SEARCHING IN GRAPHS

We already did the most basic form of search: (neighbor query)
SEARCHING IN GRAPHS

We already did the most basic form of search: (neighbor query)

In general, given a vertex \( s \), we want to locate some vertex \( t \), find a path in \( G \).
SEARCHING IN GRAPHS

We already did the most basic form of search: (neighbor query)

In general, given a vertex \( s \),

we want to locate some vertex \( t \), find a path in \( G \)

or we want to visit all vertices,
in a "local" organized manner.
BREADTH FIRST SEARCH (BFS) - The polite way to search
BREADTH FIRST SEARCH (BFS)

Start by checking if $t$ is a neighbor of $s$. Look one step away from $s$. 
BREADTH FIRST SEARCH (BFS)

Start by checking if \( t \) is a neighbor of \( s \).

Look one step away from \( s \).

If yes, done. If not, then check all neighbors-of-neighbors.
BREADTH FIRST SEARCH (BFS)

Start by checking if $t$ is a neighbor of $s$.

- Look one step away from $s$.

If yes, done. If not, then check all neighbors-of-neighbors

- One step from each.

Either done, or repeat (dig deeper)
**BREADTH FIRST SEARCH (BFS)**

Start by checking if \( t \) is a neighbor of \( s \).

→ look one step away from \( s \).

If yes, done. If not, then check all neighbors-of-neighbors

→ one step from each.

Either done, or repeat (dig deeper) ... only on unexplored neighbors!
BREADTH FIRST SEARCH (BFS)

Start by checking if \( t \) is a neighbor of \( s \).

\[ \rightarrow \text{look one step away from } s. \]

If yes, done. If not, then check \( \text{all } \) neighbors-of-neighbors \( \rightarrow \text{one step from each.} \)

Either done, or repeat (dig deeper) ... only on unexplored neighbors!
Search follows a tree pattern.
Search follows a tree pattern.

BFS extends depth by 1 at all possible nodes

always processing nodes closer to $s$ first
Search follows a tree pattern.

BFS extends depth by 1 at all possible nodes

- always processing nodes closer to $s$ first

- each node is processed only once (e.g., $u$)
If \( s \) and \( t \) are in the same connected component then the search will find \( t \).
\[ d(s,t) = 4 \]

If \( s \) and \( t \) are in the same connected component then the search will find \( t \).

Even better, BFS will find the shortest path \( s \rightarrow t \).
If $s$ and $t$ are in the same connected component then the search will find $t$.

Even better, BFS will find a shortest path $s \rightarrow t$ (prove by contradiction)
If $s$ and $t$ are in the same connected component then the search will find $t$.

Even better, BFS will find the shortest path $s \rightarrow t$. (prove by contradiction)

Time? (supposing we can tell instantly whether a vertex is "new")

d($s, t) = 4$
If \( s \& t \) are in the same connected component then the search will find \( t \).

Even better, BFS will find a \textbf{the} shortest path \( s \rightarrow t \).
(prove by contradiction)

\( \text{time?} \) (supposing we can tell instantly whether a vertex is "new")

\( O(|E|) \) (in component of \( s \))
ALGO:

1. check $\text{Adj}[s] : v_1, \ldots, v_k$

    $\leftarrow$ if $v_i = t$ DONE
Algo:

(i) check $\text{Adj}[s] : v_1 \ldots v_k$

if $v_i = t$ done

if $v_i \neq t$

mark as visited

put in queue : $Q$

Why a queue?
Algorithm:

1. mark \( s \)
2. check \( \text{Adj}[s] : v_1, \ldots, v_k \)
   - if \( v_i = t \) then done
   - mark as visited
   - if \( v_i \neq t \) put in queue : \( Q \)

\( Q : \overset{\text{in}}{v_1, v_2, v_3, v_4, v_5} \overset{\text{out}}{} \)
Algorithm:

(0) mark s

(1) check Adj[s]: v₁, ..., vₖ
   - if vᵢ = t done
   - if vᵢ ≠ t: mark as visited
   - put in queue: Q

(2) While Q not empty,
   - remove first vertex vₖ in Q
   - check Adj[vₖ]: u₀, ..., uₚ
Algo:

0) mark s
1) check Adj[s] : v₁...,vₖ
   → if vᵢ=t done
   → if vᵢ≠t put in queue : Q

2) While Q not empty,
   - remove first vertex vᵢ in Q
   - check Adj[vᵢ] : u₀...uₚ
     → if uᵢ=t done
     → if uᵢ≠t & unmarked
        put uᵢ in Q, mark uᵢ
**Algorithm:**

1. mark $s$
2. check $\text{Adj}[s]$: $v_1, ..., v_k$
   - if $v_i = t$ done
   - if $v_i \neq t$ mark as visited
     - put in queue $Q$
3. While $Q$ not empty,
   - remove first vertex $v_f$ in $Q$
   - check $\text{Adj}[v_f]$: $u_0, ..., u_p$
     - if $u_i = t$ done
     - if $u_i \neq t$ & unmarked
       - put $u_i$ in $Q$
       - mark $u_i$
Algorithm:

1. Check \( \text{Adj}[s] : v_1, \ldots, v_k \)
   - If \( v_i = t \) done
   - If \( v_i \neq t \) mark as visited
   - Put in queue: \( Q \)

2. While \( Q \) not empty,
   - Remove first vertex \( v_f \) in \( Q \)
   - Check \( \text{Adj}[v_f] : u_0, u_2, v_3, u_4 \)
     - If \( u_i = t \) done
     - If \( u_i \neq t \) & unmarked
       - Put \( u_i \) in \( Q \)
       - Mark \( u_i \)
Algorithm:

1. check $\text{Adj}[s] : v_1, \ldots, v_k$
   - if $v_i = t$ done
   - if $v_i \neq t$ mark as visited
     - put in queue $Q$

2. While $Q$ not empty,
   - remove first vertex $v_f$ in $Q$
   - check $\text{Adj}[v_f] : u_0, u_5$
     - if $u_i = t$ done
     - if $u_i \neq t$ & unmarked put $u_i$ in $Q$, mark $u_i$
Algo:

(0) mark \( s \)

(1) check Adj[\( s \)]: \( v_1 \) \( \ldots \) \( v_k \)
   if \( v_i = t \) done
   if \( v_i \neq t \)
      mark as visited
      put in queue : \( Q \)

(2) While \( Q \) not empty,
   - remove first vertex \( v_f \) in \( Q \)
   - check Adj[\( v_f \)]: \( u_1 \) \( \ldots \) \( u_p \)
      if \( u_i = t \) done
      if \( u_i \neq t \) & unmarked
         put \( u_i \) in \( Q \), mark \( u_i \)

Q: \( \overset{\text{in}}{v_1} v_2 v_3 v_4 v_5 \overset{\text{out}}{v_2 v_3 v_4 v_5 u_1 u_2 u_3 v_3 v_4 v_5 u_1 u_2 u_3 u_4 v_4 v_5 u_1 u_2 u_3 u_4 u_5 v_5 u_1 u_2 u_3 u_4 u_5 u_6} \)
Algorithm:

1. Check $\text{Adj}[s]: v_1, \ldots, v_k$
   - If $v_i = t$, done
   - If $v_i \neq t$, mark as visited
   - Put in queue $Q$

2. While $Q$ not empty,
   - Remove first vertex $v_f$ in $Q$
   - Check $\text{Adj}[v_f]: u_1, \ldots, u_p$
     - If $u_i = t$, done
     - If $u_i \neq t$ & unmarked, put $u_i$ in $Q$, mark $u_i$
Algo:

1. mark $s$
2. check $\text{Adj}[s] = v_1, ... , v_k$
   - if $v_i = t$ then done
   - else mark as visited
     - if $v_i \neq t$ then put in queue $Q$

3. While $Q$ not empty,
   - remove first vertex $v_f$ in $Q$
   - check $\text{Adj}[v_f] = u_1, ... , u_p$
     - if $u_i = t$ then done
     - else if $u_i \neq t$ and unmarked then put $u_i$ in $Q$, mark $u_i$
Algorithm:

1. Mark $s$
2. Check $\text{Adj}[s] : v_1, \ldots, v_k$
   - If $v_i = t$ done
   - Mark as visited
   - If $v_i \neq t$ put in queue: $Q$

3. While $Q$ not empty,
   - Remove first vertex $v_f$ in $Q$
   - Check $\text{Adj}[v_f] : u_1, \ldots, u_p$
     - If $u_i = t$ done
     - If $u_i \neq t$ & unmarked put $u_i$ in $Q$, mark $u_i$

$Q$: $v_1, v_2, v_3, v_4, v_5 \leftarrow \text{in}$

$Q$: $v_2, v_3, v_4, v_5$, $u_1, u_2, u_3$ \underline{remove}

$Q$: $v_3, v_4, v_5$, $u_1, u_2, u_3, u_4$

$Q$: $v_4, v_5$, $u_1, u_2, u_3, u_4, u_5$

$Q$: $v_5$, $u_1, u_2, u_3, u_4, u_5, u_6$

$Q$: $u_1, u_2, u_3, u_4, u_5, u_6$

$Q$: $u_2, u_3, u_4, u_5, u_6, x_1, x_2, x_3, x_4$

etc
(0) mark \( s \)

(1) check \( \text{Adj}[s] : v_1, ..., v_k \)

→ if \( v_i = t \) done
→ if \( v_i \neq t \)
   • mark as visited
   • put in queue \( : Q \)

(2) While \( Q \) not empty,

   - remove first vertex \( v_f \) in \( Q \)
   - check \( \text{Adj}[v_f] : u_1, ..., u_p \)
     → if \( u_i = t \) done
     → if \( u_i \neq t \) & unmarked
        put \( u_i \) in \( Q \).
        mark \( u_i \)
- mark $s$ & put in $Q$
- $\text{depth}(s) = 0$

While $Q$ not empty,

- $x = \text{dequeue}(Q)$
  - check $\text{Adj}[x]: u, ..., u_p$
    - if $u_i = t$ done
    - if $u_i \neq t$ mark as visited
      - put in queue $Q$

(0) mark $s$

(1) check $\text{Adj}[s]: v, ..., v_k$
  - if $v_i = t$ done
  - if $v_i \neq t$ mark as visited
    - put in queue $Q$

(2) While $Q$ not empty,
  - remove first vertex $v_f$ in $Q$
    - check $\text{Adj}[v_f]: u, ..., u_p$
      - if $u_i = t$ done
        - if $u_i \neq t$ & unmarked
          - put $u_i$ in $Q$
just the search process