Breadth-First Search

Version of October 11, 2014

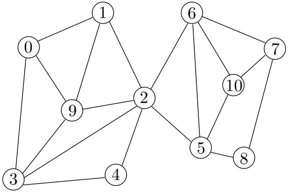
Representations of Graphs: Adjacency List

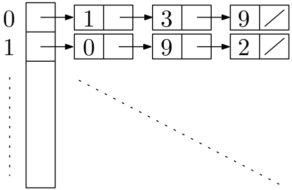
- V: set of vertices, E: set of edges. (We will sometimes also simultaneously use V to denote the number of vertices, and E to denote the number of edges.)
- Adjacency list representation: $O(V + E)$ storage
  $Adj[u]$ — linked list of all v such that $(u, v) \in E$.

- 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 $(u, v) \in E$.
  - $Adj[0] = \{1, 3, 9\}$; $Adj[1] = \{0, 9, 2\}$; ...

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.



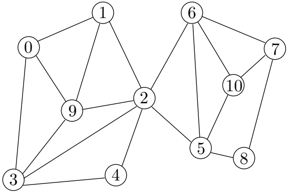


Representations of Graphs: Adjacency Matrix

- 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.

Representations of Graphs: Adjacency Matrix

- 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.



	0	1	2	3	4	5	6	7	8	9	10
0	0	1	0	1	0	0	0	0	0	1	0
1	1	0	1	0	0	0	0	0	0	1	0
2	0	1	0	1	1	1	1	0	0	1	0
3	1	0	1	0	1	0	0	0	0	1	0
4	0	0	1	1	0	0	0	0	0	0	0
5	0	0	1	0	0	0	1	0	1	0	1
6	0	0	1	0	0	1	0	1	0	0	1
7	0	0	0	0	0	0	1	0	1	0	1
8	0	0	0	0	0	1	0	1	0	0	0
9	1	1	1	1	0	0	0	0	0	0	0
10	0	0	0	0	0	1	1	1	0	0	0

The Breadth-First Search (BFS) Algorithm

What does Breadth-First Search (BFS) do?

What does Breadth-First Search (BFS) do?

- Traverse all vertices in graph,

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.

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

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

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

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

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

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

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

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(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

    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)

color[s] = ; pred[s] = NULL; d[s] = 0;

BFSVisit(s)

color[s] $= \mathrm{GRAY}$; pred[s] $= \mathrm{NULL}$; d[s] $= 0$;

BFSVisit(s)

color[s] $= \mathrm{GRAY}$; pred[s] $= \mathrm{NULL}$; d[s] $= 0$;
Q $= \emptyset$; Enqueue(Q,s);
while $Q \neq \emptyset$ **do**

BFSVisit(s)

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

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);
```

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
```

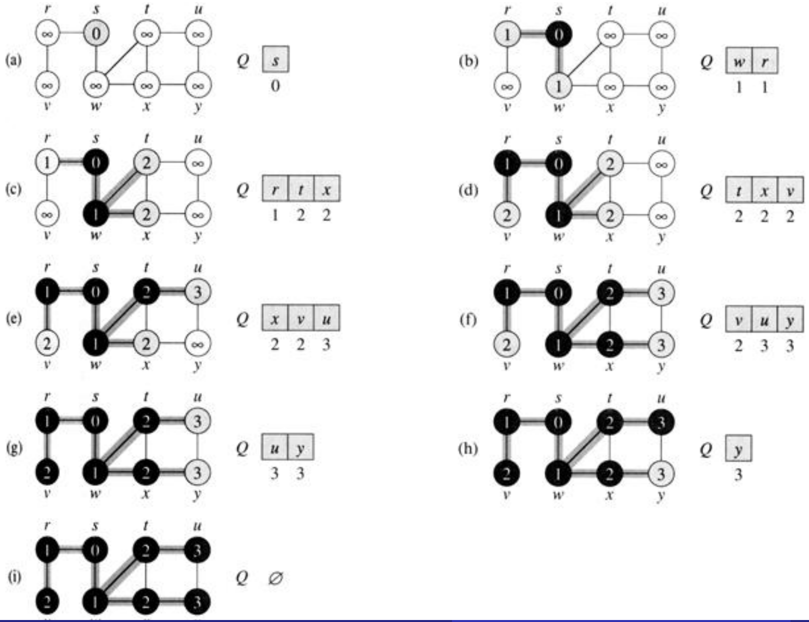
BFSVisit(s)

```
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) Q s 0
(b) Q w r 1 1
(c) Q r t x 1 2 2
(d) Q t x v 2 2 2
(e) Q x v u 2 2 3
(f) Q v u y 2 3 3
(g) Q u y 3 3
(h) Q y 3
(i) Q ∅
r s t u
v w x y

The outputs of BFS:

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 outputs of BFS:

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

The BFS Forest:

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:

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

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:

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

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

The 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:

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

- 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 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:

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

- 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.

Running Time of BFS

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?

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?
2. Finding connected components