CS310 - Advanced Data Structures and Algorithms

Spring 2014 – Class 21

April 24, 2014

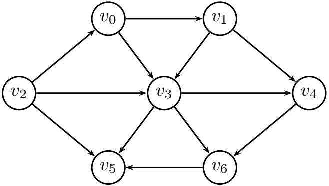
Graph Traversal

- Recall tree traversals from CS210.
- For a quick review, look at Fig. 18.23, pg. 668.
- In the preorder traversal, we start at the root, and plunge down the leftmost side, then back and down the next child of root, down, down, back up, back up, next child, down, down, back up 3x, no more children of root, done.
- You should also know the postorder and inorder, but the preorder is the one that generalizes to “depth-first search” (DFS) of a directed graph.

Depth-First Search (DFS)

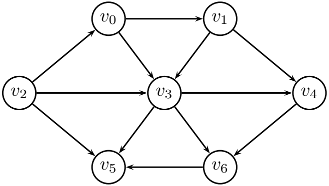
- DFS starting from V_2, etc. where we use the lower numbered adjacent vertex first. (We need some way of deciding which edge to visit first.
- In implementations, we simply follow the order of elements on the adjacency list for the vertex.)
- See how we can follow out-edges down and down:

$V_2 \rightarrow V_0 \rightarrow V_1 \rightarrow V_3 \rightarrow V_4 \rightarrow V_6 \rightarrow V_5$

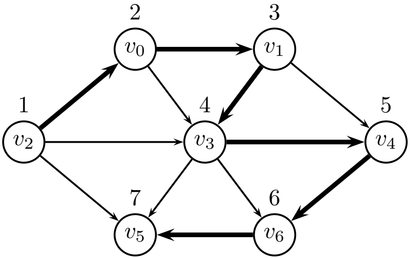


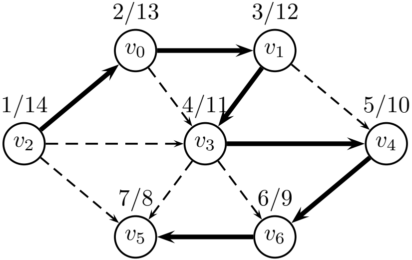
Depth-First Search (DFS)

- We're stuck with no place to go at V_5.
- We can backtrack back to V_6, back to V_4, back to V_3, see another adjacent vertex, V_6, but already visited, back to V_1, see another adj vertex, V_4, but already visited, back to V_0, see another adj vertex, V_3, but already visited, back to V_2, but V_0 and V_5 were already visited. Done.
- Notice that we don't necessarily have to start from V_2.



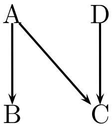
Depth-First Search (DFS)





Another DFS Example

For a simpler graph, consider the following:



- All the edges point downward.
- The DFS starting from A visits A, then B, then backtracks and visits C, backtracks back to A and is done for one tree.
- It starts over at D, but finds no unvisited vertices, so D is alone as the second tree.
- So the DFS visitation order is A, B, C, D.
- Notify vertices in DFS visitation order

Pseudocode for DFS

```
 Set up Set of ''unvisited'' vertices, initially
containing all the vertices.
for each vertex v
    // notify DFS tree starting
    if (v is in unvisited)
        dfs(v)
```

```
 dfs(vertex v)
// notify DFS visitation of v
remove v from unvisited set
for each vertex n adjacent to v
    if (n is in unvisited)
        dfs(n)
```

DFS Implementation

- Java: see DFS.java handout. This is generic <V,E> code, and uses Observable-to-Observer notifications.
- We can use DFS to do topological sort, by changing when we do the vertex notification.
- For a topological sort of this graph, we need A and D before C, so the DFS order isn't good with D at the end.
- It turns out we need to notify the points of last visit to each node: B, C, A, D
- Then the reverse of this is a topological sort.

DFS – Some Comments

- We can do a DFS of any directed (or undirected) graph, and if it's acyclic, the DFS yields a topological sort.
- If there is a cycle in the graph, it doesn't cause an infinite loop because a DFS doesn't revisit a vertex.
- In general a DFS works in phases, finding trees, so the whole thing finds a forest.

DFS on a DAG

- The first tree is a tree being traversed in preorder.
- Once this traversal returns to a vertex, it has classified all vertices it has visited as "downwind" from it, i.e. belonging to the right of it in topological sort. (And no edge can be inbound from those, because of the acyclic requirement.)
- The extreme is V2, the last revisited after going over all the edges, and it should be first in the top sort order for that tree.
- The second-to-last vertex re-visited should be second, and so on.

DFS – Some Comments

- The following trees in the forest have only inter-group edges back to previous groups, so they need to be ordered last-first in top-sort order.
- The ability of the DFS to turn a graph into a forest of trees is useful in many algorithms.
- Trees are a lot easier to work with than general graphs.
- Note that a graph does not have to be acyclic to do a DFS. It's just that you can only get a topological sort out of it if it's acyclic.

DFS – Some Comments

- The graph sources have cs310.DFS.java to do DFS traversals, and DFSTest.java as a sample application (no Swing).
- DFSDemo creates random graphs and shows them using Swing.
- The handout shows the code needed to set up for a simple screen display of the final graph as specified in pa6.
- You can use DFS.java to do the topological sort problem on pa6 if you want, but hide a modified DFS class as a private inner class.

Breadth-first search (BFS)

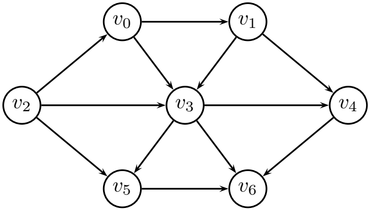
- Sometimes we want to visit all the adjacent nodes to some node before we visit other nodes. This is called Breadth-first search (BFS).
- Like DFS, we start from a source node.

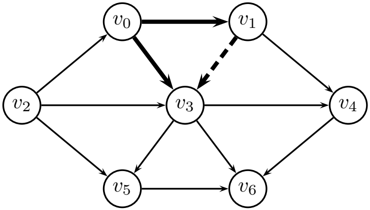
Pseudo-code:

```
 enqueue the source node
while the queue is not empty
    dequeue a node
    display the node
    for all unvisited children of the node
        enqueue the child node
```

BFS Example

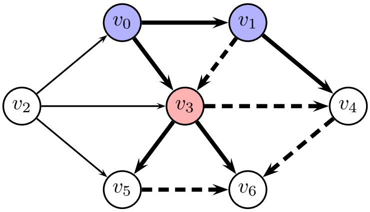
- visit V_0: we see that V_0 has two outbound edges, to V_1 and V_3
- visit edge V_0-V_1 (tree edge, since V_1 unseen. Consider V_1 seen now, but not yet processed)
- visit edge V_0-V_3 (tree edge, V_3 now seen)
- visit V_1: we see that V_1 has two outbound edges, to V_3 and V_4
- visit edge V_1-V_3 (non-tree edge, since V_3 seen before)

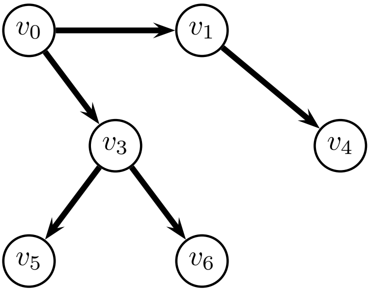




BFS Example

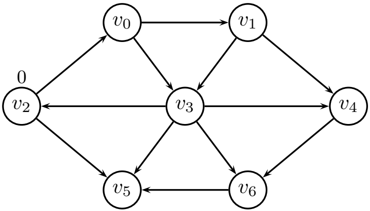
- visit edge V_1-V_4 (tree edge)
- visit V_3: we see that V_3 has 3 outbound edges, to V_4, V_5, and V_6
- visit edge V_3-V_4 (non-tree edge, since V_4 seen before)
- visit edge V_3-V_5 (tree edge)
- visit edge V_3-V_6 (tree edge)
- (now finished with neighbors of V_0, start looking at neighbors-of-neighbors, all previously seen)
- visit V_4 (...)
- ...



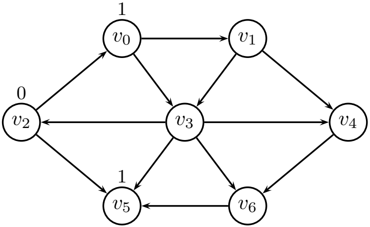


BFS Starting from V_2

Initialize: Mark first vertex as reachable with 0 edges

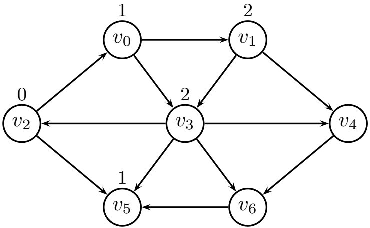


The graph after all the vertices whose path length from the starting vertex is 1 have been found

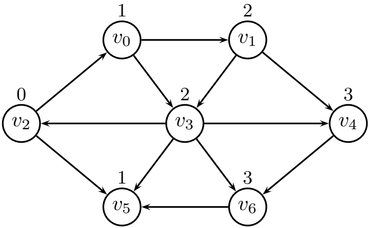


BFS Starting from V_2

The graph after all the vertices whose path length from the starting vertex is 2 have been found



The graph after BFS is completed



BFS Example

- We end up with a BFS with 3 levels, V_2 on the top level, V_0 and V_5 on the second level,
- V_1 and V_3 on the third, and V_4 and V_6 on the fourth level.
- If we started from V_0, V_2 would not be visited.
- As in DFS, we have a tree defined by the BFS using a subset of edges from the graph.
- The nodes at the second level are all one hop away from the source node, V_2.
- The nodes on the third level are all 2 hops away from V_2. Thus we are finding how far, in hops, the various nodes are from V_2.

BFS Example

- If a node is not reachable from the source, its an infinite distance from it.
- The example shown before (Figure 14.16) differs by one edge from the DAG used for the topological sort (Figure 14.30).
- The one edge difference (V_3 to V_2 instead of V_2 to V_3) causes cycles in Figure 14.16, $V_2 \to V_0 \to V_3 \to V_2$ and a longer one $V_2 \to V_0 \to V_1 \to V_3 \to V_2$.

Implementing BFS

- Since BFS (or DFS) define a tree, we can think of the "lower" nodes as children of their parent node in the tree.
- For the BFS, we find the children of a node, and then go back over them to find their children.
- We can put them on a queue as we find them, and when we run out of children, go back and dequeue them and work on their children.

DFS vs. BFS

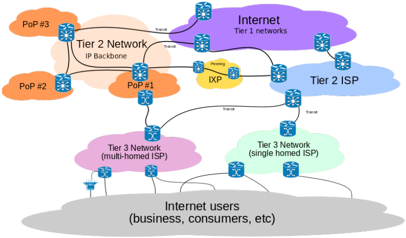
- DFS is like preorder tree traversal, plunging further and further from the source node until we can't go any further, then back.
- Can do cycle detection, can be used to topologically sort an acyclic graph.
- BFS: explore nodes adjacent to the source node, then nodes adjacent to those (that haven't been visited yet), and so on.
- Good for finding all neighbors, all neighbors of neighbors, etc. – hop counts.

Unweighted Single-Source Shortest-path Problem (SSSP)

- Find the shortest path (measured by number of edges) from a designated vertex S to every vertex.
- A special case of the weighted shortest-path problem with all weights=1.
- Has a more efficient solution.

Example: Routing in the Internet

- protocols control sending, receiving of msgs e.g., TCP, IP, HTTP, FTP, PPP
- Internet: "network of networks"
  - loosely hierarchical
  - public Internet versus private intranet
- Internet standards
  - RFC: Request for comments
  - IETF: Internet Engineering Task Force



Theory

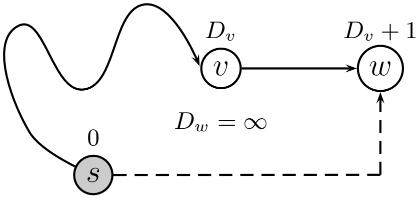
- Use a breadth-first search (BFS) strategy
  - Process vertices in layers
  - Closest to the start are evaluated first
  - Those most distant are evaluated last
- The shortest path from S to V_2 is a path of length 0.
- Start looking for all vertices that are distance 1 from S.
- Then we find each vertex whose shortest path from S is 2 ...

Single Source Shortest Path (SSSP)

- We don't want to search the whole graph looking for the next nodes to work on.
- It's clear that the nodes marked for the first time are the ones that soon should be re-examined for the next hop.
- To be efficient, we can save them on a queue, and then pull them off the queue for the next hop
- We will see these are just the nodes in BFS visitation order.

Search for the Shortest Path

- D_v = cost of a path to vertex v
- $D_w = D_v + 1$ if v is adjacent to w
- Starting with $Dw = \infty$, we can get to w by following a path to v and extending the path by the edge (v,w)

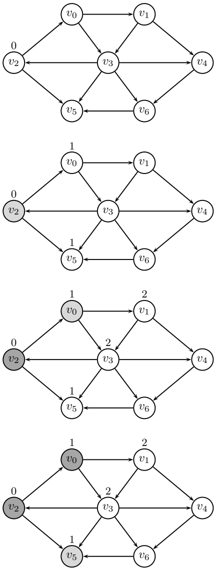


Pseudocode for Unweighted SSSP Algorithm

```
 Given a directed graph G and source node S.
Set up storage for a distance for each node, Dn:
For each node n, make Dn be INFINITY.
Set D_S = 0
Create queue q.
Enqueue source node S on q.
while the q is not empty
    Dequeue a node from q and call it v
    Loop though nodes w adjacent to v
        if D_w is INFINITY (not yet marked)
            set D_w to D_v + 1
            enqueue w on q
```

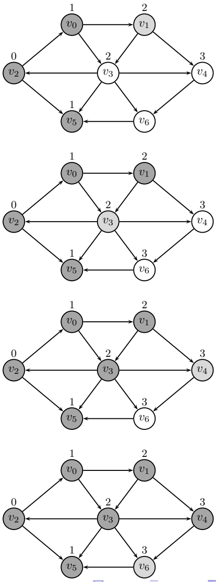
Running Example

- In panel 1, the eyeball node v is not yet set, and $q = (V_2)$ from initialization.
- $D_0 = \infty$, $D_1 = \infty$, $D_2 = 0$, $D_3 = \infty$, $D_4 = \infty$, $D_5 = \infty$, $D_6 = \infty$,
- In panel 2, V_2 is dequeued and becomes the eyeball node v, and V_0 and V_5 are enqueued, so $q = (V_0, V_5)$
- When V_0 was enqueued, $D_w = D_0$ and $D_v = D_2 = 0$, so $D_0 = 0 + 1 = 1$



Running Example

- When V_5 was enqueued, $D_w = D_5$ and $D_v = D_2 = 0$, so $D_5 = 0 + 1 = 1$
- In panel 3, V_0 is dequeued and becomes the eyeball node, and V_1 and V_3 are enqueued, so q = (V_5, V_1, V_3)
- When V_1 was enqueued, $D_w = D_1$ and $D_v = D_0 = 1$, so $D_0 = 1 + 1 = 2$
- When V_3 was enqueued, $D_w = D_3$ and $D_v = D_0 = 1$, so $D_3 = 1 + 1 = 2$
- In 4, V_5 is dequeued and becomes the eyeball node, and nothing is enqueued, so q = (V_1, V_3)
- and so on.



Running Example

- V_0 and V_5 end up with distances 1 from V_2, as previously claimed from looking at the BFS.
- Similarly, V_1 and V_3 end up with distances 2 from V_2.
- This algorithm finds the distances fine, but what about the paths that provide these shortest distances, source to destination?
- Crucial idea: Every time we mark a node with a (final) distance, we are also performing the last step of the shortest path.
- So all we need to do is save the previous node along with the distance.
- Then we can get the penultimate step from the previous node the same way, and so on.

Pseudo-Code

```
 Given a directed graph G and source node S.
Set up storage for a distance for each node, Dn:
For each node n, make Dn be INFINITY.
Set D_S = 0
Create queue q.
Enqueue source node S on q.
while the q is not empty
    Dequeue a node from q and call it v
    Loop though nodes w adjacent to v
        if D_w is INFINITY (not yet marked)
            set D_w to D_v + 1
            set prev-of-w to v <--added for path
            enqueue w on q
```

Example

- when V_0 is enqueued above, now we also save the fact that prev-of-V_0 is the current v, V_2.
- when V_5 is enqueued above, now we also save the fact that prev-of-V_5 is the current v, V_2.
- when V_1 is enqueued above, now we also save the fact that prev-of-V_1 is the current v, V_0.
- when V_3 is enqueued above, now we also save the fact that prev-of-V_3 is the current v, V_0.
- So with these 4 facts, we can construct the best path from V_2 to V_3:
  - prev-of-V_3 is V_0
  - prev-of-V_0 is V_2
- Thus the best path from V_2 to V_3 is $V_2 \rightarrow V_0 \rightarrow V_3$.

- Our challenge is to write this with our Graph API.
- The result of the algorithm is a distance and previous node for each node in the graph.
- Unlike Weiss's setup, we can't store the distances, etc. in the Graph itself.
- Again we set up a Map from V to what we need, in this case the combo of distance and prev-of Node.

Distinfo Class

```
static public class DistInfo {
    public final static int INFINITY =
        Integer.MAX_VALUE;
    private int dist = INFINITY;
    private Object prev = null; // a vertex object
    public int getDist() { return dist; }
    public Object getPrev() { return prev; }
    public String toString() {
        return "" + (dist == INFINITY ? ‘‘INF’’ :
            "" + dist)
        + ‘‘ (prev = ’’
        + (prev == null ?  ‘‘null’’ :  prev) + ‘‘)’’;
    }
}
```

- This is a simple class, grouping two numbers, and defining one constant.
- The constant is needed to express the idea that a node is unreachable, that is, it's a special value for "dist".
- This class doesn't even have a constructor defined, so it gets a default constructor.
- The initializers of the fields (INFINITY, null) are set by that default constructor.
- This will be in the generic class for Unweighted<V,E>, and we could have V instead of Object, DistInfo<V> instead of DistInfo, etc.

Distinfo Class

- We will be using this DistInfo class outside Unweighted as well as inside, defining DistInfo objects that get passed back from the unweighted method to its caller.
- Here they are the values in a Map, where the keys are the vertices. Since the caller needs to interpret them, the class needs to be public.
- DistInfo is a class that is a helper to Unweighted, so we make it a public static class to express its relationship.
- The client calls it Unweighted.DistInfo. DistInfo is "static" because its objects are independent objects, not tied to any outer class object.

Distinfo Class

- There is no outer class object in this case anyway.
- Instead of a generic class, Unweighted has a generic method in a class that's never instantiated.
- We see "getters" here, getDist() and getPrev(), but no "setters" such as setDist() – why?
- Answer: because Unweighted will be setting these fields, and as an outer class, can access private fields of its nested classes.
- This way, the outside classes can't set anything, but the right class can, so it's nicely encapsulated.

Distinfo Class

- Look at cs310.Unweighted.java. Note that Unweighted is itself not a generic class, because there's no $< .. >$ in "public class Unweighted ..."
- Instead, the generic types are introduced at the method level.
- See pg. 152-153 for discussion of this use of generics.
- Note that Unweighted is a pure app of Graph, unlike Weiss's code, which needs to be a method of his Graph class to access vertexMap.
- Our Graph API is much more encapsulated.