Graph Traversal with DFS/BFS

Tyler Moore

CS 2123, The University of Tulsa

Some slides created by or adapted from Dr. Kevin Wayne. For more information see

http://www.cs.princeton.edu/~wayne/kleinberg-tardos

Graph Traversal

- One of the most fundamental graph problems is to traverse every edge and vertex in a graph.
- For correctness, we must do the traversal in a systematic way so that we dont miss anything.
- For efficiency, we must make sure we visit each edge at most twice.
- Since a maze is just a graph, such an algorithm must be powerful enough to enable us to get out of an arbitrary maze.

Marking Vertices

- The key idea is that we must mark each vertex when we first visit it, and keep track of what have not yet completely explored.
- Each vertex will always be in one of the following three states:
  1. undiscovered the vertex in its initial, virgin state.
  2. discovered the vertex after we have encountered it, but before we have checked out all its incident edges.
  3. processed the vertex after we have visited all its incident edges.
- A vertex cannot be processed before we discover it, so over the course of the traversal the state of each vertex progresses from undiscovered to discovered to processed.

To Do List

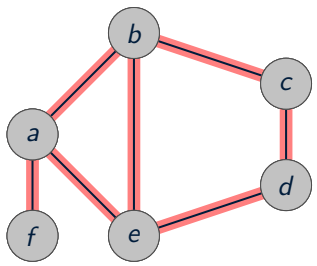
- We must also maintain a structure containing all the vertices we have discovered but not yet completely explored.
- Initially, only a single start vertex is considered to be discovered.
- To completely explore a vertex, we look at each edge going out of it. For each edge which goes to an undiscovered vertex, we mark it discovered and add it to the list of work to do.
- Note that regardless of what order we fetch the next vertex to explore, each edge is considered exactly twice, when each of its endpoints are explored.

In what order should we process vertices?

1. First-in-first-out: if we use a queue to process discovered vertices, we use **breadth-first search**
2. Last-in-first-out: if we use a stack to process discovered vertices, we use **depth-first search**

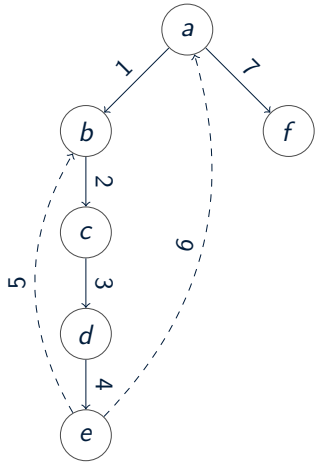
Depth-First Traversal

Undirected Graph



Discovered: a b c d e f
Processed: e d c b f a

Depth-First Search Tree



Recursive Depth-First Traversal Code

```
def rec_dfs(G, s, S=None):
    if S is None: S = set()# Initialize the history
    S.add(s)              # We've visited s
    for u in G[s]:        # Explore neighbors
        if u in S: continue# Already visited: Skip
        rec_dfs(G, u, S)  # New: Explore recursively

    >>> rec_dfs_tested(G, 0)
    [0, 1, 2, 3, 4, 5, 6, 7]
```


Iterative Depth-First Traversal Code

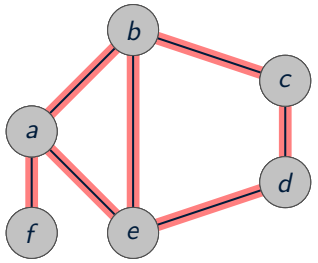
```
def iter_dfs(G, s):
    S, Q = set(), []          # Visited-set and queue
    Q.append(s)               # We plan on visiting s
    while Q:                  # Planned nodes left?
        u = Q.pop()           # Get one
        if u in S: continue# Already visited? Skip it
        S.add(u)              # We've visited it now
        Q.extend(G[u])        # Schedule all neighbors
        yield u               # Report u as visited

    >>> list(iter_dfs(G, 0))
    [0, 5, 7, 6, 2, 3, 4, 1]
```

What does the `yield` command do?

- When you execute a function, it normally `returns` values
- Generators only execute code when iterated
- Useful when you don't need to keep the list of values you loop over
- Especially helpful when the object you're iterating over is very large, and you don't want to keep the entire object in memory
- Nice explanation on Stack Overflow: `http://stackoverflow.com/questions/231767/the-python-yield-keyword-explained`

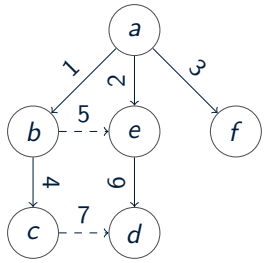
Breadth-First Traversal

Undirected Graph



Discovered: a b e f c d
Processed: a b e f c d

Breadth-First Search Tree



Breadth-First Traversal Code

```
from collections import deque
def iter_bfs(G, s, S = None):
    S, Q = set(), deque()  # Visited-set and queue
    Q.append(s)            # We plan on visiting s
    while Q:               # Planned nodes left?
        u = Q.popleft()        # Get one
        if u in S: continue# Already visited? Skip it
        S.add(u)           # We've visited it now
        Q.extend(G[u])     # Schedule all neighbors
        yield u            # Report u as visited
```

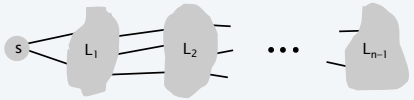

Graph Traversal Exercises

Correctness of Graph Traversal

- Every edge and vertex in the connected component is eventually visited. Why?
- Suppose it's not correct, ie. there exists an unvisited vertex A whose neighbor B was visited.
- When B was visited, each of its neighbors was added to the list to be processed. Since A is a neighbor of B, it must be visited before the algorithm completes.

Breadth-first search

BFS intuition. Explore outward from s in all possible directions, adding nodes one "layer" at a time.



BFS algorithm.

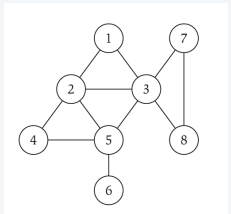
- $L_0 = \{ s \}$.
- L_1 = all neighbors of L_0.
- L_2 = all nodes that do not belong to L_0 or L_1, and that have an edge to a node in L_1.
- L_{i+1} = all nodes that do not belong to an earlier layer, and that have an edge to a node in L_i.

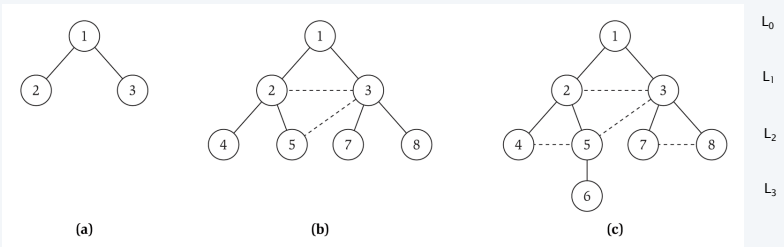
Theorem. For each i, L_i consists of all nodes at distance exactly i from s. There is a path from s to t iff t appears in some layer.

18

Breadth-first search

Property. Let T be a BFS tree of $G = (V, E)$, and let (x, y) be an edge of G. Then, the level of x and y differ by at most 1.





19

Breadth-first search: analysis

Theorem. The above implementation of BFS runs in $O(m + n)$ time if the graph is given by its adjacency representation.

Pf.

- Easy to prove $O(n^2)$ running time:
  - at most n lists $L[i]$
  - each node occurs on at most one list; for loop runs $\leq n$ times
  - when we consider node u, there are $\leq n$ incident edges (u, v), and we spend $O(1)$ processing each edge
- Actually runs in $O(m + n)$ time:
  - when we consider node u, there are $degree(u)$ incident edges (u, v)
  - total time processing edges is $\Sigma_{u \in V} \, degree(u) \; = \; 2m$. ▪

each edge (u, v) is counted exactly twice in sum: once in degree(u) and once in degree(v)

20

Identifying the shortest path between nodes in a graph

- Fact: a path from node s to t in a breadth-first traversal is the shortest path from s to t.
- But how can we programmatically identify the shortest path between two nodes?
  - By keeping track of each node's parent when traversing the graph

Identifying the shortest path between nodes in a graph

```
def bfs_parents(G, s):
    from collections import deque
    P, Q = {s: None}, deque([s])# Parents and FIFO queue
    while Q:
        u = Q.popleft()          # Constant-time for deque
        for v in G[u]:
            if v in P: continue # Already has parent
            P[v] = u             # Reached from u: u is pa
            Q.append(v)
    return P
```


Identifying the shortest path between nodes in a graph

```
N2 = {
    'a': ['b', 'c'],
    'b': ['d'],
    'c': ['e', 'f'],
    'd': ['e'],                  # d
    'e': ['f', 'g'],                 # e
    'f': ['d'],
    'g': ['f']
}
>>> P = bfs_parents(N2, 'a')
>>> P
{'a': None, 'c': 'a', 'b': 'a', 'e': 'c', 'd': 'b', 'g': 'e', 'f': 'c'}
```


Identifying the shortest path between nodes in a graph

```
#get shortest path from a to g
path = ['g']
u = 'g'
while P[u] != 'a':
    if P[u] is None: #give up if we find the root
        print 'path not found'
        break
    path.append(P[u])
    u = P[u]

path.append(P[u]) #don't forget to add the source
path.reverse()    #reorder the path to start from url1
>>> print path
['a', 'c', 'e', 'g']
```