Applications of BFS and DFS

CSE 2011

Fall 2009

Some Applications of BFS and DFS

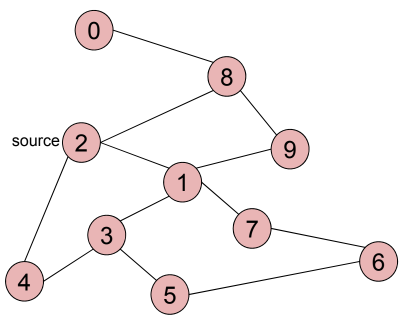
- BFS
  - To find the shortest path from a vertex *s* to a vertex *v* in an unweighted graph
  - To find the length of such a path
  - To construct a BSF tree/forest from a graph
  - To find out if a strongly connected directed graph contains cycles
- DFS
  - To find a path from a vertex *s* to a vertex *v.*
  - To find the length of such a path.
  - To construct a DSF tree/forest from a graph.

Finding Shortest Paths Using BFS

Finding Shortest Paths

- The BFS code we have seen
  - find outs if there exists a path from a vertex s to a vertex v
  - prints the vertices of a graph (connected/strongly connected).
- What if we want to find
  - the shortest path from *s* to a vertex *v* (or to every other vertex)?
  - the length of the shortest path from *s* to a vertex *v*?
- In addition to array *flag*[ ], use an array named *prev*[ ], one element per vertex.
  - *prev*[*w*] = *v* means that vertex *w* was visited right after *v*

Example



Adjacency List

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

Visited Table (T/F)

		prev[ ]
0	T	8
1	T	2
2	T	-
3	T	1
4	T	2
5	T	3
6	T	7
7	T	1
8	T	2
9	T	8

***prev*[ ] now can be traced backward to report the path!**

BFS and Finding Shortest Path

```
Algorithm BFS(s)
    for each vertex v
        do flag(v) := false;
            pred[v] := −1;                 ← initialize all pred[v] to -1
    Q = empty queue;
    flag[s] := true;
    enqueue(Q, s);
    while Q is not empty
        do v := dequeue(Q);                already got shortest path from s to v
            for each w adjacent to v
                do if flag[w] = false
                    then flag[w] := true;
                        pred[w] := v;      ← record where you came from
                        enqueue(Q, w)
```

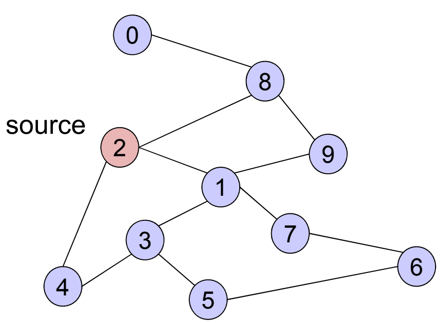
Shortest Path Algorithm

for each *w* adjacent to *v*
  if *flag*[*w*] = false {
    flag[*w*] = true;
    prev*[*w*] = *v; // visited *w* right after *v*
    enqueue(*w*);
  }

- To print the shortest path from *s* to a vertex *u*, start with *prev*[*u*] and backtrack until reaching the source *s*.
  - Running time of backtracking = ?
- To find the length of the shortest path from *s* to *u*, start with *prev*[*u*], backtrack and increment a counter until reaching *s*.
  - Running time = ?

7

Example



Adjacency List

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

Visited Table (T/F)

		prev[ ]
0	F	-
1	F	-
2	F	-
3	F	-
4	F	-
5	F	-
6	F	-
7	F	-
8	F	-
9	F	-

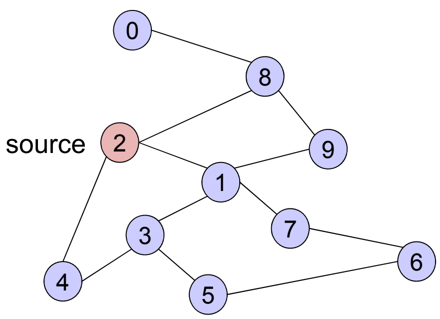
Initialize visited table (all false)

Initialize prev[ ] to -1

Q ={ }

Initialize **Q** to be empty

8



Adjacency List

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

Visited Table (T/F)

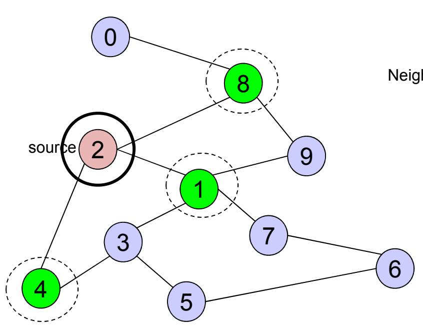
		prev
0	F	-
1	F	-
2	T	-
3	F	-
4	F	-
5	F	-
6	F	-
7	F	-
8	F	-
9	F	-

Flag that 2 has been visited.

Q = { 2 }

Place source 2 on the queue.

9



Neighbors→

Adjacency List

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

Visited Table (T/F)

		prev
0	F	-
1	T	2
2	T	-
3	F	-
4	T	2
5	F	-
6	F	-
7	F	-
8	T	2
9	F	-

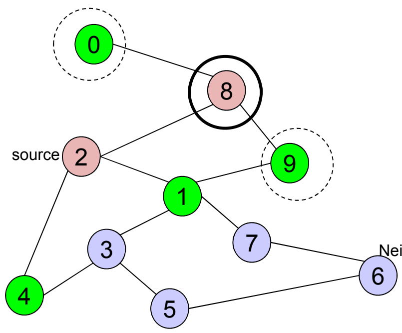
Mark neighbors as visited.

Record in prev that we came from 2.

Q = {2} → { 8, 1, 4 }

Dequeue 2.
Place all unvisited neighbors of 2 on the queue

10



Adjacency List

Neighbors → 8

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

Visited Table (T/F)

		prev
0	T	8
1	T	2
2	T	-
3	F	-
4	T	2
5	F	-
6	F	-
7	F	-
8	T	2
9	T	8

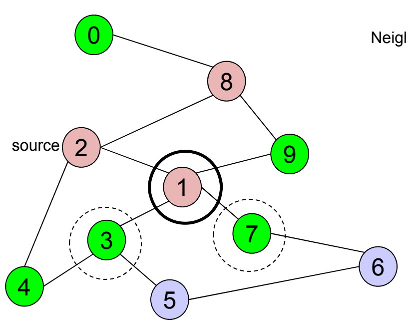
Mark new visited Neighbors.

Record in prev that we came from 8.

Q = { 8, 1, 4 } → { 1, 4, 0, 9 }

Dequeue 8.
-- Place all unvisited neighbors of 8 on the queue.
-- Notice that 2 is not placed on the queue again, it has been visited!

11



Adjacency List

Neighbors → 1

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

Visited Table (T/F)

		prev
0	T	8
1	T	2
2	T	-
3	T	1
4	T	2
5	F	-
6	F	-
7	T	1
8	T	2
9	T	8

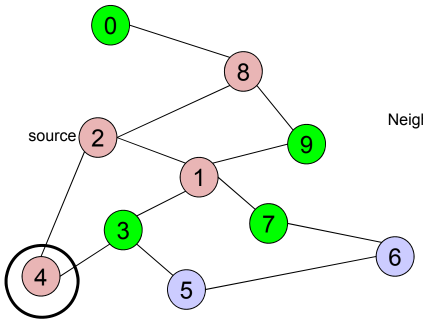
Mark new visited Neighbors.

Record in prev that we came from 1.

Q = { 1, 4, 0, 9 } → { 4, 0, 9, 3, 7 }

Dequeue 1.
-- Place all unvisited neighbors of 1 on the queue.
-- Only nodes 3 and 7 haven't been visited yet.

12



Adjacency List

Neighbors →

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

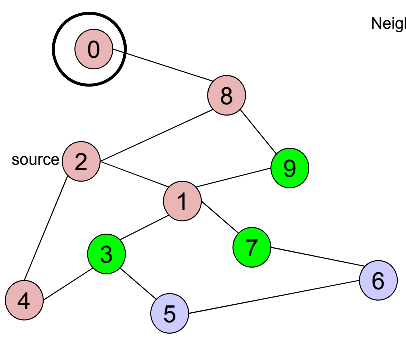
Visited Table (T/F)

Vertex	Visited	*prev*
0	T	8
1	T	2
2	T	-
3	T	1
4	T	2
5	F	-
6	F	-
7	T	1
8	T	2
9	T	8

Q = { 4, 0, 9, 3, 7 } → { 0, 9, 3, 7 }

Dequeue 4.
-- 4 has no unvisited neighbors!

13



Adjacency List

Neighbors →

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

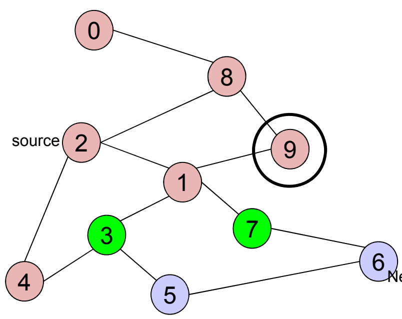
Visited Table (T/F)

Vertex	Visited	*prev*
0	T	8
1	T	2
2	T	-
3	T	1
4	T	2
5	F	-
6	F	-
7	T	1
8	T	2
9	T	8

Q = { 0, 9, 3, 7 } → { 9, 3, 7 }

Dequeue 0.
-- 0 has no unvisited neighbors!

14



Adjacency List

Neighbors → 9

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

Visited Table (T/F)

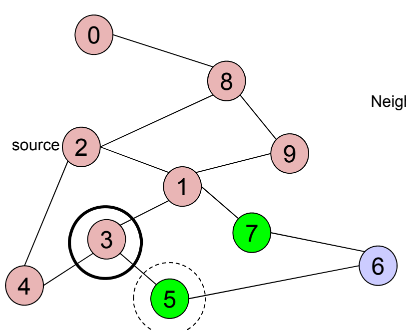
		prev
0	T	8
1	T	2
2	T	-
3	T	1
4	T	2
5	F	-
6	F	-
7	T	1
8	T	2
9	T	8

prev

Q = { 9, 3, 7 } → { 3, 7 }

Dequeue 9.
-- 9 has no unvisited neighbors!

15



Adjacency List

Neighbors → 3

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

Visited Table (T/F)

		prev
0	T	8
1	T	2
2	T	-
3	T	1
4	T	2
5	T	3
6	F	-
7	T	1
8	T	2
9	T	8

prev

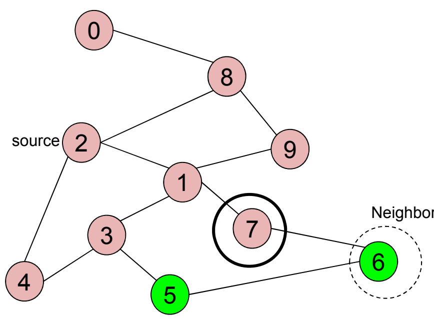
Mark new visited Vertex 5.

Record in prev that we came from 3.

Q = { 3, 7 } → { 7, 5 }

Dequeue 3.
-- place neighbor 5 on the queue.

16



Adjacency List

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

Visited Table (T/F)

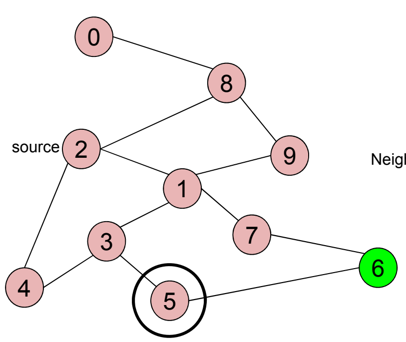
		prev
0	T	8
1	T	2
2	T	-
3	T	1
4	T	2
5	T	3
6	T	7
7	T	1
8	T	2
9	T	8

Mark new visited Vertex 6.

Record in prev that we came from 7.

Q = { 7, 5 } → { 5, 6 }

Dequeue 7.
-- place neighbor 6 on the queue.



Adjacency List

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

Visited Table (T/F)

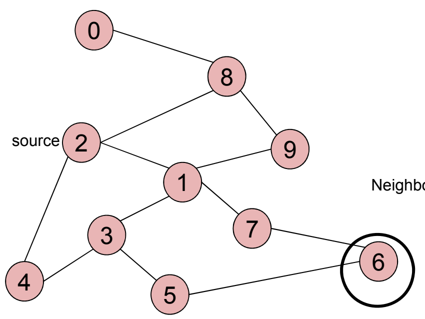
		prev
0	T	8
1	T	2
2	T	-
3	T	1
4	T	2
5	T	3
6	T	7
7	T	1
8	T	2
9	T	8

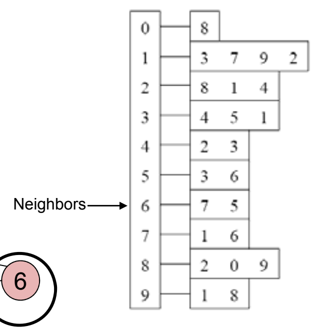
Q = { 5, 6} → { 6 }

Dequeue 5.
-- no unvisited neighbors of 5.

Adjacency List

Visited Table (T/F)





		prev
0	T	8
1	T	2
2	T	-
3	T	1
4	T	2
5	T	3
6	T	7
7	T	1
8	T	2
9	T	8

prev

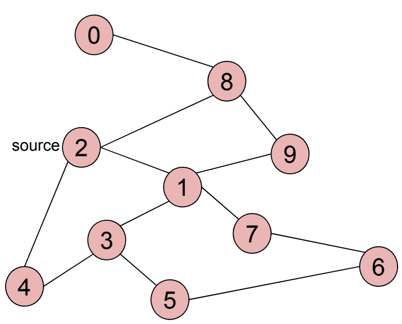
Q = { 6 } → { }

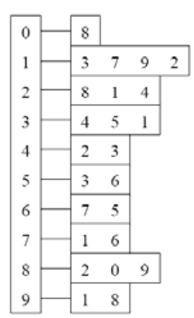
Dequeue 6.
-- no unvisited neighbors of 6.

BFS Finished

Adjacency List

Visited Table (T/F)





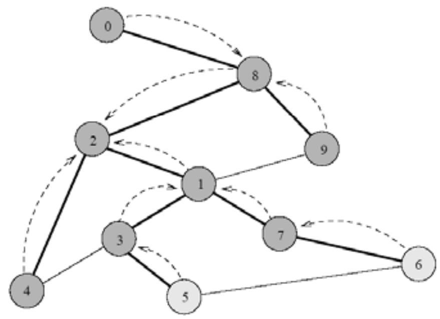
		prev[ ]
0	T	8
1	T	2
2	T	-
3	T	1
4	T	2
5	T	3
6	T	7
7	T	1
8	T	2
9	T	8

***prev*[ ]**

Q = { } **STOP!!! Q is empty!!!**

***prev*[ ] now can be traced backward to report the path!**

Example of Path Reporting



nodes	visited from
0	8
1	2
2	-
3	1
4	2
5	3
6	7
7	1
8	2
9	8

Try some examples; report path from s to v:
Path(2-0) ⇒
Path(2-6) ⇒
Path(2-1) ⇒

21

Path Reporting

- Given a vertex *w*, report the shortest path from *s* to *w*

  currentV = *w*;
  while (*prev*[*currentV*] ≠ –1) {
      output *currentV*; // or add to a list
      currentV = *prev*[*currentV*];
  }
  output *s*; // or add to a list

- The above code prints the path in *reverse* order.

22

Path Reporting (2)

- To output the path in the right order,
  - Print the list in reverse order.
  - Use a stack instead of a list.
  - Use a recursive method (implicit use of a stack).

printPath (w) {
    if ($prev[w] \neq -1$)
        printPath ($prev[w]$);
    output w;
}

23

Finding Shortest Path Length

- To find the length of the shortest path from s to u, start with $prev[u]$, backtrack and increment a counter until reaching the source s.
  - Running time of backtracking = ?
- Following is a faster way to find the length of the shortest path from s to u (at the cost of using more space)
  - Allocate an array $d[\ ]$, one element per vertex.
  - When BSF algorithm ends, $d[u]$ records the length of the shortest path from s to u.
  - Running time of finding path length = ?

24

Recording the Shortest Distance

Algorithm *BFS*(s)

```
for each vertex v
    do flag(v) := false;
       pred[v] := −1;  d[v] = ∞;
Q = empty queue;
flag[s] := true;   d[s] = 0;
enqueue(Q, s);
while Q is not empty
    do v := dequeue(Q);
       for each w adjacent to v
           do if flag[w] = false
                 then flag[w] := true;
                      pred[w] := v;  d[w] = d[v] + 1;
                      enqueue(Q, w)
```

d[v] stores shortest distance from s to v

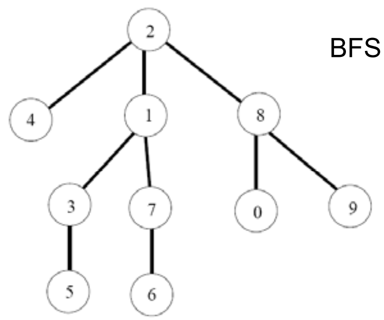
25

Computing Spanning Trees

26

Trees

- Tree: a connected graph without cycles.
- Given a connected graph, remove the cycles ⇒ a tree.
- The paths found by *BFS*(*s*) form a rooted tree (called a *spanning tree*), with the starting vertex as the root of the tree.



BFS tree for vertex $s = 2$

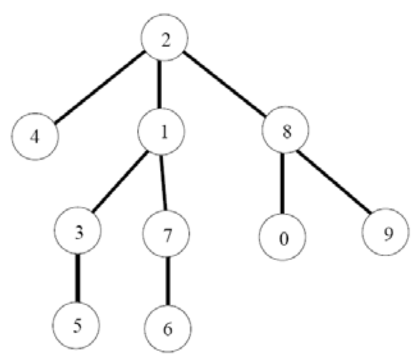
What would a level-order traversal of the tree tell you?

27

Computing a BFS Tree

- Use BFS on a vertex ***BFS(v)*** with array *prev*[ ]

- The paths from source *s* to the other vertices form a tree



28

Computing Spanning Forests

29

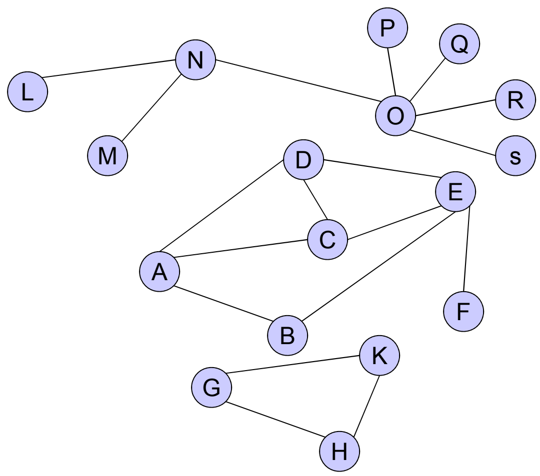
Computing a BFS Forest

- A forest is a set of trees.
- A connected graph gives a tree (which is itself a forest).
- A connected component also gives us a tree.
- A graph with *k* components gives a forest of *k* trees.

30

Example

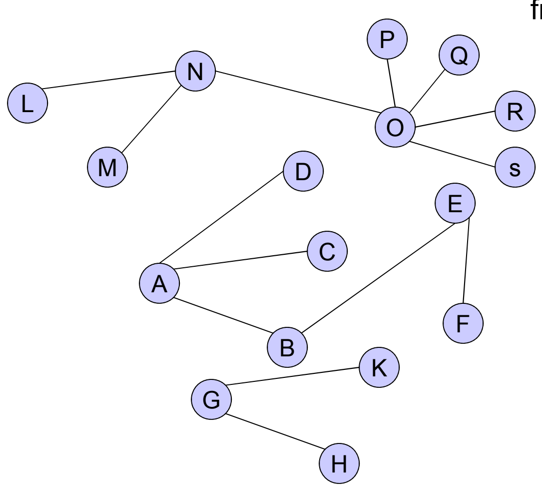
A graph with 3 components



31

Example of a Forest

We removed the cycles from the previous graph.



A forest with 3 trees

32

Computing a BFS Forest

- Use BFS method on a graph ***BFSearch***(***G***), which calls ***BFS(v)***
- Use ***BFS(v)*** with array *prev*[ ].
  - The paths originating from *v* form a tree.
- ***BFSearch***(***G***) examines all the components to compute all the trees in the forest.

33

Testing for Cycles

34

Testing for Cycles

- Method *isCyclic*(*v*) returns true if a ***directed graph*** (with only one component) contains a cycle, and returns false otherwise.

```
for each vertex v
    do flag[v] := false;
Q = empty queue;
flag[s] := true;
enqueue(Q, s);
while Q is not empty
   do v := dequeue(Q);
      for each w adjacent to v
          do if flag[w] = false
               then flag[w] := true;
                    enqueue(Q, w)
             else return true;
return false;
```

35

Finding Cycles

- To output the cycle just detected, use info in *prev*[ ].
- NOTE: The code above *applies only to* ***directed*** *graphs*.
- Homework: Explain why that code does not work for undirected graphs.

36

Finding Cycles in Undirected Graphs

- To detect/find cycles in an ***undirected*** graph, we need to classify the edges into 3 categories during program execution:
  - unvisited edge: never visited.
  - discovery edge: visited for the very **first** time.
  - cross edge: edge that forms a cycle.
- Code fragment 13.10, p. 605.
- When the BFS algorithm terminates, the discovery edges form a spanning tree.
- If there exists a cross edge, the undirected graph contains a cycle.

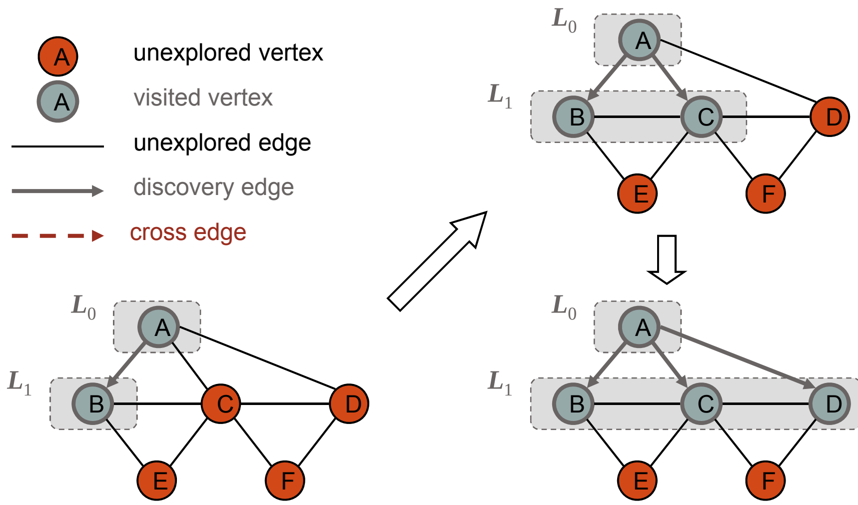
BFS Algorithm (in textbook)

- The algorithm uses a mechanism for setting and getting "labels" of vertices and edges

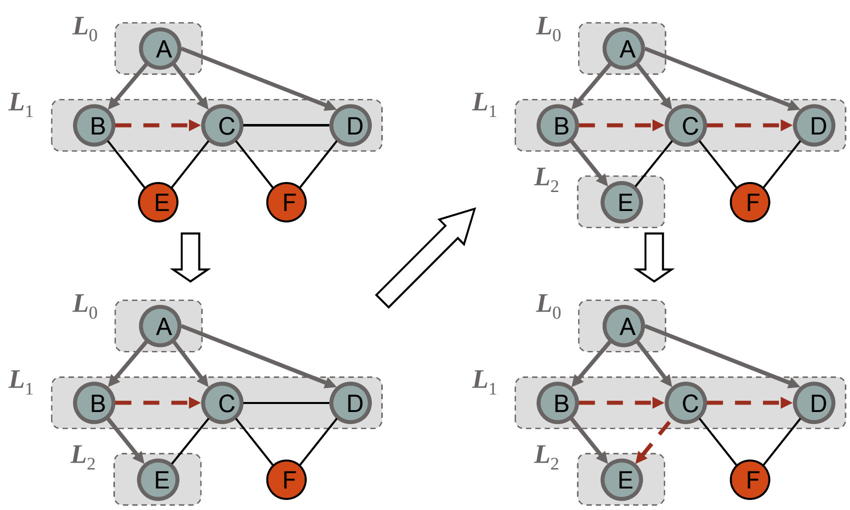
```
Algorithm BFS(G)
    Input graph G
    Output labeling of the edges
        and partition of the
        vertices of G
  for all u ∈ G.vertices()
    setLabel(u, UNEXPLORED)
  for all e ∈ G.edges()
    setLabel(e, UNEXPLORED)
  for all v ∈ G.vertices()
    if getLabel(v) = UNEXPLORED
        BFS(G, v)
```

```
Algorithm BFS(G, s)
  L0 ← new empty sequence
  L0.insertLast(s)
  setLabel(s, VISITED)
  i ← 0
  while ¬Li.isEmpty()
    Li+1 ← new empty sequence
    for all v ∈ Li.elements()
      for all e ∈ G.incidentEdges(v)
        if getLabel(e) = UNEXPLORED
          w ← opposite(v,e)
          if getLabel(w) = UNEXPLORED
            setLabel(e, DISCOVERY)
            setLabel(w, VISITED)
            Li+1.insertLast(w)
          else
            setLabel(e, CROSS)
    i ← i +1
```

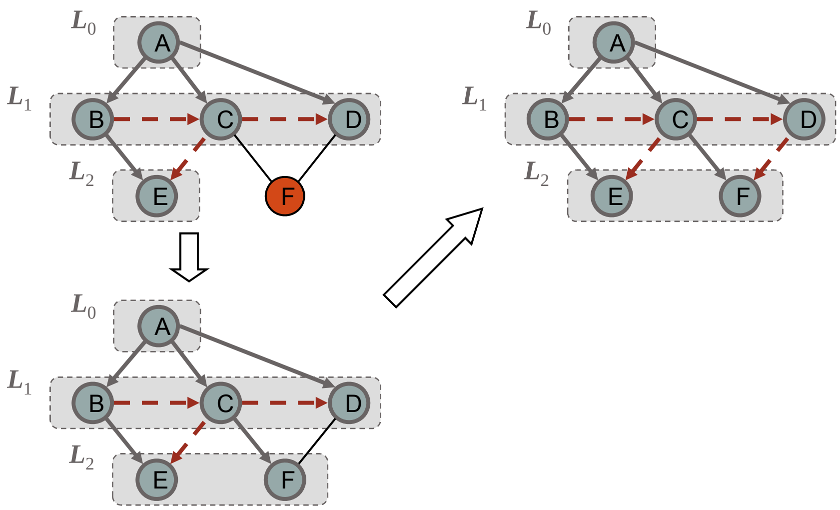
Example



Example (2)



Example (3)



DFS Applications

Applications of DFS

- Is there a path from source *s* to a vertex *v*?
- Is an undirected graph connected?
- Is a directed graph strongly connected?
- To output the contents (e.g., the vertices) of a graph
- To find the connected components of a graph
- To find out if a graph contains cycles and report cycles.
- To construct a DSF tree/forest from a graph

43

DFS Algorithm

Algorithm $DFS(s)$
1. **for** each vertex v
2. **do** $\boxed{flag[v] := \text{false};}$
3. $RDFS(s)$;

Flag all vertices as not visited

Algorithm $RDFS(v)$
1. $\boxed{flag[v] := \text{true};}$
2. **for** each neighbor w of v
3. **do if** $flag[w] = \text{false}$
4. **then** $RDFS(w)$;

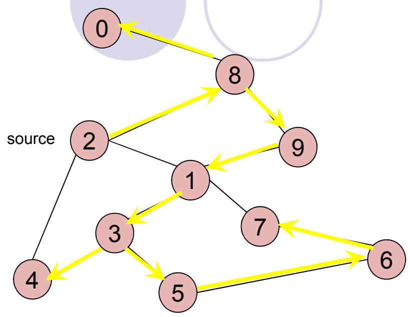
Flag yourself as visited

For unvisited neighbors, call *RDFS*(*w*) recursively

We can also record the paths using *prev*[ ].
Where do we insert the code for *prev*[ ]?

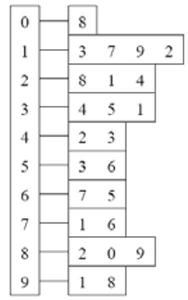
44

DFS Path Tracking



DFS find out path too

Adjacency List



Visited Table (T/F)

		Pred
0	T	8
1	T	9
2	T	-
3	T	1
4	T	3
5	T	3
6	T	5
7	T	6
8	T	2
9	T	8

Algorithm *Path*(w)

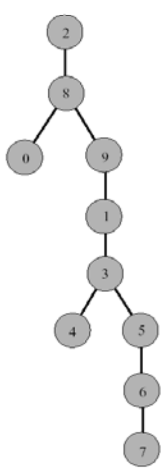
```
if pred[w] ≠ −1
    then
        Path(pred[w]);
output w
```

Try some examples.
Path(0) ->
Path(6) ->
Path(7) ->

45

DFS Tree

Resulting DFS-tree.
Notice it is much "deeper" than the BFS tree.



Captures the structure of the recursive calls

- when we visit a neighbor w of v, we add w as child of v
- whenever DFS returns from a vertex v, we climb up in the tree from v to its parent

46

Finding Cycles Using DFS

- Similar to using BFS.
- For undirected graphs, classify the edges into 3 categories during program execution: unvisited edge, discovery edge, and back (cross) edge.
  - Code Fragment 13.1, p. 595.
  - If there exists a back edge, the undirected graph contains a cycle.

47

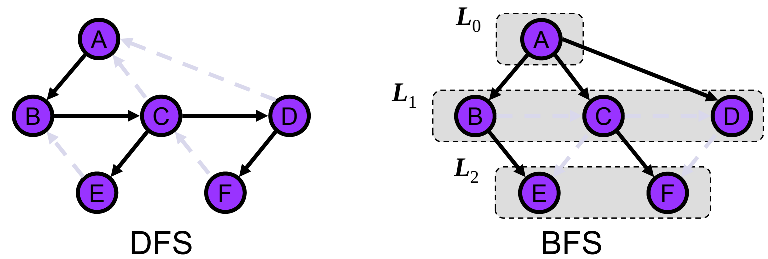
Applications – DFS vs. BFS

- What can BFS do and DFS can't?
  - Finding shortest paths (in unweighted graphs)
- What can DFS do and BFS can't?
  - Finding out if a connected undirected graph is *biconnected*
    - A connected undirected graph is biconnected if there are no vertices whose removal disconnects the rest of the graph

48

DFS vs. BFS

Applications	DFS	BFS
Spanning forest, connected components, paths, cycles	√	√
Shortest paths		√
Biconnected components	√	



Final Exam

- Review: December 8.
- Final Exam: December 11, 7PM - 10PM
- Material:
  - All lectures notes and corresponding sections in the textbook.
  - Assignments 1 and 2.
  - Homework and review questions.