No vertices have self-loops.

Asymmetry: No self-loops and at most one (directed) edge between any two vertices.

Symmetry: A binary relation R is ***symmetric*** iff aRb implies bRa for all a, b in the domain of R. So if there is an edge from a to b, there is also one in the reverse direction. So edges may as well be represented without arrows, indicating that they can be followed in either direction.

Transitivity: Short-circuits—for any path through the graph, there is an arrow from the first vertex to the last vertex on the path.

We can define some new relations based on paths. Let R be the edge relation of a digraph. Define relations R^* and R^+ on the vertices by the conditions that for all vertices a, b:

$$\begin{aligned} a\, R^*\, b &::= \text{there is a path in } R \text{ from } a \text{ to } b, \\ a\, R^+\, b &::= \text{there is a positive length path in } R \text{ from } a \text{ to } b. \end{aligned}$$

R^* is called the *path relation*[2] of R. It follows from the definition of path that R^* is transitive. It is also reflexive (because of the length-zero paths) and it contains the graph of R (because of the length-one paths). R^+ is called the *positive-length path relation*; it also contains graph (R) and is transitive.

3.2 Directed Acyclic Graphs

Definition 3.2. A *cycle* in a digraph is a path that begins and ends at the same vertex. Note that by convention, a single vertex is considered to be a cycle of length 0 that begins and ends at the vertex. A *directed acyclic graph (DAG)* is a directed graph with no positive length cycles.

A *simple cycle* in a digraph is a cycle whose vertices are distinct except for the beginning and end vertices.

DAG's are an economical way to represent partial orders. For example, the direct prerequisite relation between MIT subjects described in Week 3 Notes was used to determine the partial order of indirect prerequisites on subjects. This indirect prerequisite partial order is precisely the positive length path relation of the direct prerequisites.

Lemma 3.3. *If D is a DAG, then D^+ is a strict partial order.*

Proof. We know that D^+ is transitive. Also, a positive length path from a vertex to itself would be a cycle, so there are no such paths. This means D^+ is irreflexive, which implies it is a strict partial order (see Week 3, Tuesday, Class Problem 4). □

It's easy to check that conversely, the graph of any strict partial order is a DAG.

Problem 1. Verify that any strict partial order is a DAG.

The divisibility partial order can also be more economically represented by the path relation in a DAG. The DAG for divisibility on $\{1, 2, \ldots, 12\}$ is shown in Figure 2; the arrowheads are omitted in the Figure, and edges are understood to point upwards.

The minimum edge DAG representing a finite partial order is unique, and is easy to find. This is not hard to verify:

Problem 2. If a and b are distinct nodes of a digraph, then a is said to *cover* b if there is an edge from a to b and there is no other path from a to b. If a covers b, the edge from a to b is called a covering edge.

[2] In many texts, R^* is called the *transitive closure* of R.

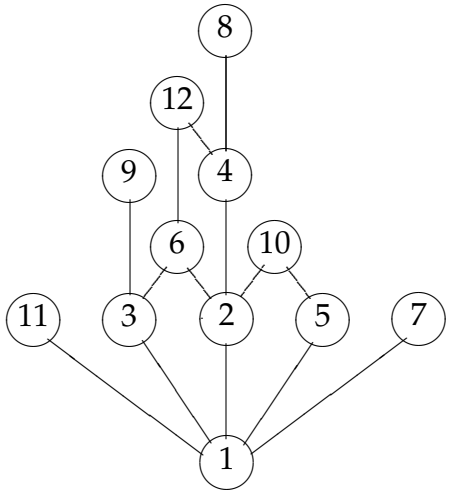


Figure 2: DAG for Divisibility on $\{1, 2, \ldots, 12\}$.

(a) Show that if two DAG's have the same positive path relation, then they have the same set of covering edges.

(b) For any DAG, D, let $\widehat{D}$ be the subgraph of D consisting of only the covering edges. Show that if D is finite and has no self-loops, then D and $\widehat{D}$ have the same positive path relation, that is $D^+ = \widehat{D}^+$.

(c) Conclude that if D is a finite DAG, then $\widehat{D}$ is the unique DAG with the smallest number of edges among all digraphs with the same positive path relation.

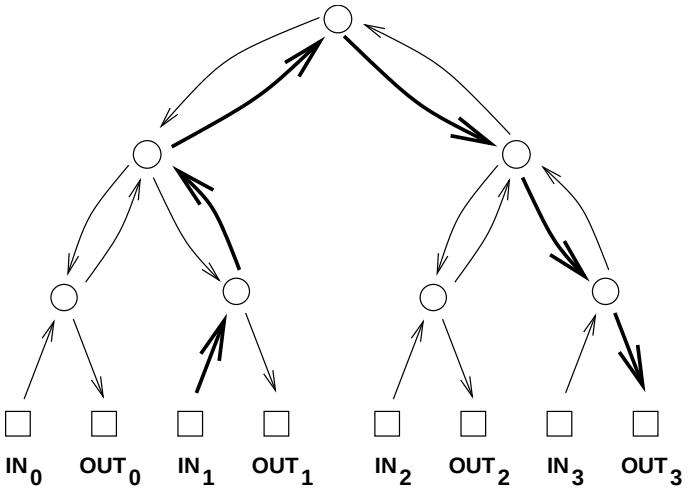
(d) Show that the previous result is not true in general for the infinite DAG corresponding to the total order on the rational numbers.

4 Communication Networks

Modeling communication networks is an important application of graphs in Computer Science. Here, vertices represent computers, processors, and switches; edges will represent wires, fiber, or other transmission lines through which data flows. For some communication networks, like the internet, the corresponding graph is enormous and largely chaotic. However, there do exist more organized networks, such as certain telephone switching networks and the communication networks inside parallel computers. For these, the corresponding graphs are highly structured. In these Notes, we'll look at some of the nicest and most commonly used communication networks.

4.1 Complete Binary Tree

Let's start with a ***complete binary tree***. Here is an example with 4 inputs and 4 outputs.



The kinds of communication networks we consider aim to transmit packets of data between computers, processors, telephones, or other devices. The term ***packet*** refers to some roughly fixed-size quantity of data— 256 bytes or 4096 bytes or whatever. In this diagram and many that follow, the squares represent ***terminals***, sources and destinations for packets of data. The circles represent ***switches***, which direct packets through the network. A switch receives packets on incoming edges and relays them forward along the outgoing edges. Thus, you can imagine a data packet hopping through the network from an input terminal, through a sequence of switches joined by directed edges, to an output terminal.

Recall that there is a unique simple path between every pair of vertices in a tree. So the natural way to route a packet of data from an input terminal to an output in the complete binary tree is along the corresponding directed path. For example, the route of a packet traveling from input 1 to output 3 is shown in bold.

4.2 Latency and Diameter

Latency is a critical issue in communication networks. This is the largest delay between the time a packet is sent from an input until it arrives at its designated output. Assuming it takes one time unit to travel across a wire with no delays at switches, the delay of a packet is the number of wires it crosses going from input to output, that is, the packet delay is the *length* of the path the packet follows.

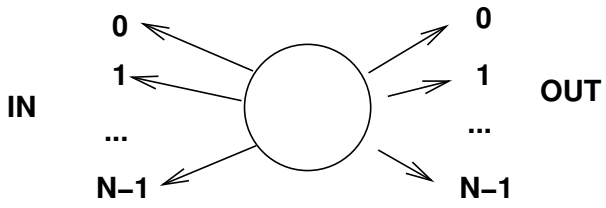
The latency of a network will depend on how packets are routed, but generally packets are routed to go from input to output by the shortest path possible. With a shortest path routing, the worst case delay is the distance between the input and output that are farthest apart. This is called the ***diameter*** of the network. In other words, the diameter of a network[3] is the maximum length of any shortest path between an input and an output. For example, in the complete binary tree above, the distance from input 1 to output 3 is six. No input and output are farther apart than this, so the diameter of this tree is also six.

[3]The usual definition of *diameter* for a general *graph* (simple or directed) is the largest distance between *any* two vertices, but in the context of a communication network we're only interested in the distance between inputs and outputs, not between arbitrary pairs of vertices.

We're going to consider several different communication networks. For a fair comparison, let's assume that each network has N inputs and N outputs, where N is a power of two. For example, the diameter of a complete binary tree with N inputs and outputs is $2 \log N + 2$. (All logarithms in this lecture— and in most of Computer Science —are base 2.) This is quite good, because the logarithm function grows very slowly. We could connect up $2^{10} = 1024$ inputs and outputs using a complete binary tree and still have a latency of only $2 \log(2^{10}) + 2 = 22$.

4.3 Switch Size

One way to reduce the diameter of a network is to use larger switches. For example, in the complete binary tree, most of the switches have three incoming edges and three outgoing edges, which makes them 3×3 switches. If we had 4×4 switches, then we could construct a complete *ternary* tree with an even smaller diameter. In principle, we could even connect up all the inputs and outputs via a single monster switch:



This isn't very productive, however, since we've just concealed the original network design problem inside this abstract switch. Eventually, we'll have to design the internals of the monster switch using simpler components, and then we're right back where we started. So the challenge in designing a communication network is figuring out how to get the functionality of an $N \times N$ switch using elementary devices, like 3×3 switches. Following this approach, we can build arbitrarily large networks just by adding in more building blocks.

4.4 Switch Count

Another goal in designing a communication network is to use as few switches as possible since routing hardware has a cost. The number of switches in a complete binary tree is $1 + 2 + 4 + 8 + \ldots + N$, since there is 1 switch at the top (the "root switch"), 2 below it, 4 below those, and so forth. By the formula for the sum of a geometric series (see Slides 3W, Feb 21, 2007) the total number of switches is $2N - 1$, which is nearly the best possible with 3×3 switches.

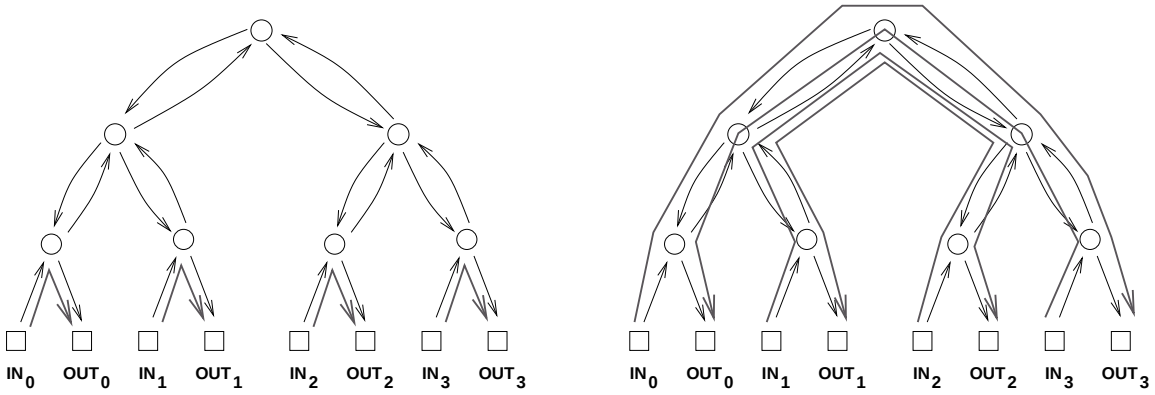
4.5 Congestion

The complete binary tree has a fatal drawback: the root switch is a bottleneck. At best, this switch must handle an enormous amount of traffic: every packet traveling from the left side of the network to the right or vice-versa. Passing all these packets through a single switch could take a long time. At worst, if this switch fails, the network is broken into two equal-sized pieces. The *max congestion* of a network is a measure of its bottlenecks; defining max congestion requires some preliminary definitions.

A ***permutation*** is a function π that maps each number in the set $\{0, 1, \ldots, N-1\}$ to another number in the set such that no two numbers are mapped to the same value. In other words, π is a bijection from $\{0, 1, \ldots, N-1\}$ to itself.

For each permutation π, there is a corresponding ***permutation routing problem***. In this problem, one packet starts out at each input; in particular, the packet starting at input i is called packet i. The challenge is to direct each packet i through the network from input i to output $\pi(i)$.

A solution to a permutation routing problem is a specification of the path taken by each of the N packets. In particular, the path taken by packet i from input i to output $\pi(i)$ is denoted $P_{i,\pi(i)}$. For example, if $\pi(i) = i$, then there is an easy solution: let $P_{i,\pi(i)}$ be the path from input i up through one switch and back down to output i. On the other hand, if $\pi(i) = (N-1) - i$, then each path $P_{i,\pi(i)}$ must begin at input i, loop all the way up through the root switch, and then travel back down to output $(N-1) - i$. These two situations are illustrated below.



We can distinguish between a "good" set of paths and a "bad" set based on congestion. The ***congestion*** of a set of paths $P_{0,\pi(0)}, \ldots, P_{N-1,\pi(N-1)}$ is equal to the largest number of paths that pass through a single switch. For example, the congestion of the set of paths in the diagram at left is 1, since at most 1 path passes through each switch. However, the congestion of the paths on the right is 4, since 4 paths pass through the root switch (and the two switches directly below the root). Generally, lower congestion is better since packets can be delayed at an overloaded switch.

By extending the notion of congestion, we can also distinguish between "good" and "bad" networks with respect to bottleneck problems. The ***max congestion*** of a network is the *maximum* over all permutations π of the *minimum* over all paths $P_{i,\pi(i)}$ of the congestion of the paths.

You may find it helpful to think about max congestion in terms of a value game. You design your spiffy, new communication network; this defines the game. Your opponent makes the first move in the game: she inspects your network and specifies a permutation routing problem that will strain your network.You move second: given her specification, you choose the precise paths that the packets should take through your network; you're trying to avoid overloading any one switch. Then her next move is to pick a switch with as large as possible a number of packets passing through it; this number is her score in the competition. The max congestion of your network is the largest score she can ensure; in other words, it is precisely the max-value of this game.

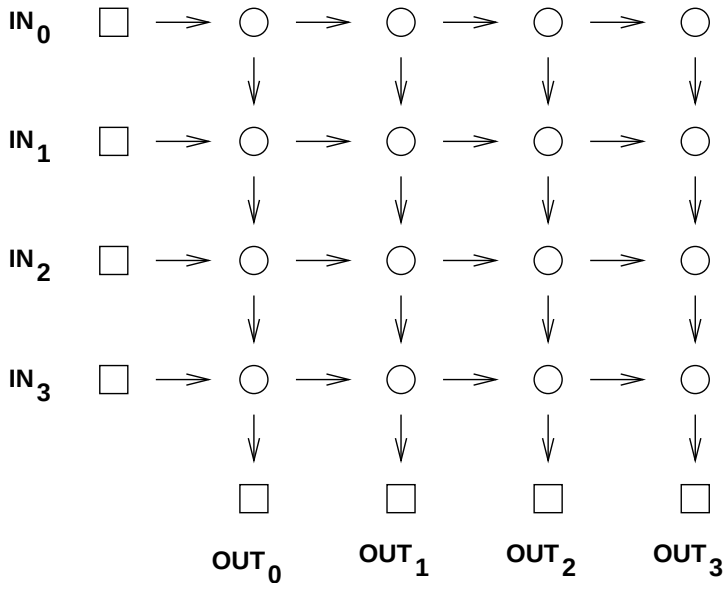
For example, if your enemy were trying to defeat the complete binary tree, she would choose a permutation like $\pi(i) = (N-1) - i$. Then for *every* packet i, you would be forced to select a path $P_{i,\pi(i)}$ passing through the root switch. Thus, the max congestion of the complete binary tree is N— which is horrible!

Let's tally the results of our analysis so far:

network	diameter	switch size	# switches	congestion
complete binary tree	$2\log N + 2$	3×3	$2N - 1$	N

4.6 2-D Array

Let's look at an another communication network. This one is called a ***2-dimensional array*** or ***grid***.



Here there are four inputs and four outputs, so $N = 4$.

The diameter in this example is 8, which is the number of edges between input 0 and output 3. More generally, the diameter of an array with N inputs and outputs is $2N$, which is much worse than the diameter of $2\log N + 2$ in the complete binary tree. On the other hand, replacing a complete binary tree with an array almost eliminates congestion.

Theorem 4.1. *The congestion of an N-input array is 2.*

Proof. First, we show that the congestion is at most 2. Let π be any permutation. Define $P_{i,\pi(i)}$ to be the path extending from input i rightward to column $\pi(i)$ and then downward to output $\pi(i)$. Thus, the switch in row i and column $\pi(i)$ transmits at most two packets: the packet originating at input i and the packet destined for column $\pi(i)$.

Next, we show that the congestion is at least 2. In any permutation routing problem where $\pi(0) = 0$ and $\pi(N-1) = N-1$, two packets must pass through the lower left switch. □

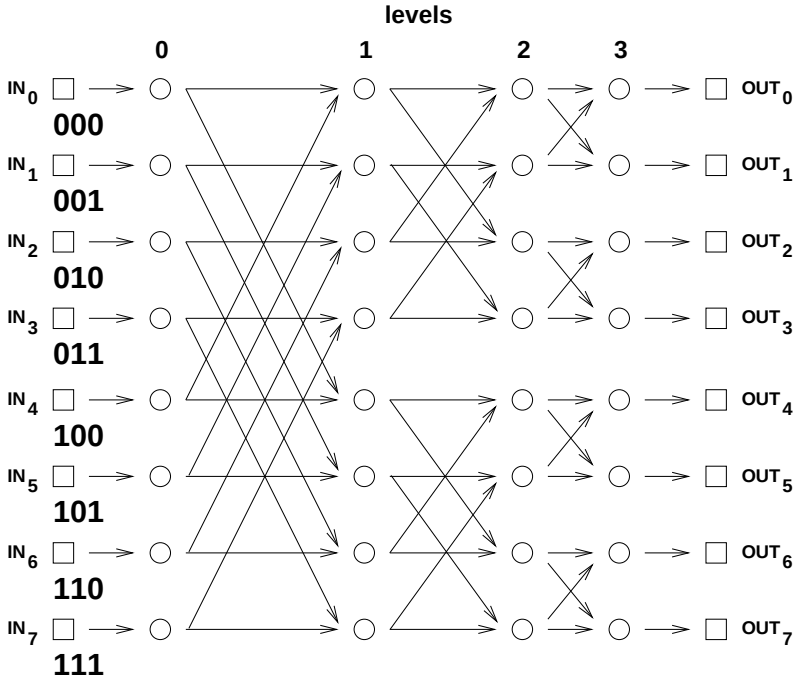
Now we can record the characteristics of the 2-D array.

network	diameter	switch size	# switches	congestion
complete binary tree	$2\log N + 2$	3×3	$2N - 1$	N
2-D array	$2N$	2×2	N^2	2

The crucial entry here is the number of switches, which is N^2. This is a major defect of the 2-D array; a network of size $N = 1000$ would require a *million* 2×2 switches! Still, for applications where N is small, the simplicity and low congestion of the array make it an attractive choice.

4.7 Butterfly

The Holy Grail of switching networks would combine the best properties of the complete binary tree (low diameter, few switches) and of the array (low congestion). The ***butterfly*** is a widely-used compromise between the two. Here is a butterfly network with $N = 8$ inputs and outputs.



The structure of the butterfly is certainly more complicated than that of the complete binary tree or 2-D array! Let's work through the various parts of the butterfly.

All the terminals and switches in the network are arranged in N rows. In particular, input i is at the left end of row i, and output i is at the right end of row i. Now let's label the rows in *binary*; thus, the label on row i is the binary number $b_1 b_2 \ldots b_{\log N}$ that represents the integer i.

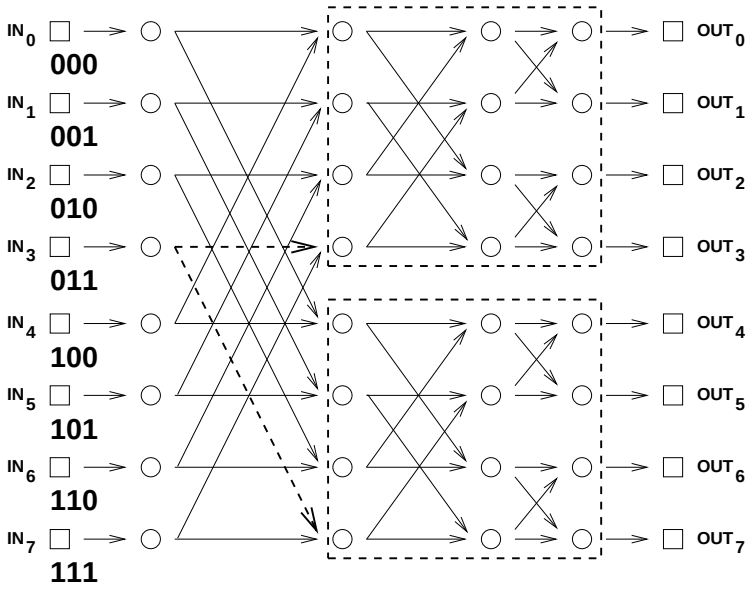
Between the inputs and the outputs, there are $\log(N) + 1$ levels of switches, numbered from 0 to $\log N$. Each level consists of a column of N switches, one per row. Thus, each switch in the network is uniquely identified by a sequence $(b_1, b_2, \ldots, b_{\log N}, l)$, where $b_1 b_2 \ldots b_{\log N}$ is the switch's row in binary and l is the switch's level.

All that remains is to describe how the switches are connected up. The basic connection pattern is expressed below in a compact notation:

$$(b_1, b_2, \ldots, b_{l+1}, \ldots, b_{\log N}, l) \begin{cases} \nearrow & (b_1, b_2, \ldots, b_{l+1}, \ldots, b_{\log N}, l+1) \\ \searrow & (b_1, b_2, \ldots, \overline{b_{l+1}}, \ldots, b_{\log N}, l+1) \end{cases}$$

This says that there are directed edges from switch $(b_1, b_2, \ldots, b_{\log N}, l)$ to two switches in the next level. One edge leads to the switch in the *same* row, and the other edge leads to the switch in the row obtained by *inverting* bit $l+1$. For example, referring back to the illustration of the size $N = 8$ butterfly, there is an edge from switch $(0, 0, 0, 0)$ to switch $(0, 0, 0, 1)$, which is in the same row, and to switch $(1, 0, 0, 1)$, which is the row obtained by inverting bit $l + 1 = 1$.

The butterfly network has a recursive structure; specifically, a butterfly of size $2N$ consists of two butterflies of size N, which are shown in dashed boxes below, and one additional level of switches. Each switch in the new level has directed edges to a pair of corresponding switches in the smaller butterflies; one example is dashed in the figure.



Despite the relatively complicated structure of the butterfly, there is a simple way to route packets. In particular, suppose that we want to send a packet from input $x_1 x_2 \ldots x_{\log N}$ to output $y_1 y_2 \ldots y_{\log N}$. (Here we are specifying the input and output numbers in binary.) Roughly, the plan is to "correct" the first bit by level 1, correct the second bit by level 2, and so forth. Thus, the sequence of switches visited by the packet is:

$$\begin{aligned}(x_1, x_2, x_3, \ldots, x_{\log N}, 0) &\to (y_1, x_2, x_3, \ldots, x_{\log N}, 1) \\ &\to (y_1, y_2, x_3, \ldots, x_{\log N}, 2) \\ &\to (y_1, y_2, y_3, \ldots, x_{\log N}, 3) \\ &\to \quad \ldots \\ &\to (y_1, y_2, y_3, \ldots, y_{\log N}, \log N)\end{aligned}$$

In fact, this is the *only* path from the input to the output!

The congestion of the butterfly network turns out to be around $\sqrt{N}$; more precisely, the congestion is $\sqrt{N}$ if N is an even power of 2 and $\sqrt{N/2}$ if N is an odd power of 2. (You'll prove this fact for homework.)

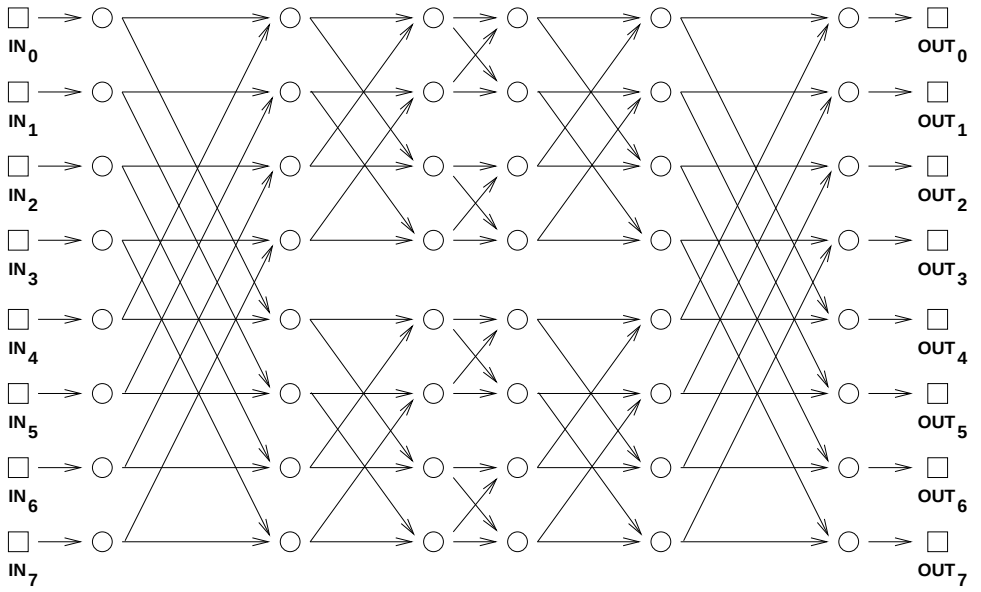
Let's add the butterfly data to our comparison table:

network	diameter	switch size	# switches	congestion
complete binary tree	$2\log N + 2$	3×3	$2N - 1$	N
2-D array	$2N$	2×2	N^2	2
butterfly	$\log N + 2$	2×2	$N(\log(N) + 1)$	$\sqrt{N}$ or $\sqrt{N/2}$

The butterfly has lower congestion than the complete binary tree. And it uses fewer switches and has lower diameter than the array. However, the butterfly does not capture the best qualities of each network, but rather is a compromise somewhere between the two. So our quest for the Holy Grail of routing networks goes on.

4.8 Beneš Network

In the 1960's, a researcher at Bell Labs named Beneš had a remarkable idea. He noticed that by placing *two* butterflies back-to-back, he obtained a marvelous communication network:



This doubles the number of switches and the diameter, of course, but completely eliminates congestion problems! The proof of this fact relies on a clever induction argument that we'll come to in a moment. Let's first see how the Beneš network stacks up:

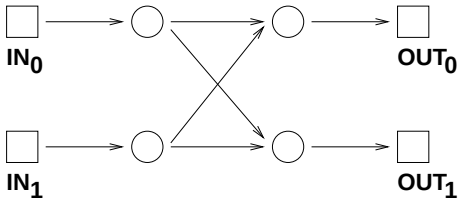
network	diameter	switch size	# switches	congestion
complete binary tree	$2\log N + 2$	3×3	$2N - 1$	N
2-D array	$2N$	2×2	N^2	2
butterfly	$\log N + 2$	2×2	$N(\log(N) + 1)$	$\sqrt{N}$ or $\sqrt{N/2}$
Beneš	$2\log N + 1$	2×2	$2N \log N$	1

The Beneš network has small size and diameter, and completely eliminates congestion. The Holy Grail of routing networks is in hand!

Theorem 4.2. *The congestion of the N-input Beneš network is 1.*

Proof. We use induction. Let $P(a)$ be the proposition that the congestion of the size 2^a Beneš network is 1.

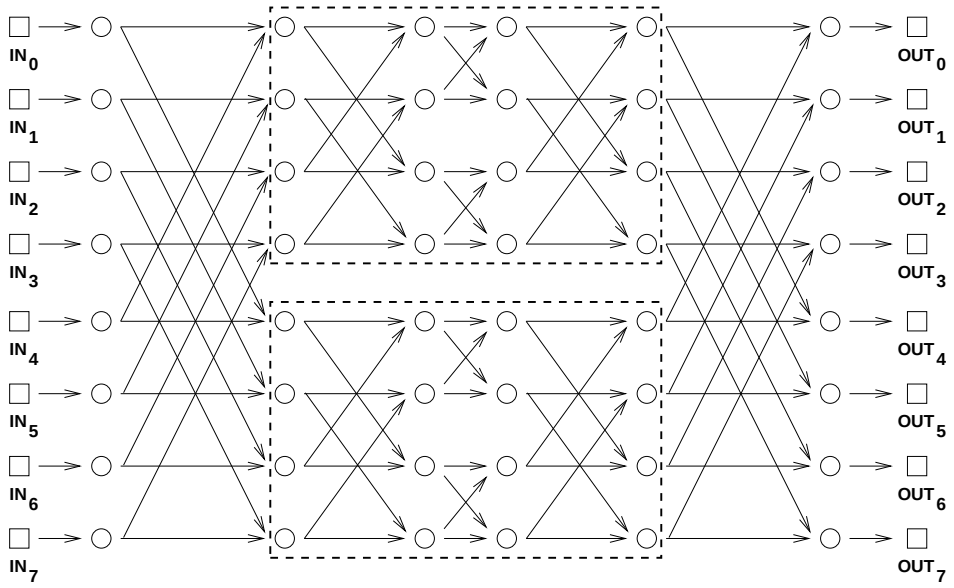
Base case. We must show that the congestion of the size $N = 2^1 = 2$ Beneš network is 1. This network is shown below:



There are only two possible permutation routing problems for a 2-input network. If $\pi(0) = 0$ and $\pi(1) = 1$, then we can route both packets along the straight edges. On the other hand, if $\pi(0) = 1$ and $\pi(1) = 0$, then we can route both packets along the diagonal edges. In both cases, a single packet passes through each switch.

Inductive step. We must show that $P(a)$ implies $P(a + 1)$, where $a \geq 1$. Thus, we assume that the congestion of an N-input Beneš network is 1 in order to prove that the congestion of a $2N$-input Beneš network is also 1.

Digression. Time out! Let's work through an example, develop some intuition, and then complete the proof. Notice that inside a Beneš network of size $2N$ lurk two Beneš subnetworks of size N. (This follows from our earlier observation that a butterfly of size $2N$ contains two butterflies of size N.) In the Beneš network shown below, the two subnetworks are in dashed boxes.

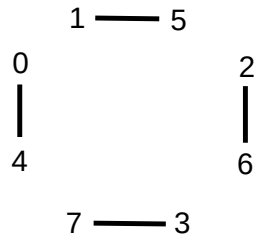


By the inductive assumption, the subnetworks can each route an arbitrary permutation with congestion 1. So if we can guide packets safely through just the first and last levels, then we can rely on induction for the rest! Let's see how this works in an example. Consider the following

permutation routing problem:

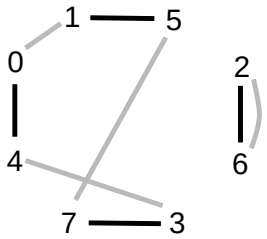
$$\begin{aligned} \pi(0) &= 1 & \pi(4) &= 3 \\ \pi(1) &= 5 & \pi(5) &= 6 \\ \pi(2) &= 4 & \pi(6) &= 0 \\ \pi(3) &= 7 & \pi(7) &= 2 \end{aligned}$$

We can route each packet to its destination through either the upper subnetwork or the lower subnetwork. However, the choice for one packet may constrain the choice for another. For example, we can not route both packet 0 *and* packet 4 through the same network since that would cause two packets to collide at a single switch, resulting in congestion. So one packet must go through the upper network and the other through the lower network. Similarly, packets 1 and 5, 2 and 6, and 3 and 7 must be routed through different networks. Let's record these constraints in a graph. The vertices are the 8 packets. If two packets must pass through different networks, then there is an edge between them. Thus, our constraint graph looks like this:



Notice that at most one edge is incident to each vertex.

The output side of the network imposes some further constraints. For example, the packet destined for output 0 (which is packet 6) and the packet destined for output 4 (which is packet 2) can not both pass through the same network; that would require both packets to arrive from the same switch. Similarly, the packets destined for outputs 1 and 5, 2 and 6, and 3 and 7 must also pass through different switches. We can record these additional constraints in our graph with gray edges:



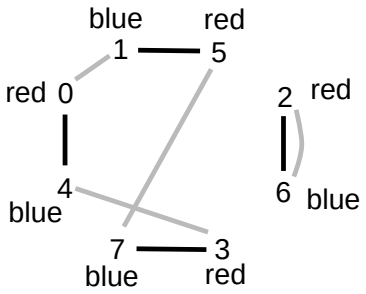
Notice that at most one new edge is incident to each vertex. The two lines drawn between vertices 2 and 6 reflect the two different reasons why these packets must be routed through different networks. However, we intend this to be a simple graph; the two lines still signify a single edge.

Now here's the key insight: *a 2-coloring of the graph corresponds to a solution to the routing problem*. In particular, suppose that we could color each vertex either red or blue so that adjacent vertices are colored differently. Then all constraints are satisfied if we send the red packets through the upper network and the blue packets through the lower network.

The only remaining question is whether the constraint graph is 2-colorable, which is easy to verify:

Problem 3. Prove that if graphs $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ both have maximum degree 1, then the graph $G = (V, E_1 \cup E_2)$ is 2-colorable.

For example, here is a 2-coloring of the constraint graph:



The solution to this graph-coloring problem provides a start on the packet routing problem:

We can complete the routing in the two smaller Beneš networks by induction! Back to the proof. **End of Digression.**

Let π be an arbitrary permutation of $\{0, 1, \ldots, 2N-1\}$. Let $G_1 = (V, E_1)$ be a graph where the vertices are packets 0, 1, ..., $2N-1$ and there is an edge u—v if $|u - v| = N$. Let $G_2 = (V, E_2)$ be a graph with the same vertices and an edge u—v if $|\pi(u) - \pi(v)| = N$. But according to Problem 3, the graph $G = (V, E_1 \cup E_2)$ is 2-colorable, so color the vertices red and blue. Route red packets through the upper subnetwork and blue packets through the lower subnetwork. Since for each edge in E_1, one vertex goes to the upper subnetwork and the other to the lower subnetwork, there will not be any conflicts in the first level. Since for each edge in E_2, one vertex comes from the upper subnetwork and the other from the lower subnetwork, there will not be any conflicts in the last level. We can complete the routing within each subnetwork by the induction hypothesis $P(a)$. □