DSC40B: Theoretical Foundations of Data Science II

Lecture 11: *Breadth-first-search (BFS) in graphs: part I*

Instructor: Yusu Wang

Graph search strategies

- How do we
  - find a path to go from node u to node v in the graph?
  - check whether the graph is connected?
  - compute how many connected components a graph has?

- We want a graph search strategy
  - which is a strategy to explore the graph systematically
    - sometimes called a graph traversal strategy

- Different graph search strategies have different properties
  - e.g, Breadth-first search (BFS) and Depth-first search (DFS)

General high-level ideas

- Each node has one of the following three states:
  - undiscovered
  - pending (discovered, but has not finished exploring it)
    - we say that a node is ``discovered" when seeing it first time, at which point its status is changed from undiscovered to pending.
  - visited (done with exploring all its neighbors)

- At the beginning, all nodes are undiscovered
- At any moment,
  - the search strategy will choose next node to visit (explore) from the list of pending nodes
  - if a node is "visited", then all its neighbors should be in "pending" or "visited"

- How do we decide which is the next node to visit?
  - Breadth-first search:
    - choose the ``oldest” pending node
    - namely, the one was discovered earliest among all pending nodes
  - Depth-first search:
    - choose the “newest” pending node
    - namely, the one that was discovered last among all pending nodes

Breadth-first search (BFS): the algorithm

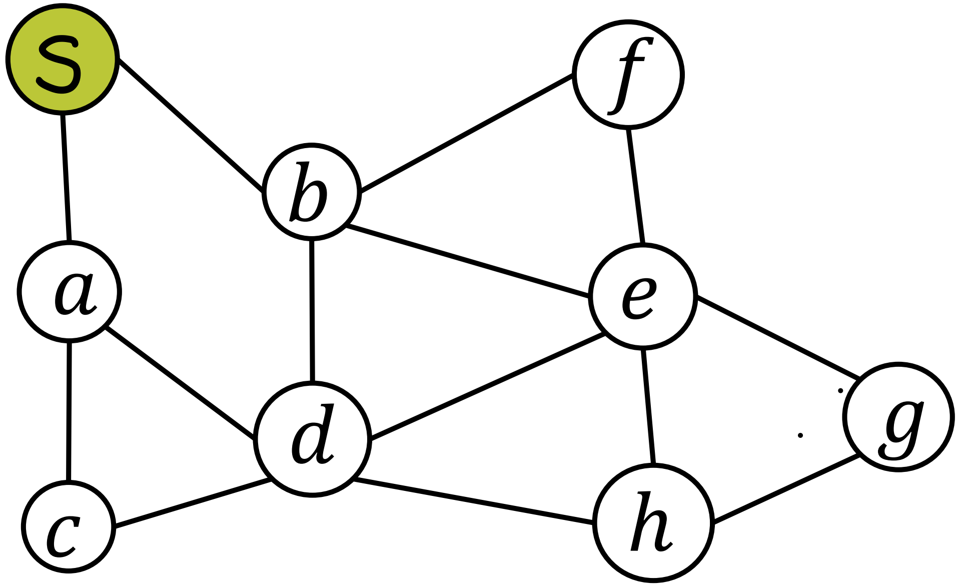
Breadth-first search

- BFS(G, s)
  - It will perform breadth-first search in G starting from a graph node s called the *source node*.

- Idea:
  - All nodes are initialized as undiscovered, other than the source node, which is initialized as pending (i.e, discovered, and yet to be processed)
  - At each step:
    - take the oldest pending node to explore
    - mark all its *undiscovered neighbors* as pending
    - then mark this node to be visited
  - Repeat till there is no more pending nodes to explore

Convention:
In this class, neighbors are produced in sorted order. (In general, depending on your implementation, they don't have to be.)

Example



undiscovered

pending

visited

How to implement the idea?

- Recall: the key to BFS is at any moment, it will choose the oldest pending node to explore

- Need to maintain pending nodes:
  - Need a FIFO (first-in first-out) data structure, which is a standard `queue' data structure
    - A queue data structure can support the following in $\Theta(1)$ time:
    - Q.Enqueue(a): it adds a new element to the end of the current queue
    - b = Q.Dequeue(): it returns the element b at the beginning of the current queue.

- Need to maintain status:
  - we can use an array to store status if all nodes are indexed from 0 to $n-1$
  - or we can use a hash table (e.g, dict from python) to store it

Pseudocode of BFS

```
BFS(G, s)
/* perform BFS starting from source node s ∈ V in graph G = (V, E)          */
for each node v ∈ V do
    v.status = 'undiscovered';
end
s.status = 'pending';
Q.init()     /* initialize Q to be an empty queue                          */
Q.Enqueue(s);
while len(Q) > 0 do
    u = Q.Dequeue();
    for each neighbor v of u do
        if v.status = 'undiscovered' then
            v.status = 'pending';
            Q.Enqueue(v);
        end
    end
    u.status = 'visited';
end
```

Implementation of BFS in python

- To get the standard \`queue' data structure
  - In python, we need to use deque
    - from collections import deque (“deck”).
    - .popleft(), .pop(), .append()
    - list doesn’t have right time complexity!
    - import queue isn’t what you want!

- To maintain \`status' of nodes
  - we can use a hash table (e.g, dict from python) to store it

Python code for BFS

```python
from collections import deque

def bfs(graph, source):
    """Start a BFS at `source`."""
    status = {node: 'undiscovered' for node in graph.nodes}

    status[source] = 'pending'
    pending = deque([source])

    # while there are still pending nodes
    while pending:
        u = pending.popleft()
        for v in graph.neighbors(u):
            # explore edge (u,v)
            if status[v] == 'undiscovered':
                status[v] = 'pending'
                # append to right
                pending.append(v)
        status[u] = 'visited'
```

Conventions:

All nodes in graph.neighbors(u) are sorted in increasing (alphabetical) orders

Remarks

- The same algorithm works for both undirected and directed graphs
- Claim:
  - BFS(G, s) will visit exactly the set of nodes that are reachable from the source s in the graph G
  - Why?
- Hence some nodes may not be visited in the end,
  - and these are the nodes not reachable from source s
- Can be used to help answer questions such as:
  - Is an input undirected graph connected?
  - Is there a path from u to v ?

Full BFS and analysis

- Note that BFS(G, s) only visits nodes reachable from s
- So if G is disconnected, then it will not visit all nodes

- How to explore all nodes?
  - ``Re-start" from an undiscovered node, till all nodes are discovered
  - Will need to call BFS() potentially multiple times, but need to maintain and pass status between calls

Full-BFS to visit all nodes

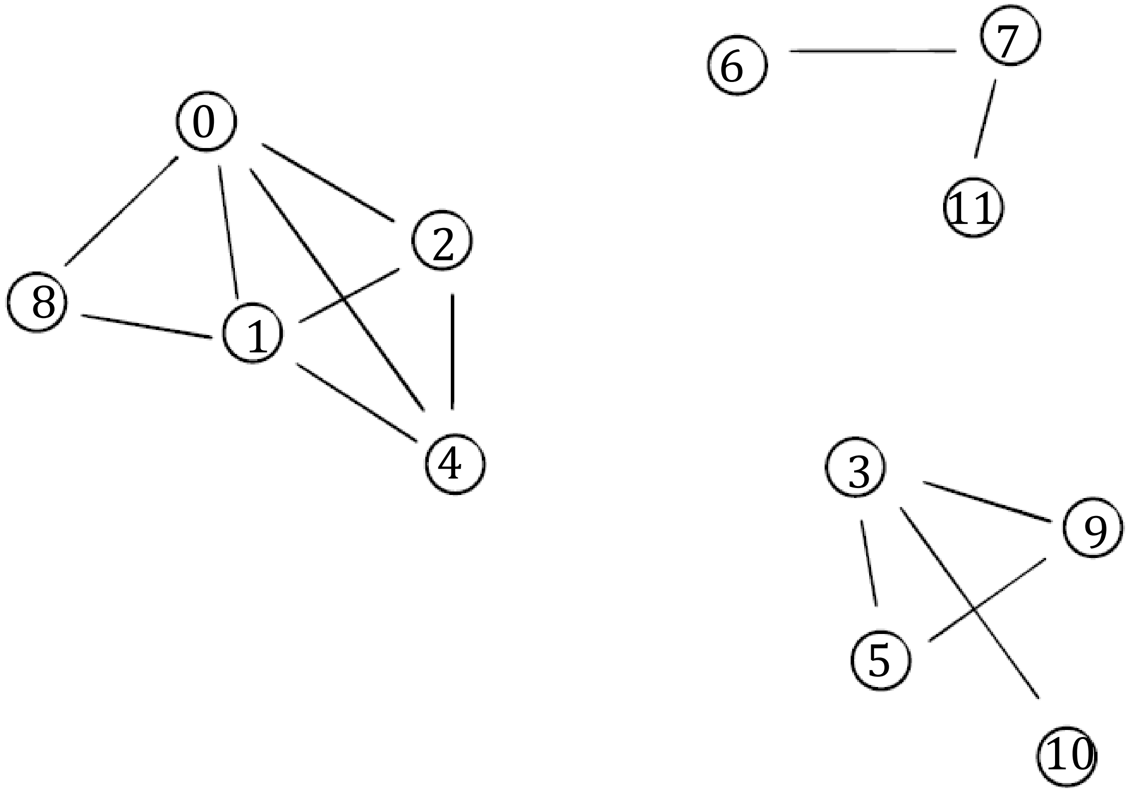
- Modify BFS() to accept statuses as well:

```
def bfs(graph, source, status=None):
    """Start a BFS at `source`."""
    if status is None:
        status = {node: 'undiscovered' for node in graph.nodes}
    # ...
```

- Full-BFS() procedure to visit all nodes

```
def full_bfs(graph):
    status = {node: 'undiscovered' for node in graph.nodes}
    for node in graph.nodes:
        if status[node] == 'undiscovered'
            bfs(graph, node, status)
```

Example



Observation

- If the input is an undirected graph with k components
  - then line 5 of the full_bfs() algorithm (namely, calling bfs) will be executed exactly k times.

```
1   def full_bfs(graph):
2       status = {node: 'undiscovered' for node in graph.nodes}
3       for node in graph.nodes:
4           if status[node] == 'undiscovered'
5               bfs(graph, node, status)
```

Time complexity analysis

- Analyzing full-BFS is conceptually easier than BFS
  - We can use a global argument to count the operations

- Note that time complexity on full-BFS obviously will be upper-bound for the time complexity of BFS

Overall algorithms

```
def full_bfs(graph):
    status = {node: 'undiscovered' for node in graph.nodes}
    for node in graph.nodes:
        if status[node] == 'undiscovered'
            bfs(graph, node, status)
```

```
def bfs(graph, source, status=None):
    """Start a BFS at `source`."""
    if status is None:
        status = {node: 'undiscovered' for node in graph.nodes}
    status[source] = 'pending'
    pending = deque([source])

    # while there are still pending nodes
    while pending:
        u = pending.popleft()
        for v in graph.neighbors(u):
            # explore edge (u,v)
            if status[v] == 'undiscovered':
                status[v] = 'pending'
                # append to right
                pending.append(v)
        status[u] = 'visited'
```

Time complexity for full-BFS

- Each node can be added to the queue exactly once
- Each edge will be explored exactly
  - twice if the input is a undirected graph
  - once if the input is a directed graph
- Initializing status takes $\Theta(|V|)$ time at the beginning

- Hence overall:
  - Time complexity of full-BFS $\Theta(|V| + |E|)$
    - If $|V| < |E|$, then the time is $\Theta(|E|)$
    - If $|V| \geq |E|$, then the time is $\Theta(|V|)$

- As a graph traversal strategy (namely we want to have a way to systematically visit all nodes in the graph)
  - The time complexity is optimal
  - as $|V| + |E|$ is the size needed to even represent input graph.

Time complexity for BFS

- Only for BFS(G, s)
  - Time complexity is $\Theta(|V| + m_s)$ where m_s= #edges in the component of G containing s
  - Note that $m_s = O(|E|)$,
  - Hence the time complexity for BFS is $O(|V| + |E|)$.

FIN