Math 203
Topics for first exam

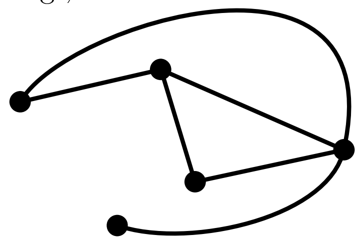
Chapter 1: Street Networks

Operations research

Goal: determine how to carry out a sequence of tasks most efficiently
Efficient: least cost, least time, least distance, ...

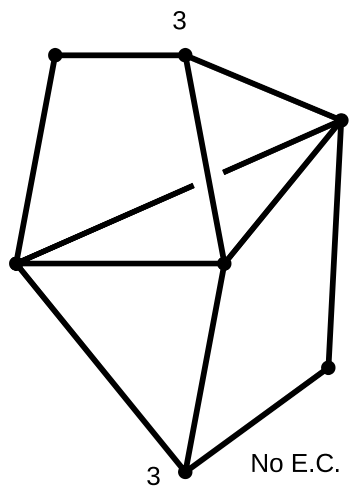
Example: reading parking meters - need to walk down each street that has meters on it. Efficient: try not to walk down the same sidewalk twice!

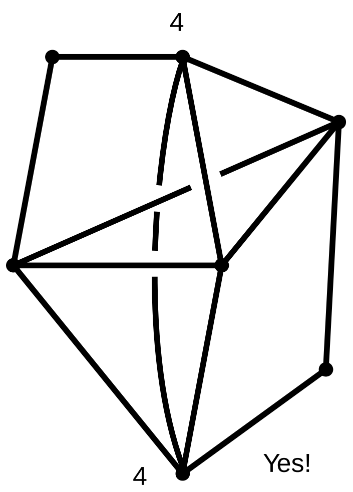
Model of the problem: a graph = a bunch of points (vertices) connected together by line segments/curves (edges).
E.g., vertices = intersections, edges = sidewalks



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 will cross each edge exactly once = **Euler circuit**.
Does every graph have an Euler circuit? No...
The **valence** of a vertex is the number of (ends of) edges that meet at the vertex.
A graph is **connected** if for any pair of edges, you can find a path in the graph going from one edge to the other (i.e., the graph doesn't come in pieces...)

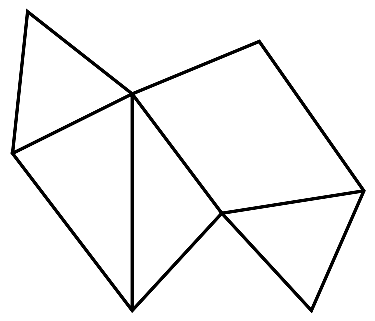
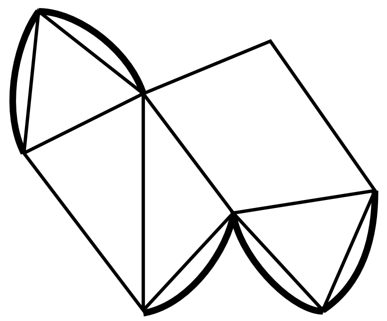
Euler's Theorem A graph has an Euler circuit if and only if (i.e., exactly when) it is both connected and every vertex has even valence.





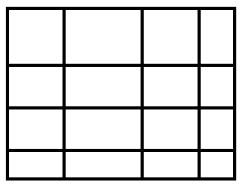
Finding an Euler circuit: start anywhere, and imagine erasing each edge *just before* you cross it; if erasing it disconnects what's left of the graph, **don't go that way!** Pick a different edge to continue along.
If the graph has no Euler circuit, what is the best route you can find?
The sum of the valences of a graph = 2×(the number of edges) is *even*. So the number of vertices with *odd* valence is also even. If we pair them up, find a path 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 (Matching Problem)? (Practice, practice, practice?) With $2n$ vertices of odd valence, 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 (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.



Complicate things? Introduce one-way streets (add arrows to edges = a **directed** graph = digraph)! Take into account the **lengths** of edges, when finding the most efficient circuit.

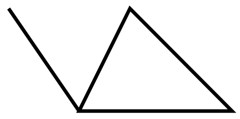
Chapter 2: Visiting Vertices

Different problem, different goal: finding efficient delivery routes. Instead of crossing every street exactly once, try to visit every vertex exactly once!

vertices = destinations , edges = routes between destinations.

A circuit that passes through each vertex exactly once is called a **Hamiltonian circuit**.

Not every graph has a Hamiltonian circuit:



Go out to the point; there's no way back!

For **complete graphs** (every pair of vertices is joined by an edge), finding Hamiltonian circuits is easy; keep walking to a new vertex until you visit them all, then return to start.

Think: each edge is labelled with a number (=distance b/w vertices, cost of shipping between destinations....). New goal: find **least cost** Hamiltonian circuit. Problem: e.g., for a complete graph with 12 vertices, there are $11 \times 10 \times \cdots \times 2 \times 1 = 11!$ (=) Hamiltonian circuits! How do you find the best one?

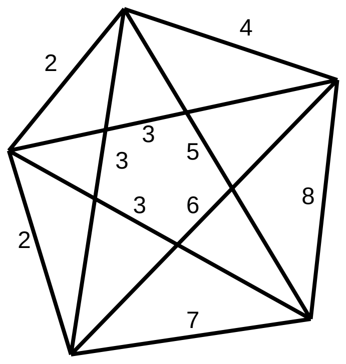
You don't. But we can think of fast ways (algorithms) to find circuits that probably won't do too badly:

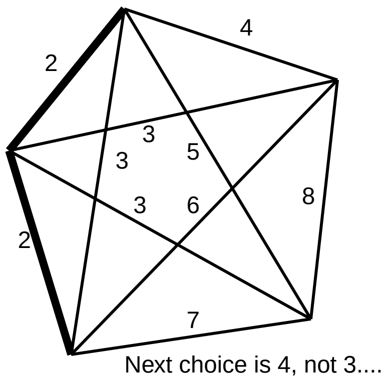
Nearest neighbor algorithm: Start at a vertex. Start building a Hamiltonian circuit by walking to the closest vertex (shortest length edge); repeat the process, walking to new vertices, until all have been visited, then walk back home.

Sorted edges algorithm: Idea: use the shortest edges you can. Sort the edges in increasing order of length. Pick edges by starting with the shortest one; keep choosing the shortest edge remaining that can be used with the ones we picked to build a Hamiltonian circuit. I.e. **discard** the shortest edge remaining if:

(a) you will have three edges sharing a vertex, or

(b) you will build a circuit that doesn't include all of the vertices.





These are both "greedy" algorithms; at each stage they both choose the shortest edge meeting certain criteria.

Different problem, different goal: stringing phone lines between cities.

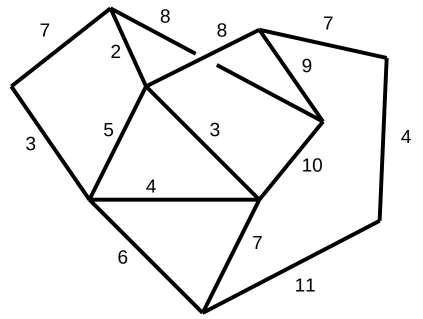
Complete graph with edges labelled (by distance, cost to string wire between,...) What is the most efficient way to wire them together? Route calls through other cities - don't need a direct connection. Just need to wire a **path** between each pair. What is the shortest total collection of edges that includes a path between every pair?

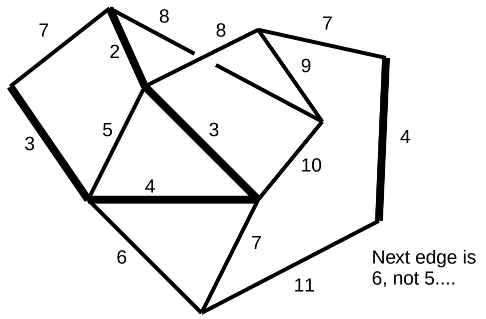
Idea: **Don't** build circuits! They waste wire (remove an edge, and all the cities in the circuit are still wired together...)

A collection of edges that form 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 cost spanning tree** for the graph.

Kruskal's algorithm: Steal from sorted edges algorithm! Pick the shortest edge; continue to add the shortest edge that **doesn't** build a circuit. Stop when all vertices are connected together (Fact: for n vertices, you will need exactly $n-1$ edges.)

This applies just as well to graphs that aren't complete (just pretend that the missing edges each cost \$1,000,000 - then the algorithm would never look at them).





Fact: Kurskal's algorithm will **always find** the minimum cost spanning tree.

Critical Path Analysis:

Idea: We have a project which involves several tasks, some of which must be completed before others can be begun (e.g., constructing a house, cooking dinner). What is the minimum amount of time required to complete all of the tasks?

We assume we have as many people as we need to assign each task to.

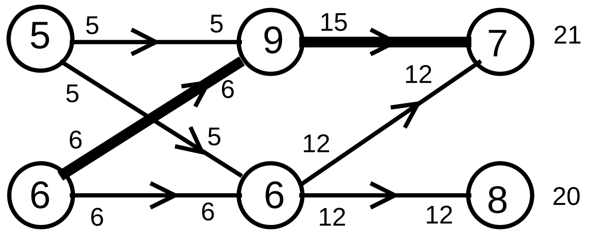
Model the progect with a directed graph. Tasks = vertices; a directed edge points from one task to anopther that cannot be begun until the first is finished (e.g., build foundation $\rightarrow$

install refrigerator) : the **order-requirement digraph**.

Basic idea: every directed path through the order-requirement digraph (**ORD**) represents a collection of tasks each of which must wait for the previous tasks to finish before starting. So the time to finish *just those tasks* is the sum of the times to do the tasks. The whole project cannot be completed in less than this time.

A **critical path** for a digraph is the longest (in terms of total completion time) directed path in the graph.

Fact: The shortest completion time for the collection of tasks is *equal* to the total time along the critical path(s).



Finding the completion time: set up the ORD so that every arrow goes left to right. Working from left to right, assign to each arrow the minimum completion time for the task it points **from**. This time is the largest of the times on arrows pointing **to** the task, plus the length of time for the task. For tasks not required for anything else, write the corresponding time to the right of the vertex. The largest number you write is the shortest completion time. The sequence of vertices that forced this number to be as high as it is represents your critical path.

Tasks on the critical path(s) represent bottlenecks; places to focus efforts to save time. Shortening a task that does not lie on *all* of the critical paths will not decrease the completion time!

Chapter 3: Planning and Scheduling

In the real world, we do not have an unlimited number of people (= processors) to assign to our tasks. How do things work if we have a set number of processors?

simplifying assumptions:

– each processor can do any of the tasks (no specialists)

– once a task is started, it is completed without stopping

– when a processor finishes, it will immediately start a task if one is available

– an order-requirement digraph tells us when a task is available (i.e., if it's prerequisite tasks have been completed)

– a **priority list** tells us which jobs we *want* to do first (e.g., shortest first, longest first, one the get us paid first...)

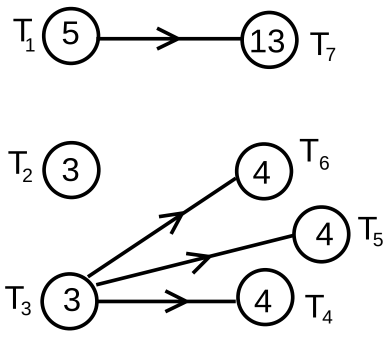
The goal of our scheduling procedure might vary depending on what we want to focus on:

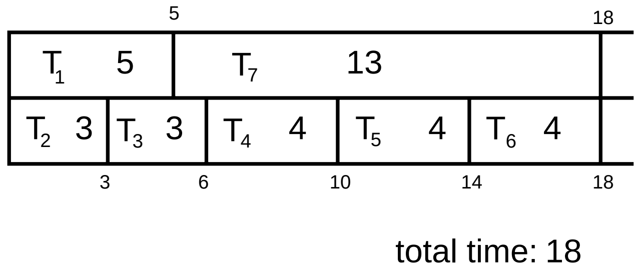
1. Shortest completion time with available resources
2. Shortest amount of idle time for each processor
3. Determine the minimum number of processors needed to complete the project in a fixed amount of time.

An easy algorithm to find the best completion time has not been (will never be?) found.

List processing algorithm: As each processor finishes a task, assign it the highest priority task that the order requirement digraph says is available. If no task is available, stand idle until one is available.

Priority list: $T_1, T_2, T_3, T_4, T_5, T_6, T_7$





Some strange behavior:

– shorten each task in the project above by 1: longer completion time!
– add a third processor: longer completion time!
– ignore the order requirement digraph: longer completion time!

Moral: We chose a very bad priority list....

Critical path analysis tells us that tasks on the critical path(s) are potential bottlenecks, slowing down overall completion time. The first task on a critical path should be given highest priority.

Critical path scheduling: Build the priority list using critical paths. The first task(s) on critical path(s) are given highest priority. Pretend that the first task is *completed*, and find the **new critical path(s)** for the remaining tasks. Give the first task on the new path the next highest priority. Continue through the ORD until every task has been placed on the priority list. Then schedule the tasks using the ORD and this priority list.

Independent tasks: Get rid of the ORD!

Many projects naturally have no ORD; e.g., processing a large number of photocopying orders, grading term papers (time=length?). The individual tasks may be completed in any order; no task needs to wait until another has finished. These are called **independent tasks**.

A fairly efficient way to schedule independent tasks is using a **decreasing time** (priority) **list**. Idea: a common source of inefficiency is having one processor working alone long after all others have finished. A long task left until the end will tie up only one processor, while everyone else sits idle. So, make sure that the last task is as short as possible! Sort the tasks in **decreasing** order of time to complete, and use this as your priority list.

Fact: If T is the best possible completion time for m processors, then the decreasing time list wil give a completion time of **at most** $(4/3 - 1/(3m))T$ (e.g., for 2 processors, this is $(7/6)T$).

Bin Packing: Set the completion time! How many processors do you need to finish the project in a set amount of time T? (Note: no single task can take more than time T...!)

Think: objects of differing heights (= completion times) being stacked on top of one another (= being assigned to each processor) into bins of height T (=for a total time of at most T each). How many bins (=how many processors) will you need to hold (= finish) all of the tasks?

Some likely procedures:

Next fit: keep adding items to a bin until the next one won't fit, then move on to the next bin.

First fit: put each item into the **first** bin it will fit in.

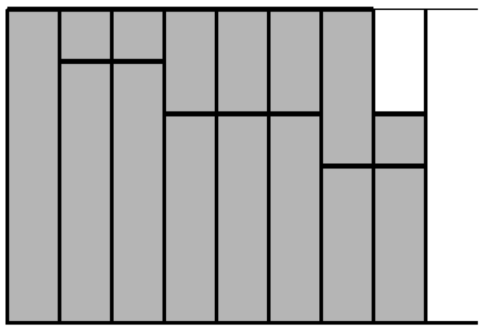
Worst fit: put each item into the open bin with the most available space (if possible), otherwise open a new bin.

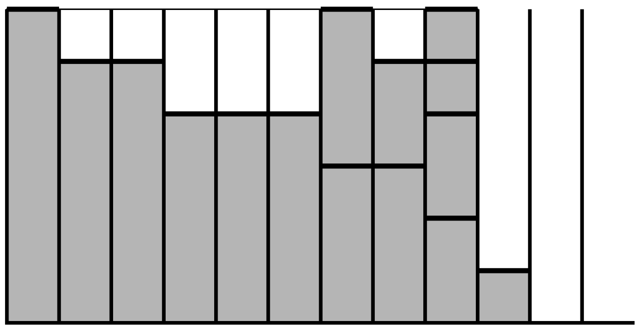
These can all be inefficient; large objects appearing late in the list will not be packed efficiently. (We all know that when packing, the **big** stuff goes in first, and the little stuff is packed around it...)

More efficient: create a decreasing time list **first**, so that big stuff gets packed away first. List all objects in decreasing order of size, first. Otherwise, proceed as before. This gives:

Next fit decreasing, First fit decreasing, Worst fit decreasing.

3,4,5,2,1,1,5,3,2,6,3,4,4,2,1 reorder: 6,5,5,4,4,4,3,3,3,2,2,2,1,1,1


first fit decreasing (8 bins)


next fit decreasing (10 bins)