

CHAPTER 1

GRAPH THEORY

Excerpted from *Math in Society*, by David Lippman

Graph Theory and Network Flows

In the modern world, planning efficient routes is essential for business and industry, with applications as varied as product distribution, laying new fiber optic lines for broadband internet, and suggesting new friends within social network websites like Facebook.

This field of mathematics started nearly 300 years ago as a look into a mathematical puzzle (we'll look at it in a bit). The field has exploded in importance in the last century, both because of the growing complexity of business in a global economy and because of the computational power that computers have provided us.

Sec. 1.1 Graphs

Example 1

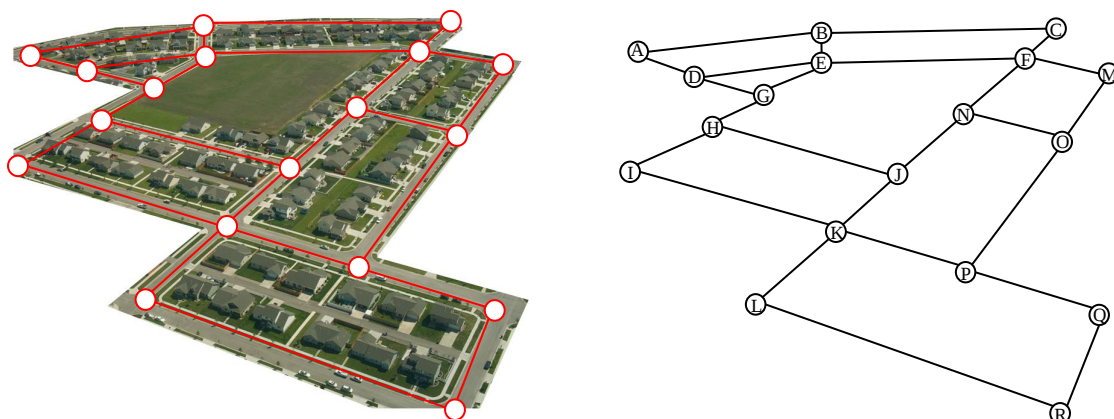
Here is a portion of a housing development from Missoula, Montana¹. As part of her job, the development's lawn inspector has to walk down every street in the development making sure homeowners' landscaping conforms to the community requirements.



¹ Sam Beebe. <http://www.flickr.com/photos/sbeebe/2850476641/>, CC-BY

Naturally, she wants to minimize the amount of walking she has to do. Is it possible for her to walk down every street in this development without having to do any backtracking? While you might be able to answer that question just by looking at the picture for a while, it would be ideal to be able to answer the question for any picture regardless of its complexity.

To do that, we first need to simplify the picture into a form that is easier to work with. We can do that by drawing a simple line for each street. Where streets intersect, we will place a dot.

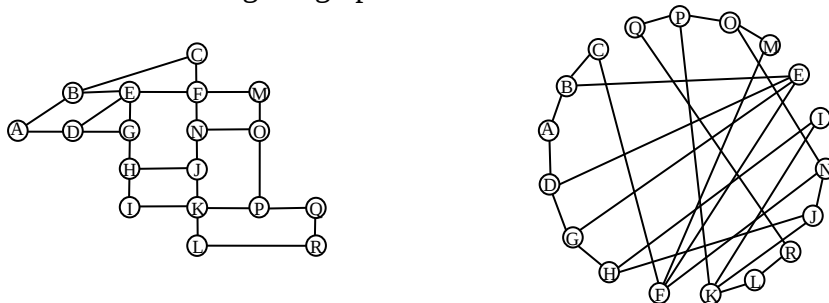


This type of simplified picture is called a **graph**.

Graphs, Vertices, and Edges

A **graph** consists of a set of dots, called **vertices**, and a set of **edges** connecting pairs of vertices.

While we drew our original graph to correspond with the picture we had, there is nothing particularly important about the layout when we analyze a graph. Both of the graphs below are equivalent to the one drawn above since they show the same edge connections between the same vertices as the original graph.

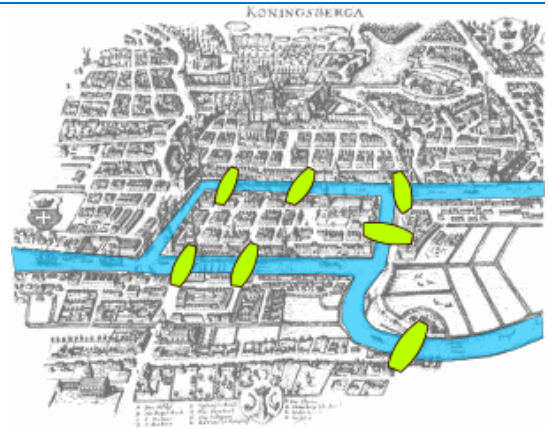


You probably already noticed that we are using the term *graph* differently than you may have used the term in the past to describe the graph of a mathematical function.

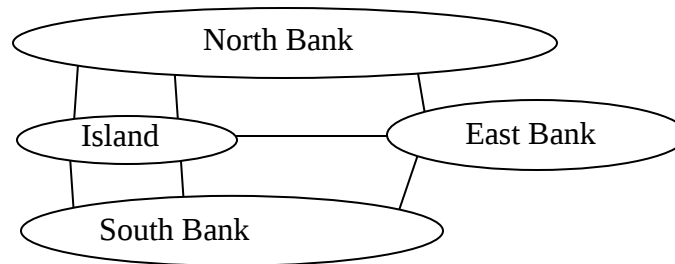
Example 2

Back in the 18th century in the Prussian city of Königsberg, a river ran through the city and seven bridges crossed the forks of the river. The river and the bridges are highlighted in the picture to the right².

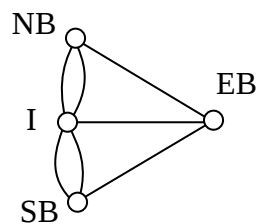
As a weekend amusement, townsfolk would see if they could find a route that would take them across every bridge once and return them to where they started.



Leonard Euler (pronounced OY-lur), one of the most prolific mathematicians ever, looked at this problem in 1735, laying the foundation for graph theory as a field in mathematics. To analyze this problem, Euler introduced edges representing the bridges:



Since the size of each land mass it is not relevant to the question of bridge crossings, each can be shrunk down to a vertex representing the location:



Notice that in this graph there are *two* edges connecting the north bank and island, corresponding to the two bridges in the original drawing. Depending upon the interpretation of edges and vertices appropriate to a scenario, it is entirely possible and reasonable to have more than one edge connecting two vertices.

While we haven't answered the actual question yet of whether or not there is a route which crosses every bridge once and returns to the starting location, the graph provides the foundation for exploring this question.

² Bogdan Giuscă. http://en.wikipedia.org/wiki/File:Königsberg_bridges.png

Definitions

While we loosely defined some terminology earlier, we now will try to be more specific.

Vertex

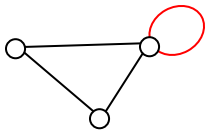
A vertex is a dot in the graph that could represent an intersection of streets, a land mass, or a general location, like “work” or “school”. Vertices are often connected by edges. Note that vertices only occur when a dot is explicitly placed, not whenever two edges cross. Imagine a freeway overpass – the freeway and side street cross, but it is not possible to change from the side street to the freeway at that point, so there is no intersection and no vertex would be placed.

Edges

Edges connect pairs of vertices. An edge can represent a physical connection between locations, like a street, or simply that a route connecting the two locations exists, like an airline flight.

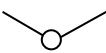
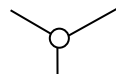
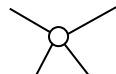
Loop

A loop is a special type of edge that connects a vertex to itself. Loops are not used much in street network graphs.



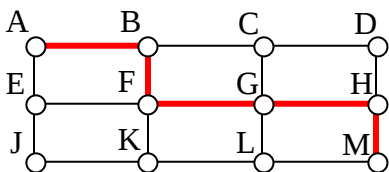
Degree of a vertex

The degree of a vertex is the number of edges meeting at that vertex. It is possible for a vertex to have a degree of zero or larger.

Degree 0	Degree 1	Degree 2	Degree 3	Degree 4
				

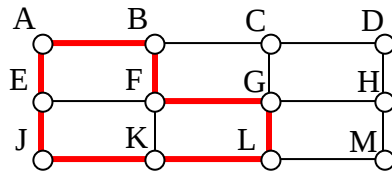
Path

A path is a sequence of vertices using the edges. Usually we are interested in a path between two vertices. For example, a path from vertex A to vertex M is shown below. It is one of many possible paths in this graph.

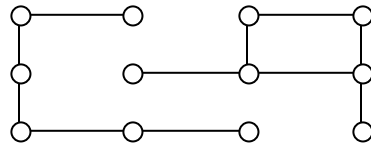


Circuit

A circuit is a path that begins and ends at the same vertex. A circuit starting and ending at vertex A is shown below.

**Connected**

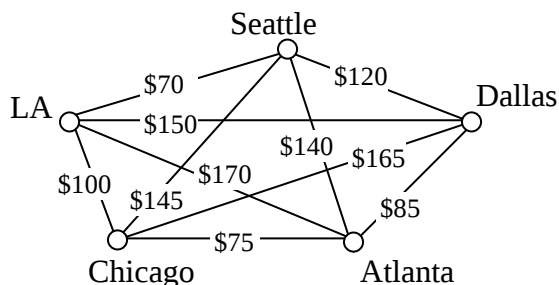
A graph is connected if there is a path from any vertex to any other vertex. Every graph drawn so far has been connected. The graph below is **disconnected**; there is no way to get from the vertices on the left to the vertices on the right.

**Weights**

Depending upon the problem being solved, sometimes weights are assigned to the edges. The weights could represent the distance between two locations, the travel time, or the travel cost. It is important to note that the distance between vertices in a graph does not necessarily correspond to the weight of an edge.

Try it Now 1

1. The graph below shows 5 cities. The weights on the edges represent the airfare for a one-way flight between the cities.
 - a. How many vertices and edges does the graph have?
 - b. Is the graph connected?
 - c. What is the degree of the vertex representing LA?
 - d. If you fly from Seattle to Dallas to Atlanta, is that a path or a circuit?
 - e. If you fly from LA to Chicago to Dallas to LA, is that a path or a circuit?



Sec. 1.2 Shortest Path

When you visit a website like Google Maps or use your Smartphone to ask for directions from home to your Aunt's house in Pasadena, you are usually looking for a shortest path between the two locations. These computer applications use representations of the street maps as graphs, with estimated driving times as edge weights.

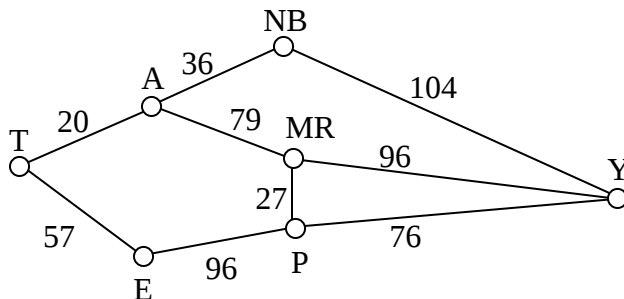
While often it is possible to find a shortest path on a small graph by guess-and-check, our goal in this chapter is to develop methods to solve complex problems in a systematic way by following **algorithms**. An algorithm is a step-by-step procedure for solving a problem. Dijkstra's (pronounced dike-stra) algorithm will find the shortest path between two vertices.

Dijkstra's Algorithm

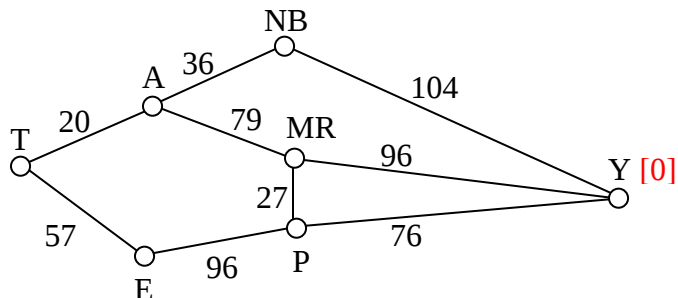
1. Mark the ending vertex with a distance of zero. Designate this vertex as current.
2. Find all vertices leading to the current vertex. Calculate their distances to the end. Since we already know the distance the current vertex is from the end, this will just require adding the most recent edge. Don't record this distance if it is longer than a previously recorded distance.
3. Mark the current vertex as visited. We will never look at this vertex again.
4. Mark the vertex with the smallest distance as current, and repeat from step 2.

Example 3

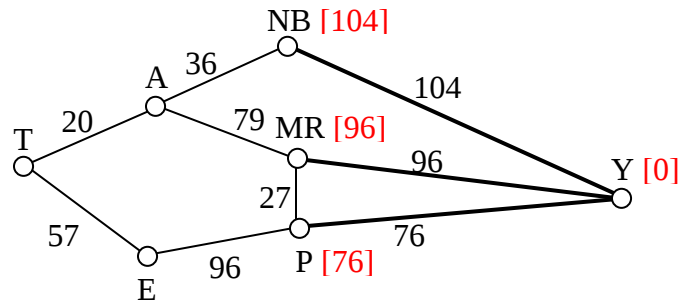
Suppose you need to travel from Tacoma, WA (vertex T) to Yakima, WA (vertex Y). Looking at a map, it looks like driving through Auburn (A) then Mount Rainier (MR) might be shortest, but it's not totally clear since that road is probably slower than taking the major highway through North Bend (NB). A graph with travel times in minutes is shown below. An alternate route through Eatonville (E) and Packwood (P) is also shown.



Step 1: Mark the ending vertex with a distance of zero. The distances will be recorded in [brackets] after the vertex name.



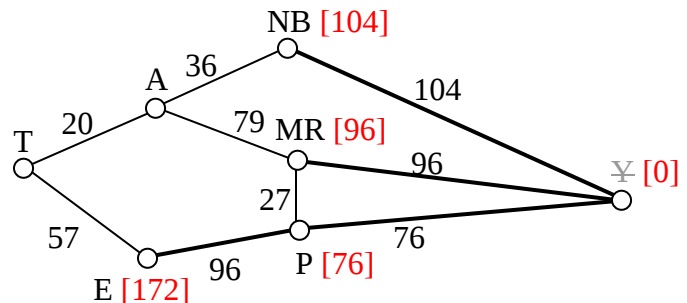
Step 2: For each vertex leading to Y, we calculate the distance to the end. For example, NB is a distance of 104 from the end, and MR is 96 from the end. Remember that distances in this case refer to the travel time in minutes.



Step 3 & 4: We mark Y as visited, and mark the vertex with the smallest recorded distance as current. At this point, P will be designated current. Back to step 2.

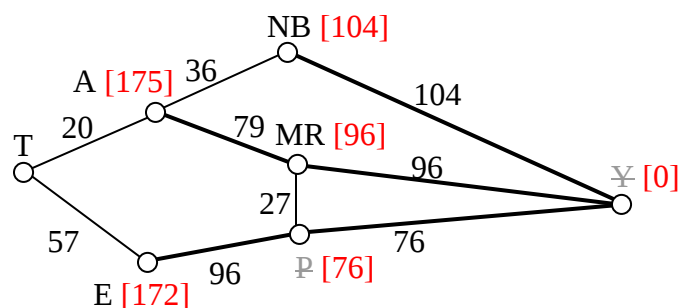
Step 2 (#2): For each vertex leading to P (and not leading to a visited vertex) we find the distance from the end. Since E is 96 minutes from P, and we've already calculated P is 76 minutes from Y, we can compute that E is $96 + 76 = 172$ minutes from Y.

If we make the same computation for MR, we'd calculate $76 + 27 = 103$. Since this is larger than the previously recorded distance from Y to MR, we will *not* replace it.



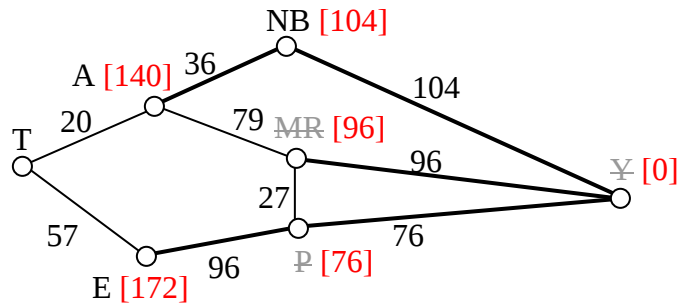
Step 3 & 4 (#2): We mark P as visited, and designate the vertex with the smallest recorded distance as current: MR. Back to step 2.

Step 2 (#3): For each vertex leading to MR (and not leading to a visited vertex) we find the distance to the end. The only vertex to be considered is A, since we've already visited Y and P. Adding MR's distance 96 to the length from A to MR gives the distance $96 + 79 = 175$ minutes from A to Y.



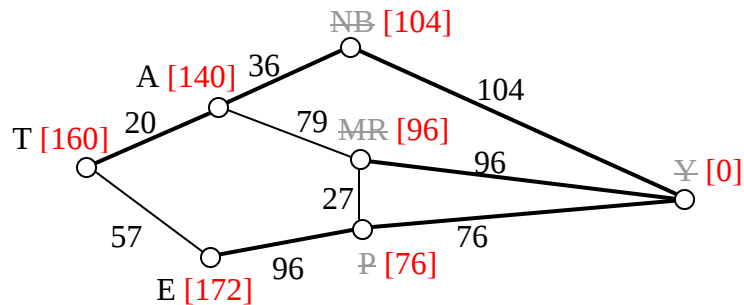
Step 3 & 4 (#3): We mark MR as visited, and designate the vertex with smallest recorded distance as current: NB. Back to step 2.

Step 2 (#4): For each vertex leading to NB, we find the distance to the end. We know the shortest distance from NB to Y is 104 and the distance from A to NB is 36, so the distance from A to Y through NB is $104 + 36 = 140$. Since this distance is shorter than the previously calculated distance from Y to A through MR, we replace it.



Step 3 & 4 (#4): We mark NB as visited, and designate A as current, since it now has the shortest distance.

Step 2 (#5): T is the only non-visited vertex leading to A, so we calculate the distance from T to Y through A: $20 + 140 = 160$ minutes.



Step 3 & 4 (#5): We mark A as visited, and designate E as current.

Step 2 (#6): The only non-visited vertex leading to E is T. Calculating the distance from T to Y through E, we compute $172 + 57 = 229$ minutes. Since this is longer than the existing marked time, we do not replace it.

Step 3 (#6): We mark E as visited. Since all vertices have been visited, we are done.

From this, we know that the shortest path from Tacoma to Yakima will take 160 minutes. Tracking which sequence of edges yielded 160 minutes, we see the shortest path is T-A-NB-Y.

Dijkstra's algorithm is an **optimal algorithm**, meaning that it always produces the actual shortest path, not just a path that is pretty short, provided one exists. This algorithm is also **efficient**, meaning that it can be implemented in a reasonable amount of time. Dijkstra's algorithm takes around V^2 calculations, where V is the number of vertices in a graph³. A graph with 100 vertices would take around 10,000 calculations. While that would be a lot to do by hand, it is not a lot for computer to handle. It is because of this efficiency that your car's GPS unit can compute driving directions in only a few seconds.

³ It can be made to run faster through various optimizations to the implementation.

In contrast, an **inefficient** algorithm might try to list all possible paths then compute the length of each path. Trying to list all possible paths could easily take 10^{25} calculations to compute the shortest path with only 25 vertices; that's a 1 with 25 zeros after it! To put that in perspective, the fastest computer in the world would still spend over 1000 years analyzing all those paths.

Example 4

A shipping company needs to route a package from Washington, D.C. to San Diego, CA. To minimize costs, the package will first be sent to their processing center in Baltimore, MD then sent as part of mass shipments between their various processing centers, ending up in their processing center in Bakersfield, CA. From there it will be delivered in a small truck to San Diego.

The travel times, in hours, between their processing centers are shown in the table below. Three hours has been added to each travel time for processing. Find the shortest path from Baltimore to Bakersfield.

	Baltimore	Denver	Dallas	Chicago	Atlanta	Bakersfield
Baltimore	*			15	14	
Denver		*		18	24	19
Dallas			*	18	15	25
Chicago	15	18	18	*	14	
Atlanta	14	24	15	14	*	
Bakersfield		19	25			*

While we could draw a graph, we can also work directly from the table.

Step 1: The ending vertex, Bakersfield, is marked as current.

Step 2: All cities connected to Bakersfield, in this case Denver and Dallas, have their distances calculated; we'll mark those distances in the column headers.

Step 3 & 4: Mark Bakersfield as visited. Here, we are doing it by shading the corresponding row and column of the table. We mark Denver as current, shown in bold, since it is the vertex with the shortest distance.

	Baltimore	Denver [19]	Dallas [25]	Chicago	Atlanta	Bakersfield [0]
Baltimore	*			15	14	
Denver		*		18	24	19
Dallas			*	18	15	25
Chicago	15	18	18	*	14	
Atlanta	14	24	15	14	*	
Bakersfield		19	25			*

Step 2 (#2): For cities connected to Denver, calculate distance to the end. For example, Chicago is 18 hours from Denver, and Denver is 19 hours from the end, the distance for Chicago to the end is $18+19 = 37$ (Chicago to Denver to Bakersfield). Atlanta is 24 hours from Denver, so the distance to the end is $24+19 = 43$ (Atlanta to Denver to Bakersfield).

Step 3 & 4 (#2): We mark Denver as visited and mark Dallas as current.

	Baltimore	Denver [19]	Dallas [25]	Chicago [37]	Atlanta [43]	Bakersfield [0]
Baltimore	*			15	14	
Denver		*		18	24	19
Dallas			*	18	15	25
Chicago	15	18	18	*	14	
Atlanta	14	24	15	14	*	
Bakersfield		19	25			*

Step 2 (#3): For cities connected to Dallas, calculate the distance to the end. For Chicago, the distance from Chicago to Dallas is 18 and from Dallas to the end is 25, so the distance from Chicago to the end through Dallas would be $18+25 = 43$. Since this is longer than the currently marked distance for Chicago, we do not replace it. For Atlanta, we calculate $15+25 = 40$. Since this is shorter than the currently marked distance for Atlanta, we replace the existing distance.

Step 3 & 4 (#3): We mark Dallas as visited, and mark Chicago as current.

	Baltimore	Denver [19]	Dallas [25]	Chicago [37]	Atlanta [40]	Bakersfield [0]
Baltimore	*			15	14	
Denver		*		18	24	19
Dallas			*	18	15	25
Chicago	15	18	18	*	14	
Atlanta	14	24	15	14	*	
Bakersfield		19	25			*

Step 2 (#4): Baltimore and Atlanta are the only non-visited cities connected to Chicago. For Baltimore, we calculate $15+37 = 52$ and mark that distance. For Atlanta, we calculate $14+37 = 51$. Since this is longer than the existing distance of 40 for Atlanta, we do not replace that distance.

Step 3 & 4 (#4): Mark Chicago as visited and Atlanta as current.

	Baltimore [52]	Denver [19]	Dallas [25]	Chicago [37]	Atlanta [40]	Bakersfield [0]
Baltimore	*			15	14	
Denver		*		18	24	19
Dallas			*	18	15	25
Chicago	15	18	18	*	14	
Atlanta	14	24	15	14	*	
Bakersfield		19	25			*

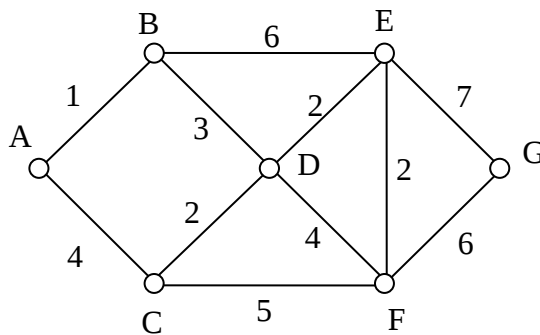
Step 2 (#5): The distance from Atlanta to Baltimore is 14. Adding that to the distance already calculated for Atlanta gives a total distance of $14 + 40 = 54$ hours from Baltimore to Bakersfield through Atlanta. Since this is larger than the currently calculated distance, we do not replace the distance for Baltimore.

Step 3 & 4 (#5): We mark Atlanta as visited. All cities have been visited and we are done.

The shortest route from Baltimore to Bakersfield will take 52 hours, and will route through Chicago and Denver.

Try it Now 2

Find the shortest path between vertices A and G in the graph below.



Sec. 1.3 Euler Circuits and the Chinese Postman Problem

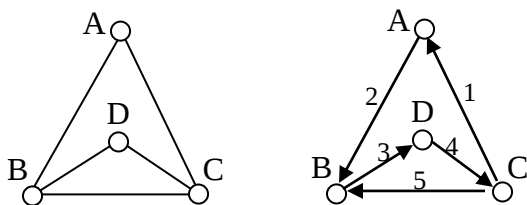
In the first section, we created a graph of the Königsberg bridges and asked whether it was possible to walk across every bridge once. Because Euler first studied this question, these types of paths are named after him.

Euler Path

An **Euler path** is a path that uses every edge in a graph with no repeats. Being a path, it does not have to return to the starting vertex.

Example 5

In the graph shown below, there are several Euler paths. One such path is CABDCB. The path is shown in arrows to the right, with the order of edges numbered.

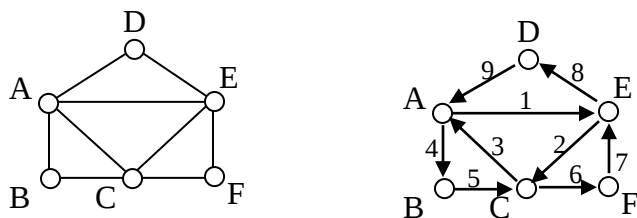


Euler Circuit

An **Euler circuit** is a circuit that uses every edge in a graph with no repeats. Being a circuit, it must start and end at the same vertex.

Example 6

The graph below has several possible Euler circuits. Here's a couple, starting and ending at vertex A: ADEACEFCBA and AECABCFEDA. The second is shown in arrows.



Look back at the example used for Euler paths – does that graph have an Euler circuit? A few tries will tell you no; that graph does not have an Euler circuit. When we were working with shortest paths, we were interested in the optimal path. With Euler paths and circuits, we're primarily interested in whether an Euler path or circuit *exists*.

Why do we care if an Euler circuit exists? Think back to our housing development lawn inspector from the beginning of the chapter. The lawn inspector is interested in walking as little as possible. The ideal situation would be a circuit that covers every street with no repeats. That's an Euler circuit! Luckily, Euler solved the question of whether or not an Euler path or circuit will exist.

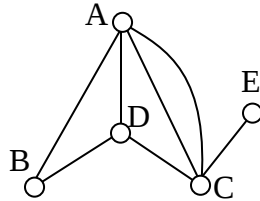
Euler's Path and Circuit Theorems

A graph will contain an Euler path if it contains at most two vertices of odd degree.

A graph will contain an Euler circuit if all vertices have even degree

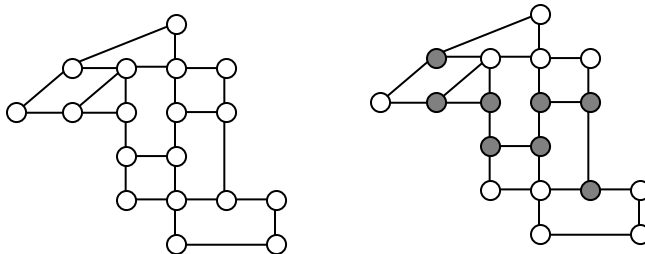
Example 7

In the graph below, vertices A and C have degree 4, since there are 4 edges leading into each vertex. B is degree 2, D is degree 3, and E is degree 1. This graph contains two vertices with odd degree (D and E) and three vertices with even degree (A, B, and C), so Euler's theorems tell us this graph has an Euler path, but not an Euler circuit.



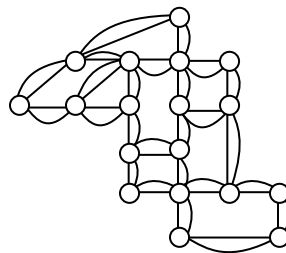
Example 8

Is there an Euler circuit on the housing development lawn inspector graph we created earlier in the chapter? All the highlighted vertices have odd degree. Since there are more than two vertices with odd degree, there are no Euler paths or Euler circuits on this graph. Unfortunately our lawn inspector will need to do some backtracking.



Example 9

When it snows in the same housing development, the snowplow has to plow both sides of every street. For simplicity, we'll assume the plow is out early enough that it can ignore traffic laws and drive down either side of the street in either direction. This can be visualized in the graph by drawing two edges for each street, representing the two sides of the street.



Notice that every vertex in this graph has even degree, so this graph does have an Euler circuit.

Now we know how to determine if a graph has an Euler circuit, but if it does, how do we find one? While it usually is possible to find an Euler circuit just by pulling out your pencil and trying to find one, the more formal method is **Fleury's algorithm**.

Fleury's Algorithm

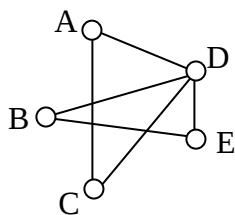
1. Start at any vertex if finding an Euler circuit. If finding an Euler path, start at one of the two vertices with odd degree.
2. Choose any edge leaving your current vertex, provided deleting that edge will not separate the graph into two disconnected sets of edges.
3. Add that edge to your circuit, and delete it from the graph.
4. Continue until you're done.

Example 10

Let's find an Euler Circuit on this graph using Fleury's algorithm, starting at vertex A.

Original Graph.

Choosing edge AD.

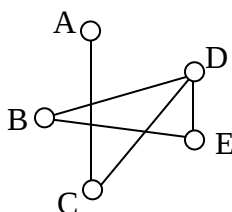


Circuit so far: AD

AD deleted. D is current.

Can't choose DC since that would disconnect graph.

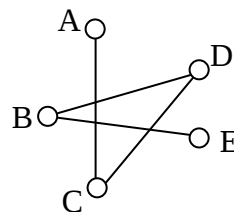
Choosing DE



Circuit so far: ADE

E is current.

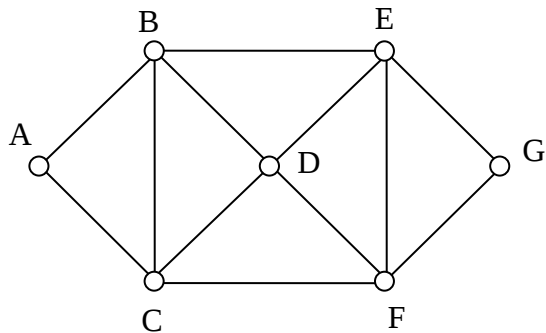
From here, there is only one option, so the rest of the circuit is determined.



Circuit: ADEBDCA

Try it Now 3

Does the graph below have an Euler Circuit? If so, find one.



Eulerization and the Chinese Postman Problem

Not every graph has an Euler path or circuit, yet our lawn inspector still needs to do her inspections. Her goal is to minimize the amount of walking she has to do. In order to do that, she will have to duplicate some edges in the graph until an Euler circuit exists.

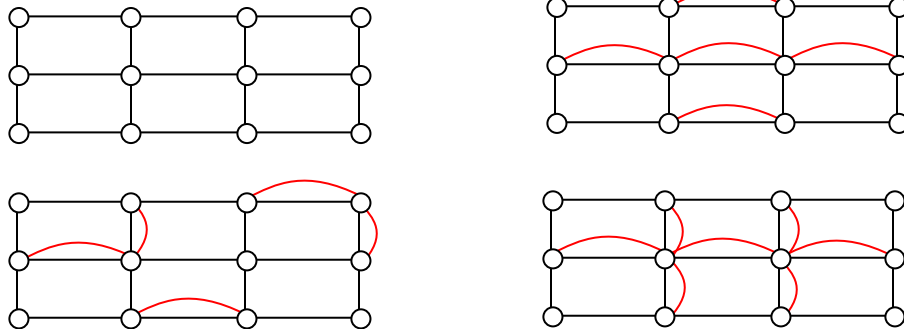
Eulerization

Eulerization is the process of adding edges to a graph to create an Euler circuit on a graph. To eulerize a graph, edges are duplicated to connect pairs of vertices with odd degree. Connecting two odd degree vertices increases the degree of each, giving them both even degree. When two odd degree vertices are not directly connected, we can duplicate all edges in a path connecting the two.

Note that we can only duplicate edges, not create edges where there wasn't one before. Duplicating edges would mean walking or driving down a road twice, while creating an edge where there wasn't one before is akin to installing a new road!

Example 11

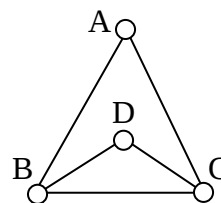
For the rectangular graph shown, three possible eulerizations are shown. Notice in each of these cases the vertices that started with odd degrees have even degrees after eulerization, allowing for an Euler circuit.



In the example above, you'll notice that the last eulerization required duplicating seven edges, while the first two only required duplicating five edges. If we were eulerizing the graph to find a walking path, we would want the eulerization with minimal duplications. If the edges had weights representing distances or costs, then we would want to select the eulerization with the minimal total added weight.

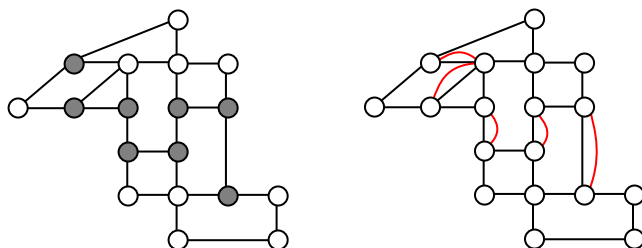
Try it Now 4

Eulerize the graph shown, then find an Euler circuit on the eulerized graph.



Example 12

Looking again at the graph for our lawn inspector from Examples 1 and 8, the vertices with odd degree are shown highlighted. With eight vertices, we will always have to duplicate at least four edges. In this case, we need to duplicate five edges since two odd degree vertices are not directly connected. Without weights we can't be certain this is the eulerization that minimizes walking distance, but it looks pretty good.



The problem of finding the optimal eulerization is called the Chinese Postman Problem, a name given by an American in honor of the Chinese mathematician Mei-Ko Kwan who first studied the problem in 1962 while trying to find optimal delivery routes for postal carriers. This problem is important in determining efficient routes for garbage trucks, school buses, parking meter checkers, street sweepers, and more.

Unfortunately, algorithms to solve this problem are fairly complex. Some simpler cases are considered in the exercises.

Sec. 1.4 Hamiltonian Circuits and the Traveling Salesman Problem

In the last section, we considered optimizing a walking route for a postal carrier. How is this different than the requirements of a package delivery driver? While the postal carrier needed to walk down every street (edge) to deliver the mail, the package delivery driver instead needs to visit every one of a set of delivery locations. Instead of looking for a circuit that covers every edge once, the package deliverer is interested in a circuit that visits every vertex once.

Hamiltonian Circuits and Paths

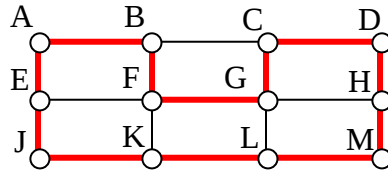
A **Hamiltonian circuit** is a circuit that visits every vertex once with no repeats. Being a circuit, it must start and end at the same vertex. A **Hamiltonian path** also visits every vertex once with no repeats, but does not have to start and end at the same vertex.

Hamiltonian circuits are named for William Rowan Hamilton who studied them in the 1800's.

Example 13

One Hamiltonian circuit is shown on the graph below. There are several other Hamiltonian circuits possible on this graph. Notice that the circuit only has to visit every vertex once; it does not need to use every edge.

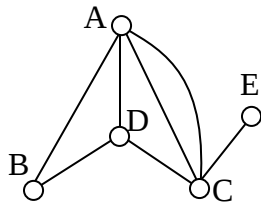
This circuit could be notated by the sequence of vertices visited, starting and ending at the same vertex: ABFGCDHMLKJEA. Notice that the same circuit could be written in reverse order, or starting and ending at a different vertex.



Unlike with Euler circuits, there is no nice theorem that allows us to instantly determine whether or not a Hamiltonian circuit exists for all graphs.⁴

Example 14

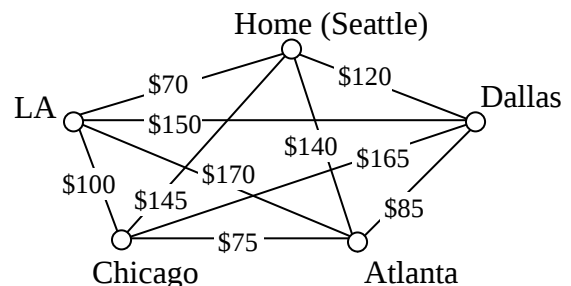
Does a Hamiltonian path or circuit exist on the graph below?



We can see that once we travel to vertex E there is no way to leave without returning to C, so there is no possibility of a Hamiltonian circuit. If we start at vertex E we can find several Hamiltonian paths, such as ECDAB and ECABD.

With Hamiltonian circuits, our focus will not be on existence, but on the question of optimization; given a graph where the edges have weights, can we find the optimal Hamiltonian circuit; the one with lowest total weight.

This problem is called the **Traveling salesman problem** (TSP) because the question can be framed like this: Suppose a salesman needs to give sales pitches in four cities. He looks up the airfares between each city, and puts the costs in a graph. In what order should he travel to visit each city once then return home with the lowest cost?



⁴ There are some theorems that can be used in specific circumstances, such as Dirac's theorem, which says that a Hamiltonian circuit must exist on a graph with n vertices if each vertex has degree $n/2$ or greater.

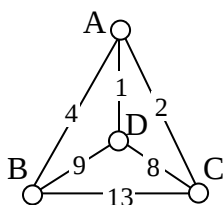
To answer this question of how to find the lowest cost Hamiltonian circuit, we will consider some possible approaches. The first option that might come to mind is to just try all different possible circuits.

Brute Force Algorithm (a.k.a. exhaustive search)

1. List all possible Hamiltonian circuits
2. Find the length of each circuit by adding the edge weights
3. Select the circuit with minimal total weight.

Example 15

Apply the Brute force algorithm to find the minimum cost Hamiltonian circuit on the graph below.



To apply the Brute force algorithm, we list all possible Hamiltonian circuits and calculate their weight:

Circuit	Weight
ABCD A	$4+13+8+1 = 26$
ABDCA	$4+9+8+2 = 23$
ACBDA	$2+13+9+1 = 25$

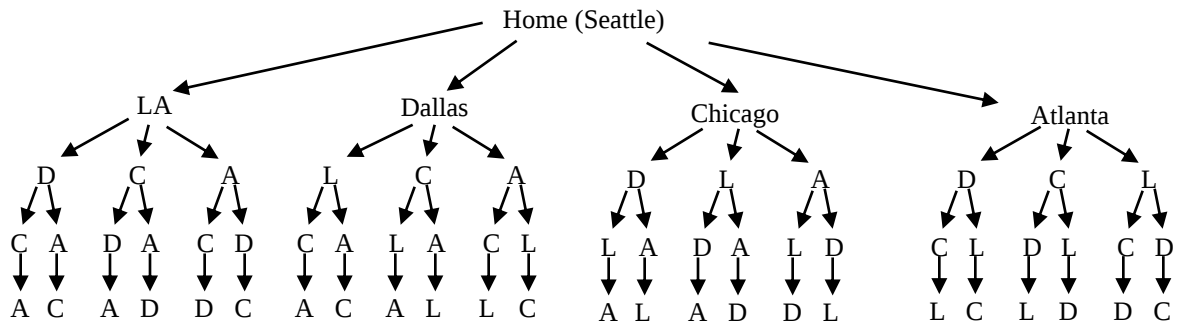
Note: These are the unique circuits on this graph. All other possible circuits are the reverse of the listed ones or start at a different vertex, but result in the same weights.

From this we can see that the second circuit, ABDCA, is the optimal circuit.

The Brute force algorithm is optimal; it will always produce the Hamiltonian circuit with minimum weight. Is it efficient? To answer that question, we need to consider how many Hamiltonian circuits a graph could have. For simplicity, let's look at the worst-case possibility, where every vertex is connected to every other vertex. This is called a **complete graph**.

Suppose we had a complete graph with five vertices like the air travel graph above. From Seattle there are four cities we can visit first. From each of those, there are three choices. From each of those cities, there are two possible cities to visit next. There is then only one choice for the last city before returning home.

This can be shown visually:



Counting the number of routes, we can see there are $4 \cdot 3 \cdot 2 \cdot 1 = 24$ routes. For six cities there would be $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$ routes.

Number of Possible Circuits

For n vertices in a complete graph, there will be $(n-1)! = (n-1)(n-2)(n-3)\cdots 3 \cdot 2 \cdot 1$ routes. Half of these are duplicates in reverse order, so there are $\frac{(n-1)!}{2}$ unique circuits.

The exclamation symbol, $!$, is read “factorial” and is shorthand for the product shown.

Example 16

How many circuits would a complete graph with 8 vertices have?

A complete graph with 8 vertices would have $(8-1)! = 7! = 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 5040$ possible Hamiltonian circuits. Half of the circuits are duplicates of other circuits but in reverse order, leaving 2520 unique routes.

While this is a lot, it doesn’t seem unreasonably huge. But consider what happens as the number of cities increase:

Cities	Unique Hamiltonian Circuits
9	$8!/2 = 20,160$
10	$9!/2 = 181,440$
11	$10!/2 = 1,814,400$
15	$14!/2 = 43,589,145,600$
20	$19!/2 = 60,822,550,204,416,000$

As you can see the number of circuits is growing extremely quickly. If a computer looked at one billion circuits a second, it would still take almost two years to examine all the possible circuits with only 20 cities! Certainly Brute Force is **not** an efficient algorithm.

Unfortunately, no one has yet found an efficient *and* optimal algorithm to solve the TSP, and it is very unlikely anyone ever will. Since it is not practical to use brute force to solve the problem, we turn instead to **heuristic algorithms**; efficient algorithms that give approximate solutions. In other words, heuristic algorithms are fast, but may or may not produce the optimal circuit.

Nearest Neighbor Algorithm (NNA)

1. Select a starting point.
2. Move to the nearest unvisited vertex (the edge with smallest weight).
3. Repeat until the circuit is complete.

Example 17

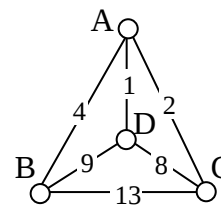
Consider our earlier graph, shown to the right.

Starting at vertex A, the nearest neighbor is vertex D with a weight of 1.

From D, the nearest neighbor is C, with a weight of 8.

From C, our only option is to move to vertex B, the only unvisited vertex, with a cost of 13.

From B we return to A with a weight of 4.



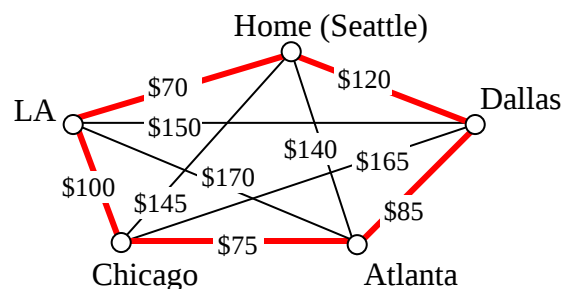
The resulting circuit is ADCBA with a total weight of $1+8+13+4 = 26$.

We ended up finding the worst circuit in the graph! What happened? Unfortunately, while it is very easy to implement, the NNA is a **greedy** algorithm, meaning it only looks at the immediate decision without considering the consequences in the future. In this case, following the edge AD forced us to use the very expensive edge BC later.

Example 18

Consider again our salesman. Starting in Seattle, the nearest neighbor (cheapest flight) is to LA, at a cost of \$70. From there:

LA to Chicago: \$100
 Chicago to Atlanta: \$75
 Atlanta to Dallas: \$85
 Dallas to Seattle: \$120
 Total cost: \$450



In this case, nearest neighbor did find the optimal circuit.

Going back to our first example, how could we improve the outcome? One option would be to redo the nearest neighbor algorithm with a different starting point to see if the result changed. Since nearest neighbor is so fast, doing it several times isn't a big deal.

Repeated Nearest Neighbor Algorithm (RNNA)

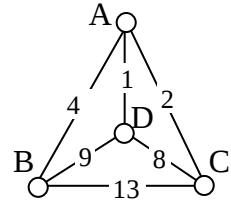
1. Do the Nearest Neighbor Algorithm starting at each vertex
2. Choose the circuit produced with minimal total weight

Example 19

We will revisit the graph from Example 17.

Starting at vertex A resulted in a circuit with weight 26.

Starting at vertex B, the nearest neighbor circuit is BADCB with a weight of $4+1+8+13 = 26$. This is the same circuit we found starting at vertex A. No better.



Starting at vertex C, the nearest neighbor circuit is CADBC with a weight of $2+1+9+13 = 25$. Better!

Starting at vertex D, the nearest neighbor circuit is DACBA. Notice that this is actually the same circuit we found starting at C, just written with a different starting vertex.

The RNNA was able to produce a slightly better circuit with a weight of 25, but still not the optimal circuit in this case. Notice that even though we found the circuit by starting at vertex C, we could still write the circuit starting at A: ADBCA or ACBDA.

Try it Now 5

The table below shows the time, in milliseconds, it takes to send a packet of data between computers on a network. If data needed to be sent in sequence to each computer, then notification needed to come back to the original computer, we would be solving the TSP. The computers are labeled A-F for convenience.

	A	B	C	D	E	F
A	--	44	34	12	40	41
B	44	--	31	43	24	50
C	34	31	--	20	39	27
D	12	43	20	--	11	17
E	40	24	39	11	--	42
F	41	50	27	17	42	--

- a. Find the circuit generated by the NNA starting at vertex B.
- b. Find the circuit generated by the RNNA.

While certainly better than the basic NNA, unfortunately, the RNNA is still greedy and will produce very bad results for some graphs. As an alternative, our next approach will step back and look at the “big picture” – it will select first the edges that are shortest, and then fill in the gaps.

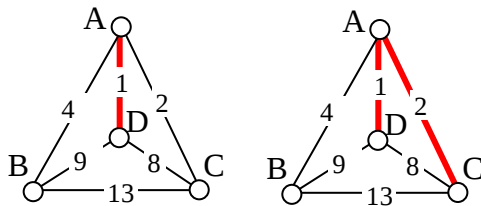
Sorted Edges Algorithm (a.k.a. Cheapest Link Algorithm)

1. Select the cheapest unused edge in the graph.
2. Repeat step 1, adding the cheapest unused edge to the circuit, unless:
 - a. adding the edge would create a circuit that doesn't contain all vertices, or
 - b. adding the edge would give a vertex degree 3.
3. Repeat until a circuit containing all vertices is formed.

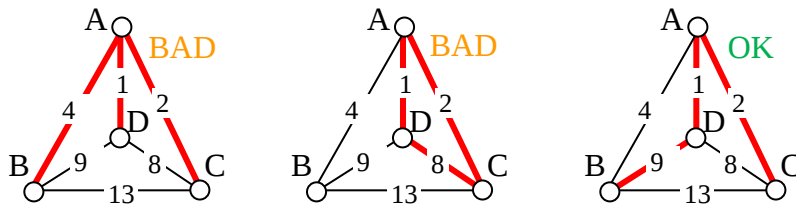
Example 20

Using the four vertex graph from earlier, we can use the Sorted Edges algorithm.

The cheapest edge is AD, with a cost of 1. We highlight that edge to mark it selected. The next shortest edge is AC, with a weight of 2, so we highlight that edge.

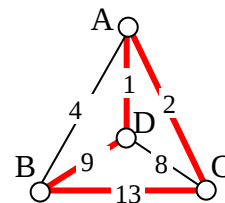


For the third edge, we'd like to add AB, but that would give vertex A degree 3, which is not allowed in a Hamiltonian circuit. The next shortest edge is CD, but that edge would create a circuit ACDA that does not include vertex B, so we reject that edge. The next shortest edge is BD, so we add that edge to the graph.



We then add the last edge to complete the circuit: ACBDA with weight 25.

Notice that the algorithm did not produce the optimal circuit in this case; the optimal circuit is ACDBA with weight 23.



While the Sorted Edge algorithm overcomes some of the shortcomings of NNA, it is still only a heuristic algorithm, and does not guarantee the optimal circuit.

Example 21

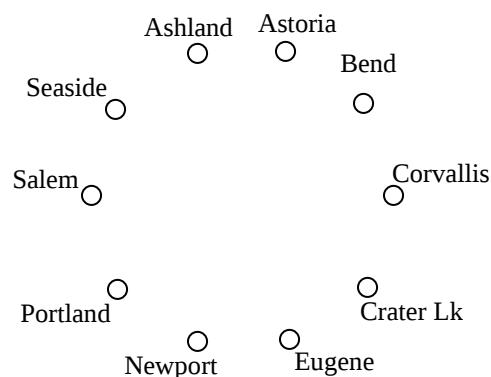
Your teacher's band, *Derivative Work*, is doing a bar tour in Oregon. The driving distances are shown below. Plan an efficient route for your teacher to visit all the cities and return to the starting location. Use NNA starting at Portland, and then use Sorted Edges.

	Ashland	Astoria	Bend	Corvallis	Crater Lake	Eugene	Newport	Portland	Salem	Seaside
Ashland	-	374	200	223	108	178	252	285	240	356
Astoria	374	-	255	166	433	199	135	95	136	17
Bend	200	255	-	128	277	128	180	160	131	247
Corvallis	223	166	128	-	430	47	52	84	40	155
Crater Lake	108	433	277	430	-	453	478	344	389	423
Eugene	178	199	128	47	453	-	91	110	64	181
Newport	252	135	180	52	478	91	-	114	83	117
Portland	285	95	160	84	344	110	114	-	47	78
Salem	240	136	131	40	389	64	83	47	-	118
Seaside	356	17	247	155	423	181	117	78	118	-

Using NNA with a large number of cities, you might find it helpful to mark off the cities as they're visited to keep from accidentally visiting them again. Looking in the row for Portland, the smallest distance is 47, to Salem. Following that idea, our circuit will be:

Portland to Salem 47
 Salem to Corvallis 40
 Corvallis to Eugene 47
 Eugene to Newport 91
 Newport to Seaside 117
 Seaside to Astoria 17
 Astoria to Bend 255
 Bend to Ashland 200
 Ashland to Crater Lake 108
 Crater Lake to Portland 344
 Total trip length: 1266 miles

Using Sorted Edges, you might find it helpful to draw an empty graph, perhaps by drawing vertices in a circular pattern. Adding edges to the graph as you select them will help you visualize any circuits or vertices with degree 3.



We start adding the shortest edges:

Seaside to Astoria	17 miles
Corvallis to Salem	40 miles
Portland to Salem	47 miles
Corvallis to Eugene	47 miles

The graph after adding these edges is shown to the right. The next shortest edge is from Corvallis to Newport at 52 miles, but adding that edge would give Corvallis degree 3.

Continuing on, we can skip over any edge pair that contains Salem or Corvallis, since they both already have degree 2.

Portland to Seaside	78 miles
Eugene to Newport	91 miles
Portland to Astoria	(reject – closes circuit)
Ashland to Crater Lk	108 miles

The graph after adding these edges is shown to the right. At this point, we can skip over any edge pair that contains Salem, Seaside, Eugene, Portland, or Corvallis since they already have degree 2.

Newport to Astoria	(reject – closes circuit)
Newport to Bend	180 miles
Bend to Ashland	200 miles

At this point the only way to complete the circuit is to add:

Crater Lk to Astoria	433 miles
----------------------	-----------

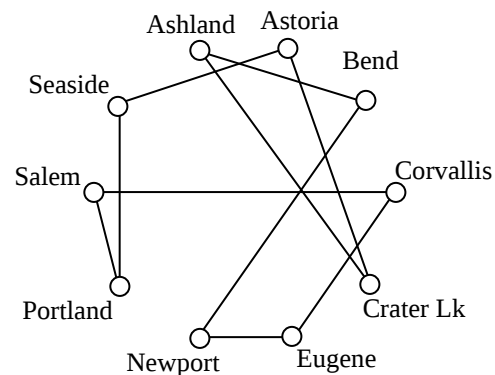
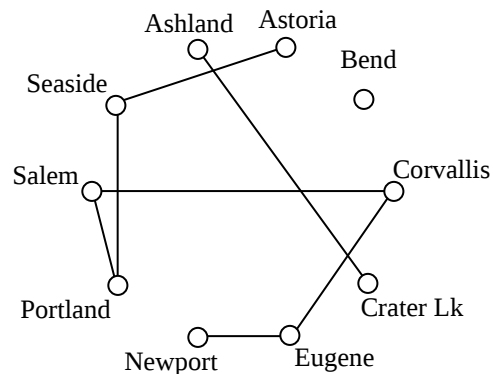
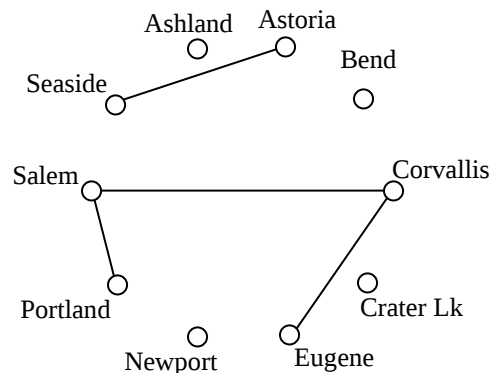
The final circuit, written to start at Portland, is:

Portland, Salem, Corvallis, Eugene, Newport, Bend, Ashland, Crater Lake, Astoria, Seaside, Portland.

Total trip length: 1241 miles.

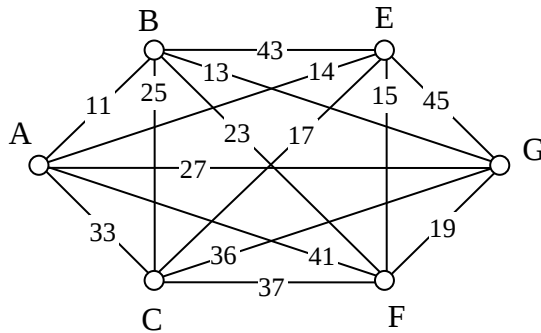
While better than the NNA route, neither algorithm produced the optimal route. The following route can make the tour in 1069 miles:

Portland, Astoria, Seaside, Newport, Corvallis, Eugene, Ashland, Crater Lake, Bend, Salem, Portland

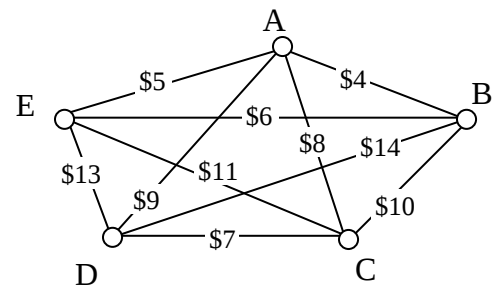


Try it Now 6

Find the circuit produced by the Sorted Edges algorithm using the graph below.

**Sec. 1.5 Networks and Spanning Trees**

A company requires reliable internet and phone connectivity between their five offices (named A, B, C, D, and E for simplicity) in New York, so they decide to lease dedicated lines from the phone company. The phone company will charge for each link made. The costs, in thousands of dollars per year, are shown in the graph.

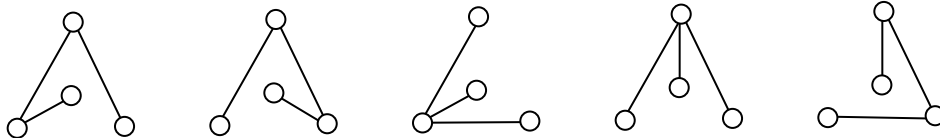


In this case, we don't need to find a circuit, or even a specific path; all we need to do is make sure we can make a call from any office to any other. In other words, we need to be sure there is a path from any vertex to any other vertex.

Spanning Tree

A **spanning tree** is a connected graph using all vertices in which there are no circuits. In other words, there is a path from any vertex to any other vertex, but no circuits.

Some examples of spanning trees are shown below. Notice there are no circuits in the trees, and it is fine to have vertices with degree higher than two.



Usually we have a starting graph to work from, like in the phone example above. In this case, we form our spanning tree by finding a **subgraph** – a new graph formed using all the vertices but only some of the edges from the original graph. No edges will be created where they didn't already exist.

Of course, any random spanning tree isn't really what we want. We want the **minimum cost spanning tree (MCST)**.

Minimum Cost Spanning Tree (MCST)

The minimum cost spanning tree is the spanning tree with the smallest total edge weight.

A nearest neighbor style approach doesn't make as much sense here since we don't need a circuit, so instead we will take an approach similar to sorted edges.

Kruskal's Algorithm

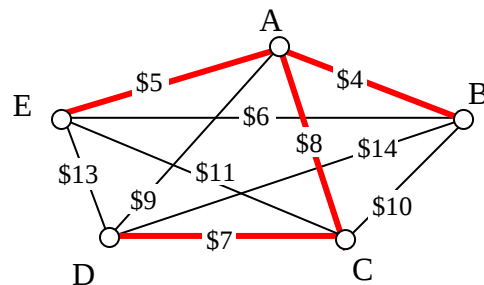
1. Select the cheapest unused edge in the graph.
2. Repeat step 1, adding the cheapest unused edge, unless:
 - a. adding the edge would create a circuit
3. Repeat until a spanning tree is formed

Example 22

Using our phone line graph from above, begin adding edges:

AB	\$4	OK
AE	\$5	OK
BE	\$6	reject – closes circuit ABEA
DC	\$7	OK
AC	\$8	OK

At this point we stop – every vertex is now connected, so we have formed a spanning tree with cost \$24 thousand a year.



Remarkably, Kruskal's algorithm is both optimal and efficient; we are guaranteed to always produce the optimal MCST.

Example 23

The power company needs to lay updated distribution lines connecting the ten Oregon cities below to the power grid. How can they minimize the amount of new line to lay?

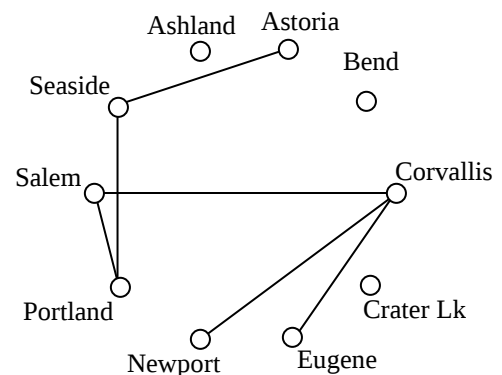
	Ashland	Astoria	Bend	Corvallis	Crater Lake	Eugene	Newport	Portland	Salem	Seaside
Ashland	-	374	200	223	108	178	252	285	240	356
Astoria	374	-	255	166	433	199	135	95	136	17
Bend	200	255	-	128	277	128	180	160	131	247
Corvallis	223	166	128	-	430	47	52	84	40	155
Crater Lake	108	433	277	430	-	453	478	344	389	423
Eugene	178	199	128	47	453	-	91	110	64	181
Newport	252	135	180	52	478	91	-	114	83	117
Portland	285	95	160	84	344	110	114	-	47	78
Salem	240	136	131	40	389	64	83	47	-	118
Seaside	356	17	247	155	423	181	117	78	118	-

Using Kruskal's algorithm, we add edges from cheapest to most expensive, rejecting any that close a circuit. We stop when the graph is connected.

Seaside to Astoria 17 miles
 Corvallis to Salem 40 miles
 Portland to Salem 47 miles
 Corvallis to Eugene 47 miles
 Corvallis to Newport 52 miles
 Salem to Eugene reject – closes circuit
 Portland to Seaside 78 miles

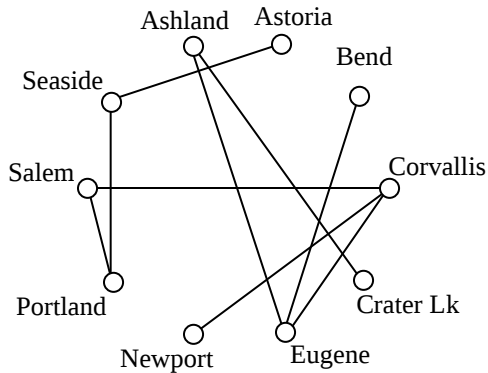
The graph up to this point is shown to the right.
 Continuing,

Newport to Salem reject
 Corvallis to Portland reject
 Eugene to Newport reject
 Portland to Astoria reject
 Ashland to Crater Lk 108 miles
 Eugene to Portland reject
 Newport to Portland reject
 Newport to Seaside reject
 Salem to Seaside reject
 Bend to Eugene 128 miles
 Bend to Salem reject



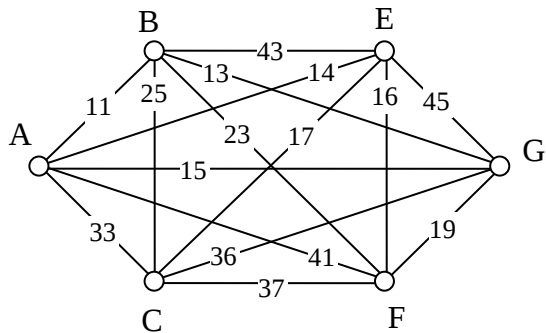
Astoria to Newport	reject
Salem to Astoria	reject
Corvallis to Seaside	reject
Portland to Bend	reject
Astoria to Corvallis	reject
Eugene to Ashland	178 miles

This connects the graph. The total length of cable to lay would be 695 miles.



Try it Now 7

Find a minimum cost spanning tree on the graph below using Kruskal's algorithm.



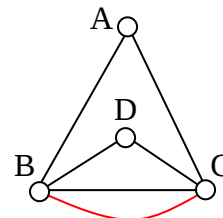
Try it Now Answers

- 5 vertices, 10 edges
- Yes, it is connected
- The vertex is degree 4
- A path
- A circuit

2. The shortest path is ABDEG, with length 13

3. Yes, all vertices have even degree so this graph has an Euler Circuit. There are several possibilities. One is: ABEGFCDFEDBCA

4. This graph can be eulerized by duplicating the edge BC, as shown. One possible Euler circuit on the eulerized graph is ACDBCBA



Try it Now Answers Continued

5. At each step, we look for the nearest location we haven't already visited.

From B the nearest computer is E with time 24.

From E, the nearest computer is D with time 11.

From D the nearest is A with time 12.

From A the nearest is C with time 34.

From C, the only computer we haven't visited is F with time 27

From F, we return back to B with time 50.

The NNA circuit from B is BEDACFB with time 158 milliseconds.

Using NNA again from other starting vertices:

Starting at A: ADEBCFA: time 146

Starting at C: CDEBAFC: time 167

Starting at D: DEBCFAD: time 146

Starting at E: EDACFBE: time 158

Starting at F: FDEBCAF: time 158

The RNNA found a circuit with time 146 milliseconds: ADEBCFA. We could also write this same circuit starting at B if we wanted: BCFADEB or BEDACFB.

6.

AB: Add, cost 11

BG: Add, cost 13

AE: Add, cost 14

EF: Add, cost 15

EC: Skip (degree 3 at E)

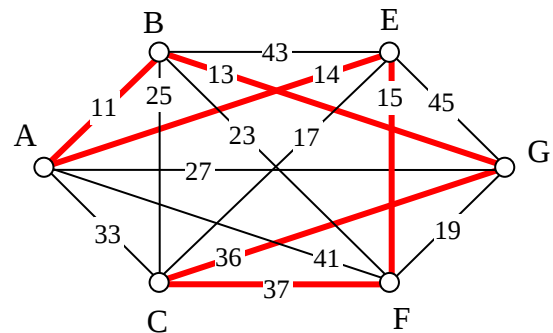
FG: Skip (would create a circuit not including C)

BF, BC, AG, AC: Skip (would cause a vertex to have degree 3)

GC: Add, cost 36

CF: Add, cost 37, completes the circuit

Final circuit: ABGCFEA



7.

AB: Add, cost 11

BG: Add, cost 13

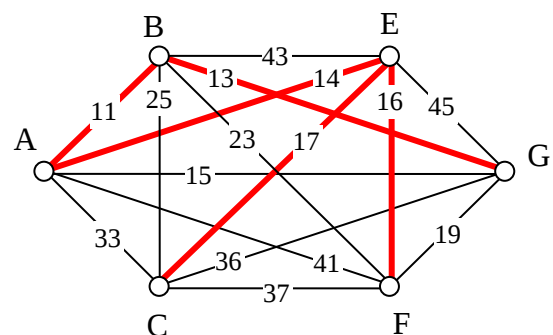
AE: Add, cost 14

AG: Skip, would create circuit ABGA

EF: Add, cost 16

EC: Add, cost 17

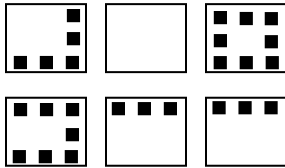
This completes the spanning tree



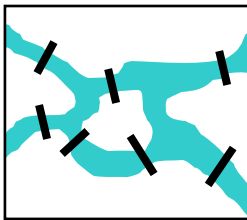
Exercises

Skills

1. To deliver mail in a particular neighborhood, the postal carrier needs to walk along each of the streets with houses (the dots). Create a graph with edges showing where the carrier must walk to deliver the mail.



2. Suppose that a town has 7 bridges as pictured below. Create a graph that could be used to determine if there is a path that crosses all bridges once.



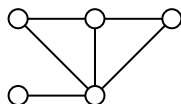
3. The table below shows approximate driving times (in minutes, without traffic) between five cities in the Dallas area. Create a weighted graph representing this data.

	Plano	Mesquite	Arlington	Denton
Fort Worth	54	52	19	42
Plano		38	53	41
Mesquite			43	56
Arlington				50

4. Shown in the table below are the one-way airfares between 5 cities⁵. Create a graph showing this data.

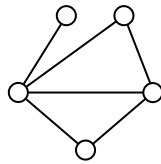
	Honolulu	London	Moscow	Cairo
Seattle	\$159	\$370	\$654	\$684
Honolulu		\$830	\$854	\$801
London			\$245	\$323
Moscow				\$329

5. Find the degree of each vertex in the graph below.

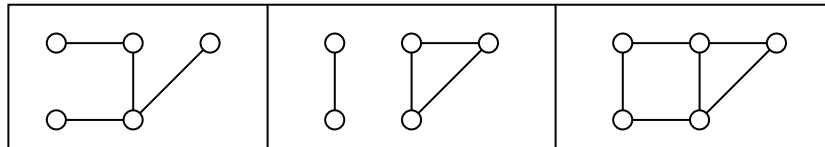


⁵ Cheapest fares found when retrieved Sept 1, 2009 for travel Sept 22, 2009

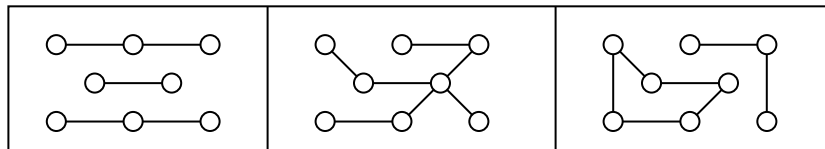
6. Find the degree of each vertex in the graph below.



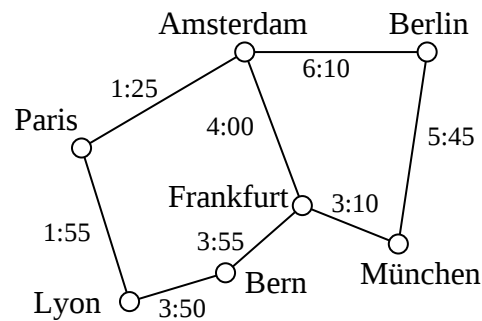
7. Which of these graphs are connected?



8. Which of these graphs are connected?

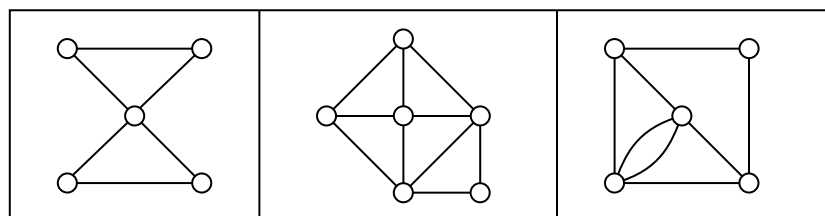


9. Travel times by rail for a segment of the Eurail system is shown below with travel times in hours and minutes⁶. Find path with shortest travel time from Bern to Berlin by applying Dijkstra's algorithm.



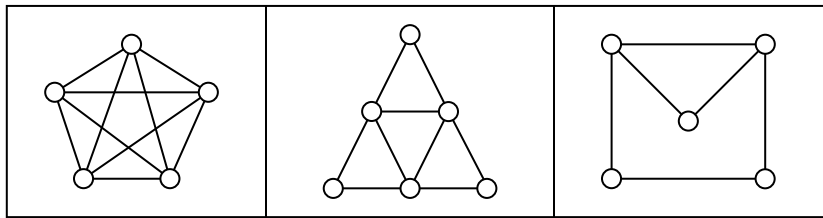
10. Using the graph from the previous problem, find the path with shortest travel time from Paris to München.

11. Does each of these graphs have an Euler circuit? If so, find it.

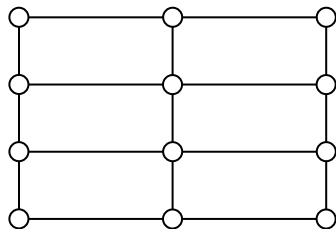


⁶ From <http://www.eurail.com/eurail-railway-map>

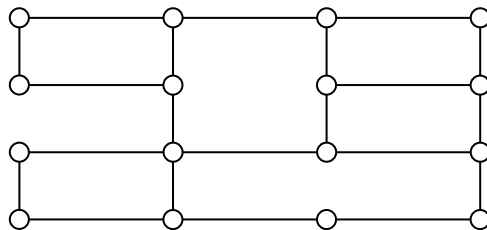
12. Does each of these graphs have an Euler circuit? If so, find it.



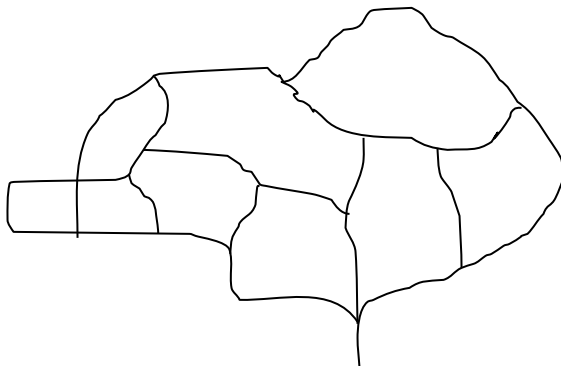
13. Eulerize this graph using as few edge duplications as possible. Then, find an Euler circuit.



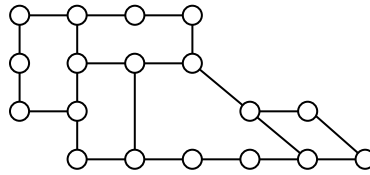
14. Eulerize this graph using as few edge duplications as possible. Then, find an Euler circuit.



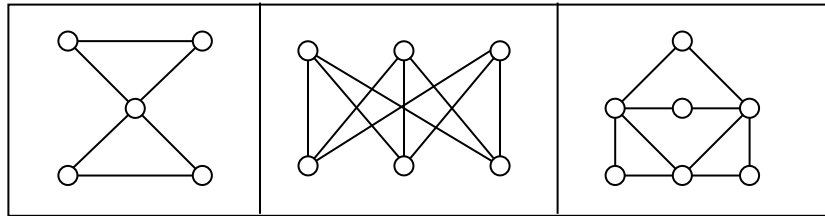
15. The maintenance staff at an amusement park need to patrol the major walkways, shown in the graph below, collecting litter. Find an efficient patrol route by finding an Euler circuit. If necessary, eulerize the graph in an efficient way.



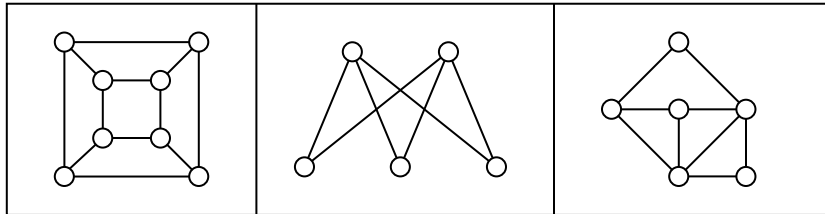
16. After a storm, the city crew inspects for trees or brush blocking the road. Find an efficient route for the neighborhood below by finding an Euler circuit. If necessary, eulerize the graph in an efficient way.



17. Does each of these graphs have at least one Hamiltonian circuit? If so, find one.



18. Does each of these graphs have at least one Hamiltonian circuit? If so, find one.

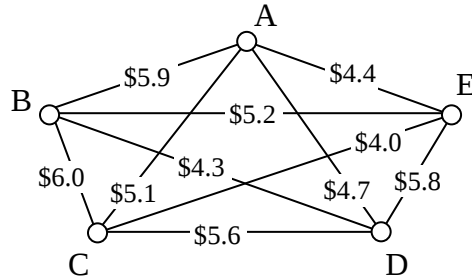


19. A company needs to deliver product to each of their 5 stores around the Dallas, TX area. Driving distances between the stores are shown below. Find a route for the driver to follow, returning to the distribution center in Fort Worth:
- Using Nearest Neighbor starting in Fort Worth
 - Using Repeated Nearest Neighbor
 - Using Sorted Edges

	Plano	Mesquite	Arlington	Denton
Fort Worth	54	52	19	42
Plano		38	53	41
Mesquite			43	56
Arlington				50

20. A salesperson needs to travel from Seattle to Honolulu, London, Moscow, and Cairo. Use the table of flight costs from problem #4 to find a route for this person to follow:
- Using Nearest Neighbor starting in Seattle
 - Using Repeated Nearest Neighbor
 - Using Sorted Edges

21. When installing fiber optics, some companies will install a sonet ring; a full loop of cable connecting multiple locations. This is used so that if any part of the cable is damaged it does not interrupt service, since there is a second connection to the hub. A company has 5 buildings. Costs (in thousands of dollars) to lay cables between pairs of buildings are shown below. Find the circuit that will minimize cost:
- Using Nearest Neighbor starting at building A
 - Using Repeated Nearest Neighbor
 - Using Sorted Edges



22. A tourist wants to visit 7 cities in Israel. Driving distances, in kilometers, between the cities are shown below⁷. Find a route for the person to follow, returning to the starting city:
- Using Nearest Neighbor starting in Jerusalem
 - Using Repeated Nearest Neighbor
 - Using Sorted Edges

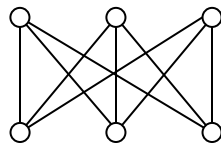
	Jerusalem	Tel Aviv	Haifa	Tiberias	Beer Sheba	Eilat
Jerusalem	--					
Tel Aviv	58	--				
Haifa	151	95	--			
Tiberias	152	134	69	--		
Beer Sheba	81	105	197	233	--	
Eilat	309	346	438	405	241	--
Nazareth	131	102	35	29	207	488

23. Find a minimum cost spanning tree for the graph you created in problem #3
24. Find a minimum cost spanning tree for the graph you created in problem #22
25. Find a minimum cost spanning tree for the graph from problem #21

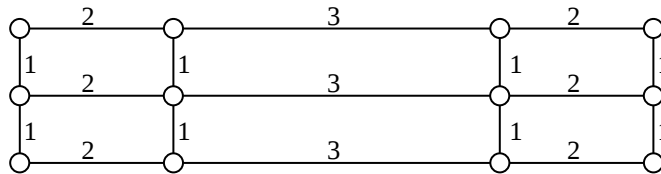
⁷ From <http://www.ddtravel-acc.com/Israel-cities-distance.htm>

Concepts

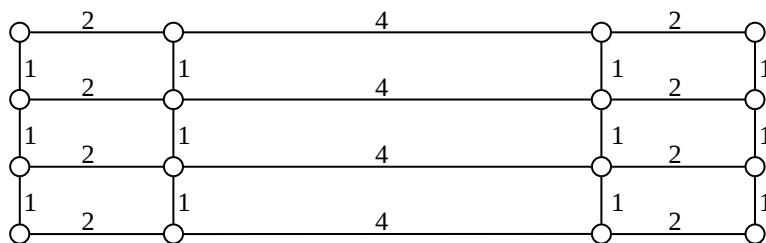
26. Can a graph have one vertex with odd degree? If not, are there other values that are not possible? Why?
27. A complete graph is one in which there is an edge connecting every vertex to every other vertex. For what values of n does complete graph with n vertices have an Euler circuit? A Hamiltonian circuit?
28. Create a graph by drawing n vertices in a row, then another n vertices below those. Draw an edge from each vertex in the top row to every vertex in the bottom row. An example when $n=3$ is shown below. For what values of n will a graph created this way have an Euler circuit? A Hamiltonian circuit?



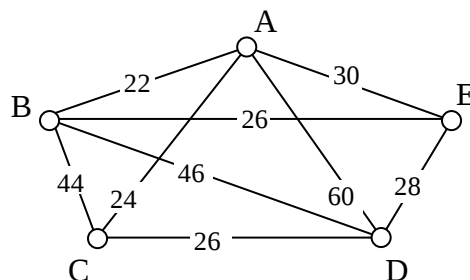
29. Eulerize this graph in the most efficient way possible, considering the weights of the edges.



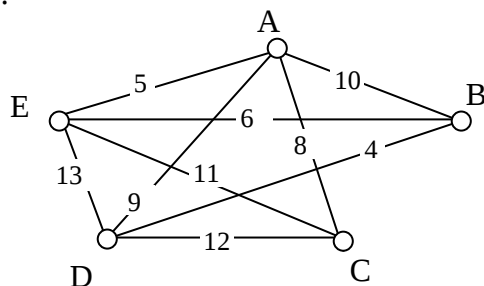
30. Eulerize this graph in the most efficient way possible, considering the weights of the edges.



31. Eulerize this graph in the most efficient way possible, considering the weights of the edges.



32. Eulerize this graph in the most efficient way possible, considering the weights of the edges.



Explorations

33. Social networks such as Facebook and LinkedIn can be represented using graphs in which vertices represent people and edges are drawn between two vertices when those people are “friends.” The table below shows a friendship table, where an X shows that two people are friends.

	A	B	C	D	E	F	G	H	I
A		X	X			X	X		
B			X		X				
C					X				
D					X				X
E							X		X
F								X	X
G								X	
H									X

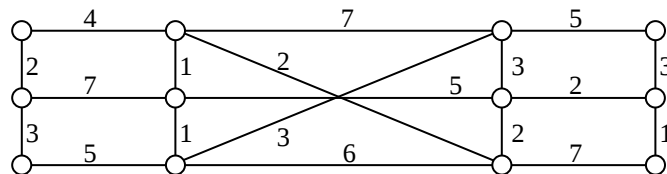
- Create a graph of this friendship table
- Find the shortest path from A to D. The length of this path is often called the “degrees of separation” of the two people.
- Extension: Split into groups. Each group will pick 10 or more movies, and look up their major actors (www.imdb.com is a good source). Create a graph with each actor as a vertex, and edges connecting two actors in the same movie (note the movie name on the edge). Find interesting paths between actors, and quiz the other groups to see if they can guess the connections.

34. A spell checker in a word processing program makes suggestions when it finds a word not in the dictionary. To determine what words to suggest, it tries to find similar words. One measure of word similarity is the Levenshtein distance, which measures the number of substitutions, additions, or deletions that are required to change one word into another. For example, the words spit and spot are a distance of 1 apart; changing spit to spot requires one substitution (i for o). Likewise, spit is distance 1 from pit since the change requires one deletion (the s). The word spite is also distance 1 from spit since it requires one addition (the e). The word soot is distance 2 from spit since two substitutions would be required.

- Create a graph using words as vertices, and edges connecting words with a Levenshtein distance of 1. Use the misspelled word “moke” as the center, and try to find at least 10 connected dictionary words. How might a spell checker use this graph?
- Improve the method from above by assigning a weight to each edge based on the likelihood of making the substitution, addition, or deletion. You can base the weights on any reasonable approach: proximity of keys on a keyboard, common language errors, etc. Use Dijkstra’s algorithm to find the length of the shortest path from each word to “moke”. How might a spell checker use these values?

35. The graph below contains two vertices of odd degree. To eulerize this graph, it is necessary to duplicate edges connecting those two vertices.

- Use Dijkstra’s algorithm to find the shortest path between the two vertices with odd degree. Does this produce the most efficient eulerization and solve the Chinese Postman Problem for this graph?



- Suppose a graph has n odd vertices. Using the approach from part a, how many shortest paths would need to be considered? Is this approach going to be efficient?

Scheduling

Excerpted from *Math in Society*, by David Lippman

Scheduling

An event planner has to juggle many workers completing different tasks, some of which must be completed before others can begin. For example, the banquet tables would need to be arranged in the room before the catering staff could begin setting out silverware. The event planner has to carefully schedule things so that everything gets done in a reasonable amount of time.

The problem of scheduling is fairly universal. Contractors need to schedule workers and subcontractors to build a house as quickly as possible. A magazine needs to schedule writers, editors, photographers, typesetters, and others so that an issue can be completed on time.

Section 2.1 Getting Started

To begin thinking about scheduling, let us consider an auto shop that is converting a car from gas to electric. A number of steps are involved. A time estimate for each task is given.

- Task 1: Remove engine and gas parts (2 days)
- Task 2: Steam clean the inside of the car (0.5 day)
- Task 3: Buy an electric motor and speed controller (2 days for travel)
- Task 4: Construct the part that connects the motor to the car's transmission (1 day)
- Task 5: Construct battery racks (2 days)
- Task 6: Install the motor (0.5 day)
- Task 7: Install the speed controller (0.5 day)
- Task 8: Install the battery racks (0.5 day)
- Task 9: Wire the electricity (1 day)

Some tasks have to be completed before others – we certainly can't install the new motor before removing the old engine! There are some tasks, however, that can be worked on simultaneously by two different people, like constructing the battery racks and installing the motor.

To help us visualize the ordering of tasks, we will create a **digraph**.

Digraph

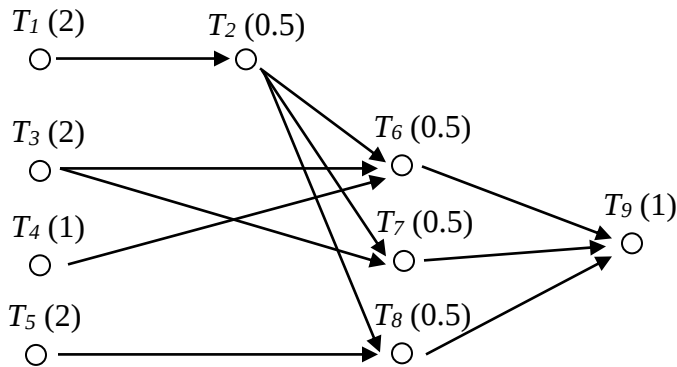
A digraph is a graphical representation of a set of tasks in which tasks are represented with dots, called vertices, and arrows between vertices are used to show ordering.

For example, this digraph shows that Task 1, notated T_1 for compactness, needs to be completed before Task 2. The number in parentheses after the task name is the time required for the task.



Example 1

A complete digraph for our car conversion would look like this:



The time it takes to complete this job will partially depend upon how many people are working on the project.

Processors

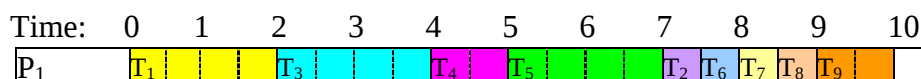
In scheduling jargon, the workers completing the tasks are called **processors**. While in this example the processors are humans, in some situations the processors are computers, robots, or other machines.

For simplicity, we are going to make the very big assumptions that every processor can do every task, that they all would take the same time to complete it, and that only one processor can work on a task at a time.

Finishing Time

The **finishing** time is how long it will take to complete all the tasks. The finishing time will depend upon the number of processors and the specific schedule.

If we had only one processor working on this task, it is easy to determine the finishing time; just add up the individual times. We assume one person can't work on two tasks at the same time, ignore things like drying times during which someone could work on another task. Scheduling with one processor, a possible schedule would look like this, with a finishing time of 10 days.



In this schedule, all the ordering requirements are met. This is certainly not the only possible schedule for one processor, but no other schedule could complete the job in less time. Because of this, this is an **optimal schedule** with **optimal finishing time** – there is nothing better.

Optimal Schedule

An optimal schedule is the schedule with the shortest possible finishing time.

For two processors, things become more interesting. For small digraphs like this, we probably could fiddle around and guess-and-check a pretty good schedule. Here would be a possibility:

Time:	0	1	2	3	4	5	6	7	8	9	10
P ₁	T ₁		T ₃		T ₆	T ₉					
P ₂	T ₅		T ₄	T ₂	T ₈	T ₇					

With two processors, the finishing time was reduced to 5.5 days. What was processor 2 doing during the last day? Nothing, because there were no tasks for the processor to do. This is called **idle time**.

Idle Time

Idle time is time in the schedule when there are no tasks available for the processor to work on, so they sit idle.

Is this schedule optimal? Could it have been completed in 5 days? Because every other task had to be completed before task 9 could start, there would be no way that both processors could be busy during task 9, so it is not possible to create a shorter schedule.

So how long will it take if we use three processors? About $10/3 = 3.33$ days? Again we will guess-and-check a schedule:

Time:	0	1	2	3	4	5	6	7	8	9	10
P ₁	T ₁		T ₂	T ₆	T ₈	T ₉					
P ₂	T ₃				T ₇						
P ₃	T ₄	T ₅									

With three processors, the job still took 4.5 days. It is a little harder to tell whether this schedule is optimal. However, it might be helpful to notice that since Task 1, 2, 6, and 9 have to be completed sequentially, there is no way that this job could be completed in less than $2+0.5+0.5+1 = 4$ days, regardless of the number of processors. Four days is, for this digraph, the absolute minimum time to complete the job, called the **critical time**.

Critical Time

The critical time is the absolute minimum time to complete the job, regardless of the number of processors working on the tasks.

Critical time can be determined by looking at the longest sequence of tasks in the digraph, called the critical path

Sec. 2.2 Adding an algorithm

Up until now, we have been creating schedules by guess-and-check, which works well enough for small schedules, but would not work well with dozens or hundreds of tasks. To create a more procedural approach, we might begin by somehow creating a **priority list**. Once we have a priority list, we can begin scheduling using that list and the **list processing algorithm**.

Priority List

A priority list is a list of tasks given in the order in which we desire them to be completed.

The List Processing Algorithm turns a priority list into a schedule

List Processing Algorithm

1. On the digraph or priority list, circle all tasks that are ready, meaning that all pre-requisite tasks have been completed.
2. Assign to each available processor, in order, the first ready task. Mark the task as in progress, perhaps by putting a single line through the task.
3. Move forward in time until a task is completed. Mark the task as completed, perhaps by crossing out the task. If any new tasks become ready, mark them as such.
4. Repeat until all tasks have been scheduled.

Example 2

Using our digraph from above, schedule it using the priority list below:

T₁, T₃, T₄, T₅, T₆, T₇, T₈, T₂, T₉

Time 0: Mark ready tasks

Priority list: T₁, T₃, T₄, T₅, T₆, T₇, T₈, T₂, T₉

We assign the first task, T₁ to the first processor, P₁, and the second ready task, T₃, to the second processor. Making those assignments, we mark those tasks as in progress:

Priority list: ~~T₁~~, ~~T₃~~, T₄, T₅, T₆, T₇, T₈, T₂, T₉

Schedule up to here:

Time:	0	1	2	3	4	5	6	7	8	9	10
P ₁	T ₁										
P ₂	T ₃										

We jump to the time when the next task completes, which is at time 2.

Time 2: Both processors complete their tasks. We mark those tasks as complete. With Task 1 complete, Task 2 becomes ready:

Priority list: ~~T₁~~, ~~T₃~~, ~~T₄~~, ~~T₅~~, T₆, T₇, T₈, T₂, T₉
 We assign the next ready task on the list, T₄ to P₁, and T₅ to P₂.

Priority list: ~~T₁~~, ~~T₃~~, ~~T₄~~, ~~T₅~~, T₆, T₇, T₈, T₂, T₉

Time:	0	1	2	3	4	5	6	7	8	9	10
P ₁	T ₁			T ₄							
P ₂	T ₃		T ₅								

Time 3: Processor 1 has completed T₄. Completing T₄ does not make any other tasks ready (note that all the rest require that T₂ be completed first).

Priority list: ~~T₁~~, ~~T₃~~, ~~T₄~~, ~~T₅~~, T₆, T₇, T₈, T₂, T₉

Since the next three tasks are not yet ready, we assign the next ready task, T₂ to P₁

Priority list: ~~T₁~~, ~~T₃~~, ~~T₄~~, ~~T₅~~, T₆, T₇, T₈, ~~T₂~~, T₉

Time:	0	1	2	3	4	5	6	7	8	9	10
P ₁	T ₁		T ₄	T ₂							
P ₂	T ₃		T ₅								

Time 3.5: Processor 1 has completed T₂. Completing T₂ causes T₆ and T₇ to become ready. We assign T₆ to P₁

Priority list: ~~T₁~~, ~~T₃~~, ~~T₄~~, ~~T₅~~, ~~T₆~~, ~~T₇~~, T₈, ~~T₂~~, T₉

Time:	0	1	2	3	4	5	6	7	8	9	10
P ₁	T ₁		T ₄	T ₂	T ₆						
P ₂	T ₃		T ₅								

Time 4: Both processors complete their tasks. The completion of T₅ allows T₈ to become ready. We assign T₇ to P₁ and T₈ to P₂.

Priority list: ~~T₁~~, ~~T₃~~, ~~T₄~~, ~~T₅~~, ~~T₆~~, ~~T₇~~, ~~T₈~~, ~~T₂~~, T₉

Time:	0	1	2	3	4	5	6	7	8	9	10
P ₁	T ₁		T ₄	T ₂	T ₆	T ₇					
P ₂	T ₃		T ₅		T ₈						

Time 4.5: Both processors complete their tasks. T_9 becomes ready, and is assigned to P_1 . There is no ready task for P_2 to work on, so P_2 idles.

Priority list: ~~T_1~~ , ~~T_3~~ , ~~T_4~~ , ~~T_5~~ , ~~T_6~~ , ~~T_7~~ , ~~T_8~~ , ~~T_2~~ , ~~T_9~~

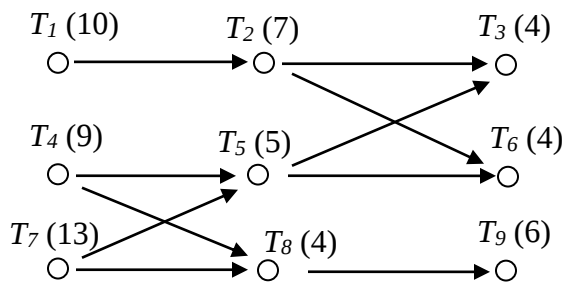
Time:	0	1	2	3	4	5	6	7	8	9	10
P_1	T_1		T_4	T_2	T_6	T_7	T_9				
P_2	T_3		T_5		T_8						

With the last task completed, we have a completed schedule, with finishing time 5.5 days.

Try it Now 1

Using the digraph below, create a schedule using the priority list:

$T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8, T_9$



It is important to note that the list processing algorithm itself does not influence the resulting schedule – the schedule is completely determined by the priority list followed. The list processing, while do-able by hand, could just as easily be executed by a computer. The interesting part of scheduling, then, is how to choose a priority list that will create the best possible schedule.

Choosing a priority list

Next, we will explore two algorithms for selecting a priority list.

Sec. 2.3 Decreasing time algorithm

The decreasing time algorithm takes the approach of trying to get the very long tasks out of the way as soon as possible by putting them first on the priority list.

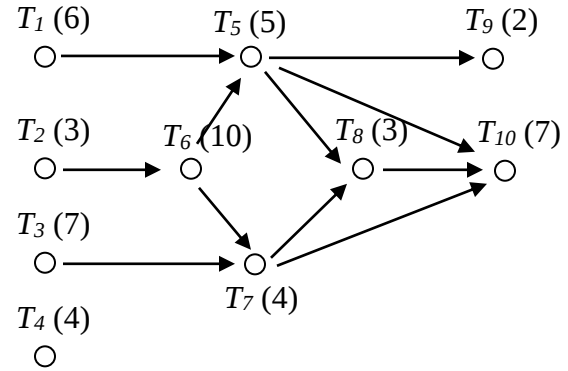
Decreasing Time Algorithm

Create the priority list by listing the tasks in order from longest completion time to shortest completion time.

Example 3

Consider the scheduling problem represented by the digraph below. Create a priority list using the decreasing time list algorithm, then use it to schedule for two processors using the list processing algorithm.

To use the decreasing time list algorithm, we create our priority list by listing the tasks in order from longest task time to shortest task time. If there is a tie, we will list the task with smaller task number first (not for any good reason, but just for consistency).



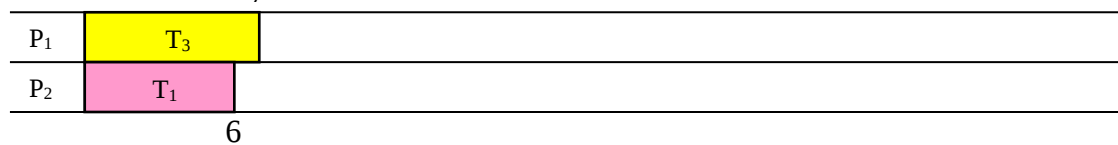
For this digraph, the decreasing time algorithm would create a priority list of:

$$T_6(10), T_3(7), T_{10}(7), T_1(6), T_5(5), T_4(4), T_7(4), T_2(3), T_8(3), T_9(2)$$

Once we have the priority list, we can create the schedule using the list processing algorithm. With two processors, we'd get:

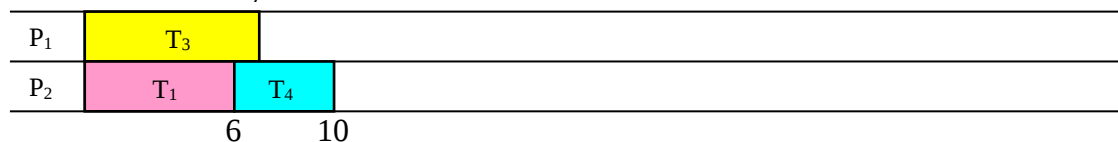
Time 0: We identify ready tasks, and assign T_3 to P_1 and T_1 to P_2

Priority list: T_6 , ~~T_3~~ , T_{10} , ~~T_1~~ , T_5 , T_4 , T_7 , T_2 , T_8 , T_9



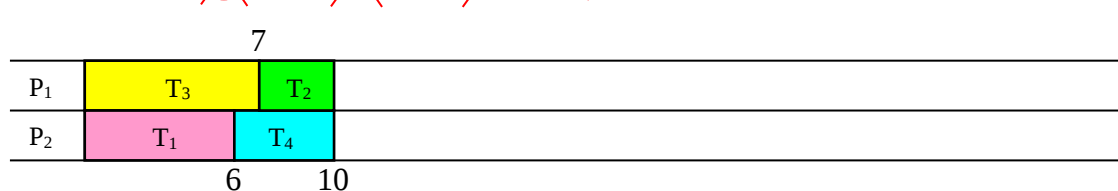
Time 6: P_2 completes T_1 . No new tasks become ready, so T_4 is assigned to P_2 .

Priority list: T_6 , ~~T_3~~ , T_{10} , ~~T_1~~ , T_5 , ~~T_4~~ , T_7 , T_2 , T_8 , T_9



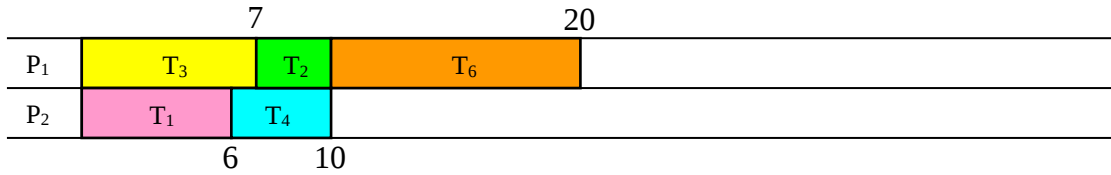
Time 7: P_1 completes T_3 . No new tasks become ready, so T_2 is assigned to P_1 .

Priority list: T₆, ~~T₃~~, T₁₀, ~~T₁~~, T₅, ~~T₄~~, T₇, ~~T₂~~, T₈, T₉



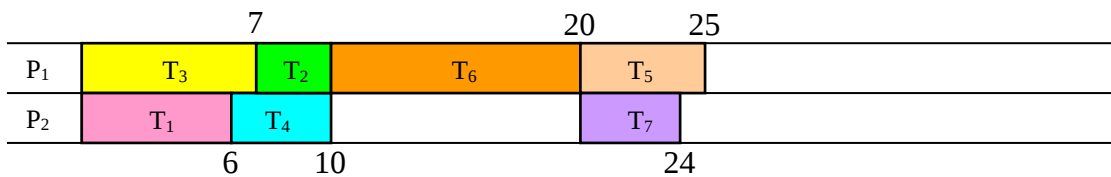
Time 10: Both processors complete their tasks. T_6 becomes ready, and is assigned to P_1 . No other tasks are ready, so P_2 idles.

Priority list: ~~T_6~~ , ~~T_3~~ , T_{10} , ~~T_1~~ , T_5 , ~~T_4~~ , T_7 , ~~T_2~~ , T_8 , T_9



Time 20: With T_6 complete, T_5 and T_7 become ready, and are assigned to P_1 and P_2 respectively.

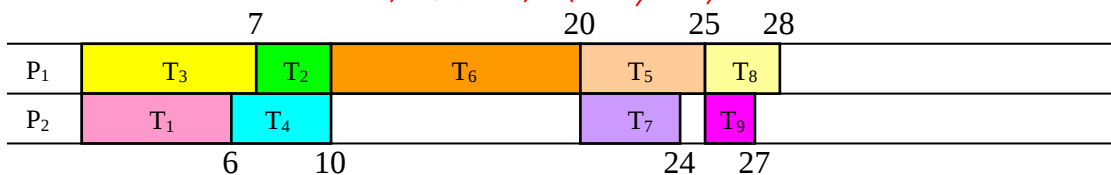
Priority list: ~~T_6~~ , ~~T_3~~ , T_{10} , ~~T_1~~ , ~~T_5~~ , ~~T_4~~ , ~~T_7~~ , ~~T_2~~ , T_8 , T_9



Time 24: P_2 completes T_7 . No new items become ready, so P_2 idles.

Time 25: P_1 completes T_5 . T_8 and T_9 become ready, and are assigned.

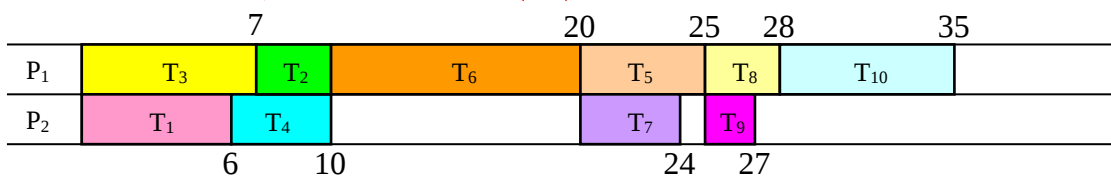
Priority list: ~~T_6~~ , ~~T_3~~ , T_{10} , ~~T_1~~ , ~~T_5~~ , ~~T_4~~ , ~~T_7~~ , ~~T_2~~ , ~~T_8~~ , ~~T_9~~



Time 27: T_9 is completed. No items ready, so P_2 idles.

Time 28: T_8 is completed. T_{10} becomes ready, and is assigned to P_1 .

Priority list: ~~T_6~~ , ~~T_3~~ , ~~T_{10}~~ , ~~T_1~~ , ~~T_5~~ , ~~T_4~~ , ~~T_7~~ , ~~T_2~~ , ~~T_8~~ , ~~T_9~~



This is our completed schedule, with a finishing time of 35.

Using the decreasing time algorithm, the priority list led to a schedule with a finishing time of 35. Is this good? It certainly looks like there was a lot of idle time in this schedule. To get some idea how good or bad this schedule is, we could compute the critical time, the minimum time to complete the job. To find this, we look for the sequence of tasks with the highest total completion time. For this digraph that sequence would appear to be: $T_2, T_6, T_5, T_8, T_{10}$, with total sequence time of 28. From this we can conclude that our schedule isn't horrible, but there is a possibility that a better schedule exists.

Try it Now 2

Determine the priority list for the digraph from Try it Now 1 using the decreasing time algorithm.

Sec. 2.5 Critical path algorithm

A sequence of tasks in the digraph is called a **path**. In the previous example, we saw that the critical path dictates the minimum completion time for a schedule. Perhaps, then, it would make sense to consider the critical path when creating our schedule. For example, in the last schedule, the processors began working on tasks 1 and 3 because they were longer tasks, but starting on task 2 earlier would have allowed work to begin on the long task 6 earlier.

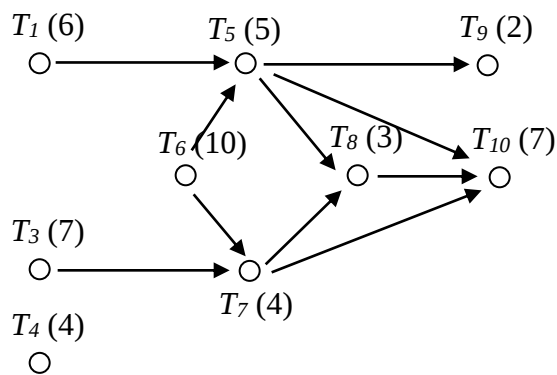
The critical path algorithm allows you to create a priority list based on idea of critical paths.

Critical Path Algorithm (version 1)

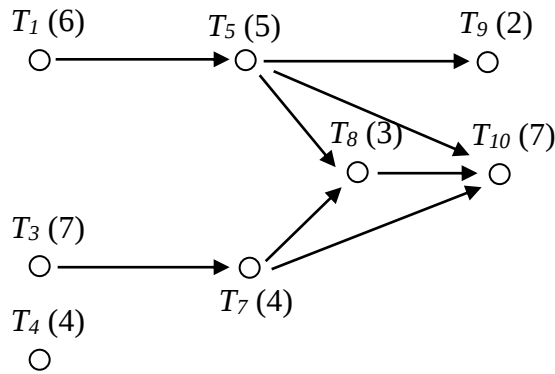
1. Find the critical path.
2. The first task in the critical path gets added to the priority list.
3. Remove that task from the digraph
4. Repeat, finding the new critical path with the revised digraph.

Example 4

The original digraph from Example 3 has critical path $T_2, T_6, T_5, T_8, T_{10}$, so T_2 gets added first to the priority list. Removing T_2 from the digraph, it now looks like:



The critical path (longest path) in the remaining digraph is now T_6, T_5, T_8, T_{10} , so T_6 is added to the priority list and removed.



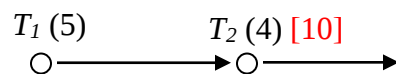
Now there are two paths with the same longest length: T_1, T_5, T_8, T_{10} and T_3, T_7, T_8, T_{10} . We can add T_1 to the priority list (or T_3 – we usually add the one with smaller item number) and remove it, and continue the process.

I'm sure you can imagine that searching for the critical path every time you remove a task from the digraph would get really tiring, especially for a large digraph. In practice, the critical path algorithm is implemented by first working from the end backwards. This is called the **backflow algorithm**.

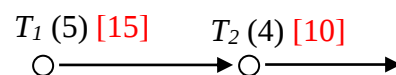
Backflow Algorithm

1. Introduce an "end" vertex, and assign it a time of 0, shown in [brackets]
2. Move backwards to every vertex that has an arrow to the end and assign it a critical time
3. From each of those vertices, move backwards and assign those vertices critical times. Notice that the critical time for the earlier vertex will be that task's time plus the critical time for the later vertex.

Example: Consider this segment of digraph.

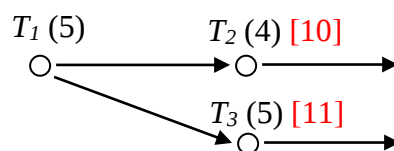


In this case, if T_2 has already been determined to have a critical time of 10, then T_1 will have a critical time of $5+10 = 15$



If you have already assigned a critical time to a vertex, replace it only if the new time is larger.

Example: In the digraph below, T_1 should be labeled with a critical time of 16, since it is the longer of $5+10$ and $5+11$.



4. Repeat until all vertices are labeled with their critical times

Once you have completed the backflow algorithm, you can easily create the critical path priority list by using the critical times you just found.

Critical Path Algorithm (version 2)

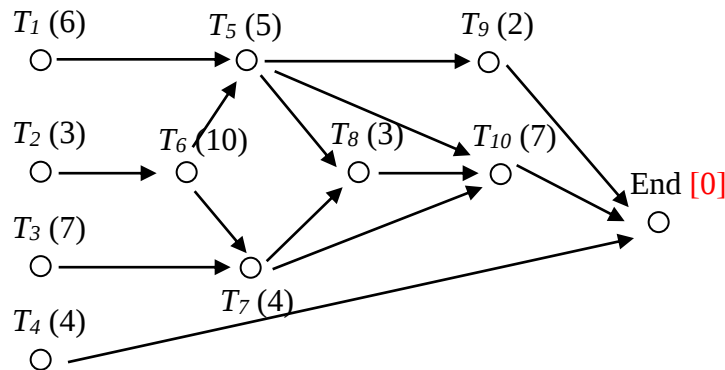
1. Apply the backflow algorithm to the digraph
2. Create the priority list by listing the tasks in order from longest critical time to shortest critical time

This version of the Critical Path Algorithm will usually be the easier to implement.

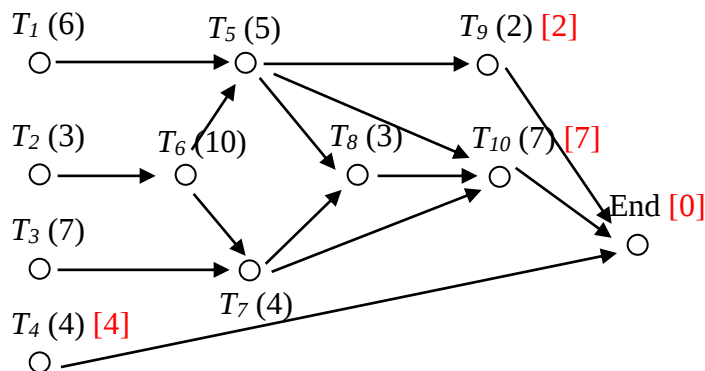
Example 5

Applying this to our digraph from the earlier example, we start applying the backflow algorithm.

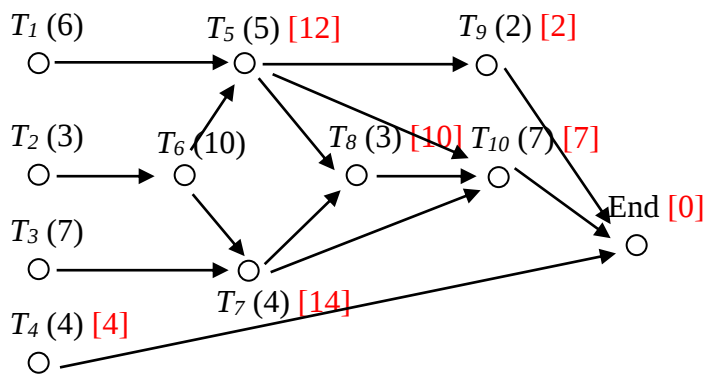
We add an end vertex and give it a critical time of 0.



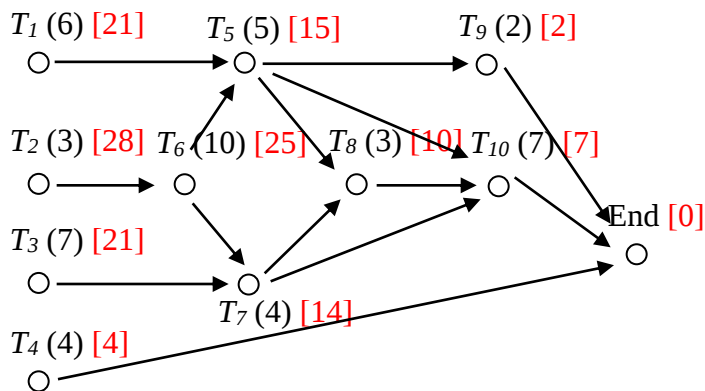
We then move back to T_4 , T_9 , and T_{10} , labeling them with their critical times



From each vertex marked with a critical time, we go back. T_7 , for example, will get labeled with a critical time 11 – the task time of 4 plus the critical time of 7 for T_{10} . For T_5 , there are two paths to the end. We use the longer, labeling T_5 with critical time $5+7 = 12$.



Continue the process until all vertices are labeled. Notice that the critical time for T_5 ended got replaced later with the even longer path through T_8 .



We can now quickly create the critical path priority list by listing the tasks in decreasing order of critical time:

Priority list: $T_2, T_6, T_1, T_3, T_5, T_7, T_8, T_{10}, T_4, T_9$

Applying this priority list using the list processing algorithm, we get the schedule:

		3		13		18	21		28	
P_1	T_2	T_6			T_5	T_8	T_{10}			
P_2	T_1	T_3			T_7	T_4	T_9			
		6			17		23			

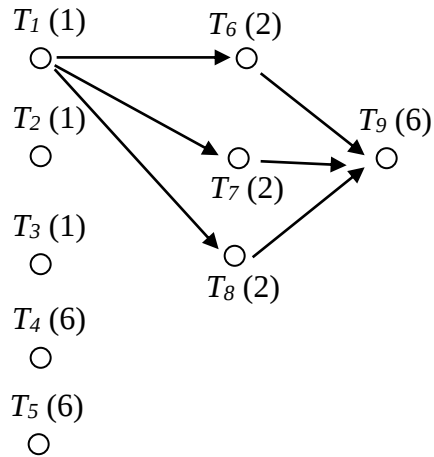
In this particular case, we were able to achieve the minimum possible completion time with this schedule, suggesting that this schedule is optimal. This is certainly not always the case.

Try it Now 3

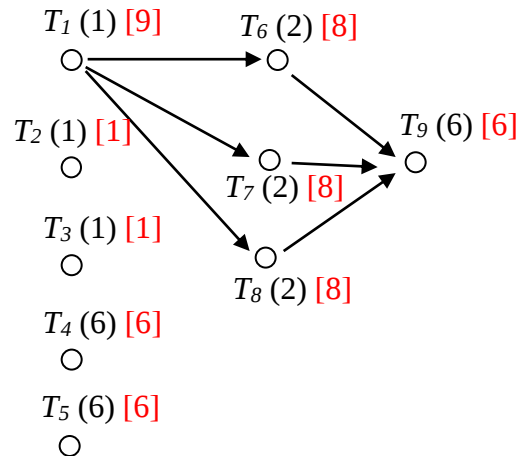
Determine the priority list for the digraph from Try it Now 1 using the critical path algorithm.

Example 6

This example is designed to show that the critical path algorithm doesn't always work wonderfully. Schedule the tasks in the digraph below on three processors using the critical path algorithm.



To create a critical path priority list, we could first apply the backflow algorithm:



This yields the critical-path priority list: T₁, T₆, T₇, T₈, T₄, T₅, T₉, T₂, T₃.

Applying the list processing algorithm to this priority list leads to the schedule:

	1	3	5	7	13	
P ₁	T ₁	T ₆	T ₇	T ₈	T ₉	
P ₂	T ₄			T ₂		
P ₃	T ₅			T ₃		
	6					

This schedule has finishing time of 13.

By observation, we can see that a much better schedule exists for the example above:

	1	3	9	
P ₁	T ₁	T ₆	T ₉	
P ₂	T ₂	T ₇	T ₅	
P ₃	T ₃	T ₈	T ₄	

In most cases the critical path algorithm will lead to a very good schedule. There are cases, like this, where it will not. Unfortunately, there is no known algorithm to always produce the optimal schedule.

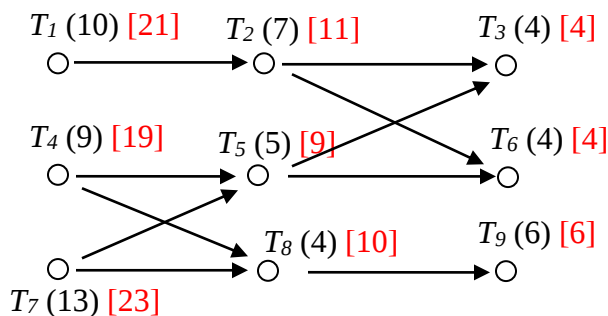
Try it Now Answers

- Time 0: Tasks 1, 4, and 7 are ready. Assign Tasks 1 and 4
Time 9: Task 4 completes. Only task 7 is ready; assign task 7
Time 10: Task 1 completes. Task 2 now ready. Assign task 2
Time 17: Task 2 completes. Nothing is ready. Processor 1 idles
Time 22: Task 7 completes. Tasks 5 and 8 are ready. Assign tasks 5 and 8
Time 26: Task 8 completes. Task 9 is ready. Assign task 9
Time 27: Task 5 completes. Tasks 3 and 6 are ready. Assign task 3
Time 31: Task 3 completes. Assign task 6
Time 35: Everything is done. Finishing time is 35 for this schedule.

	10	17	27	31	35
P ₁	T ₁	T ₂	T ₅	T ₃	
P ₂	T ₄	T ₇	T ₈	T ₉	
	6	9	22	26	32

- T₇, T₁, T₄, T₂, T₉, T₅, T₃, T₆, T₈

- Applying the backflow algorithm, we get this:



The critical path priority list is: T₇, T₁, T₄, T₂, T₈, T₅, T₉, T₃, T₆

Exercises

Skills

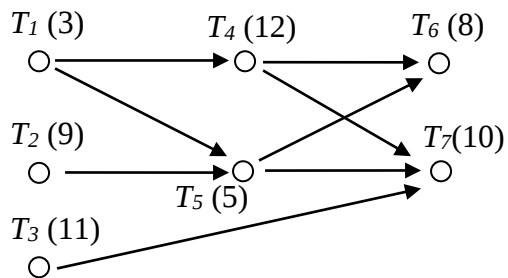
1. Create a digraph for the following set of tasks:

Task	Time required	Tasks that must be completed first
A	3	
B	4	
C	7	
D	6	A, B
E	5	B
F	5	D, E
G	4	E

2. Create a digraph for the following set of tasks:

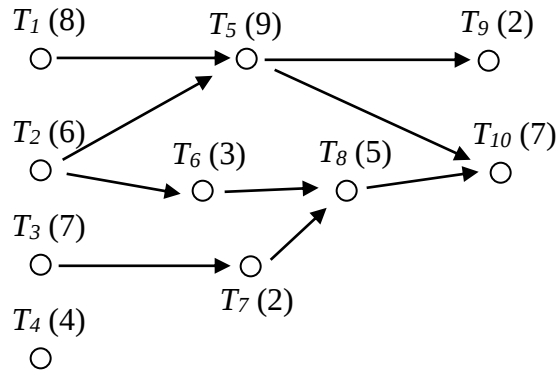
Task	Time required	Tasks that must be completed first
A	3	
B	4	
C	7	
D	6	A
E	5	A
F	5	B
G	4	D, E

Use this digraph for the next 2 problems.



3. Using the priority list $T_4, T_1, T_7, T_3, T_6, T_2, T_5$, schedule the project with two processors.
4. Using the priority list $T_5, T_2, T_3, T_7, T_1, T_4, T_6$, schedule the project with two processors.

Use this digraph for the next 4 problems.

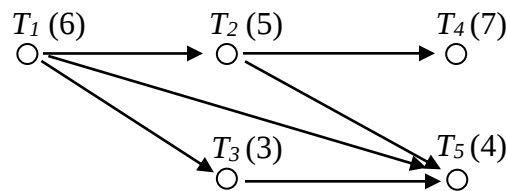


5. Using the priority list $T_4, T_3, T_9, T_{10}, T_8, T_5, T_6, T_1, T_7, T_2$ schedule the project with two processors.
6. Using the priority list $T_2, T_4, T_6, T_8, T_{10}, T_1, T_3, T_5, T_7, T_9$ schedule the project with two processors.
7. Using the priority list $T_4, T_3, T_9, T_{10}, T_8, T_5, T_6, T_1, T_7, T_2$ schedule the project with three processors.
8. Using the priority list $T_2, T_4, T_6, T_8, T_{10}, T_1, T_3, T_5, T_7, T_9$ schedule the project with three processors.
9. Use the decreasing time algorithm to create a priority list for the digraph from #3, and schedule with two processors.
10. Use the decreasing time algorithm to create a priority list for the digraph from #3, and schedule with three processors.
11. Use the decreasing time algorithm to create a priority list for the digraph from #5, and schedule with two processors.
12. Use the decreasing time algorithm to create a priority list for the digraph from #5, and schedule with three processors.
13. Use the decreasing time algorithm to create a priority list for the problem from #1, and schedule with two processors.
14. Use the decreasing time algorithm to create a priority list for the problem from #2, and schedule with two processors.
15. With the digraph from #3:
 - a. Apply the backflow algorithm to find the critical time for each task
 - b. Find the critical path for the project and the minimum completion time
 - c. Use the critical path algorithm to create a priority list and schedule on two processors.

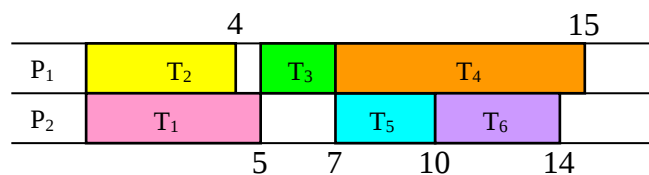
16. With the digraph from #3, use the critical path algorithm to schedule on three processors.
17. With the digraph from #5:
 - a. Apply the backflow algorithm to find the critical time for each task
 - b. Find the critical path for the project and the minimum completion time
 - c. Use the critical path algorithm to create a priority list and schedule on two processors.
18. With the digraph from #5, use the critical path algorithm to schedule on three processors.
19. Use the critical path algorithm to schedule the problem from #1 on two processors.
20. Use the critical path algorithm to schedule the problem from #2 on two processors.

Concepts

21. If an additional order requirement is added to a digraph, can the optimal finishing time ever become longer? Can the optimal finishing time ever become shorter?
22. Will an optimal schedule always have no idle time?
23. Consider the digraph below.
 - a. How many priority lists could be created for these tasks?
 - b. How many unique schedules are created by those priority lists?



24. Create a digraph and priority list that would lead to the schedule below.



25. Is it possible to create a digraph with three tasks for which every possible priority list creates a different schedule? If so, create it.
26. Is it possible to create a digraph with four tasks for which every possible priority list creates a different schedule? If so, create it.

Exploration

27. Independent tasks are ones that have no order requirements; they can be completed in any order.
- Consider three tasks, with completion times 2, 2, and 4 hours respectively. Construct two different schedules on two processors with different completion times to show that the priority list still matters with independent tasks.
 - Choose a set of independent tasks with different completion times, and implement the decreasing time list algorithm and the critical path algorithm. What do you observe?
 - Will using the decreasing time list or critical path algorithms with independent tasks always produce an optimal schedule? Why or why not?
 - Will using the decreasing time list or critical path algorithms with independent tasks always produce the same schedule? Why or why not?
28. In a group, choose ten tasks necessary to throw a birthday party for a friend or child (for example, cleaning the house or buying a cake). Determine order requirements for the tasks, create a digraph, and schedule the tasks for two people.
- 29-37: At the end of the chapter it was noted that no algorithm exists to determine if an arbitrary schedule is optimal, but there are special cases where we can determine that a schedule is indeed optimal. In each of the following scenarios, determine
- If the scenario is even possible
 - Whether or not the schedule *could* be optimal
 - Whether or not we can be *sure* that the schedule is optimal
29. A job has a critical time of 30 hours, and the finishing time for the schedule on 2 processors is 30 hours.
30. The sum of all task times for a job is 40 hours, and the finishing time for the schedule on 2 processors was 15 hours
31. The sum of all task times for a job is 100 hours, the critical time of the job was 40 hours, and the finishing time for the schedule on 2 processors was 50 hours.
32. The sum of all task times for a job is 50 hours, and the finishing time for the schedule on 2 processors was 40 hours.
33. A job has a critical time of 30 hours, and the finishing time for the schedule on 3 processors is 20 hours.
34. The sum of all task times for a job is 60 hours, and the finishing time for the schedule on 3 processors was 20 hours.
35. The critical time for a job is 25 hours, and the finishing time for the schedule on 2 processors was 30 hours.
36. The sum of all task times for a job is 20 hours, and the finishing time for the schedule on 2 processors was 25 hours.
37. Based on your observations in the previous scenarios, write guidelines for when you can determine that a schedule is optimal.