Graphs - used to model pairwise relations between entities.

\[ \text{Graph} (G = (V, E)) \]
\[ V : \text{vertices/nodes} \]
\[ E : \text{edges/arcs/links} \ (u, v) \text{ where } u, v \in V \]

"Nodes ~ cities"
"Edges ~ roads."

Undirected graph - no particular ordering of vertices of an edge.

Vertices: \( V = \{ a, b, c, d, e \} \)
Edges: \( E = \{ (a, b), (a, c), (b, c), (b, d), (b, e), (c, d), (d, e) \} \).

-or-

\( E = \{ (b, a), (c, a), (c, b), (d, b), (d, c), (e, d) \} \).

Directed graphs - pair of vertices of an edge are ordered.

"Certain roads blocked off due to snow."

Edges: \( E = \{ (a, b), (b, c), (b, d), (c, a), (c, d), (d, e), (e, b) \} \).
adjacency (undirected): u adjacent to v if (u,v) or (v,u) ∈ E

adjacency (directed): " " " " " " (v,u) ↗ (u)

path - sequence of vertices v₁, v₂, ..., vₙ such that an edge exists for every adjacent pair in the sequence.

ex: P: c → a → b → c → d → e ≡ (c,a), (a,b), (b,c), (c,d), (d,e)

path length - number of edges in path.

ex: |P| = 5

distance of vertices u, v = length of shortest path from u to v.

ex: dist of c, b = 2

(cycle - path of n vertices v₁, ..., vₙ where vₙ = v₁.)

weighted graph - edges have weights/cost.

w uv = weight of edge (u,v).

\[ \begin{array}{c c c c c}
\text{a} & 10 & \text{b} \\
\text{c} & 4 & 5 & \text{e} \\
\text{d} & 2 & 2 & \text{f} \\
\end{array} \]

"amount of time it takes to traverse the edge"

can also have weighted undirected graphs.

edge (b, a) ∈ E is a directed edge from b to a.
Path length (weighted) - sum of weights on edges of the path.
(distance ~ still length of shortest path)

Weighted distance from c to b?

not \( c - a - b = 3 + 10 \)
but \( c - d - e - b = 2 + 2 + 1 = 5 \)

Notice distance from b to c: 4 or 5.

How to represent/implement graphs?

Pairwise, so natural to use a matrix.

\[
A = \begin{pmatrix}
10 & - & - & - & - & - \\
- & 4 & 5 & - & - & - \\
3 & - & - & 2 & - & - \\
- & - & - & 2 & - & - \\
1 & - & - & - & - & - \\
- & - & - & - & - & 1
\end{pmatrix}
\]

A of size \( |V| \times |V| \).

Problem? Space \( O(|V|^2) \).

Better for when graph is dense.

Dense graph - \( |E| \approx O(|V|^2) \).
Typically graphs are sparse - \( |E| \approx O(|V|) \), so we use adjacency lists.

Adjacency list - for each vertex, store a list of adjacent vertices (and weights).
BFS(G, s) // Searches G starting at s

1. For all v, v.color = white
2. s.color = gray
3. Q.enqueue(s)
4. while (Q not empty):
   5.   v = Q.dequeue()
   6.   for each neighbor u of v:
       7.     if u.color == white
       8.       u.color = gray
       9.       Q.enqueue(u)
\[ a \rightarrow [b(10)] \]
\[ b \rightarrow [c(4), d(5)] \quad \text{Space: } O(V + E) \]
\[ c \rightarrow [a(3), d(2)] \]
\[ d \rightarrow [e(2)] \quad \text{Storing an entry for every node} \]
\[ e \rightarrow [b(1)] \quad \text{Storing every edge} \]
\[ f \rightarrow [ ] \]

Searching a graph

BFS: start at source, search 1 hop away, 2 hops away...

\[ \text{color nodes white - not discovered} \]
\[ \text{gray - discovered} \]

(Pseudocode on slides/next page)

Try on above example:

\[ \text{BFS } (G, s) \]

Q:\[ 8 \quad x \quad e \quad b \quad e \quad h \quad f \]

8 \[ x \]
\[ e \quad b \quad e \quad h \quad f \]
Run Time? \( IV, IE \)

\[
\begin{align*}
(s,a) & \quad (c,e) & \quad (e,h) \\
(s,c) & \quad (c,h) & \\
\uparrow & \quad \uparrow & \quad \uparrow \\
\mathbf{s} & \quad a & \quad c & \quad b & \quad e & \quad h & \quad f \\
\downarrow & \quad & \quad & \quad & \quad & \quad & \quad (o, h)
\end{align*}
\]

Notice every edge appears once
every node considered constant number of times.

\[
\Rightarrow \text{Total} = O(IV + IE)
\]

How to use BFS to find shortest paths in unweighted graphs:

Keep distance variable with each node.
Initially: \( s, dist = 0 \)
\( v, dist = \infty \) (for all other verts).

Dequeue a node
when checking neighbors, if \( dist = \infty \) (hasn't been discovered yet), set \( dist = \) dequeued node's \( dist + 1 \).
BFS-SP (G, s) // Finds shortest paths from s to all
// other nodes in unweighted graph G:

1. For all v, v.dist = ∞
2. s.dist = 0
3. Q.enqueue (s)
4. while (Q not empty)
5. v = Q.dequeue()
6. for each neighbor u of v
7. if (u.dist = ∞)
8. u.dist = v.dist + 1
9. Q.enqueue (u)

Q: s a d b e f

How to modify this to work for weighted graphs?

Lots of changes, so let's see what changes
Specifically:

Find shortest paths from source s to all other vertices in G, where G is:
- directed
- weighted (positive weights) \( w_{u,v} > 0 \) for all \( (u,v) \in E \).

Differences bw unweighted and weighted:

\( \begin{array}{c|c}
\text{unweighted} & \text{weighted} \\
\hline
\begin{array}{c}
\text{unweighted graph}
\end{array} & \begin{array}{c}
\text{weighted graph}
\end{array}
\end{array} \)

Once we update a distance, what is possible to have true about that node?

\( \Rightarrow \) We will never re-update \( v \)'s dist is set to 5
\( v \)'s " " " " " " 6
\( x \)'s " " " " " " So we should update \( x \).

(Traversing an edge in the future cannot yield shorter path to \( u \).)
**Difference #1**

When to update $u \text{ dist}$?

For vertex $x$, with neighbor $u$: $x \rightarrow u$

- If $u\.dist > x\.dist + w_{x,u}$
  
  $u\.dist = x\.dist + w_{x,u}$

**Another difference?**

3. **unweighted**

   ![Graph](image)

   5
   |
   V
   |
   1
   |
   X \neq 6

4. **weighted**

   ![Graph](image)

   V \rightarrow X

   100
   |
   X
   |
   7

(V) is much closer to

Always process first node in the queue w/c

next node will be 1 hop farther than previous

(or 0 hops farther)

**Difference (3)**

At each stage, process the node with the minimum distance.

**Turns out (will prove later)** when $u\.dist$ is minimum distance of all non-processed nodes $u\.dist$ is shortest distance from $5$ to $u$. 