Math 203
Topics for the Chapter 6 quiz:
Routes and Networks

To do:

Model a problem (sweeping streets, delivering mail) as finding an efficient path in a graph.

Determine if a graph has an Euler circuit/path; Eulerize a graph to find an efficient circuit/path.

Model a problem (wiring up a neighborhood, creating emergency routes, laying pipes) as finding an efficient subgraph in a graph.

Find a minimal weight maximal tree in a weighted graph.

Basic goal: determine how to carry out a sequence of tasks most efficiently.

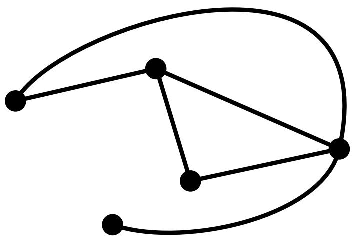
Efficient: least cost, least time, least distance, ...

Example: reading parking meters - need to walk down each sidewalk that has meters on it. Efficient: try not to walk down the same sidewalk twice!

Example: Snowplowing. Each street needs to be plowed; try not to drive down an already plowed street!

Model of the problem: a graph = a bunch of points (vertices) connected together by line segments/curves (edges).

E.g., vertices = intersections/locations; edges = sidewalks/streets/roads



A **path** in a graph is a way to walk from vertex to vertex along the edges.

A **circuit** is a path that begins and ends at the same vertex.

Finding efficient circuits: Need to cross every edge (read all the meters!), the best possible circuit will cross each edge exactly once = **Euler circuit**. An *Euler path* is a path that crosses every edge exactly once (but doesn't need to end where it began).

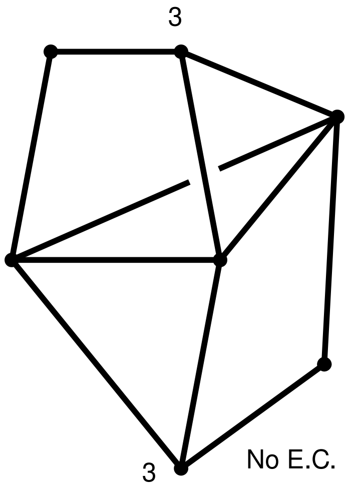
Does every graph have an Euler circuit? No...

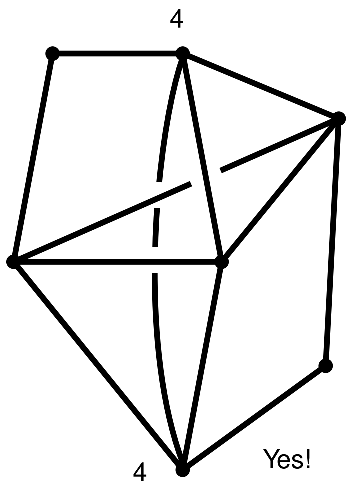
The **degree** of a vertex is the number of (ends of) edges that meet at the vertex.

A graph is **connected** if for any pair of vertices, you can find a path in the graph going from one vertex to the other (i.e., the graph doesn't come in pieces...)

Euler's Theorem: A graph has an Euler circuit if and only if (i.e., exactly when) it is both connected and every vertex has even degree. A graph has an Euler path exactly when it has at most two odd-degree vertices (the path must begin and end at the odd-degree vertices).

Basic idea: in a circuit, you have to leave a vertex the same number of times that you arrive at it! For an Euler circuit, since you don't re-use edges, you need an even number, $2n$, of edges at a vertex in order to arrive and leave n times.

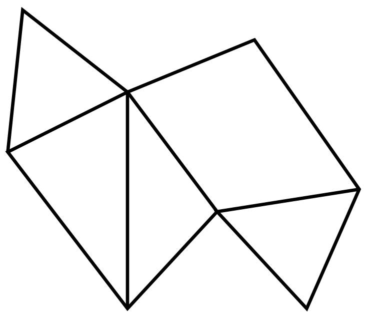


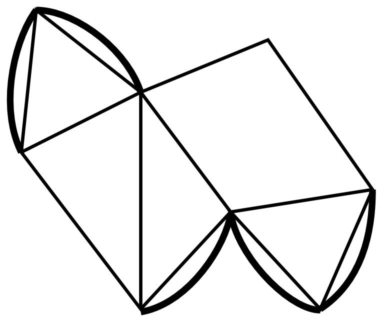


Finding an Euler circuit: don't burn your bridges! Start anywhere, and imagine erasing each edge *just before* you cross it; if erasing it disconnects what's left of the graph, **don't go that way!** (You would be cutting yourself off from the edges that are still behind you; you could never cross them.) Pick a different edge to continue along, instead. This is *Fleury's algorithm*.

If the graph has no Euler circuit, what is the best route you can find?

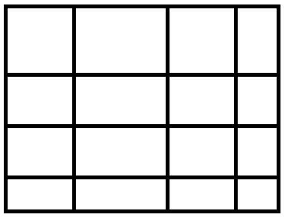
The sum of the degrees of all of the vertices of a graph = 2×(the number of edges) is *even*. So the number of vertices with *odd* degree is also even. If we pair the odd-degree vertices up, choose paths between each pair, and duplicate the edges of our graph (put in a second, parallel, edge next to the ones in our path), the resulting graph *will* have an Euler circuit (this is called **Eulerizing** the graph). The best Eulerization will have the fewest number of additional edges. This means: the sum of the lengths of the paths pairing up vertices is as small as possible. How do we find the best paths (called the Matching Problem)? We don't really know! (Practice, practice, practice?) With $2n$ vertices of odd degree, you will need to add at least n edges.





Finding the best circuit: find an Euler circuit for the Eulerized graph, and 'squeeze' it onto the original graph (imagine crossing the same edge twice, instead of walking across the added parallel edge).

For a rectangular graph, the most efficient Eulerization is usually found by walking around the outer edge; choose paths that go around the outside of the rectangle. (Edge-walker algorithm)



Basic idea: odd-degree vertices are on outer edge; pick one, and a direction to travel around the edge; duplicate the edges until you reach the next odd-degree vertex, keep going <u>without</u> duplicating edges to the next, and repeat.

Networks: Different problem, different goal. Stringing phone/electrical lines between cities/locations. The goal: connect the vertices (locations) to one another along the edges, in the most efficient way possible. What is the most efficient way to wire the vertices together? We can route calls through other cities - we don't need a direct connection. We just need to know that when we are done wiring, we have completed a **path** between each pair.

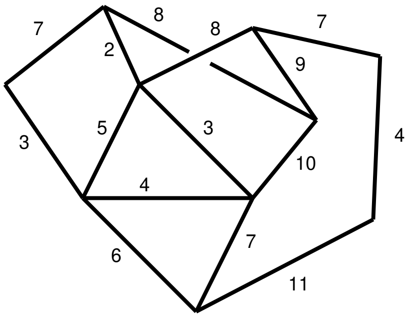
Weighted graph: graph with a weight (distance, cost to string wire, etc.) given to each edge; a *weighted graph*. The new question: what is the shortest total collection of edges that includes a path between every pair of vertices? (Emergency snow routes? Need a cleared path betwen each vertex = location.)

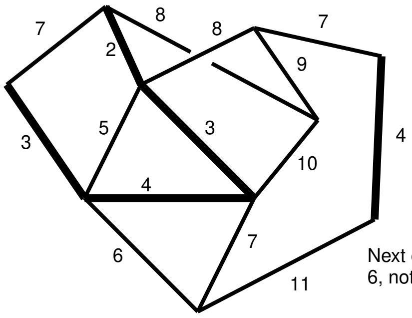
We want to find a *subgraph* (a collection of edges from the original graph) which contains all of the vertices, is connected (so every pair of vertices can be connected by a path in the subgraph), and has the smallest *total weight* = the sum of the weights of the edges being used.

Idea: **Don't** build circuits! They waste wire (remove an edge from the circuit, and all the cities in the circuit are still wired together...)

A connected subgraph that contains no circuit is called a **tree**. A tree that contains all of the vertices is called a **maximal tree**. Our shortest total collection of edges will form what is called a **minimum weight maximal tree** for the graph.

Kruskal's algorithm: Start with a subgraph consisting of just the vertices. Starting with the shortest (i.e., least weight) edge in the graph, continue to add the shortest edge to our subgraph if it **would't** form a subgraph that contains a circuit. Stop when the subgraph we have assembled is connected and contains all of the vertices. (Fact: for n vertices, you will need exactly $n-1$ edges; a tree has one less edge than it does vertices.)





Fact: Kruskal's algorithm will **always find** a minimum weight maximal tree.

Different approach, same result: never use the most expensive edge, unless you have to! Start with the <u>most expensive</u> edge. If removing it would disconnect the graph, then you *need* it to connect the vertices together, color it to add it to our maximal tree. If it doesn't disconnect the graph, you don't need it; erase it. Continue with the next most expensive edge, working until every edge has either been colored (as being needed) or erased. The edges that we colored form our minimum weight maximal tree.

For both approaches, if there are two (or more) edges to consider with the same weight, work through them in any order. You may end up with a different maximal tree at the end, but it will have the same minimal weight.