Graphs I

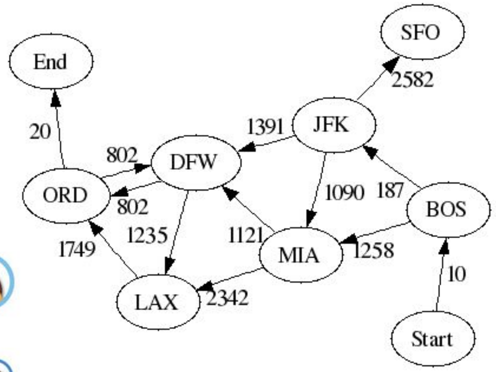
AGENDA

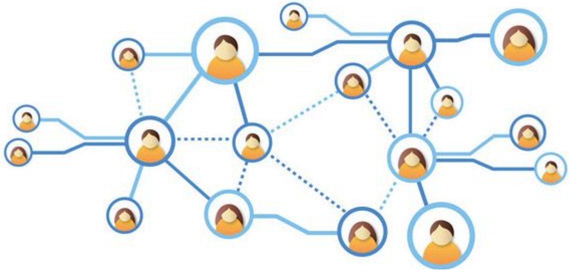
- Intro to Graphs
- Graph Properties
- Representing Graphs
- Graph Traversals

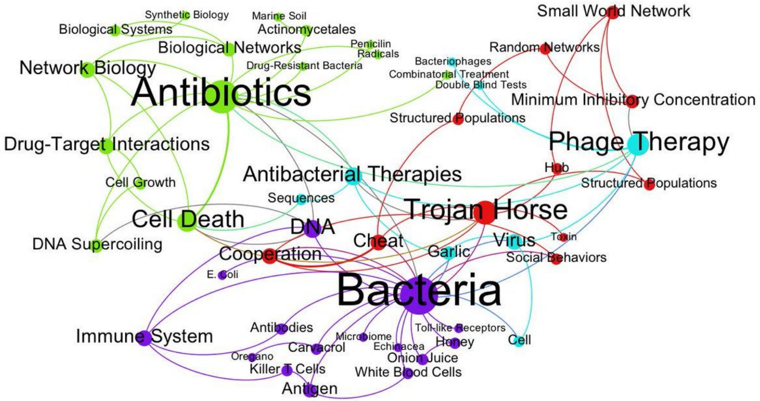
Intro to Graphs

INTRO TO GRAPHS

- A very versatile data structure that allows you to represent relationships between data
  - Social network, flight schedule, word relationships, etc.

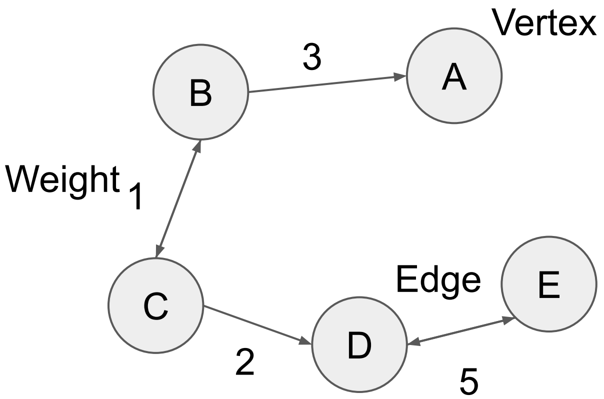






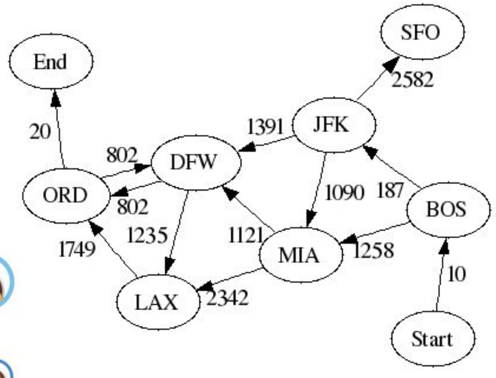
COMPONENTS OF A GRAPH

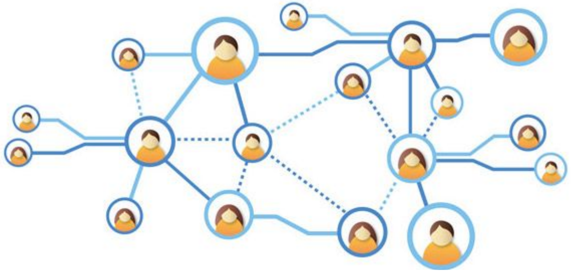
- *Vertex* - also called nodes
- *Edge* - connects a pair of nodes
    - *Unidirectional* - Path from A to B doesn't mean there's a path from B to A
    - *Bidirectional* - A is friends with B, then B is friends with A
- *Weight* - used to represent a value associated with the edge (usually a cost)

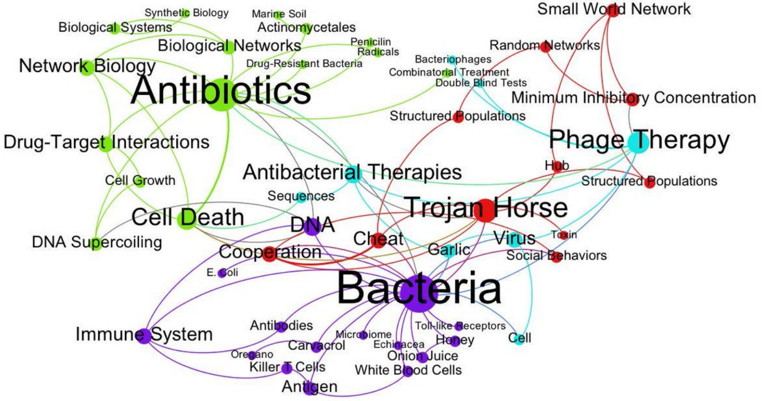


EXAMPLES OF GRAPHS

- Social Networks
    - Each node is a user, edges are friendships between nodes
    - Edges can also be to which groups you are a part of
- Transportation Systems (BART, Maps, etc.)
    - Each node is a location, each edge is a route to another one. A weight can represent time to get there
- The Internet!
    - Each page can be represented as a node, an directed edge is a link to another web page



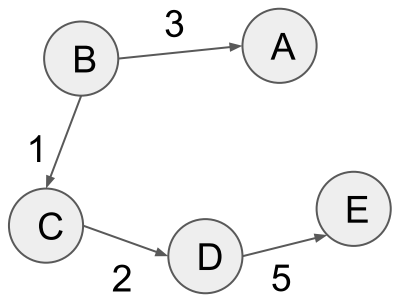


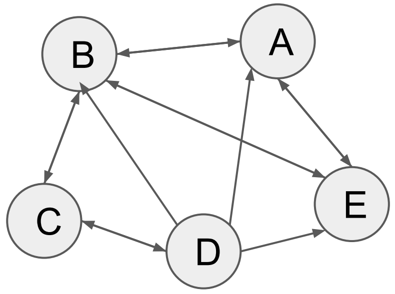


Graph Properties

GRAPH PROPERTIES

- A graph can have multiple properties
- Knowing these different properties are important so you can build/solve graph problems!

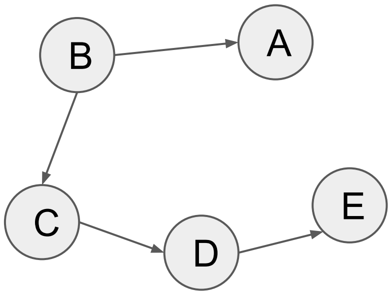




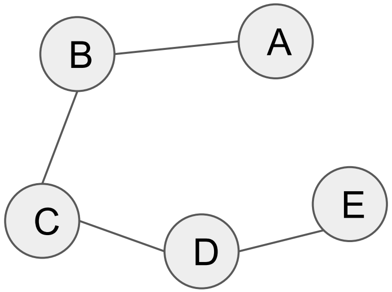
DIRECTED VS. UNDIRECTED

- A graph can be either *directed* or *undirected*
- *Directed* - An edge from A to B doesn't mean there's an edge from B to A
- *Undirected* - An edge from A to B means there's also an edge from B to A

Directed



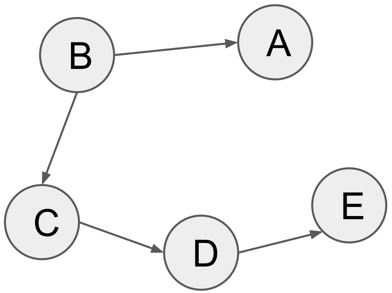
Undirected



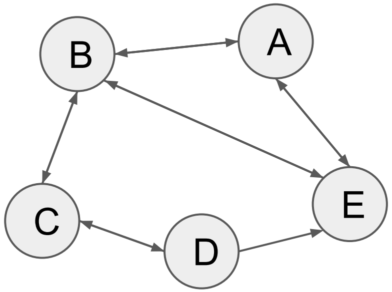
CYCLIC VS. ACYCLIC

- Applies to directed graphs
- *cyclic* - there's at least one path from a node back to itself
- *acyclic* - there are no paths such that no node can be traversed back to itself

Acyclic Graph



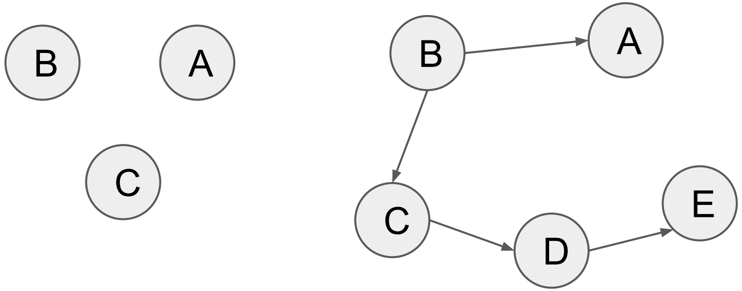
Cyclic Graph



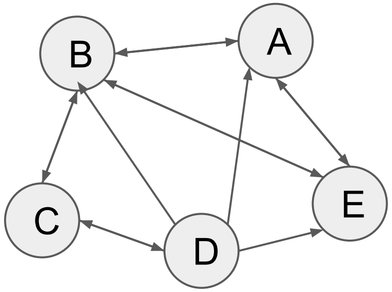
DENSE VS. SPARSE

- A graph can be sparse/dense or anything in between
- *Dense* - contains close to the maximum edges possible
- *Sparse* - contains close to the minimum edges possible

Sparse



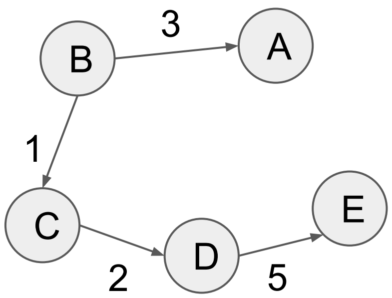
Dense



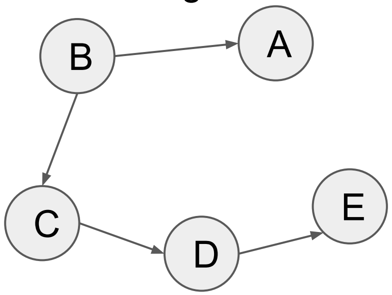
WEIGHTED VS. UNWEIGHTED

- A graph can either be weighted or unweighted
- Weight determines a value associated with an edge (usually a cost)
- *Weighted* - Each edge has an associated value
- Un*weighted* - Each edge has no associated value

Weighted



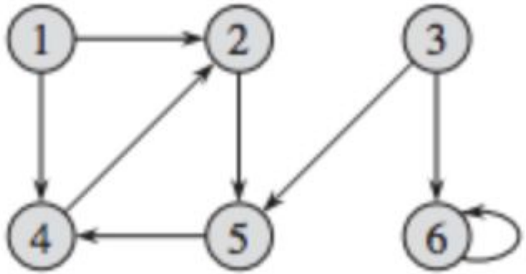
Unweighted



Representing Graphs

ADJACENCY LIST

- Use a dictionary with sets to represent the edges of a particular vertex to other neighboring vertices
- *adjacencyList[i]* is a set of all the edges to its neighbors for vertex *i*




```
{
    1: {2, 4},
    2: {5},
    3: {6, 5},
    4: {2},
    5: {4},
    6: {6}
}
```

Adjacency List Demo

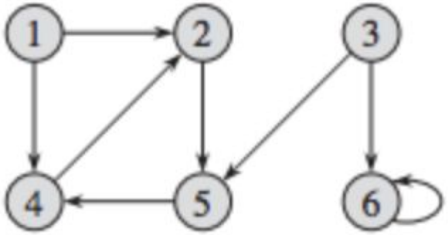
ADJACENCY LIST RUNTIME/SPACE COMPLEXITIES

- Space: O(vertices^2)
  - Imagine a dense graph
- Add vertex: O(1)
- Remove vertex: O(vertices)
- Add edge: O(1)
- Remove edge: O(1)
- Find edge: O(1)
- Get all edges: O(1)

```
{
  1: {2, 4},
  2: {5},
  3: {6, 5},
  4: {2},
  5: {4},
  6: {6}
}
```

ADJACENCY MATRIX

- Use a matrix to represent whether or not there exists an edge between two vertices
- *matrix[i][j]* is True if there exists an edge from vertex *i* to vertex *j*

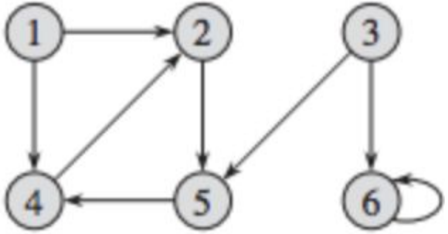


	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Adjacency Matrix Demo

ADJACENCY MATRIX RUNTIME/SPACE COMPLEXITIES

- Space: O(vertices^2)
  - Even in a sparse graph, but good for dense graphs b/c lists are space efficient
- Add vertex: O(vertices^2)
- Remove vertex: O(vertices^2)
- Add edge: O(1)
- Remove edge: O(1)
- Find edge: O(1)
- Get all edges: O(vertices)



$$\begin{array}{c|cccccc|} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 1 & 0 \\ 3 & 0 & 0 & 0 & 0 & 1 & 1 \\ 4 & 0 & 1 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 1 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 & 0 & 1 \\ \hline \end{array}$$

ADJACENCY MATRIX VS. ADJACENCY LISTS

	Space	Add Vertices	Remove Vertices	Add Edge	Remove Edge	Find Edge	Get All Edges
Adj. Matrix	$O(V^2)$	$O(V^2)$	$O(V^2)$	O(1)	O(1)	O(1)	O(V)
Adj. List	$O(V^2)$	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)

$$
\begin{array}{c|cccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & 0 & 1 & 0 & 1 & 0 & 0 \\
2 & 0 & 0 & 0 & 0 & 1 & 0 \\
3 & 0 & 0 & 0 & 0 & 1 & 1 \\
4 & 0 & 1 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 0 & 1 & 0 & 0 \\
6 & 0 & 0 & 0 & 0 & 0 & 1 \\
\hline
\end{array}
$$

```
{
  1: {2, 4},
  2: {5},
  3: {6, 5},
  4: {2},
  5: {4},
  6: {6}
}
```

ADJACENCY MATRIX VS. ADJACENCY LISTS

- The best representation mainly depends on whether or not the graph is sparse/dense and what you're optimizing for (space/runtime)
- Representing dense graphs are probably better with adjacency matrix because lists are very space efficient in comparison to dictionaries/sets
- You'll probably deal with more adjacency lists

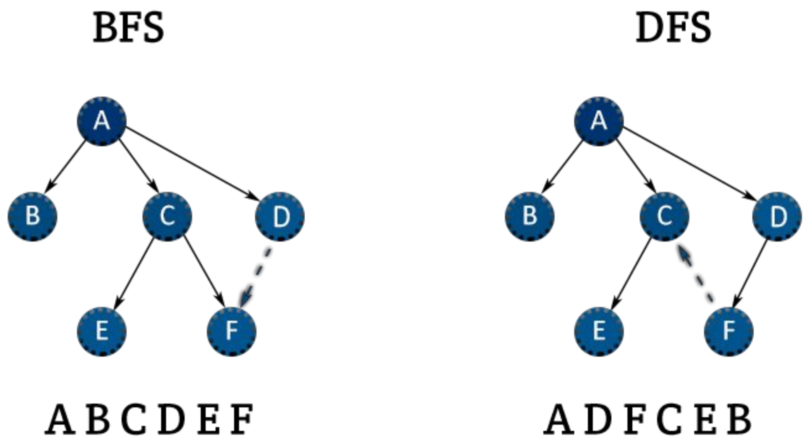
	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

```
{
    1: {2, 4},
    2: {5},
    3: {6, 5},
    4: {2},
    5: {4},
    6: {6}
}
```

Graph Traversals

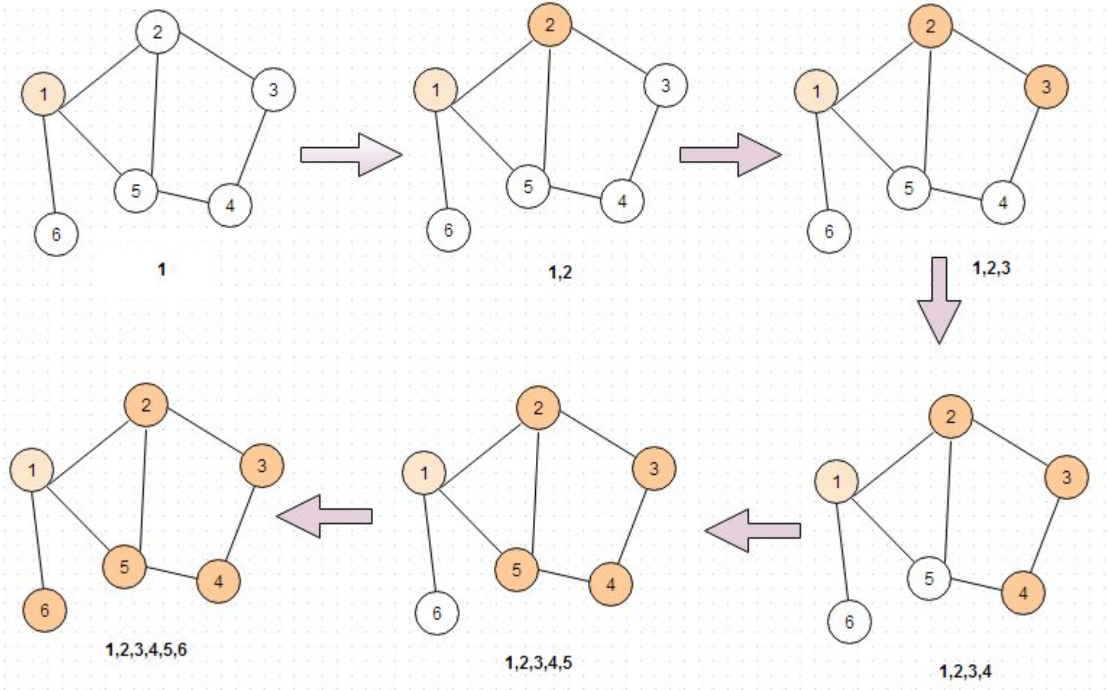
GRAPH TRAVERSALS

- There are two primary ways to traverse a graph: *Depth-first* and *Breadth-first*
- Traversal vs. Search
  - In a search, you stop once you find the node you're searching for
  - In a traversal, you traverse the entire graph



DEPTH-FIRST TRAVERSAL

- Traverse the graph in a depth-ward motion using a stack/recursion



DEPTH-FIRST TRAVERSAL ITERATIVE PSEUDOCODE

```
procedure DFS_iterative(G, v) is
    let S be a stack
    S.push(v)
    while S is not empty do
        v = S.pop()
        if v is not labeled as discovered then
            label v as discovered
            for all edges from v to w in G.adjacentEdges(v) do
                S.push(w)
```

Note: The function name should be DFT_iterative

DEPTH-FIRST TRAVERSAL RECURSIVE PSEUDOCODE

```
procedure DFS(G, v) is
    label v as discovered
    for all directed edges from v to w that are in G.adjacentEdges(v) do
        if vertex w is not labeled as discovered then
            recursively call DFS(G, w)
```

Note: The function name should be DFT_recursive

Depth-First Demo

BREADTH-FIRST

- Traverse the graph in a breadth-ward motion using a queue
- Very useful for finding **shortest path** from node to node

BREADTH-FIRST SEARCH PSEUDOCODE

```
procedure BFS(G, root) is
    let Q be a queue
    label root as discovered
    Q.enqueue(root)
    while Q is not empty do
        v := Q.dequeue()
        if v is the goal then
            return v
        for all edges from v to w in G.adjacentEdges(v) do
            if w is not labeled as discovered then
                label w as discovered
                w.parent := v
                Q.enqueue(w)
```

Breadth-First Demo