Breadth-First Search

Version of October 11, 2014
Representations of Graphs: Adjacency List

- $V$: set of vertices, $E$: set of edges. (We will sometimes also simultaneously use $V$ to denote the number of vertices, and $E$ to denote the number of edges.)

- **Adjacency list representation**: $O(V + E)$ storage
  
  $Adj[u]$: linked list of all $v$ such that $(u, v) \in E$. 

  
  \[
  \begin{align*}
  Adj[0] &= \{1, 3, 9\} \\
  Adj[1] &= \{0, 9, 2\} \\
  \end{align*}
  \]
Representations of Graphs: Adjacency List

- \( V \): set of vertices, \( E \): set of edges. (We will sometimes also simultaneously use \( V \) to denote the number of vertices, and \( E \) to denote the number of edges.)

- **Adjacency list representation**: \( O(V + E) \) storage
  \( \text{Adj}[u] \) — linked list of all \( v \) such that \((u, v) \in E\).
    - \( \text{Adj}[0] = \{1, 3, 9\} \); \( \text{Adj}[1] = \{0, 9, 2\} \); ...

10 3 4 9 2 6 7 10 5
Representations of Graphs: Adjacency List

- $V$: set of vertices, $E$: set of edges. (We will sometimes also simultaneously use $V$ to denote the number of vertices, and $E$ to denote the number of edges.)

- **Adjacency list representation**: $O(V + E)$ storage
  $Adj[u]$ — linked list of all $v$ such that $(u, v) \in E$.
  - $Adj[0] = \{1, 3, 9\}$; $Adj[1] = \{0, 9, 2\}$; ...

- Can retrieve all the neighbors of $u$ in $O(\text{degree}(u))$ time.
Adjacency matrix representation: $O(V^2)$ storage

$A = [a_{ij}], a_{ij} = 1$ if $(v_i, v_j) \in E$;

$a_{ij} = 0$ if $(v_i, v_j) \notin E$.

For undirected graph, adjacency matrix is always symmetric.
Adjacency matrix representation: $O(V^2)$ storage

$A = [a_{ij}]$, $a_{ij} = 1$ if $(v_i, v_j) \in E$;
$a_{ij} = 0$ if $(v_i, v_j) \notin E$.

For undirected graph, adjacency matrix is always symmetric.

Can check if $u$ and $v$ are connected in $O(1)$ time.
What does Breadth-First Search (BFS) do?
What does Breadth-First Search (BFS) do?

- Traverse all vertices in graph,
What does Breadth-First Search (BFS) do?

- Traverse all vertices in graph, and thereby
- Reveal properties of the graph.

Three arrays are used to keep information gathered during traversal:

1. $color[u]$: the color of each vertex visited
   - WHITE: undiscovered
   - GRAY: discovered but not finished processing
   - BLACK: finished processing

2. $pred[u]$: the predecessor pointer pointing back to the vertex from which $u$ was discovered

3. $d[u]$: the distance from the source to vertex $u$
The Breadth-First Search (BFS) Algorithm

What does Breadth-First Search (BFS) do?

- Traverse all vertices in graph, and thereby
- Reveal properties of the graph.

Three arrays are used to keep information gathered during traversal:

1. `color[u]`: the color of each vertex visited
   - WHITE: undiscovered
   - GRAY: discovered but not finished processing
   - BLACK: finished processing

2. `pred[u]`: the predecessor pointer pointing back to the vertex from which `u` was discovered

3. `d[u]`: the distance from the source to vertex `u`
What does Breadth-First Search (BFS) do?
- Traverse all vertices in graph, and thereby
- Reveal properties of the graph.

Three arrays are used to keep information gathered during traversal
1. \( \textcolor{red}{\text{color}[u]} \): the \textcolor{red}{color} of each vertex visited
The Breadth-First Search (BFS) Algorithm

What does Breadth-First Search (BFS) do?
- Traverse all vertices in graph, and thereby
- Reveal properties of the graph.

Three arrays are used to keep information gathered during traversal

1. **color[ u ]**: the color of each vertex visited
   - WHITE: undiscovered

Version of October 11, 2014
The Breadth-First Search (BFS) Algorithm

What does Breadth-First Search (BFS) do?
- Traverse all vertices in graph, and thereby
- Reveal properties of the graph.

Three arrays are used to keep information gathered during traversal

1. **color[u]**: the color of each vertex visited
   - **WHITE**: undiscovered
   - **GRAY**: discovered but not finished processing
What does Breadth-First Search (BFS) do?
- Traverse all vertices in graph, and thereby
- Reveal properties of the graph.

Three arrays are used to keep information gathered during traversal

1. \textit{color}[u]: the \textit{color} of each vertex visited
   - WHITE: undiscovered
   - GRAY: discovered but not finished processing
   - BLACK: finished processing
The Breadth-First Search (BFS) Algorithm

What does Breadth-First Search (BFS) do?
- Traverse all vertices in graph, and thereby
- Reveal properties of the graph.

Three arrays are used to keep information gathered during traversal

1. $\text{color}[u]$: the color of each vertex visited
   - WHITE: undiscovered
   - GRAY: discovered but not finished processing
   - BLACK: finished processing

2. $\text{pred}[u]$: the predecessor pointer
   - pointing back to the vertex from which $u$ was discovered
What does Breadth-First Search (BFS) do?
- Traverse all vertices in graph, and thereby
- Reveal properties of the graph.

Three arrays are used to keep information gathered during traversal

1. **color**[\(u\)] : the **color** of each vertex visited
   - WHITE: undiscovered
   - GRAY: discovered but not finished processing
   - BLACK: finished processing

2. **pred**[\(u\)] : the **predecessor** pointer
   - pointing back to the vertex from which \(u\) was discovered

3. **d**[\(u\)] : the **distance** from the source to vertex \(u\)
BFS Algorithm

BFS(G)

// Initialize
foreach u in V do
    color[u] = WHITE; // undiscovered
    pred[u] = NULL; // no predecessor
end

time =
BFS Algorithm

BFS(G)

// Initialize
foreach u in V do
  color[u] = WHITE; // undiscovered
  pred[u] = NULL; // no predecessor
end

time = 0;

foreach u in V do
BFS Algorithm

BFS(G)

// Initialize
foreach u in V do
    color[u] = WHITE; // undiscovered
    pred[u] = NULL; // no predecessor
end

time= 0;

foreach u in V do
    // start a new tree
    if
BFS Algorithm

BFS(G)

// Initialize
foreach u in V do
    color[u] = WHITE; // undiscovered
    pred[u] = NULL; // no predecessor
end

time = 0;

foreach u in V do
    // start a new tree
    if color[u] =
BFS Algorithm

BFS(G)

// Initialize
foreach u in V do
    color[u] = WHITE; // undiscovered
    pred[u] = NULL; // no predecessor
end

time= 0;

foreach u in V do
    // start a new tree
    if color[u] = WHITE then
        BFSVisit(u);
    end
end
BFS(G)

// Initialize
foreach u in V do
    color[u] = WHITE; // undiscovered
    pred[u] = NULL; // no predecessor
end

time = 0;

foreach u in V do
    // start a new tree
    if color[u] = WHITE then
        BFSVisit(u);
    end
end
BFSVisit(s)

\[ \text{color}[s] = \text{GRAY}; \text{pred}[s] = \text{NULL}; d[s] = 0; \]
BFSVisit(s)

color[s] = GRAY; pred[s] = NULL; d[s] = 0;
BFSVisit(s)

color[s] = GRAY; pred[s] = NULL; d[s] = 0;
Q = ∅; Enqueue(Q,s);
while Q ≠ ∅ do
BFSVisit(s)

color[s] = GRAY; pred[s] = NULL; d[s] = 0;
Q = ∅; Enqueue(Q,s);
while Q ≠ ∅ do
    u = Dequeue(Q);
    foreach v ∈ Adj[u] do
        if color[v] = WHITE then
            color[v] = GRAY;
            d[v] = d[u] + 1;
            pred[v] = u;
            Enqueue(Q,v);
        end
    end
end
color[u] = BLACK;
BFSVisit(s)

color[s] = GRAY; pred[s] = NULL; d[s] = 0;
Q = ∅; Enqueue(Q,s);
while Q ≠ ∅ do
    u = Dequeue(Q);
    foreach v ∈ Adj[u] do
        if color[v] = WHITE then
            color[v] = GRAY;
            d[v] = d[u] + 1 ;
            pred[v] = u;
            Enqueue(Q,v);
    color[u] = BLACK;
end
BFSVisit(s)

color[s] = GRAY; pred[s] = NULL; d[s] = 0;
Q = ∅; Enqueue(Q,s);
while Q ≠ ∅ do
    u = Dequeue(Q);
    foreach v ∈ Adj[u] do
        if color[v] = WHITE then
            color[v] = GRAY;
            d[v] = d[u] + 1;
            pred[v] = u;
            Enqueue(Q,v);
        end
    end
    color[u] = BLACK;
end
BFSVisit(s)

color[s] = GRAY; pred[s] = NULL; d[s] = 0;
Q = ∅; Enqueue(Q,s);
while Q ≠ ∅ do
    u = Dequeue(Q);
    foreach v ∈ Adj[u] do
        if color[v] = WHITE then
            color[v] = GRAY;
            d[v] = d[u] + 1;
            pred[v] = u;
            Enqueue(Q,v);
        end
    end
    color[u] = BLACK;
end

Question
Which graph representation shall we use?
BFS Example

(a) Breadth-First Search

(b) Breadth-First Search

(c) Breadth-First Search

(d) Breadth-First Search

(e) Breadth-First Search

(f) Breadth-First Search

(g) Breadth-First Search

(h) Breadth-First Search

(i) Breadth-First Search
The outputs of BFS:

Distance array: $d[v]$

Predecessor array: $pred[v]$  

The BFS Forest:

Use $pred[v]$ to define a graph $F = (V, E_f)$ as follows:

$E_f = \{(pred[v], v) | v \in V, pred[v] \neq \text{NULL}\}$

This graph has no cycles (why?), and is therefore a forest, i.e. a collection of trees. We call it a BFS Forest.

In each tree, $d[v]$ gives the shortest distance to the initial vertex of the tree. Following $pred[v]$ gives a shortest path to the initial vertex of the tree.
The BFS Algorithm

The outputs of BFS:

1. Distance array: $d[v]$
The BFS Algorithm

The outputs of BFS:

1. Distance array: $d[v]$
2. Predecessor array $pred[v]$

The BFS Forest:

Use $pred[v]$ to define a graph $F = (V, E_f)$ as follows:

$$E_f = \{ (pred[v], v) | v \in V, pred[v] \neq NULL \}$$

This graph has no cycles (why?), and is therefore a forest, i.e. a collection of trees. We call it a BFS Forest.

In each tree, $d[v]$ gives the shortest distance to the initial vertex of the tree.

Following $pred[v]$ gives a shortest path to the initial vertex of the tree.
The BFS Algorithm

The outputs of BFS:
1. Distance array: $d[v]$
2. Predecessor array $pred[v]$

The BFS Forest:
The BFS Algorithm

The outputs of BFS:

1. Distance array: $d[v]$
2. Predecessor array $pred[v]$

The BFS Forest:

Use $pred[v]$ to define a graph $F = (V, E_f)$ as follows:

$$E_f = \{(pred[v], v) | v \in V, pred[v] \neq \text{NULL}\}$$
The BFS Algorithm

The outputs of BFS:

1. Distance array: \( d[v] \)
2. Predecessor array \( pred[v] \)

The BFS Forest:

- Use \( pred[v] \) to define a graph \( F = (V, E_f) \) as follows:

\[
E_f = \{(pred[v], v) | v \in V, pred[v] \neq NULL\}
\]

- This graph has no cycles (why?), and is therefore a forest, i.e. a collection of trees. We call it a BFS Forest.
The BFS Algorithm

The outputs of BFS:
1. Distance array: \( d[v] \)
2. Predecessor array \( pred[v] \)

The BFS Forest:
- Use \( pred[v] \) to define a graph \( F = (V, E_f) \) as follows:
  \[
  E_f = \{(pred[v], v) \mid v \in V, \, pred[v] \neq \text{NULL}\}
  \]
- This graph has no cycles (why?), and is therefore a forest, i.e. a collection of trees. We call it a BFS Forest.
- In each tree, \( d[v] \) gives the shortest distance to the initial vertex of the tree.
The BFS Algorithm

The outputs of BFS:
1. Distance array: \( d[v] \)
2. Predecessor array \( \text{pred}[v] \)

The BFS Forest:
- Use \( \text{pred}[v] \) to define a graph \( F = (V, E_f) \) as follows:

\[
E_f = \{ (\text{pred}[v], v) | v \in V, \text{pred}[v] \neq \text{NULL} \}
\]

- This graph has no cycles (why?), and is therefore a forest, i.e. a collection of trees. We call it a BFS Forest.
- In each tree, \( d[v] \) gives the shortest distance to the initial vertex of the tree.
- Following \( \text{pred}[v] \) gives a shortest path to the initial vertex of the tree.
On each vertex $u$, we spend time $T_u = O(1 + \text{degree}(u))$
Running Time of BFS

On each vertex $u$, we spend time $T_u = O(1 + \text{degree}(u))$

The total running time is

$$\sum_{u \in V} T_u \leq \sum_{u \in V} (O(1 + \text{degree}(u))) = O(V + E)$$
Running Time of BFS

On each vertex $u$, we spend time $T_u = O(1 + \text{degree}(u))$

The total running time is

$$\sum_{u \in V} T_u \leq \sum_{u \in V} (O(1 + \text{degree}(u))) = O(V + E)$$

Hence, the running of BFS on a graph with $V$ vertices and $E$ edges is $O(V + E)$
Running Time of BFS

On each vertex $u$, we spend time $T_u = O(1 + \text{degree}(u))$

The total running time is

$$\sum_{u \in V} T_u \leq \sum_{u \in V} (O(1 + \text{degree}(u))) = O(V + E)$$

Hence, the running of BFS on a graph with $V$ vertices and $E$ edges is $O(V + E)$

Applications:

1. Shortest paths in a graph
Running Time of BFS

On each vertex $u$, we spend time $T_u = O(1 + \text{degree}(u))$

The total running time is

$$\sum_{u \in V} T_u \leq \sum_{u \in V} (O(1 + \text{degree}(u))) = O(V + E)$$

Hence, the running of BFS on a graph with $V$ vertices and $E$ edges is $O(V + E)$

Applications:

1. Shortest paths in a graph
   - What if the graph is weighted?
On each vertex $u$, we spend time $T_u = O(1 + \text{degree}(u))$

The total running time is

$$\sum_{u \in V} T_u \leq \sum_{u \in V} (O(1 + \text{degree}(u))) = O(V + E)$$

Hence, the running of BFS on a graph with $V$ vertices and $E$ edges is $O(V + E)$

Applications:

1. Shortest paths in a graph
   - What if the graph is weighted?
2. Finding connected components