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 $v$
**Algorithm BFS(s)**

1. for each vertex \( v \)
2. do \( \text{flag}(v) := \text{false}; \)
3. \[ \text{pred}[v] := -1; \]
4. \( Q = \text{empty queue}; \)
5. \( \text{flag}[s] := \text{true}; \)
6. \( \text{enqueue}(Q, s); \)
7. while \( Q \) is not empty
8. do \( v := \text{dequeue}(Q); \)
9. for each \( w \) adjacent to \( v \)
10. do if \( \text{flag}[w] = \text{false} \)
11. then \( \text{flag}[w] := \text{true}; \)
12. \[ \text{pred}[w] := v; \]
13. \( \text{enqueue}(Q, w) \)

---

**Example**

\[ Q = \{ \} \quad \text{STOP!!!} \quad Q \text{ is empty!!!} \]

**Adjacency List**

\[
\begin{array}{cccccccccc}
0 & 3 & 7 & 9 & 2 \\
1 & 8 & 1 & 4 \\
2 & 4 & 5 & 1 \\
3 & 2 & 3 \\
4 & 3 & 6 \\
5 & 7 & 5 \\
6 & 1 & 6 \\
7 & 2 & 0 & 9 \\
8 & 1 & 8 \\
9 &
\end{array}
\]

\[ \text{visited table (T/F)} \]

\[
\begin{array}{cccccccccc}
0 & T & 8 \\
1 & T & 2 \\
2 & T & 1 \\
3 & T & 1 \\
4 & T & 2 \\
5 & T & 3 \\
6 & T & 7 \\
7 & T & 1 \\
8 & T & 2 \\
9 & T & 8 \\
\end{array}
\]

\[ \text{prev[ ]} \]

**Visited Table (T/F)**

\[ \text{prev[ ] now can be traced backward to report the path!} \]
Shortest Path Algorithm

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

- To print the shortest path from \( s \) to a vertex \( u \), start with \( \text{prev}[u] \) 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 \( \text{prev}[u] \), backtrack and increment a counter until reaching \( s \).
  - Running time = ?

Example

\[ Q = \{ \} \]
Initialize \( Q \) to be empty
\[ Q = \{2\} \]
Place source 2 on the queue.

\[ Q = (2) \rightarrow \{8, 1, 4\} \]
Dequeue 2.
Place all unvisited neighbors of 2 on the queue.

Flag that 2 has been visited.

Mark neighbors as visited.
Record in pred that we came from 2.
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!

Dequeue 1.
-- Place all unvisited neighbors of 1 on the queue.
-- Only nodes 3 and 7 haven’t been visited yet.
\( Q = \{4, 0, 9, 3, 7\} \rightarrow \{0, 9, 3, 7\} \)

Deque 4.
-- 4 has no unvisited neighbors!

\( Q = \{0, 9, 3, 7\} \rightarrow \{9, 3, 7\} \)

Deque 0.
-- 0 has no unvisited neighbors!
Adjacency List

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>T</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>T</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

Visited Table (T/F)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>T</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>T</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>T</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>T</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>T</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>T</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>T</td>
<td>8</td>
</tr>
</tbody>
</table>

Q = \{9, 3, 7\} \rightarrow \{3, 7\}

Dequeue 9.
-- 9 has no unvisited neighbors!

Q = \{3, 7\} \rightarrow \{7, 5\}

Dequeue 3.
-- place neighbor 5 on the queue.
**Q = \{7, 5\} \rightarrow \{5, 6\}\)**

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

Mark new visited Vertex 6.
Record in Pred that we came from 7.

---

**Q = \{5, 6\} \rightarrow \{6\}\)**

Dequeue 5.
-- no unvisited neighbors of 5.
Adjacency List

```
0: 8 7 9 2
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)

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

**Q = \{6\} \rightarrow \{\}\**

Dequeue 6.

-- no unvisited neighbors of 6.

---

**BFS Finished**

```
0: 8 7 9 2
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
```

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

Q = \{\} STOP!!! Q is empty!!!
Example of Path Reporting

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

Path Reporting

- Given a vertex w, report the shortest path from s to w
  \[currentV = w;\]
  while (\(\text{prev}[currentV] \neq -1\)) {
    output \(currentV;\)  // or add to a list
    \(currentV = \text{prev}[currentV];\)
  }
output \(s;\)  // or add to a list

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

```java
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 BFS 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** $BFS(s)$
1. for each vertex $v$
2. do $flag(v) := false$;
3. $pred[v] := -1$; $d[v] = \infty$;
4. $Q = empty$ queue;
5. $flag[s] := true$; $d[s] = 0$;
6. $enqueue(Q, s)$;
7. while $Q$ is not empty
8. do $v := dequeue(Q)$;
9. for each $w$ adjacent to $v$
10. do if $flag[w] = false$
11. then $flag[w] := true$;
12. $pred[w] := v$; $d[w] = d[v] + 1$;
13. $enqueue(Q, w)$

$d[v]$ stores shortest distance from $s$ to $v$

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?
More on BFS

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

Recall the BFS Algorithm ...

**Algorithm BFS(s)**

**Input**: s is the source vertex

**Output**: Mark all vertices that can be visited from s.

1. **for** each vertex v
2. \hspace{1em} **do** flag[v] := false;
3. \hspace{1em} Q := empty queue;
4. \hspace{1em} flag[s] := true;
5. \hspace{1em} enqueue(Q, s);
6. **while** Q is not empty
7. \hspace{1em} **do** v := dequeue(Q); output ( v );
8. \hspace{1em} **for** each w adjacent to v
9. \hspace{2em} **do if** flag[w] = false
10. \hspace{3em} **then** flag[w] := true;
11. \hspace{2em} enqueue(Q, w)
Enhanced BFS Algorithm

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

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

A graph with 3 components:

Next Topics

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