Graph Implementation

Pairwise, so natural to use adjacency matrix

\[ A = \text{matrix of size } |V| \times |V| \]

\[
\begin{array}{cccccc}
  & a & b & c & d & e & f \\
 a & - & 10 & - & - & - & - \\
b & - & - & 4 & 5 & - & - \\
c & 3 & - & - & 2 & - & - \\
d & - & - & - & - & 2 & - \\
e & - & 1 & - & - & - & - \\
f & - & - & - & - & - & - \\
\end{array}
\]

Problem? Space required is \( O(|V|^2) \) but matrix is mostly empty.

⇒ Matrices better for dense graphs

\[ |E| \approx O(|V|^2) \]

Typically, graphs are sparse - \( |E| \approx O(|V|) \), so we use adjacency list - for each vertex, store a list of adjacent vertices (and edge weights).
a [b(10)]
b [c(4), d(5)]
c [a(3), d(2)]
d [e(2)]
e [b(1)]
f []

Space \( O(1V + |E|) \)

Storing an entry for every node and every edge

Searching a graph

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

(Pseudocode on slides/next page)

Try for example:

- not visited
- visited

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

Q: \([s \ e \ c \ b \ e \ h \ f]\)

\(v = \{s, a, c, b, e, h, f\}\)
BFS Run-Time in terms of |V|, |E|

\[(s, a) \quad (c, e) \quad (e, h)\]
\[(s, c) \quad (c, h) \quad (e, f)\]

\[s \quad a \quad c \quad b \quad e \quad h \quad f\]
\[(a, b)\]

Notice: every edge considered once
'' node '' constant number of times

Total: \(O(|V| + |E|)\)

How to use BFS to find shortest paths from a source \(s\) to all other nodes in an unweighted graph?

\((\Rightarrow)\) Keep distance variable with every node. Initially: \(s\).dist = 0
\(v\).dist = \(\infty\) (for all other vertices \(v\)).

• Dequeue a node, \(v\)
• When checking neighbors \(u\), if \(u\).dist = \(\infty\)
  (\(u\) hasn't been discovered yet), set \(u\).dist = \(v\).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 d c v e W f

How to modify this for weighted graphs?
Specifically:

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

Differences b/w unweighted and weighted?

For unweighted:

```
5  
1 1
```

```
\( u \neq 6 \)
```

When we update a node's distance, will that distance ever change?

No!

For any node \( u \), when \( u.d\text{dist} \) is updated, \( u.d\text{dist} \) is the final (best) distance to \( u \).

Traversing an edge in the future will not improve distances to \( v, u, x \).

Terminology:
- A node \( u \) is processed if \( u.d\text{dist} \) is the final distance to \( u \).
- When \( u.d\text{dist} \) is updated, \( u \) is processed.
For weighted?

```
  10
  +---+ 5
  |   |  \
  |   |   
  +---+ 2
      X
```

When we update a node's distance, the node may not be processed.

\( u.\text{dist} \) can be improved to 9.

So we should update \( u \) again!

**Difference #1**

*unweighted*

\[
\text{if} \ (u.\text{dist} - v) = 0
\]

*unweighted*

\[
\text{if} \ (u.\text{dist} - v) = 0
\]

When to update \( u.\text{dist} \)?

\[
\text{For vertex } v \text{ with neighbor } u \quad (v) \rightarrow (u)
\]

If \( u.\text{dist} > v.\text{dist} + w_{v,u} \),

\[
u.\text{dist} = v.\text{dist} + w_{v,u}
\]

Another difference?

First, notice for unweighted, we process nodes in a specific order: Ascending order of distance. Why does this work?
Want to process a node that is closer to s before one that is farther since the closer node may yield a shorter path to the farther one. 

True for weighted too!

For unweighted, since edge weights are 1, we visit nodes exactly in this order (1 hop away, 2 hops away, 3 hops away, ...).

Since we visit nodes in the same order they should be processed, we can use a queue to get the next node to process.
For weighted:

\[ \begin{array}{c}
\text{u visited before y, but we should process y first.}
\end{array} \]

\[ \text{Why?} \]

Can't use a queue anymore!

How to get node with minimum current distance?

Use Min Binary Heap!

**Difference (2):** Use Min Binary Heap to get the node with minimum distance.

In general,

\[ \begin{array}{c}
\text{Do not want to process u before y (even though we visit u before y)}
\end{array} \]

Will prove later - when a node is deleted from heap it is processed (dist is set to final/best).