# Graph Algorithms

An introduction to graph algorithms,
with applications in bioinformatics

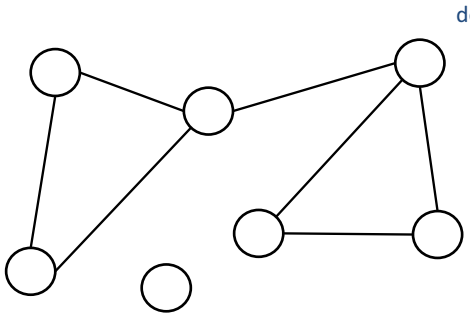
## Today's Lecture

- What's a graph?
- Why are they important in bioinformatics?
- The basics of computing on graphs
- A "top hits" selection of algorithms
- Where can I learn more?

# What's a graph?

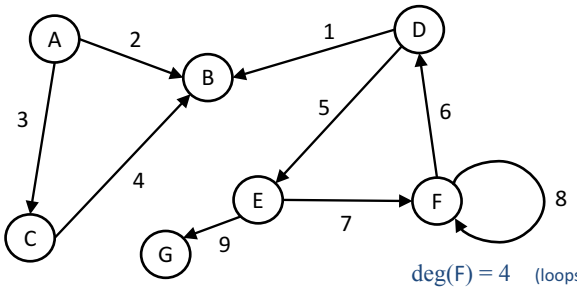
A graph is a set of vertices and edges.

$$G = (V, E)$$



degree = deg ($v$)
= # of incident edges

# A connected, labeled, directed, weighted graph

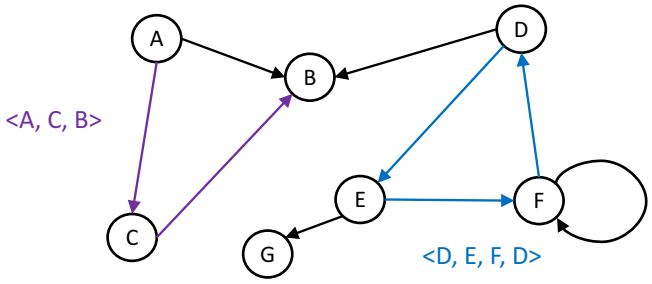



deg(F) = 4 (loops count twice)

Both edge and vertices can be labeled

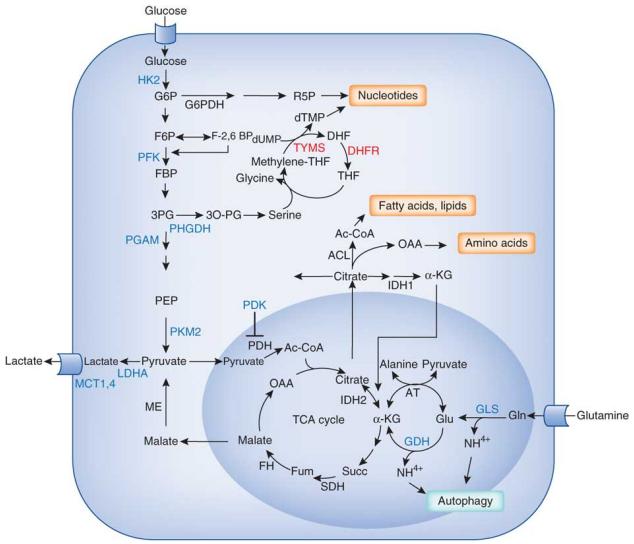
Labels often represent important values (weights, costs, distances)

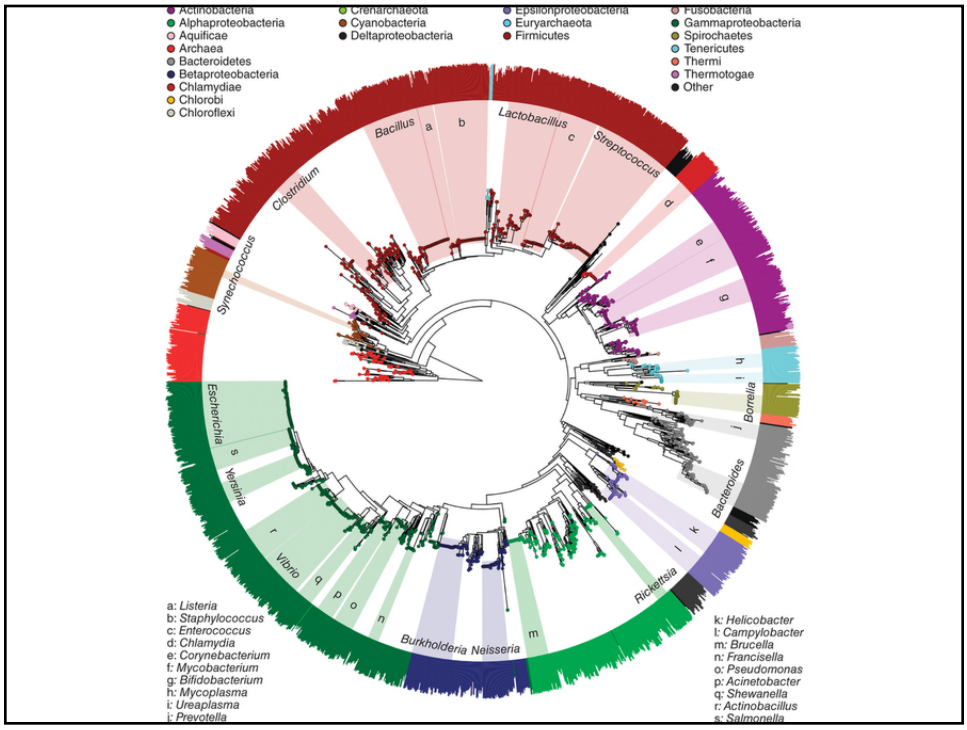
path = a sequence of edges connecting distinct vertices

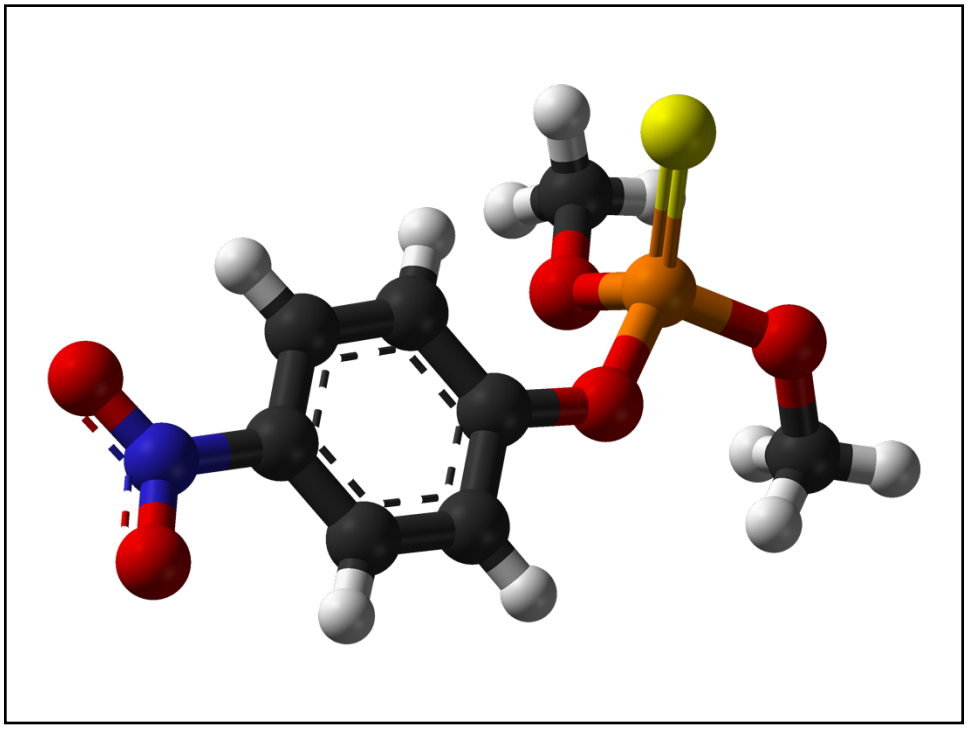


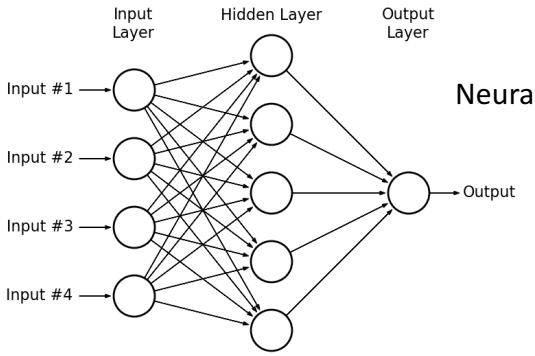

cycle = a path where the start and end vertices are the same

# Graphs in Bioinformatics



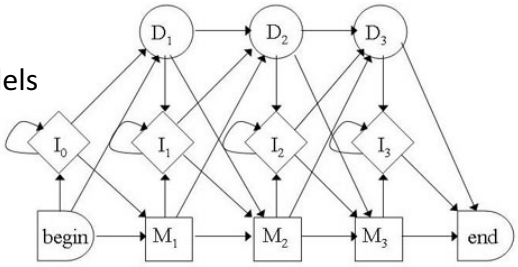

Actinobacteria
Alphaproteobacteria
Aquificae
Archaea
Bacteroidetes
Betaproteobacteria
Chlamydiae
Chlorobi
Chloroflexi
Crenarchaeota
Cyanobacteria
Deltaproteobacteria
Epsilonproteobacteria
Euryarchaeota
Firmicutes
Fusobacteria
Gammaproteobacteria
Spirochaetes
Tenericutes
Thermi
Thermotogae
Other
Bacillus
Lactobacillus
Streptococcus
Clostridium
Synechococcus
Escherichia
Yersinia
Vibrio
Burkholderia Neisseria
Rickettsia
Bacteroides
Borrelia
a: Listeria
b: Staphylococcus
c: Enterococcus
d: Chlamydia
e: Corynebacterium
f: Mycobacterium
g: Bifidobacterium
h: Mycoplasma
i: Ureaplasma
j: Prevotella
k: Helicobacter
l: Campylobacter
m: Brucella
n: Francisella
o: Pseudomonas
p: Acinetobacter
q: Shewanella
r: Actinobacillus
s: Salmonella

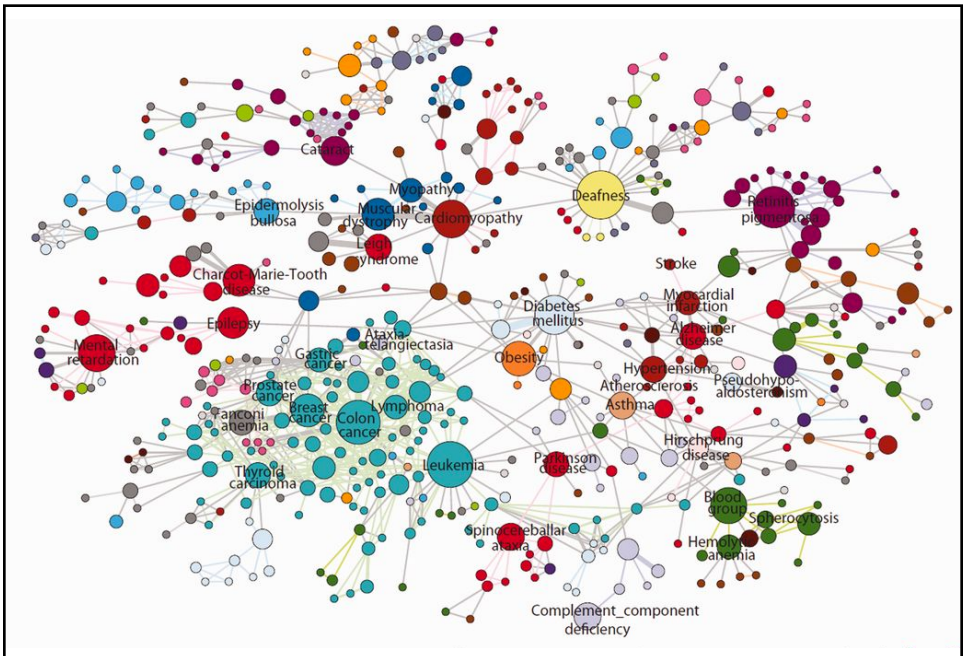




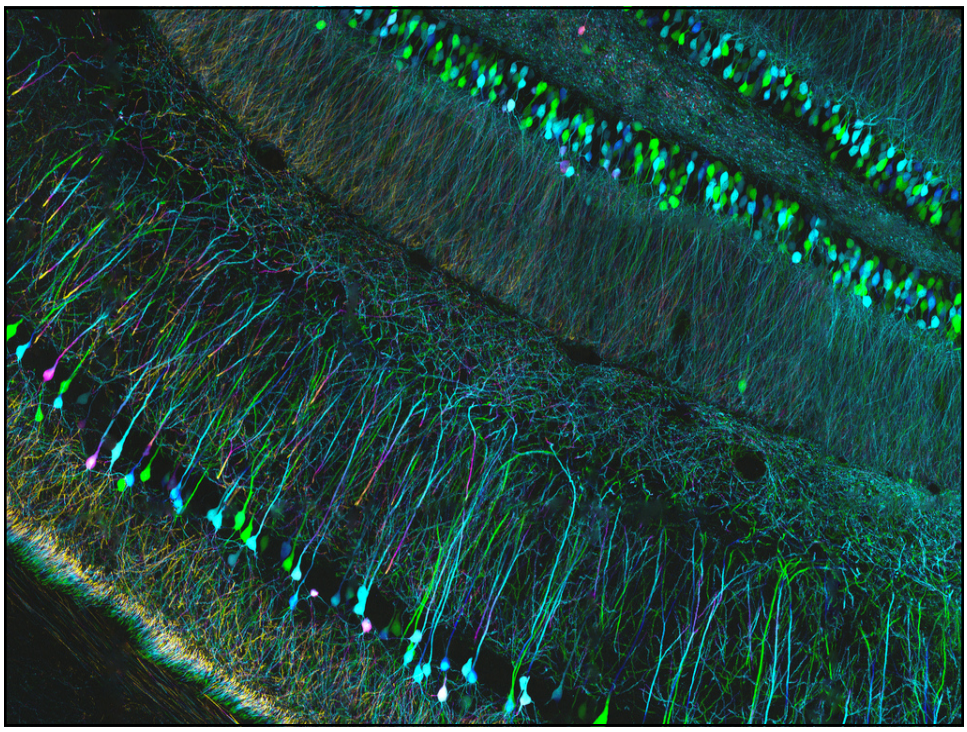

Neural networks

Hidden Markov Models

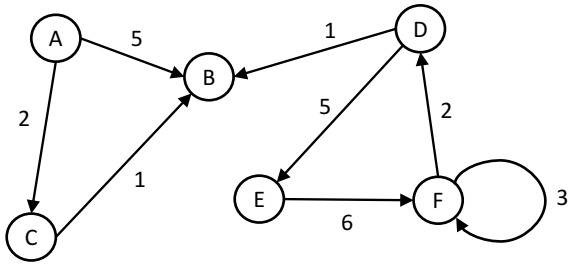



Goh & Choi, *Briefings in Functional Genomics*, Vol 2, No 6, 533-542



# Computing with graphs




| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| A | | 5 | 2 | | | |
| B | | | | | | |
| C | | 1 | | | | |
| D | | 1 | | | 5 | |
| E | | | | | | 6 |
| F | | | | 2 | | 3 |

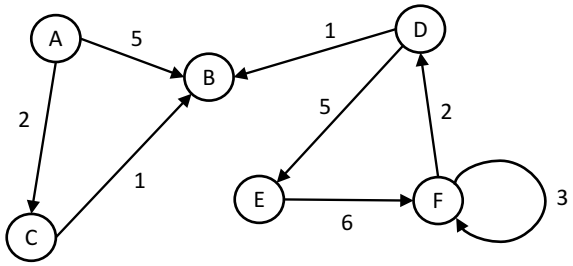
**Adjacency Matrix**

Listing all vertices adjacent to $v$: $\Theta(V)$

To determine if any two vertices $u,v$ are connected: $\Theta(1)$

Space required: $\Theta(V^2)$

# Computing with graphs




```
A: [B(5), C(2)]
B: []
C: [B(1)]
D: [B(1), E(5)]
E: [F(6)]
F: [D(2), F(3)]
```

**Adjacency Lists**

Listing all vertices adjacent to $v$: $\Theta(deg(v))$

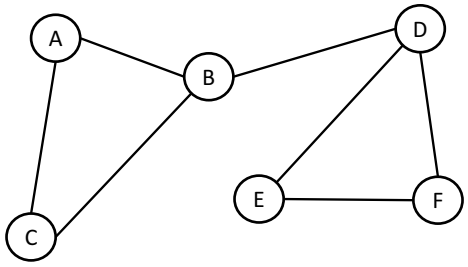
To determine if any two vertices $u,v$ are connected: $\Theta(deg(u))$

Space required: $\Theta(V + E)$

# Breadth-first Search

```
Breadth_First_Search(G, start):

  for vertex in G:
    vertex.visited = false

  Q = new empty queue
  Q.enqueue(start)

  while Q is not empty:
    v = Q.dequeue()
    for n adjacent to v:
      if !n.visited:
        # do some work
        n.visited = true
        Q.enqueue(n)
```

starting at A, what's the traversal order?

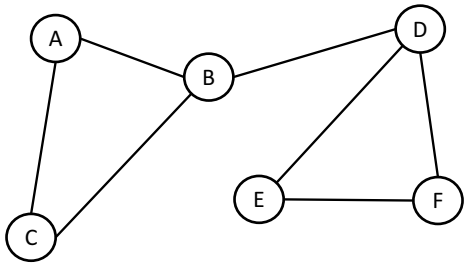



```
A: [B, C]
B: [A, C, D]
C: [A, B]
D: [B, E, F]
E: [D, F]
F: [D, E]
```

$\Theta(|V|+|E|)$

# Depth-first Search

```
Depth_First_Search(G, v):

  v.visited = true
  # do some work

  for n adjacent to v:
    if !n.visited:
      Depth_First_Search(G, n)
```



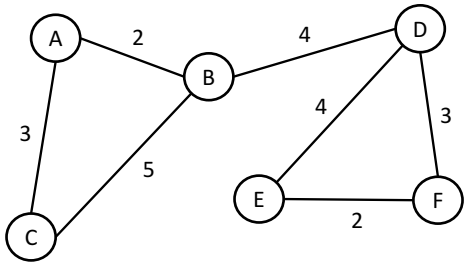

```
A: [B, C]
B: [A, C, D]
C: [A, B]
D: [B, E, F]
E: [D, F]
F: [D, E]
```

starting at A, traversal order is?

$\Theta(|V|+|E|)$

# Shortest Path (Dijkstra's Algorithm)

```
Shortest_Path(G, start):

  Q = []

  for vertex v in G:
    v.dist = Infinity
    v.prev = NULL
    Q.append(v)

  start.dist = 0

  while Q is not empty:
    v = v in Q where v.dist is smallest
    Q.remove(v)

    for each neighbor u of v:
      x = v.dist + edge_len(v, u)
      if x < u.dist:
        u.dist = x
        u.prev = v
```




```
A: [B(2), C(3)]
B: [A(2), C(5), D(4)]
C: [A(3), B(5)]
D: [B(4), E(4), F(3)]
E: [D(4), F(2)]
F: [D(3), E(2)]
```

starting at A, traversal order is?

$\Theta(|V|^2)$ in the simplest implementations

# Hamiltonians

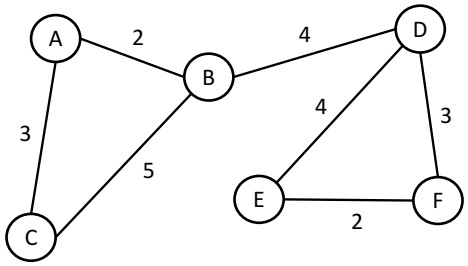
A Hamiltonian path visits each vertex exactly once

Finding a Hamiltonian path in an arbitrary graph is NP-complete

Finding a Hamiltonian path of minimum cost is known as the Traveling Salesman problem

Was shown to be solvable in linear time using a "DNA computer", but it requires a factorial number of DNA molecules.

Adleman (1994) *Science* 266(5187):1021




```
A: [B(2), C(3)]
B: [A(2), C(5), D(4)]
C: [A(3), B(5)]
D: [B(4), E(4), F(3)]
E: [D(4), F(2)]
F: [D(3), E(2)]
```

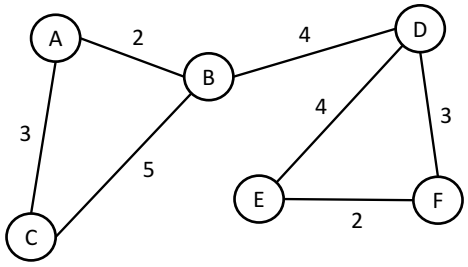
# Eulerian Paths

A Eularian path visits each edge exactly once

A Eularian cycle requires that all nodes in the graph have even degree

Finding a Eularian path/cycle can be done in linear time

What are the Eularian paths in the graph?




```
A: [B(2), C(3)]
B: [A(2), C(5), D(4)]
C: [A(3), B(5)]
D: [B(4), E(4), F(3)]
E: [D(4), F(2)]
F: [D(3), E(2)]
```

# Application to DNA sequencing

To assemble the short reads into a longer sequence, we want to find the shortest string of nucleotides that contains all of the reads

"Shortest Superstring" problem

NP-complete

# Application to DNA sequencing

Given a set of strings, $s_1, s_2, \ldots, s_n$ (the short reads), define $l(s_i, s_j)$ to be the length of the longest prefix of $s_j$ that matches a suffix of $s_i$

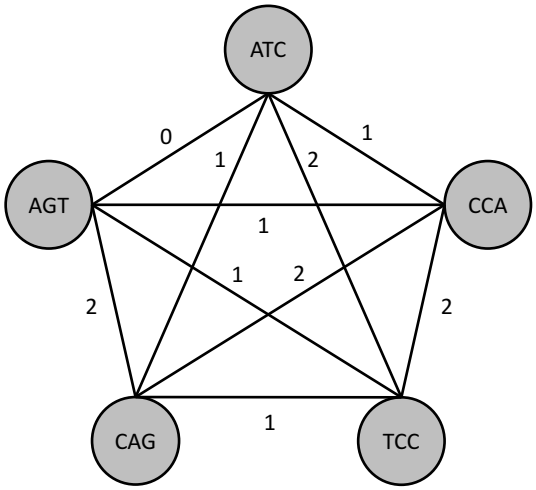
$s_i$ GGCATTACGATACAGGCTACGCACGGG

CGCACGGGGTACAGATCGGCAATCAGCA $s_j$

$l(s_i, s_j) = 8$

We can construct a graph with $n$ vertices, representing the strings $s_1, s_2, \ldots, s_n$ , and with edges between vertices $s_i$ and $s_j$ have weight $l(s_i, s_j)$

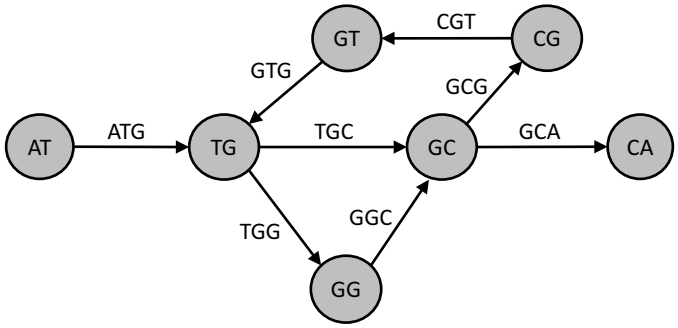
S = { ATC, CCA, CAG, TCC, AGT }




S = { ATG, TGC, GTG, GGC, GCA, GCG, CGT, TGG }

Let each vertex represent all of the $l$-1 mers of the strings

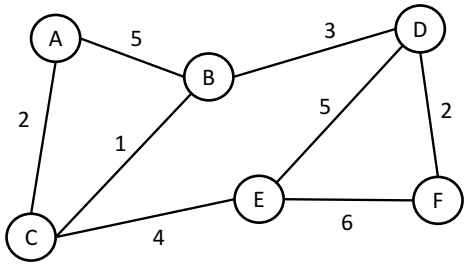
Let each edge represent an $l$- mer from S




What sequences do the Euler paths provide?

For more detail, see Pevzner, Tang, Waterman (2001) *PNAS* 98(17):9748

# Flow




```
A: [B(5), C(2)]
B: [A(5), C(1)]
C: [A(2), B(1), E(4)]
D: [B(3), E(5), F(2)]
E: [C(4), D(5), F(6)]
F: [E(6), D(2)]
```

Edge weights indicate a flow "capacity"

In the case of a metabolic network, these could be bidirectional, and weighted asymmetrically

How do we find the maximum flow from a source vertex to a sink vertex?

# Finding Max Flow

Given graph $G(V,E)$, let $c(u,v)$ = the **capacity** between vertices $u$ and $v$,
and $f(u,v)$ = the **flow** between vertices $u$ and $v$

Note that $f(u,v) \leq c(u,v)$

We define the **residual capacity** to be $c_f(u,v) = c(u,v) - f(u,v)$

We can now make a **residual graph**, $G_f = (V, E_f)$, where

$E_f$ is the set of edges such that

if $f(u,v) < c(u,v)$, there is a forward edge from $u$ to $v$, with capacity $c_f(u,v)$
if $f(u,v) > 0$, there is a backward edge from $u$ to $v$, with capacity $f(u,v)$

We define an **augmenting path** to be a set of vertices $\{v_1, v_2, \ldots, v_k\}$, such that

$v_1$ = the source vertex
$v_k$ = the sink vertex
$c_f(v_i, v_{i+1}) > 0 \quad \forall\, i$

# Ford-Fulkerson Algorithm

```
Max_Flow(G, source, sink):

  flow = 0
  F = [[0 for j in range(n)] for i in range(n)]

  while TRUE:
    c, path = Find_Path(G, source, sink, F)
    if c = 0:
      break
    flow = flow + c

    v = sink
    while v != source:
      u = path[v]
      F[u,v] += c
      F[v,u] -= c
      v = u

  return flow, F
```

finds an augmenting path with capacity c

walk the path and update flows

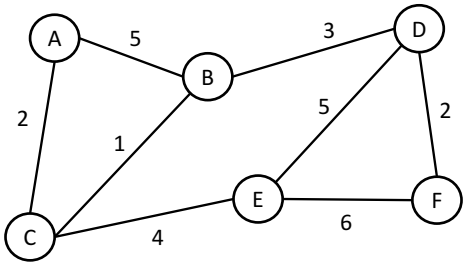
How do we find an augmenting path? BFS or DFS!

Edmunds-Karp $\Theta(|V|\,|E|^2)$

# Minimum Spanning Trees

A **spanning tree** is a subgraph of G such that includes every vertex of G and is a **tree**

A **tree** is a graph such that there is only one edge between any two vertices




What are the spanning trees of this graph?
(Hint: BFS and DFS both create them)

The **minimum spanning tree** is spanning tree with the lowest sum of edge weights

# Minimum Spanning Trees

Minimum spanning trees are often used in analyzing biological networks

How do we find the minimum spanning tree?

We start with a vertex
Then we add edges one at a time, making sure that each edge
- does not create a cycle
- the resulting subgraph is part of a minimum spanning tree

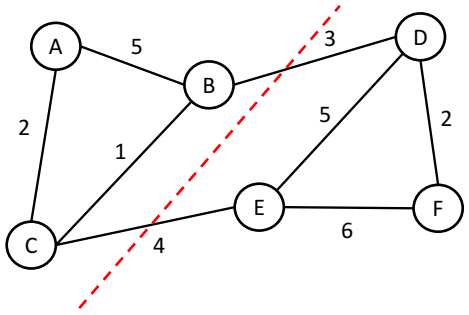
How do we know when an edge is safe to add?

# Cuts

A **cut** is a **partition** of the graph

A cut **respects** a set of edges if no edges from the set cross the cut

A **light edge** is an edge of minimum weight that does cross the cut

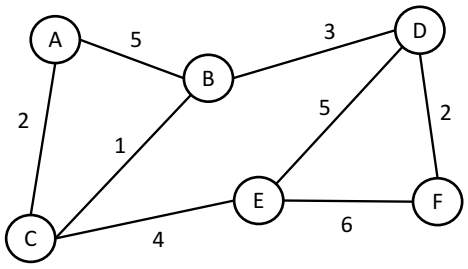



A light edge should be safe to add to our minimum spanning tree, as it doesn't create a cycle, and has a minimum weight

# Prim's Algorithm

First, we choose a vertex to be the root of the tree

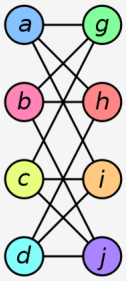
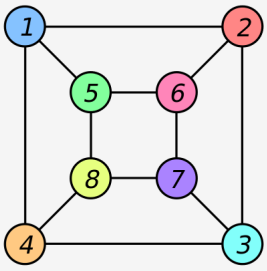
While the tree does not contain all of the vertices in the graph
- Find the lowest cost edge leaving the tree
- Add this edge to the tree




$\Theta(|V|+|E| \log|V|)$

# Subgraph Isomorphism

Two graphs are **isomorphic** if a 1-to-1 correspondence between their vertex sets exists that preserve adjacencies

| Graph G | Graph H | An isomorphism between G and H |
|---|---|---|
|  |  | $f(a) = 1$<br>$f(b) = 6$<br>$f(c) = 8$<br>$f(d) = 3$<br>$f(g) = 5$<br>$f(h) = 2$<br>$f(i) = 4$<br>$f(j) = 7$ |

# Subgraph Isomorphism

Given two graphs, $G$ and $H$, determine whether there is a subset of $G$ (a subgraph) that is isomorphic to $H$.

Lots of applications in bioinformatics:

- comparing metabolic networks in different organisms
- comparing protein and gene interaction networks
- identifying pharmacophores in chemical compounds

Unfortunately, the subgraph isomorphism problem has been proved to be NP-complete

# Ullman Algorithm

We assign vertices in the graph $H$ to the vertices in $G$, checking to make sure that each assignment has the requisite edges

This is done via a depth-first search

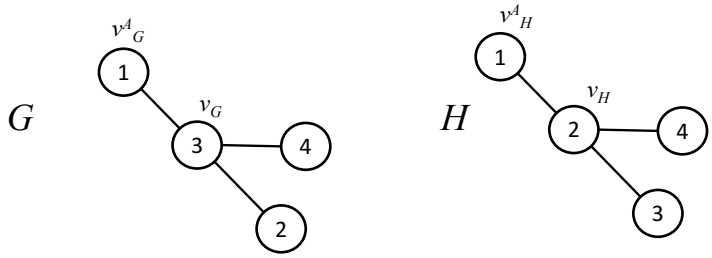
```
search(graph, subgraph, assignments):

if every edge between assigned vertices in the subgraph is not also an edge in the graph:
  return False

if all subgraph vertices have been assigned:
  return True

else:
  for all possible assignments for the next vertex of the subgraph:
    if search(graph, subgraph, possible_assignment):
      return True
```

# Ullman Algorithm

The clever part is where we can prune the possible assignments

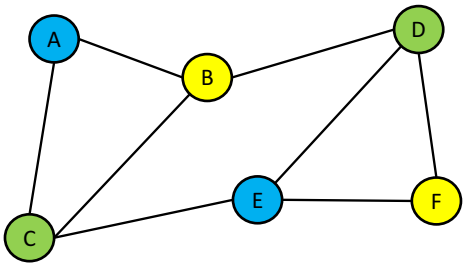
If a vertex of $G$, $v_G$, corresponds to a vertex of $H$, $v_H$, then for each adjacent vertex of $v_G$ in $G$, denoted $v^A{}_G$, there must be a vertex in $H$, $v^A{}_H$, such that

$v^A{}_H$ is adjacent to $v_H$ in $H$
and
$v^A{}_H$ corresponds to $v^A{}_G$




# Graph Coloring

A graph **coloring** is a labeling of vertices such that no pair of vertices connected by an edge share the same label




Determining the coloring using the smallest number of colors is NP-hard

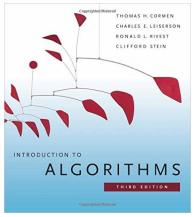
# Graph Coloring

Greedy algorithms often find low-number colorings for many graphs

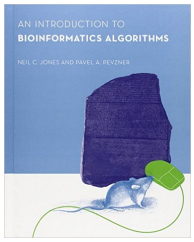
> We simply take each vertex one at a time and apply the next available color
>
> Some heuristics in the order the vertices are considered can help (eg. ordering by degree)

For a graph of maximum degree $D$, any greedy coloring will use at most $D + 1$ colors

# Graph Algorithm Resources




***Introduction to Algorithms***
Cormen, et.al.
Section VI




***An Introduction to Bioinformatics Algorithms***
Jones & Pevzner
Section 8

Wikipedia & stackoverflow.com