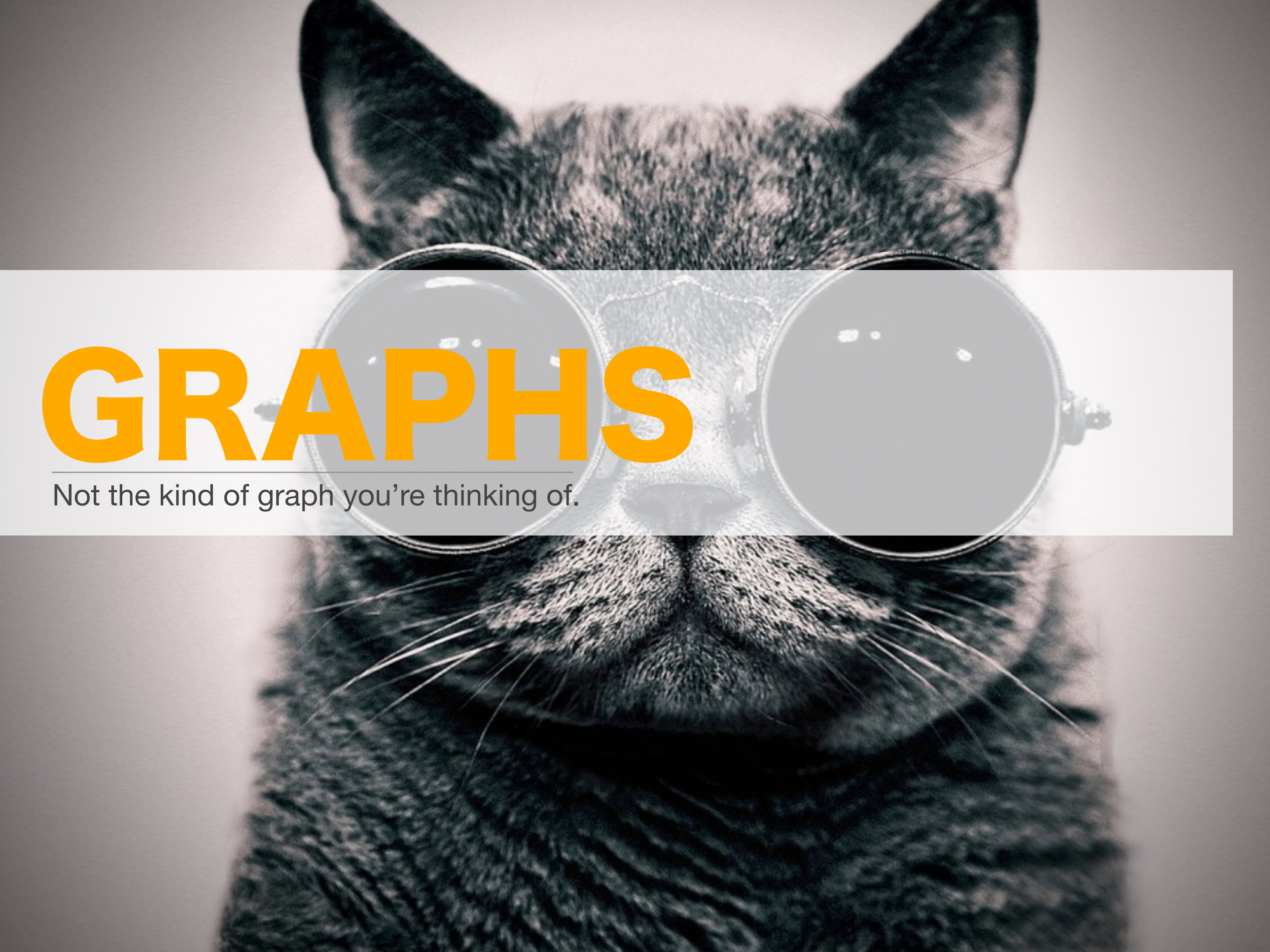


GRAPHS

Not the kind of graph you're thinking of.

Who are you exactly?

And why should I pay attention?

Founder & CEO at

RECRUITERS!!! ACTUALLY HAS FIVE YEARS OF EXPERIENCE WITH

@indexzero

Who are you exactly?

Come to EmpireJS!

http://2014.empirejs.org

GRAPHS

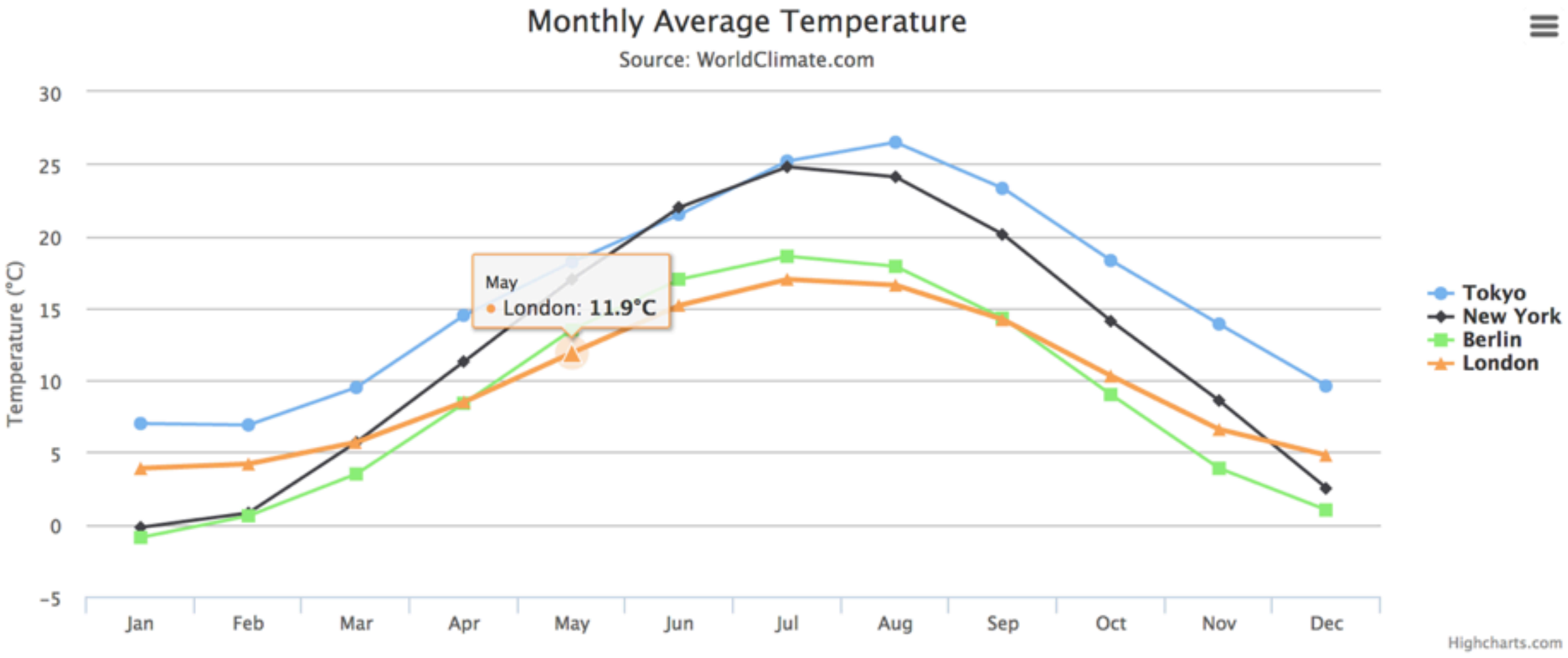
RULE

EVERYTHING

AROUND

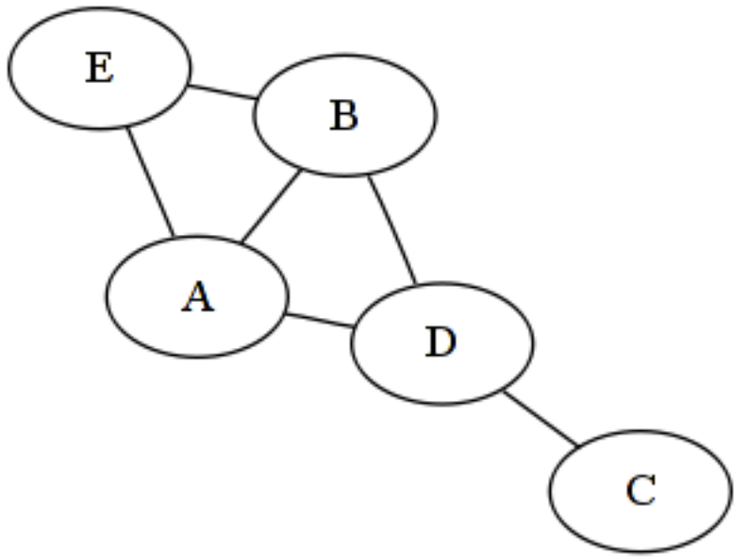
ME

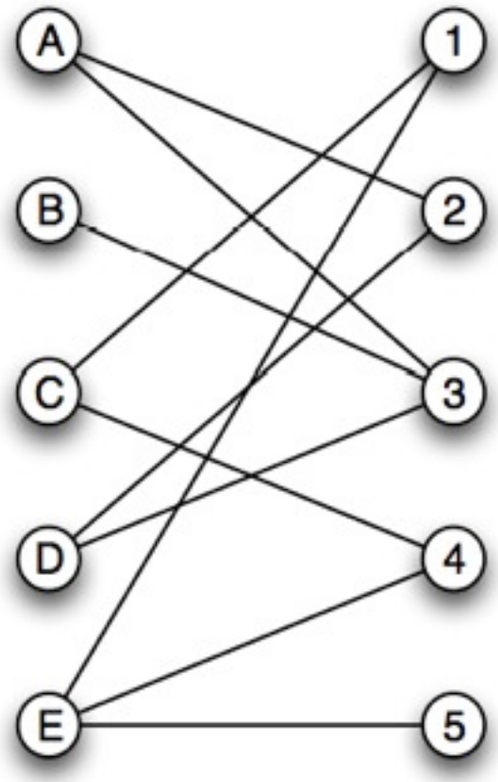
NOT THIS KIND

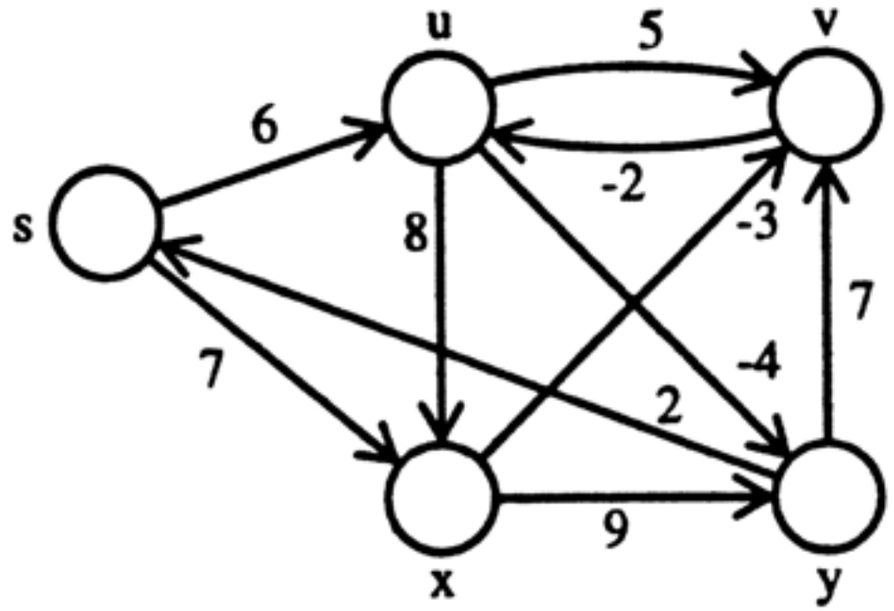


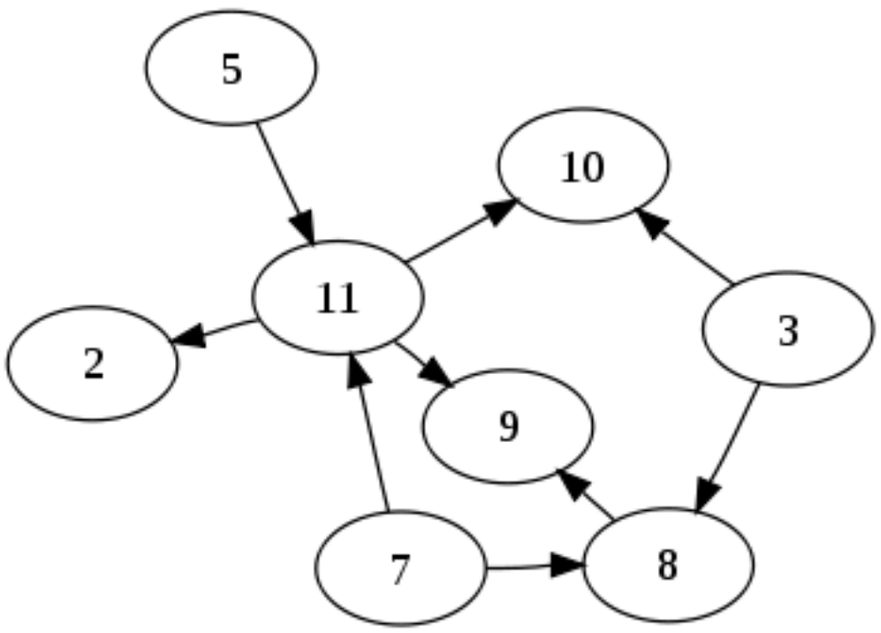
OF GRAPH

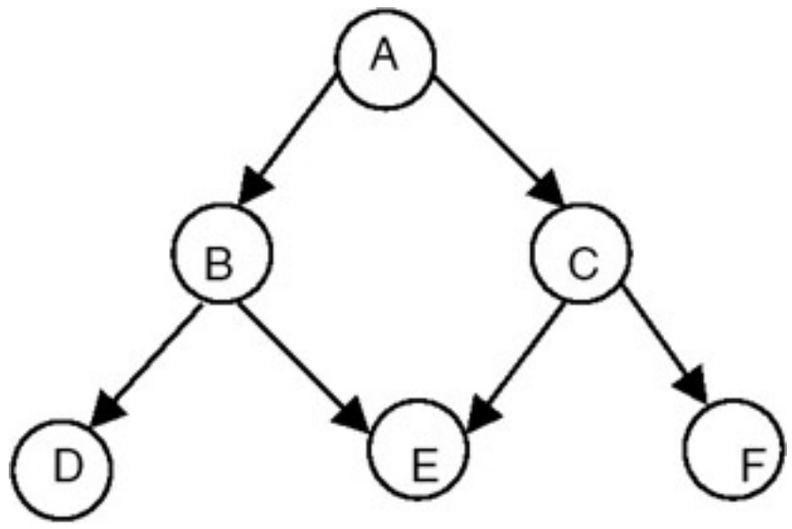
THESE KINDS











OF GRAPHS

TRANSPORATION NETWORKS

INFORMATION NETWORKS

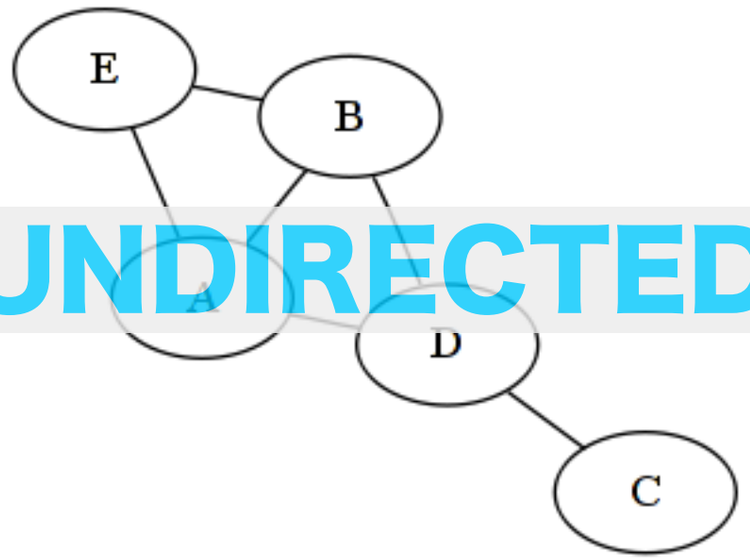
MOLECULAR CHEMISTRY

WIRELESS NETWORKS

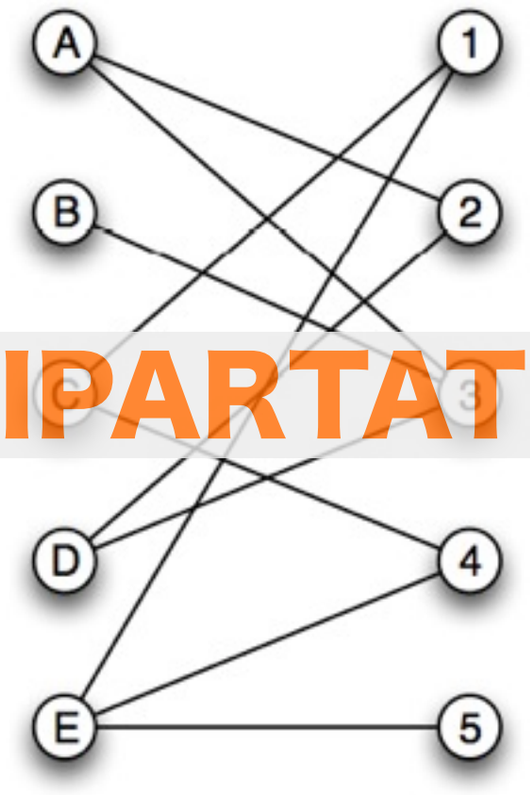
DEPENDENCY MANAGEMENT

MAJOR LEAGUE BASEBALL

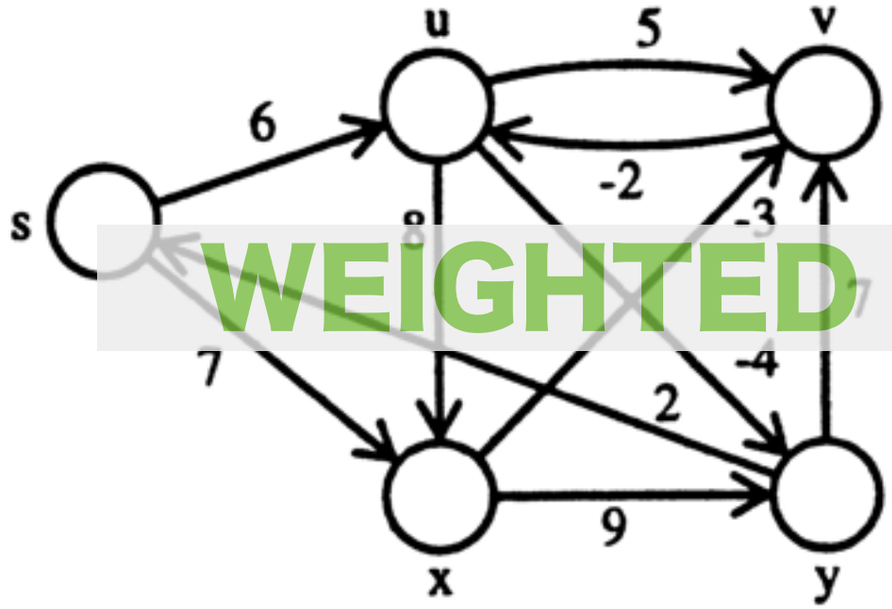
ALL OF THESE GRAPHS



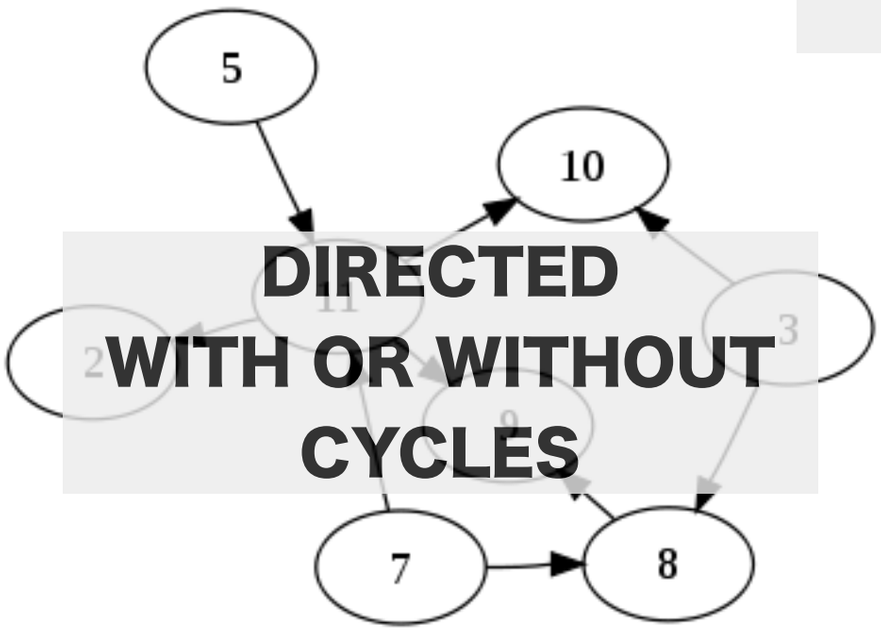
UNDIRECTED



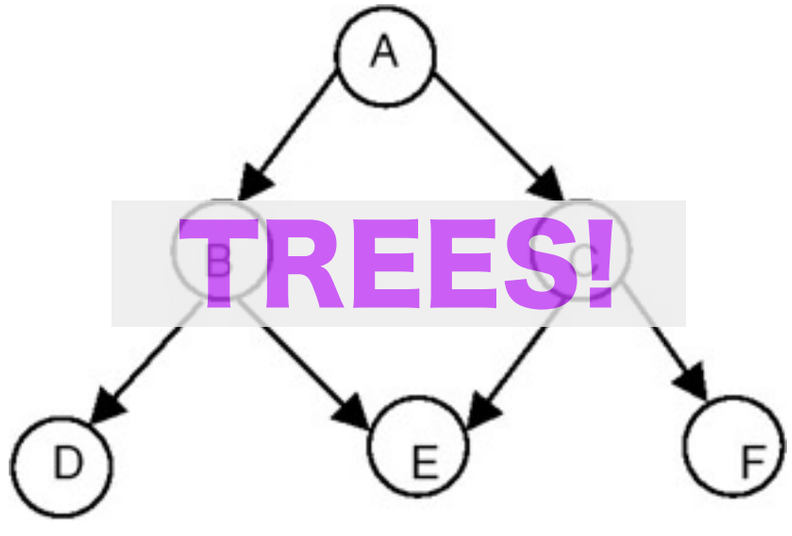
BIPARTATE



WEIGHTED



DIRECTED
WITH OR WITHOUT
CYCLES



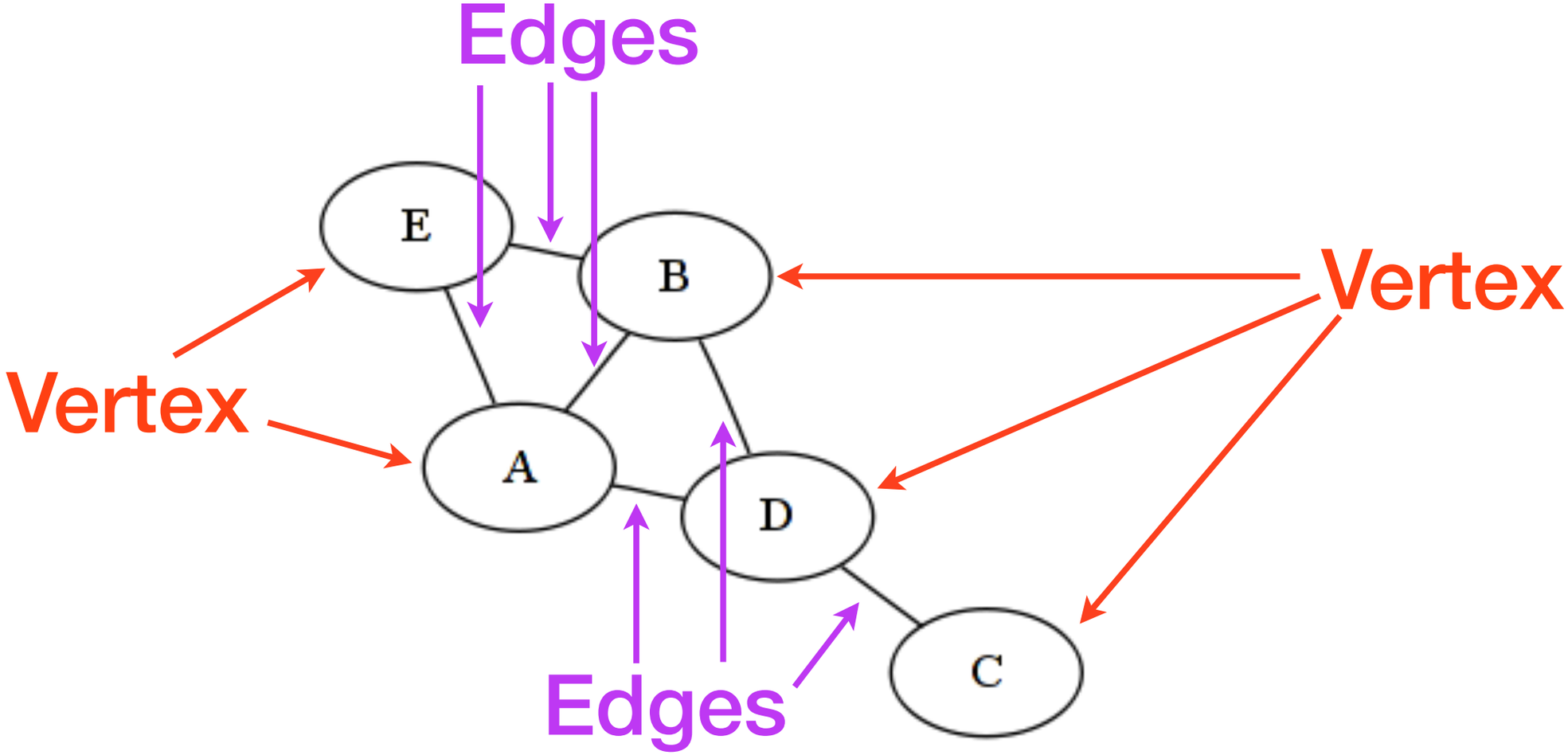
TREES!

ARE DIFFERENT

The G = (V,E) kind of Graph

It’s kind of like math. Yes, math.

Definition 1. A directed graph consists of a finite set of vertices ***V*** and a set of directed edges ***E***. An edge is an ordered pair of ***vertices (u,v)***.



The G = (V,E) kind of Graph

The math will be over soon. I promise

- A **path** is a sequence of vertices $(x_1, x_2, \ldots, x_n)$ such that consecutive vertices are adjacent (edge $(x_i, x_i + 1) \in E$ for all $1 \leq i \leq n - 1$).

- A path is **simple** when all vertices are distinct.

- A **cycle** is a simple path that ends where it starts, that is, $x_n = x_1$.

- The **distance** between between u and v is the length of the shortest path from u to v.

- An undirected graph is **connected** when there is a path between every pair of vertices.

The G = (V,E) kind of Graph

The math will be over soon. I promise

- The **connected component** of a node u in an undirected graph is the set of all nodes in the graph reachable by a path from u.

- A directed graph is **strongly connected** when, for every pair of vertices u, v, there is a path from u to v and a path from v to u.

- The **strongly connected component** of a node u in a directed graph is the set of nodes v such that there is a path from u to v and a path from v to u.

The G = (V,E) kind of Graph

Last set of definitions and notation

- $|V| = n$ = The number of ***verticies***
- $|E| = m$ = The number of ***edges***
- A graph algorithm is linear when it runs in $O(|V| + |E|) = O(n + m)$.

Graphs. How do they work?

Two common internal representations

Question: How do we represent the edges?

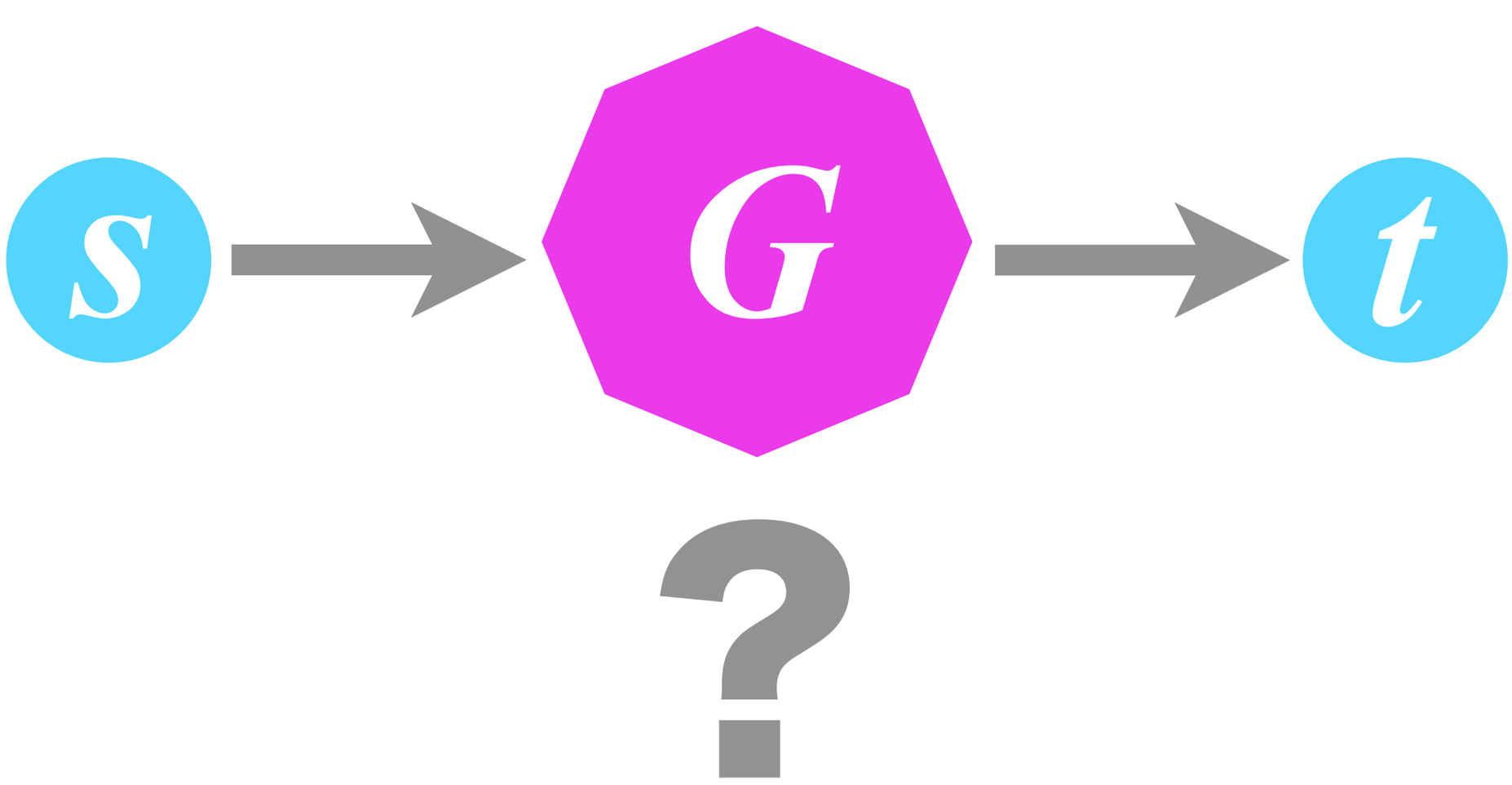
- ***Adjacency list:*** A graph with edges $|E| = m$ can be represented as a list of **m adjacency pairs.**

- ***Adjacency matrix:*** A graph of size $|V| = n$ can be represented as **an n x n matrix.**

Graph Search Algorithms!

Breadth First Search & Depth First Search

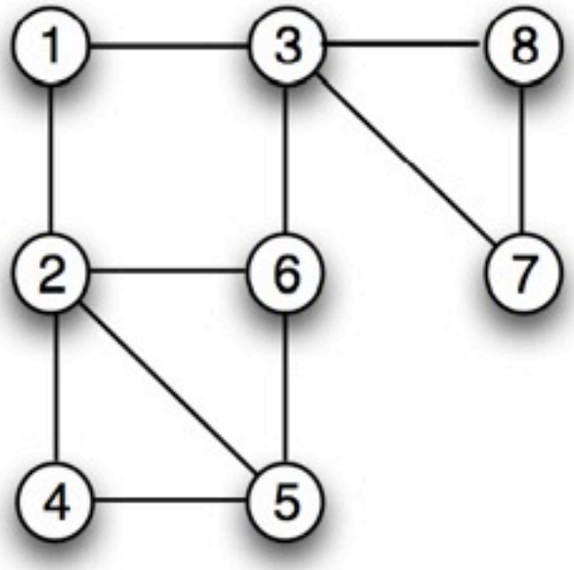
The Question. does the graph G contain an s-t path? i.e. Does a path exist between two verticies, s & t, in the graph G?



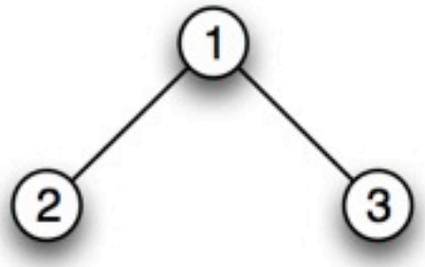
Graph Search Algorithms!

Breadth First Search

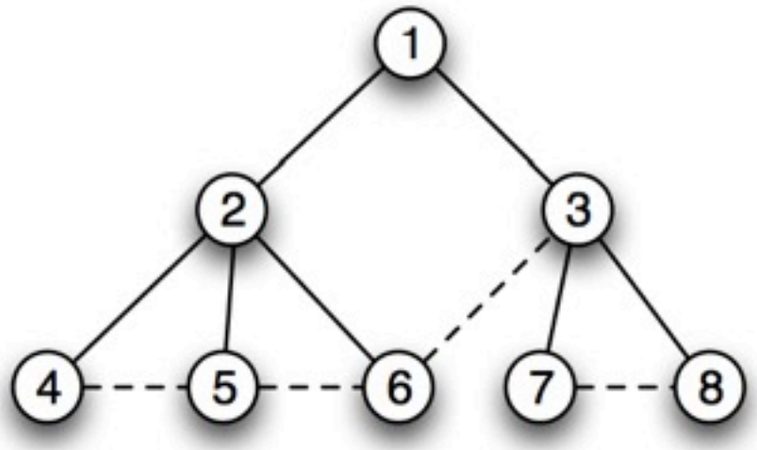
The main idea of BFS is to explore the graph starting from a vertex s outward in all possible directions, adding nodes one “layer” at a time.



(a) Graph



(b) 1st layer

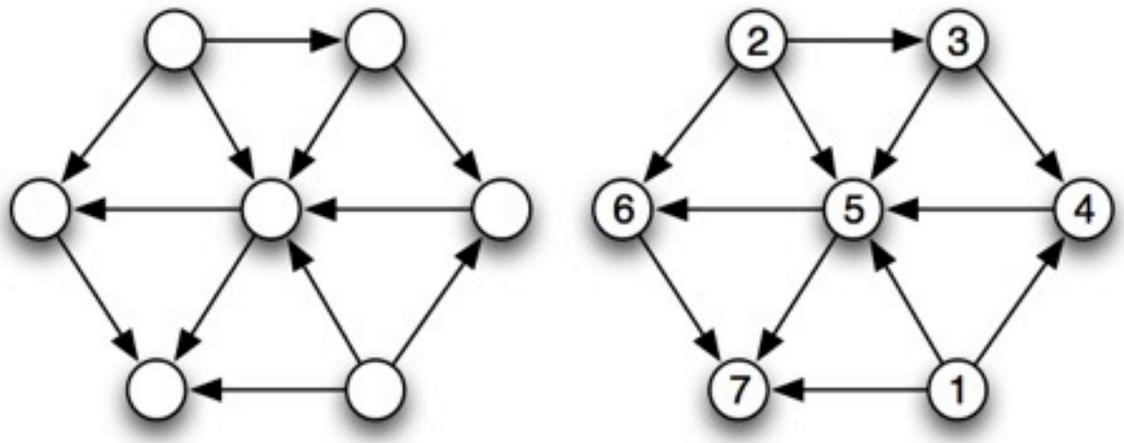


(c) 1st and 2nd layers

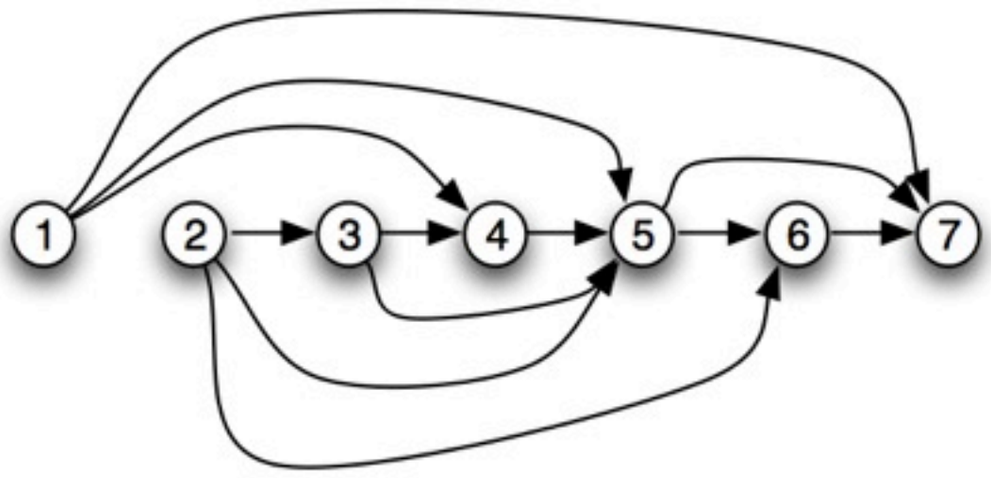
Graph Search Algorithms!

Depth First Search

1. Start at s, and try the first edge out of s, towards some node v.
2. Continue from v until you reach a “dead end”, that is, a node whose neighbors have all been explored.
3. Backtrack to the first node with an unexplored neighbor and repeat 2.



(a) A DAG and its topological order



(b) Topologically sorted DAG: edges go from left to right

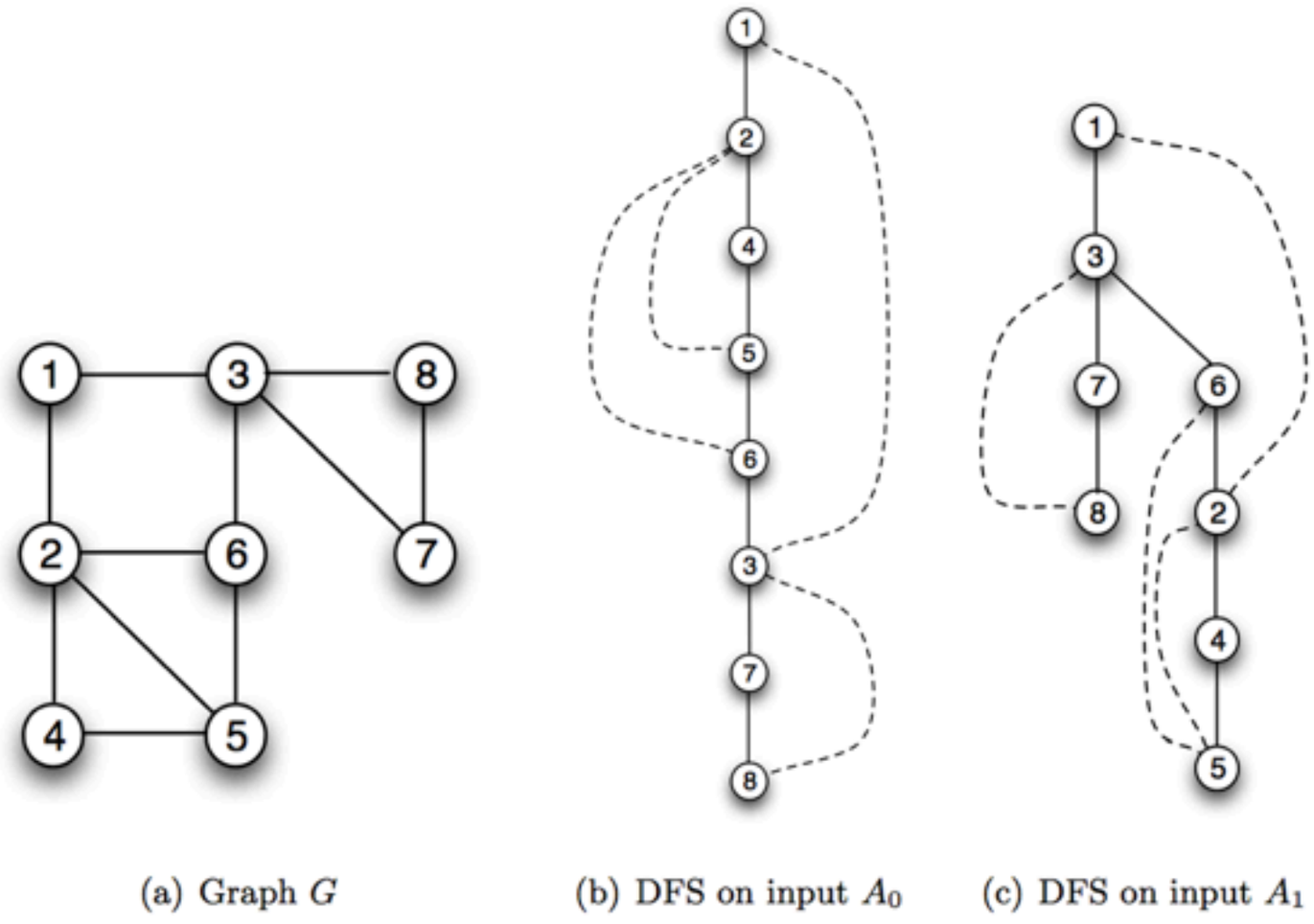
A graph is a tree!

DFS makes your graph a tree! Well .... *sort of.*

1. **Forward edges:** go from an ancestor to a descendant other than a child.

2. **Back edges:** go from a descendant to an ancestor, other than the parent.

3. **Cross edges:** go from right to left, that is,
   - from **tree to tree**; e.g., (v_5, v_4) in Figure 2(b).
   - between nodes **in the same tree** but in different branches. e.g., *(u, v)* on the right of Figure 3(b).



(a) Graph G (b) DFS on input A_0 (c) DFS on input A_1

Path Finding Algorithms!

All your paths are belong to s-t. Wait what?

The Question. what are all of the ***shortest paths in G from s***? i.e. What is the shortest path from a single source, s, and to all other verticies in G?

By knowing an answer to this, we also know:

- ***Single-pair shortest-path:*** What is the ***shortest s-t path in G?***
- ***Single-destination shortest-path:*** what are all of the ***shortest paths in G to a single sink, t***?
- ***All-pairs shortest-paths:*** What is the shortest path between ***all (u,v) vertex pairs in G***?

Graph Algorithms!

The more you know.

Did you know that the collective form of computer scientists is ...?

a dijkstra!

Seriously tho. Dijkstra was a boss.

He created an algorithm to solve this for us.

Here is how it works.

Path Finding Algorithms!

Dijkstra's algorithm and friends.

Start with your single source, s

And a set of shortest paths from s for vertices, Q

Initially, Q only contains s, with $distance(s) = 0$

In every iteration, examine $V - Q$ and select vertex v:

1. **has an incoming edge from some $vertex \in Q$**
2. **minimizes the quantity among all $v \in V - Q$**

$$d(v) = \min_{u \in Q} distance(u) + weight(u, v)$$

MIND. EXPLOSION.

I should spend all my time on npm

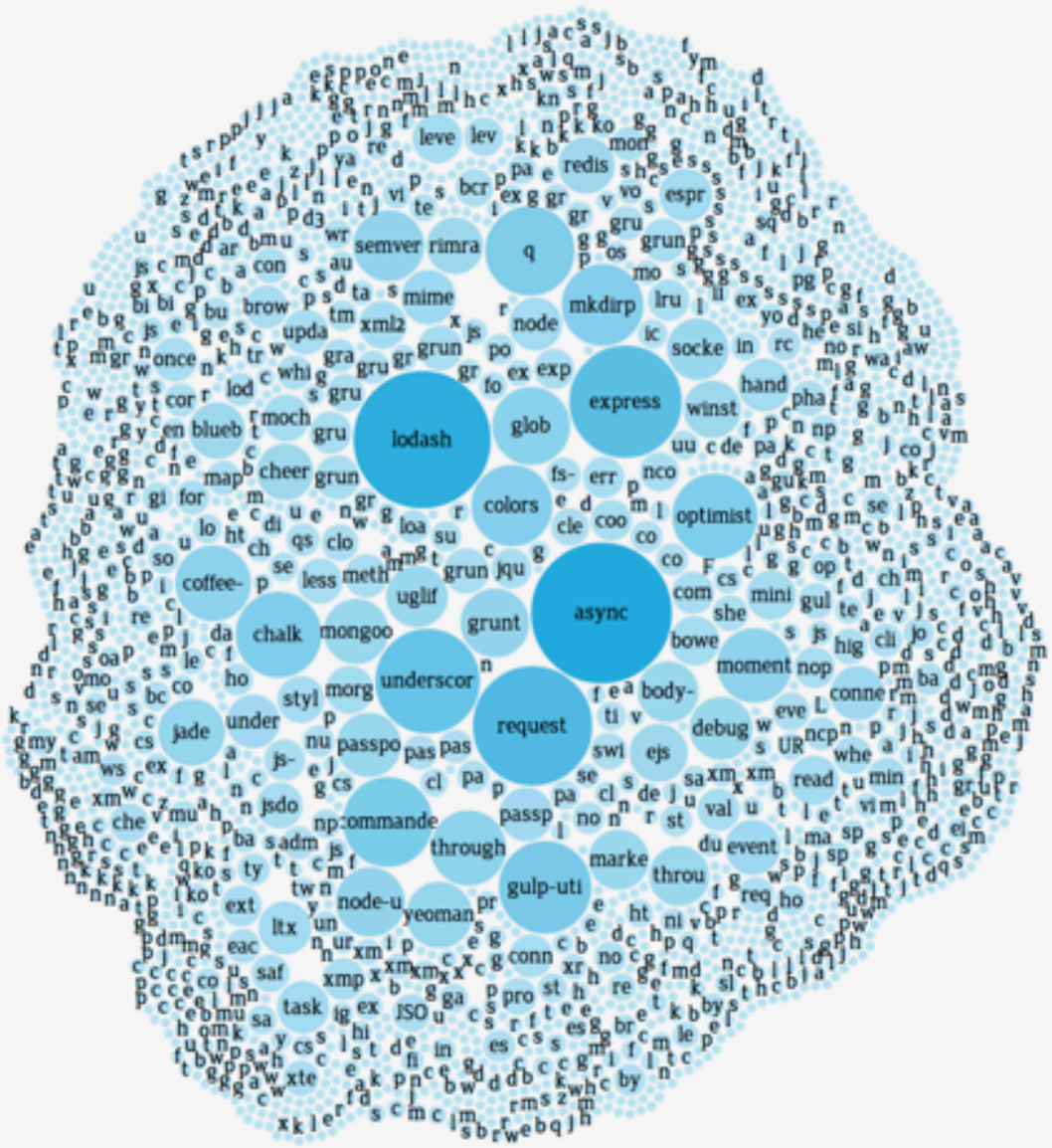
Ask me about it later if you want.

Dependency graphs

An application of graphs in “real” life!

- Basic graph representation is not extremely helpful for visualizing package relationships.

- But it does provide a basic structure for a graph search problem.



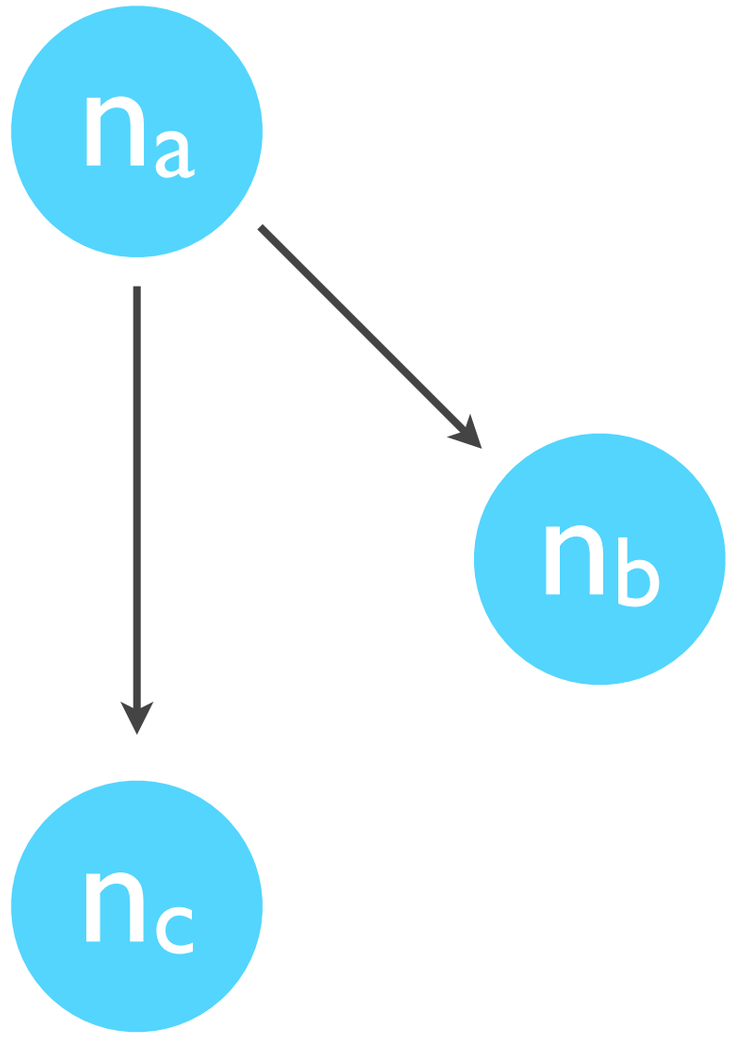
Dependency graphs

The G = (V,E) kind of graph

- To build our graph, G, we add a node (or vertex) for every package.

- We then add a colored edge from any node $\mathbf{n_A}$ to $\mathbf{n_B}$ if package A depends on package B.

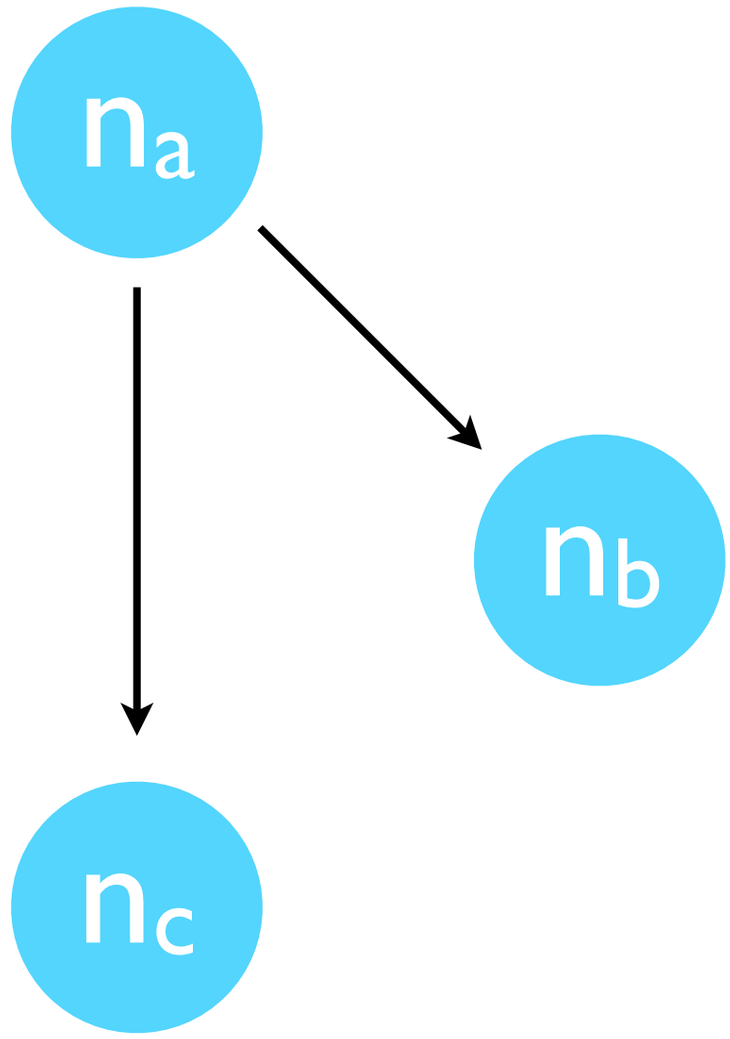
- Edges are colored by dependency type: `dependencies` or `devDependencies`



Dependency graphs

The G = (V,E) kind of graph

```
{
  "name": "package-a",
  "dependencies": {
    "package-b": "~1.0.4",
    "package-c": "~2.1.3"
  },
  "main": "./index.js"
}
```



- **Now imagine this graph for 70,000+ modules.**

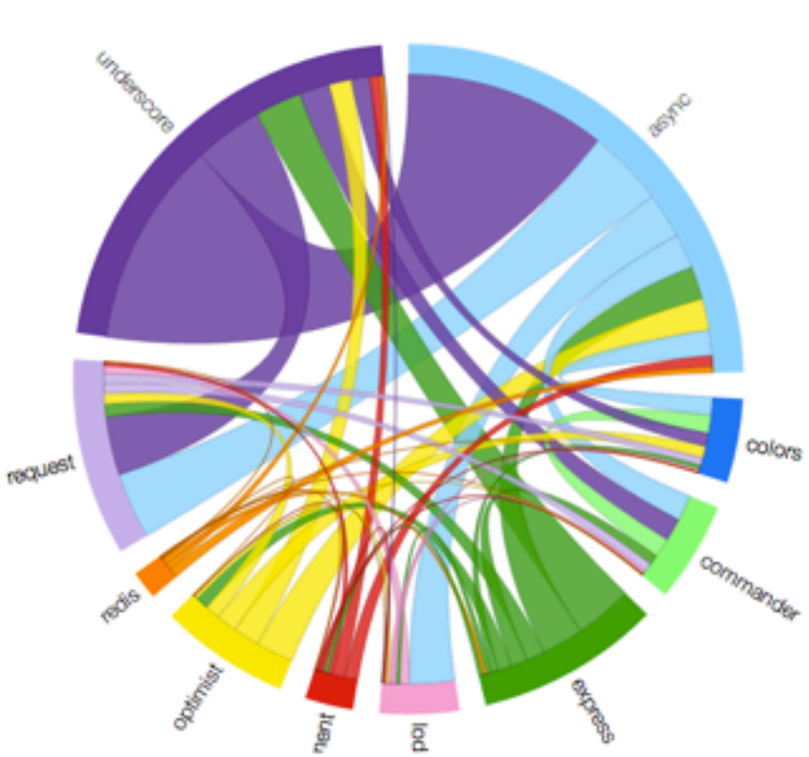
Actually.
It’s not so bad.

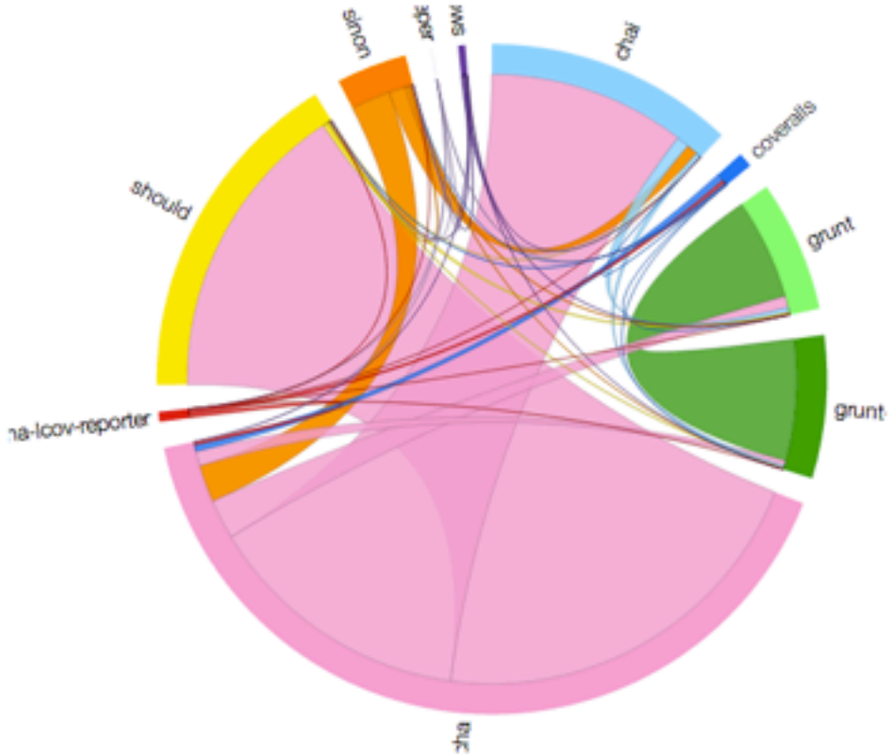
Dependency graphs

Ok, so what is this useful for?

- There are tons of useful applications of dependency graphs.
- **Lets consider one.**

Codependencies





Codepdencies?

Well, technically co(*)dependencies.

- Codependencies answer the question *“people who depend on package A also depend on ...”*

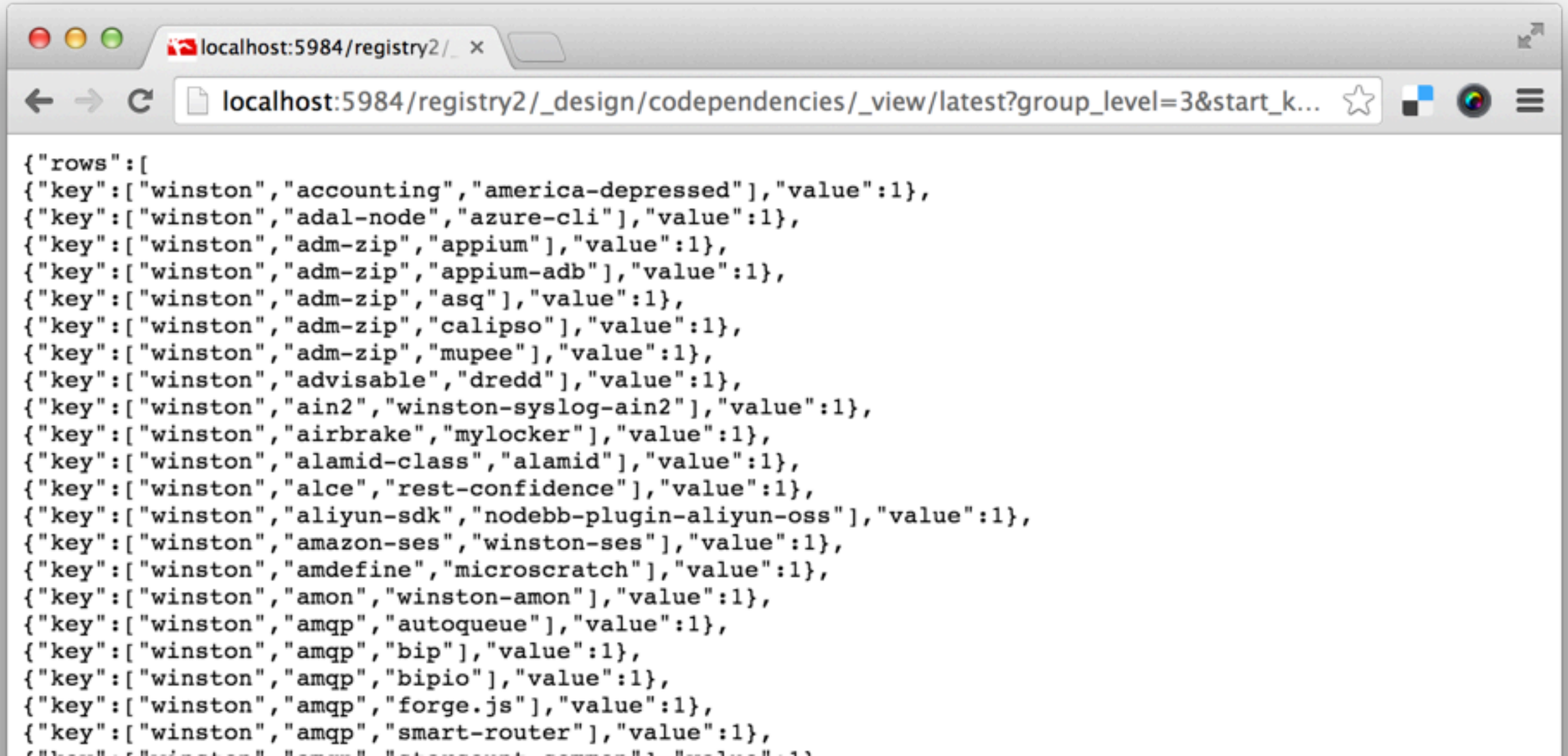
```
["package", "codep", "thru"]
```

<3 CouchDB!

Codepdencies?

Thanks CouchDB!

```
group_level=3
  &start_key=["winston"]
  &end_key=["winston",{}]
```



Codepdencies

The meat of the analysis

- So this is all well and good, but what the **heck are you doing?!?!**

```
For module NAME generate a matrix by:
- Rank codependencies based on
  number of times they appear
- For each codependency C in the
  SET of the top N:
    Rank SET - {C} by number of times
    they appear to create ROW[C]
```

Codepdencies

Understanding codependencies through winston

- This last step yields a matrix for the codependency relationship:

	asyn…	expr…	opti…	requ…	unde…
asyn…	0.0000	553.0000	534.0000	837.0000	1359.0000
expr…	553.0000	0.0000	314.0000	365.0000	648.0000
opti…	534.0000	314.0000	0.0000	335.0000	448.0000
requ…	837.0000	365.0000	335.0000	0.0000	786.0000
unde…	1359.0000	648.0000	448.0000	786.0000	0.0000

Codepdencies

Understanding codependencies through winston

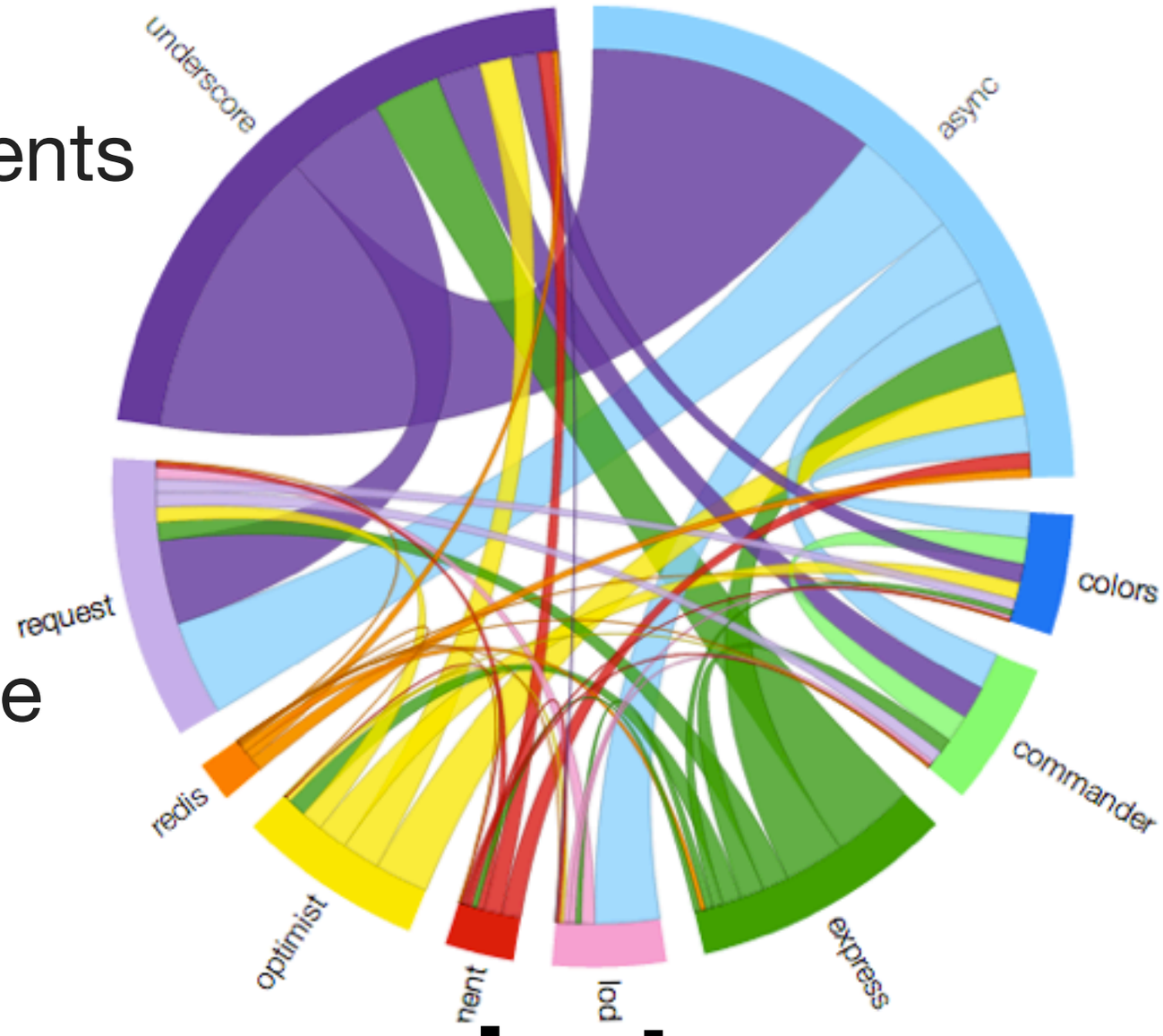
- Now we need to weight the matrix based on the overall appearance of these codependencies

```
240 | async      | 0.2761
207 | underscore | 0.2382
163 | express    | 0.1875
133 | request    | 0.1530
126 | optimist   | 0.1449
869 total
```

Codepdencies

Reading the tea leaves of dense data visualization

- **The size of the arc** represents the degree of the codependency relationship with the parent module.
- **The size of the chord** represents the degree of the codependency relationship between each pair.
- **The color of the chord** represents the “dominant” module between the pair.



I CAN HAZ MORE GRAPHS?

- Dijkstra fails under some conditions
- Minimum Spanning Trees
  - Prim's Algorithm
  - Kruskal's Algorithm
- Bipartite Matching Graphs
- Flow Networks & min s-t cuts
  - Ford-Fulkerson
  - Edmonds Karp


DONE
COMEDY