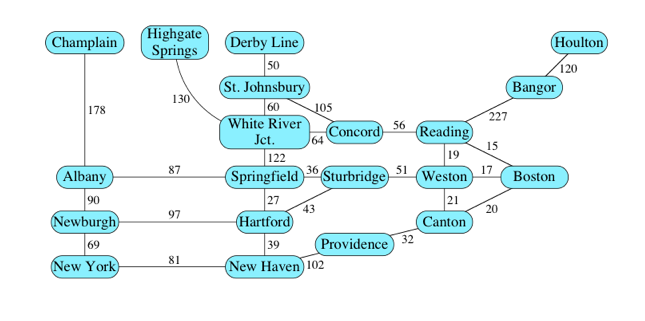


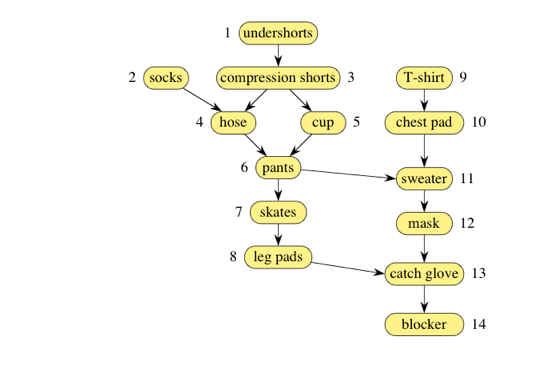
This social network is a graph. The names are the **vertices** of the graph , each line is an **edge**. Because the “know each other” relationship goes both ways, the graph is undirected. An edge between two vertices in an undirected graph is called an **incident**, and we say that the two connected vertices are **adjacent** or **neighbors**. The number of edges incident on a vertex is the **degree** of the vertex.

Audray and Emily don’t know each other. How could we make them meet and start a relationship? Well, we could introduce them through Bill. We would then say there is a **path** of three edges between Emily and Audrey. Or Gayle would introduce Audrey to Cathy, who would then introduce Audrey to Emily, and Emily to Bill. This would be a path of five edges.

Sometimes we might put numerical values on our edges. A path would then have a bigger value than another one, even if they have the same number of edges. In this example we are representing a road network. The shortest way from one point to another depends not only on the number of roads (edges) taken, but also on their lengths (values, or **weight**).



Sometimes the edges will be unidirectional. This examples illustrates the order in which a hockey player would put on his gear



For example, the arrow from the chest pad to the sweater indicates that the chest pad must be put on before the sweater.

There are no cycles in this graph: we call it a **directed acyclic graph** (**dag** for short).

For directed graphs, we are gonna say that an edge **leaves** a vortex and **enters**  another. The number of edges leaving a vertex is an **out-degree** and the number of edges entering a vertex is an **in-degree**.

We usually denote the vertex set by **V** and the edges set by **E**. If a graph has a small number of edges, we say it is **sparse**.

**Representing graphs**

There are three ways to represent a graph. While choosing which way, we need to make sure it doesn’t take up too much memory (determined through asymptotic notation), we also need to take into account how long it takes the program to determine whether an edge is part of the graph and how long it takes to find the neighbors of a given vertex.

We will indentify the vertices by numbers (from 0 to |V| - 1).

Let’s take a look at the three ways to represent a graph.

Edge List

One way is to represent a graph by an array of |E| edges. Each edge in the array will be represented by the two vertexes it touches, and we’ll add a third value if the vertex is weighted.

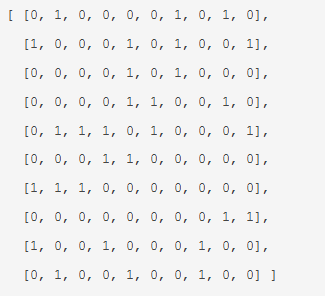
http://i.gyazo.com/4c98b994ba13d8aee85193eaabf46e89.png

It’s a simple representation, but if we want to know whether a given edge is on the graph, we have to search through the whole array, and if the array is not sorted, this amounts to a linear search through |E|.

Adjacency matrices

For a graph with |V| vertices, and adjacent matrix is a |V|x|V| matrix of 0 and 1s, with the index of the vertices on the horizontal and vertical legend. If vertex p touches vertex q (meaning that edge(p,q) is **on the map**), there will be a 1 at (p,q) and (q,p) in the matrix, and a 0 otherwise. Of course, a vertex can not be his own neighbor, so the matrix diagonal will be filled with zeroes.

Here is the matrix in javascript:



Here, if we want to find whether an edge is on the graph, we just have to look at vertices it would connect in the double array, so the search running time is constant. The problems comes form somewhere else: its sheer size. It just takes too much space, and that’s that much computational power lost (|V|² space). Second, if you want to check which vertices are adjacent to a given vertex, you have to look through all the array every time.

For an undirected graph, the matrix is **symmetric**. For a directed graph, the matrix has to be not symmetrical.

Adjacency list

This is a combination of the two previous methods. For each vertex I, we store an array of all the vertices adjacent to it. It’s just like the matrix if you will, because each subarray back then reflected which vertices were adjacent to a vertex, but since we are now using the indexes of the vertices instead of 0 and 1s, the size of the subarrays will be <= |V|. If a graph is sparse, we will have much smaller subarrays.

Worst case scenario for running time is theta(d), where d is the index of vertex i. Indeed, we go to vertex I in constant time, and d is a variable which depends on how many edges are connected to vertex i. (and d<=|V|-1)

Adjacency lists take up 2|E + V| space if the graph is undirected (each adjacency/edge [AB] occurs exactly twice, for vertex A and vertex B) and |E + V| for directed graphs.