Finding Shortest Paths Using BFS

Finding Shortest Paths

- The BFS code we have seen
  - find outs if there exist 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*

BFS and Finding Shortest Path

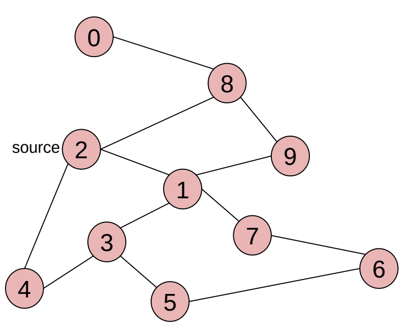
```
Algorithm BFS(s)
for each vertex v
    do flag(v) := false;
       pred[v] := −1;
Q = empty queue;
flag[s] := true;
enqueue(Q, s);
while Q is not empty
    do v := dequeue(Q);
       for each w adjacent to v
           do if flag[w] = false
                then flag[w] := true;
                 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

initialize all pred[v] to -1

already got shortest path from s to v

record where you came from

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

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

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

Shortest Path Algorithm

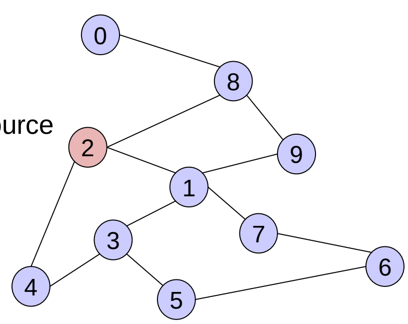
for each *w* adjacent to *v*
    if *flag*[*w*] = false {
        flag[*w*] = true;
        prev*[*w*] = *v; // visited *w* right after *v*
        enqueue(*w*);
    }

- 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 = ?

5

Example

source



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	-

prev[ ]

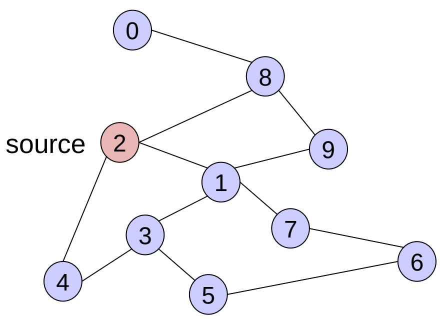
Initialize visited table (all false)

Initialize prev[ ] to -1

Q = { }

Initialize **Q** to be empty

6



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)

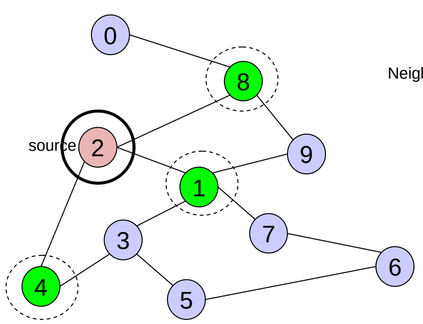
		Pred
0	F	-
1	F	-
2	T	-
3	F	-
4	F	-
5	F	-
6	F	-
7	F	-
8	F	-
9	F	-

Flag that 2 has been visited.

Q = { 2 }

Place source 2 on the queue.

7



Adjacency List

Neighbors → 2

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)

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

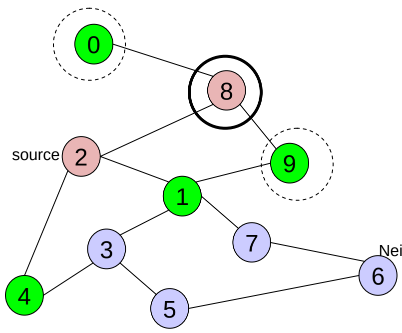
Mark neighbors as visited.

Record in pred that we came from 2.

Q = {2} → { 8, 1, 4 }

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

8



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)

	Visited	*Pred*
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

Pred

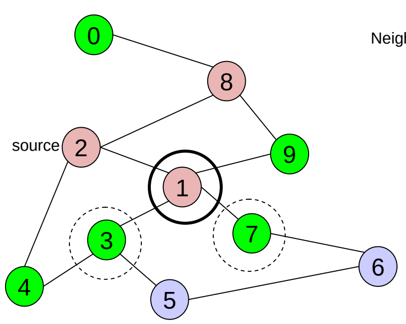
Mark new visited Neighbors.

Record in Pred 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!

9



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)

	Visited	*Pred*
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

Pred

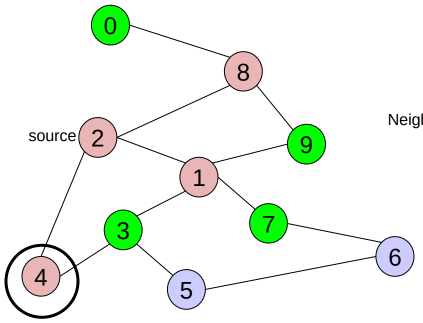
Mark new visited Neighbors.

Record in Pred that we came from 1.

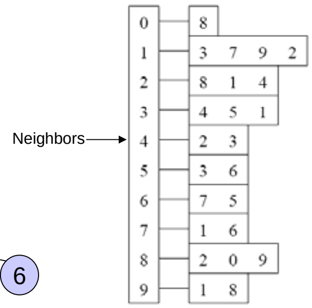
Q = { 1, 4, 0, 9 } → { 4, 0, 9, 3, 7 }

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

10



Adjacency List



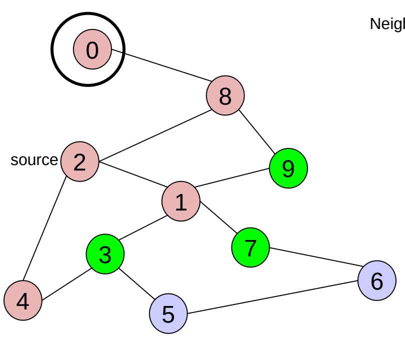
Visited Table (T/F)

		Pred
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

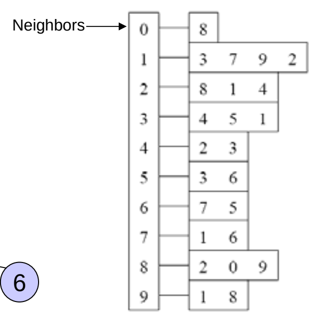
Q = { 4, 0, 9, 3, 7 } → { 0, 9, 3, 7 }

Dequeue 4.
-- 4 has no unvisited neighbors!

11



Adjacency List



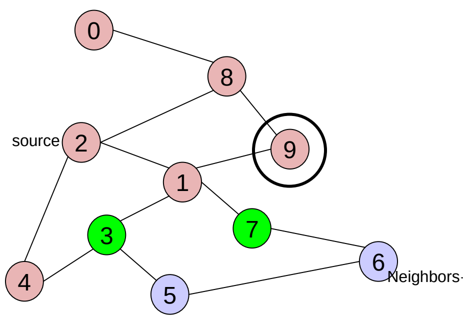
Visited Table (T/F)

		Pred
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!

12



Adjacency List

Vertex	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)

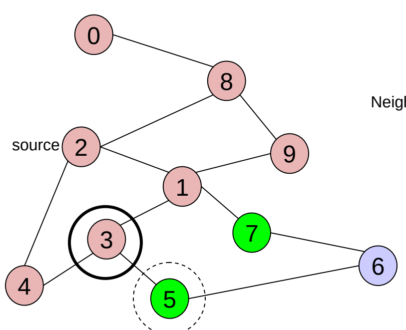
	Visited	Pred
0	T	8
1	T	2
2	T	-
3	T	1
4	T	2
5	F	-
6	F	-
7	T	1
8	T	2
9	T	8

Pred

Q = { 9, 3, 7 } → { 3, 7 }

Dequeue 9.
-- 9 has no unvisited neighbors!

13



Neighbors →

Adjacency List

Vertex	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)

	Visited	Pred
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

Pred

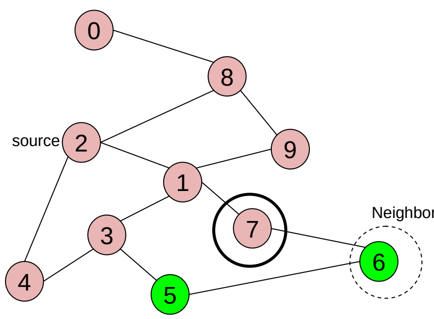
Mark new visited Vertex 5.

Record in Pred that we came from 3.

Q = { 3, 7 } → { 7, 5 }

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

14



Neighbors → 7

Adjacency List

Vertex	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)

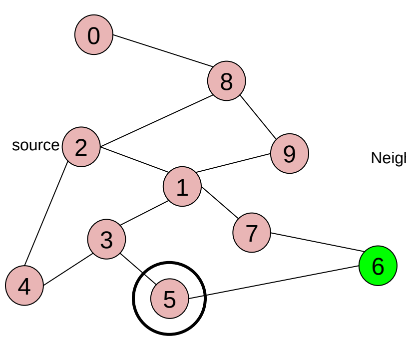
		Pred
0	T	8
1	T	2
2	T	-
3	T	1
4	T	2
5	T	3
6	T	7
7	T	1
8	T	2
9	T	8

Mark new visited Vertex 6.

Record in Pred that we came from 7.

Q = { 7, 5 } → { 5, 6 }

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



Neighbors → 5

Adjacency List

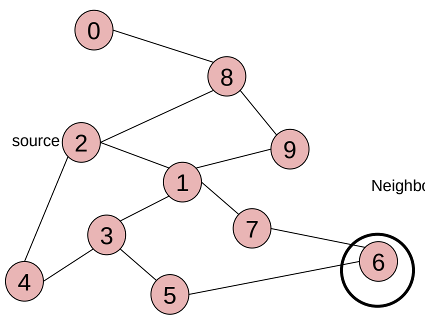
Vertex	Neighbors
0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Visited Table (T/F)

		Pred
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

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)

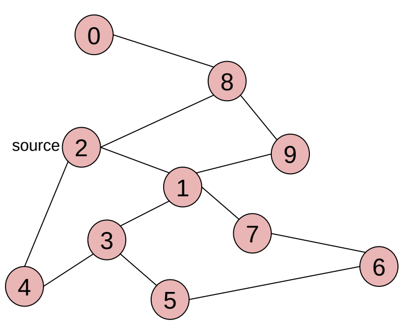
		Pred
0	T	8
1	T	2
2	T	-
3	T	1
4	T	2
5	T	3
6	T	7
7	T	1
8	T	2
9	T	8

Q = { 6 } → { }

Dequeue 6.
-- no unvisited neighbors of 6.

17

BFS Finished



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)

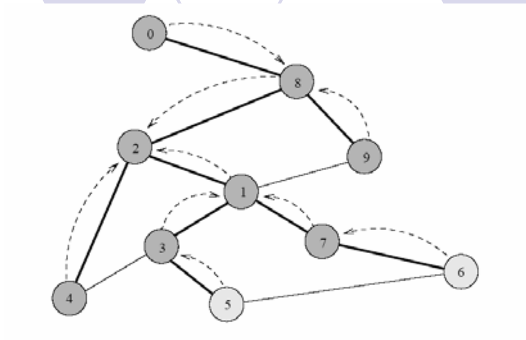
		***prev*[ ]**
0	T	8
1	T	2
2	T	-
3	T	1
4	T	2
5	T	3
6	T	7
7	T	1
8	T	2
9	T	8

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

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

18

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) ⇒

19

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.

20

Path Reporting (cont.)

- 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).

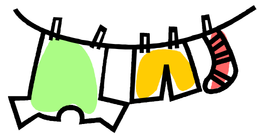
printPath (*w*) {
  if (*prev*[*w*] ≠ –1)
    printPath (*prev*[*w*]);
  output *w*;
}

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 = ?

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)
```

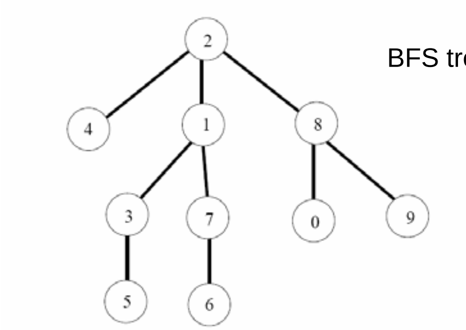


d[v] stores shortest distance from s to v

23

BFS Trees

- Tree: a connected (strongly connected) graph without cycles
- Assuming a connected (strongly connected) graph, the paths found by BFS is often drawn as a rooted tree (called BFS tree), with the starting vertex as the root of the tree.



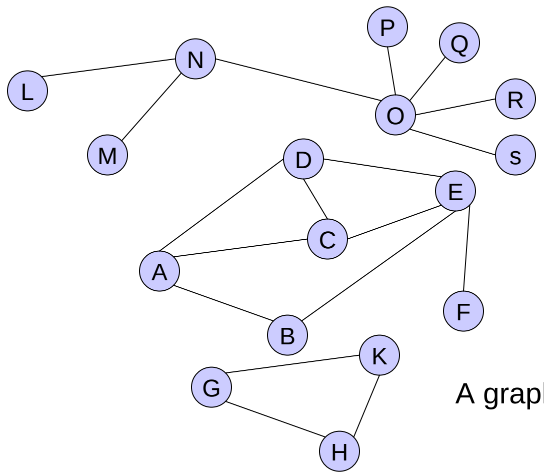
BFS tree for vertex *s* = 2

Question: What would a “level” order traversal tell you?

24

More on BFS

A graph may not be connected (strongly connected) $\Rightarrow$ enhance the above BFS code to accommodate this case.



A graph with 3 components

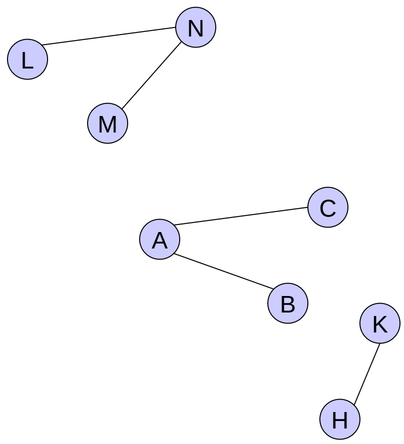
Recall the BFS Algorithm …

Algorithm *BFS*(s)
Input: s is the source vertex
Output: Mark all vertices that can be visited from s.

```
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);     output ( v );
         for each w adjacent to v
              do if flag[w] = false
                    then flag[w] := true;
                         enqueue(Q, w)
```

Enhanced BFS Algorithm

A graph with 3 components



- It turns out that we can re-use the previous BFS method to compute the connected components of a graph *G*

```
BFSearch( G ) {
i = 1;     // component number
for every vertex v
    flag[v] = false;
for every vertex v
    if ( flag[v] == false ) {
       print ( "Component " +  i++ );
       BFS( v );
    }
}
```

27

Next Topics

- To construct a BSF forest from a graph
- Depth First Search (DFS)
- Shortest path algorithms for weighted graphs:
  - Bellman-Ford
  - **Dijkstra's**

28