# CS 341: ALGORITHMS

**Lecture 10: graph algorithms I**

Readings: see website

Trevor Brown
https://student.cs.uwaterloo.ca/~cs341
trevor.brown@uwaterloo.ca

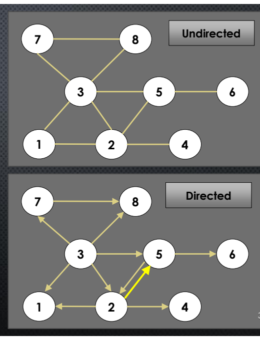


## GRAPHS



## GRAPHS

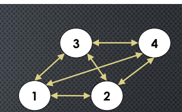
- A graph is a pair $G = (V, E)$
- $V$ contains **vertices**
- $E$ contains **edges**
  - An edge $uv$ connects two **distinct** vertices $u, v$
  - Also denoted $(u, v)$
- Graphs can be **undirected**
- … or **directed**
  - meaning $(u, v) \neq (v, u)$

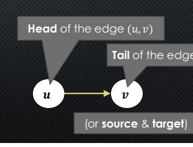





## PROPERTIES OF GRAPHS

- Number of vertices $n = |V|$
- Number of edges $m = |E| \leq n(n-1)$
  - Note $m$ is in $O(n^2)$ but **not necessarily** $\Omega(n^2)$
  - For undirected graphs, $m \leq \frac{n(n-1)}{2}$
    - (Asymptotically, no different)
- Other common terminology:
  - **vertices = nodes** edges = arcs






## A FEW MORE TERMS

- The **indegree** of a node $u$, denoted $\text{indeg}(u)$, is the number of edges **directed into** $u$
- The **outdegree**, denoted $\text{outdeg}(u)$, is the number of edges **directed out from** $u$
  - or simply $\deg(u)$ in an undirected graph
- The **neighbours** of $u$ are the nodes $u$ points to
  - Also called the **nodes adjacent to** $u$, denoted $\text{adj}(u)$


$indeg(u) = 1$
$outdeg(u) = 2$
$adj(u) = \{1,5\}$

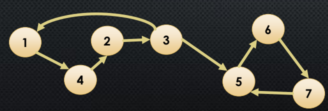


## DATA STRUCTURES FOR GRAPHS

- Two main representations
  - **Adjacency matrix**
  - **Adjacency list**
- Each has pros & cons

## ADJACENCY MATRIX REPRESENTATION

- $n \times n$ matrix $A = (a_{uv})$
  - rows & columns indexed by $V$
- $a_{uv} = 1$ if $(u, v)$ **is an edge**
- $a_{uv} = 0$ if $(u, v)$ is a **non-edge**
- Diagonal = 0 (no self edges)




Matrix $A$

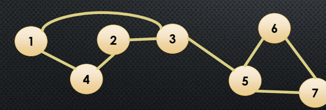
tail / head

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 2 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 3 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 4 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 7 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

7

## ADJACENCY MATRIX REPRESENTATION

- **For undirected graphs**
- $a_{uv} = 1$ if $(u, v)$ **or** $(v, u)$ is an edge
- Matrix is symmetric $A^T = A$




tail / head

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 2 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 3 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| 4 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 1 | 0 | 0 | 1 | 1 |
| 6 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| 7 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |

8

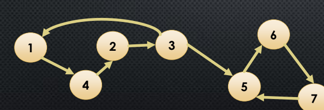
## IMPLEMENTING AN ADJACENCY MATRIX

- Suppose we are loading a graph from input
  - Assume nodes are labeled 0..n-1
  - 2D array **bool adj[n][n]**
- What if nodes are not labeled 0..n-1?
  - Rename them in a preprocessing step
- What if you don't have 2D arrays?
  - Transform 2D array index into 1D index
  - adj[u][v] → adj[u*n + v]
    (can simplify with macros in C)

9

## ADJACENCY LIST REPRESENTATION

- $n$ linked lists, one for each node
- We write $adj[u]$ to denote the list for node $u$
- $adj[u]$ contains the labels of nodes it has edges to



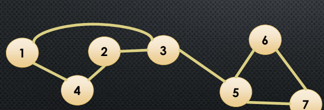

head / tail

- 1 → 4
- 2 → 3
- 3 → 1 → 5
- 4 → 2
- 5 → 6
- 6 → 7
- 7 → 5

10

## ADJACENCY LIST REPRESENTATION

- **For undirected graphs**
- If $adj[u]$ contains $v$ then $adj[v]$ also contains $u$




head / tail

- 1 → 3 → 4
- 2 → 3 → 4
- 3 → 1 → 2 → 5
- 4 → 1 → 2
- 5 → 3 → 6 → 7
- 6 → 5 → 7
- 7 → 5 → 6

11

## IMPLEMENTING ADJACENCY LISTS

- Suppose we are loading a graph from input
  - Assume nodes are labeled 0..n-1
  - Array of lists adj[n]
  - (In C++, something like an array of vector<int> would work)

12

# PROS AND CONS

| | Adjacency matrix | Adjacency list |
|---|---|---|
| Time to test whether $(u, v)$ is an edge | $O(1)$ | $O(outdeg(u))$ |
| Time to list neighbours of $u$ | $O(n)$ | $O(outdeg(u))$ |
| Space complexity | $O(n^2)$ | $O(n+m)$ |

Excellent when nodes have $O(1)$ **neighbours**

Can be better for dense graphs

Better if $o(n^2)$ edges

We call this a **sparse** graph

13

# BREADTH FIRST SEARCH

A simple introduction to graph algorithms

14

```
BreadthFirstSearch(V[1..n], adj[1..n], s)
    pred[1..n] = [null, null, ..., null]
    dist[1..n] = [infty, infty, ..., infty]
    colour[1..n] = [white, white, ..., white]
    q = new queue

    colour[s] = gray
    dist[s] = 0
    q.enqueue(s)

    while q is not empty
        u = q.dequeue()
        for v in adj[u]
            if colour[v] = white
                pred[v] = u
                colour[v] = gray
                dist[v] = dist[u] + 1
                q.enqueue(v)
        colour[u] = black

    return colour, pred, dist
```

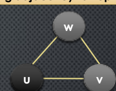
**Assuming adjacency list representation**

**Discover (enqueue)** starting node s

**Start processing** node $u$'s edges

**Discover** (enqueue) neighbour $v$

**Finish processing** $u$

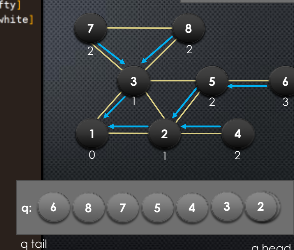



- **Undiscovered** nodes are **white**
- **Discovered** nodes are **gray**
  - **Processing** adjacent edges
- **Finished** nodes are **black**
  - Adjacent nodes have been **processed**
- **Connected graph:** each node is eventually black

15


```
BreadthFirstSearch(V[1..n], adj[1..n], s)
    pred[1..n] = [null, null, ..., null]
    dist[1..n] = [infty, infty, ..., infty]
    colour[1..n] = [white, white, ..., white]
    q = new queue

    colour[s] = gray
    dist[s] = 0
    q.enqueue(s)

    while q is not empty
        u = q.dequeue()
        for v in adj[u]
            if colour[v] = white
                pred[v] = u
                colour[v] = gray
                dist[v] = dist[u] + 1
                q.enqueue(v)
        colour[u] = black

    return colour, pred, dist
```


Example execution starting at node 1




16


```
BreadthFirstSearch(V[1..n], adj[1..n], s)
    pred[1..n] = [null, null, ..., null]
    dist[1..n] = [infty, infty, ..., infty]
    colour[1..n] = [white, white, ..., white]
    q = new queue

    colour[s] = gray
    dist[s] = 0
    q.enqueue(s)

    while q is not empty
        u = q.dequeue()
        for v in adj[u]
            if colour[v] = white
                pred[v] = u
                colour[v] = gray
                dist[v] = dist[u] + 1
                q.enqueue(v)
        colour[u] = black

    return colour, pred, dist
```


# COMPLEXITY

$O(n)$ (with adjacency lists)

$O(1)$

$O(1)$

$O(n)$ iterations

$O(1)$

$O(|adj[u]|)$ iterations

$O(1)$

$O(1)$

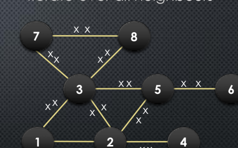
- **Naïve loop analysis:**
  - $O(n)$ iterations * $O(|adj[u]|)$ iterations
  - $|adj[u]| \leq n$, so $O(n^2)$

17


```
BreadthFirstSearch(V[1..n], adj[1..n], s)
    pred[1..n] = [null, null, ..., null]
    dist[1..n] = [infty, infty, ..., infty]
    colour[1..n] = [white, white, ..., white]
    q = new queue

    colour[s] = gray
    dist[s] = 0
    q.enqueue(s)

    while q is not empty
        u = q.dequeue()
        for v in adj[u]
            if colour[v] = white
                pred[v] = u
                colour[v] = gray
                dist[v] = dist[u] + 1
                q.enqueue(v)
        colour[u] = black

    return colour, pred, dist
```

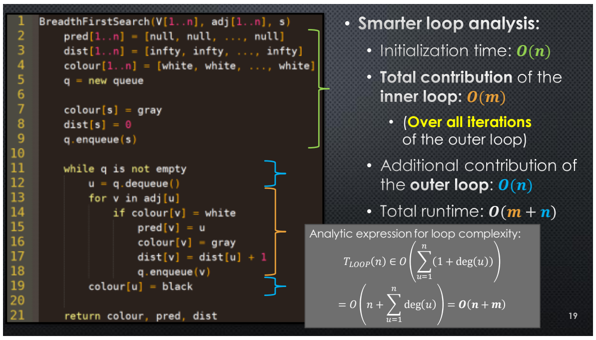

**Smarter loop analysis:**

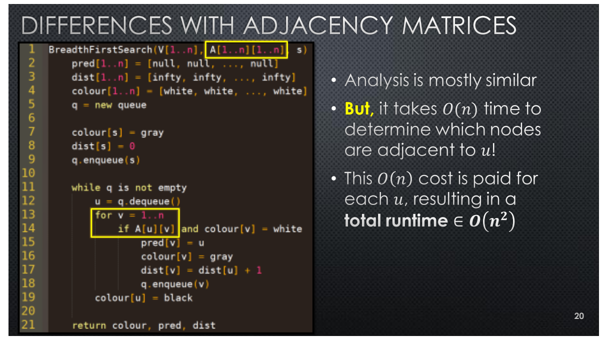
- For each $u$, iterate over all neighbours

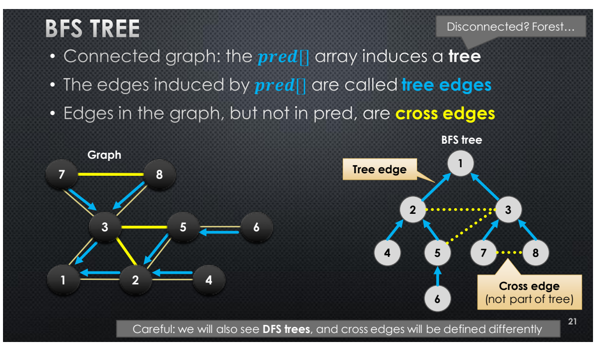


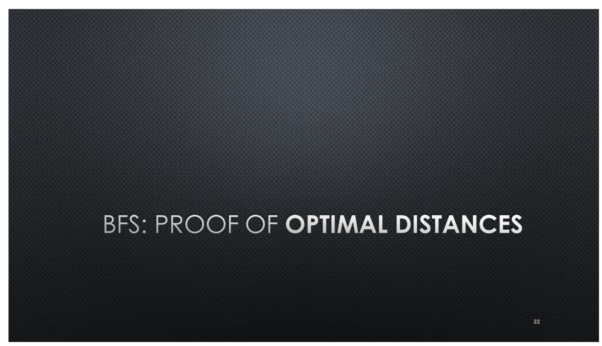

- We touch each edge twice (doing $O(1)$ work each time)
- **Total contribution** of the inner loop to the runtime: $O(m)$

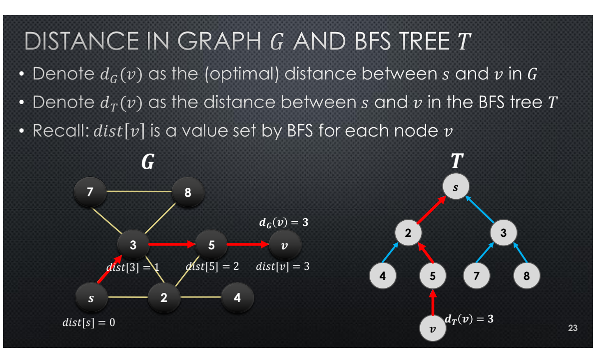
18

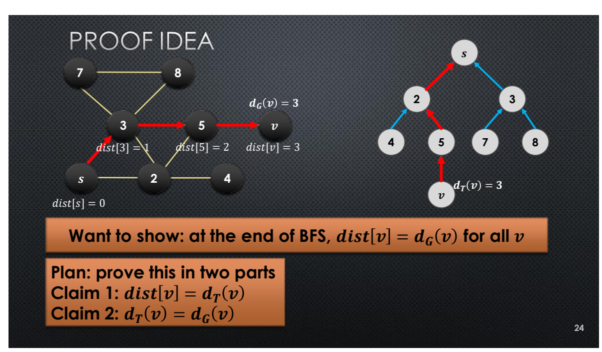

BreadthFirstSearch(V[1..n], adj[1..n], s)
pred[1..n] = [null, null, ..., null]
dist[1..n] = [infty, infty, ..., infty]
colour[1..n] = [white, white, ..., white]
q = new queue
colour[s] = gray
dist[s] = 0
q.enqueue(s)
while q is not empty
u = q.dequeue()
for v in adj[u]
if colour[v] = white
pred[v] = u
colour[v] = gray
dist[v] = dist[u] + 1
q.enqueue(v)
colour[u] = black
return colour, pred, dist
Smarter loop analysis:
Initialization time: $O(n)$
Total contribution of the inner loop: $O(m)$
(Over all iterations of the outer loop)
Additional contribution of the outer loop: $O(n)$
Total runtime: $O(m+n)$
Analytic expression for loop complexity:
$T_{LOOP}(n) \in O\left(\sum_{u=1}^{n}(1+\deg(u))\right)$
$= O\left(n + \sum_{u=1}^{n}\deg(u)\right) = O(n+m)$
19


DIFFERENCES WITH ADJACENCY MATRICES
BreadthFirstSearch(V[1..n], A[1..n][1..n], s)
pred[1..n] = [null, null, ..., null]
dist[1..n] = [infty, infty, ..., infty]
colour[1..n] = [white, white, ..., white]
q = new queue
colour[s] = gray
dist[s] = 0
q.enqueue(s)
while q is not empty
u = q.dequeue()
for v = 1..n
if A[u][v] and colour[v] = white
pred[v] = u
colour[v] = gray
dist[v] = dist[u] + 1
q.enqueue(v)
colour[u] = black
return colour, pred, dist
Analysis is mostly similar
But, it takes $O(n)$ time to determine which nodes are adjacent to $u$!
This $O(n)$ cost is paid for each $u$, resulting in a total runtime $\in O(n^2)$
20


BFS TREE
Disconnected? Forest…
Connected graph: the pred[] array induces a tree
The edges induced by pred[] are called tree edges
Edges in the graph, but not in pred, are cross edges
Graph
BFS tree
Tree edge
Cross edge (not part of tree)
Careful: we will also see DFS trees, and cross edges will be defined differently
21


BFS: PROOF OF OPTIMAL DISTANCES
22


DISTANCE IN GRAPH $G$ AND BFS TREE $T$
Denote $d_G(v)$ as the (optimal) distance between $s$ and $v$ in $G$
Denote $d_T(v)$ as the distance between $s$ and $v$ in the BFS tree $T$
Recall: $dist[v]$ is a value set by BFS for each node $v$
$G$
$T$
$d_G(v) = 3$
$dist[3] = 1$
$dist[5] = 2$
$dist[v] = 3$
$dist[s] = 0$
$d_T(v) = 3$
23


PROOF IDEA
$d_G(v) = 3$
$dist[3] = 1$
$dist[5] = 2$
$dist[v] = 3$
$dist[s] = 0$
$d_T(v) = 3$
Want to show: at the end of BFS, $dist[v] = d_G(v)$ for all $v$
Plan: prove this in two parts
Claim 1: $dist[v] = d_T(v)$
Claim 2: $d_T(v) = d_G(v)$
24

## SKETCH OF **CLAIM 1**: $dist[v] = d_T(v), \forall v \in V$

```
BreadthFirstSearch(V[1..n], adj[1..n], s)
    pred[1..n] = [null, null, ..., null]
    dist[1..n] = [infty, infty, ..., infty]
    colour[1..n] = [white, white, ..., white]
    q = new queue

    colour[s] = gray
    dist[s] = 0
    q.enqueue(s)

    while q is not empty
        u = q.dequeue()
        for v in adj[u]
            if colour[v] = white
                pred[v] = u
                colour[v] = gray
                dist[v] = dist[u] + 1
                q.enqueue(v)
        colour[u] = black

    return colour, pred, dist
```

**Key observation**: whenever we set $dist[v] \leftarrow dist[u] + 1$, $u$ is the parent of $v$ in the BFS tree.

Based on this observation, a simple inductive proof shows $\boldsymbol{dist[v] = d_T(v)}$

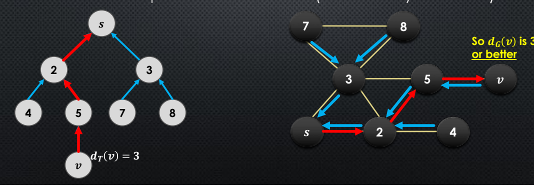
(for example, by strong induction on the nodes in the order their $dist$ values are set---left as an exercise)

25

## SKETCH OF **CLAIM 2**: $d_T(v) = d_G(v)$

- **Part 1:** $\forall v, \boldsymbol{d_G(v) \leq d_T(v)}$
  - There is a unique path $v \rightarrow \cdots \rightarrow s$ in $T$
  - And $T$ is a **subgraph of $G$**
  - So that same path also exists in $G$ (technically reversed)

To prove = we show ≤ and ≥

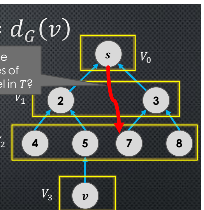



26

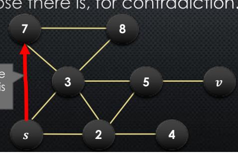
## SKETCH OF **CLAIM 2**: $d_T(v) = d_G(v)$

- **Part 2:** $\forall v, \boldsymbol{d_G(v) \geq d_T(v)}$
  - Partition $T$ into **levels** $V_i = \{v : d_T(v) = i\}$ by distance from $s$
  - **Claim:** there is **no "forward" edge in $G$** that "skips" a level from $V_i$ to $V_j, j \geq i+2$
  - Suppose there is, for contradiction…

What are the consequences of "skipping" a level in $T$?




That "skip" edge in $T$ looks like this in $G$



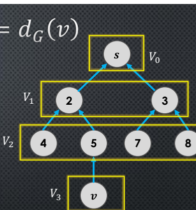

But that edge in $G$ would cause 7 to have $s$ as its parent, so $dist[7]$ would be **only 1 greater** than its parent…

Contradicts(!) the assumption that the edge points to a node with **greater distance by at least 2**

27

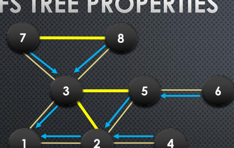
## SKETCH OF **CLAIM 2**: $d_T(v) = d_G(v)$

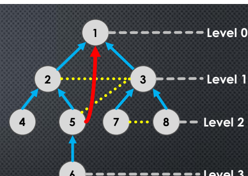
- **Part 2:** $\forall v, \boldsymbol{d_G(v) \geq d_T(v)}$
  - We've just argued that there is **no "forward" edge in $G$** that "skips" a level in $T$ from $V_i$ to $V_j, j \geq i+2$
  - Since no edge in $G$ "skips" a level in $T$, we know **at least one edge in $G$** is needed to traverse **each level** between $\boldsymbol{s \in V_0}$ and $\boldsymbol{v \in V_{d_T(v)}}$
  - There are $d_{T(v)}$ such levels, so $d_G(v) \geq d_T(v)$




28

## BFS TREE PROPERTIES




**Fact: there are no "back" edges in <u>undirected</u> graphs that "skip" a level going <u>up</u> in the BFS tree.**

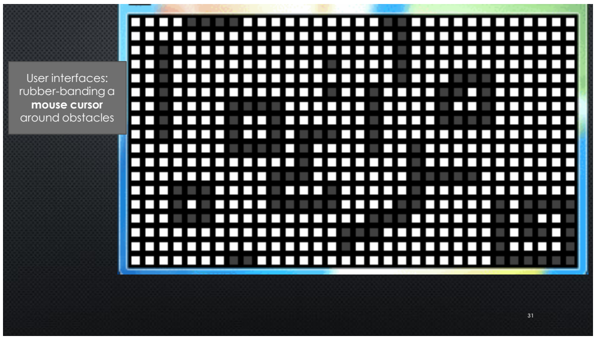
Exercise: what about directed graphs?

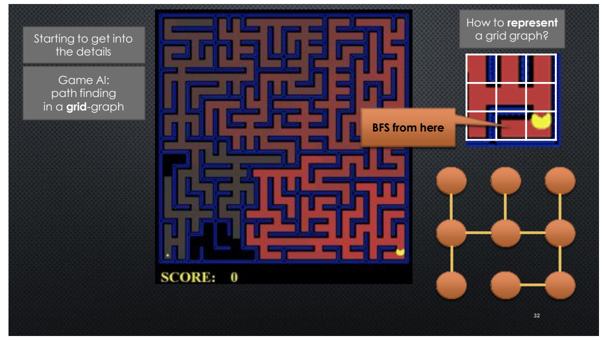
Answer in bonus slides…

29

APPLICATION:
FINDING **SHORTEST PATHS**

30

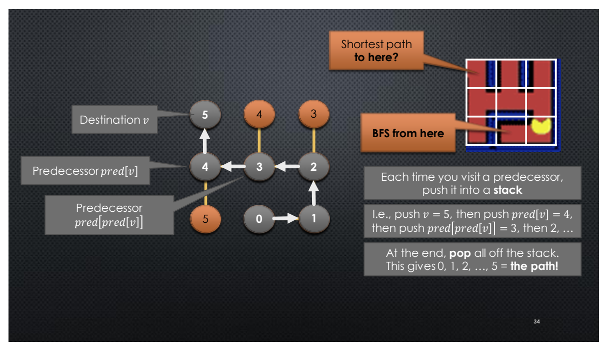


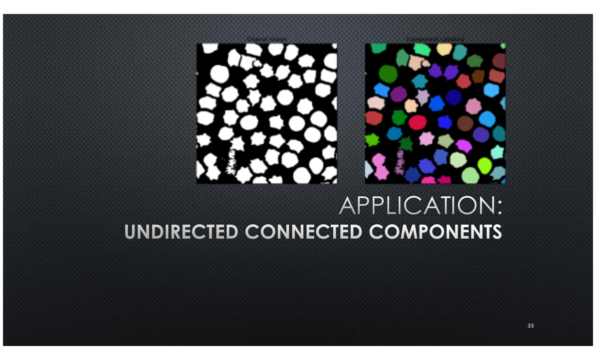



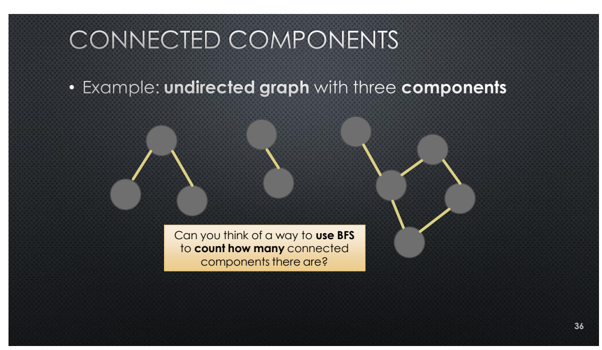

## HOW TO **OUTPUT** AN **ACTUAL PATH**

- Suppose you want to output a **path** from $s$ to $v$ with minimum distance (not just the **distance** to $v$)
- Algorithm (what do you think?)
  - Similar to extracting an answer from a DP array!
  - Work backwards through the predecessors
  - Note: this will print the path **in reverse**! Solution?

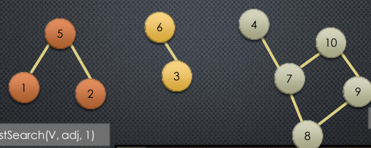
33



## CONNECTED COMPONENTS



BreadthFirstSearch(V, adj, 1)

BreadthFirstSearch(V, adj, 3)

BreadthFirstSearch(V, adj, 4)

Modified BFS that (1) reuses the same colour array for consecutive calls and (2) sets comp[u] = compNum for each node u it visits

Can be done in $O(n + m)$ time

Complexity?

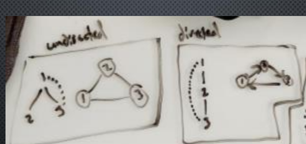
```
UndirectedConnectedComponents(adj[1..n])
    colour[1..n] = [white, ..., white]
    comp[1..n] = [0, ..., 0]
    compNum = 1
    for start = 1..n
        if colour[start] is white
            BFS(adj, start, colour, comp, compNum)
            compNum = compNum + 1
    return comp
```

37

## BONUS SLIDES

38

## ANSWER TO BFS TREE PROPERTY EXERCISE…



Bfs tree graph Bfs tree graph

Dotted = back edge

39