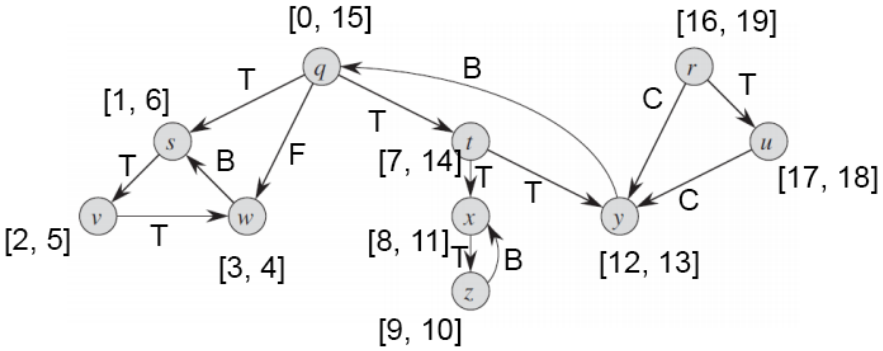
Algorithms and Data — Problem Set 2

Nick Ippoliti

October 11, 2016

1 DFS

1.1 i. & ii.



T: Tree edge

B: Back edge

F: Forward edge

C: Cross edge

2 DFS + BFS

If BFS and DFS on a graph G both return T, then G must equal T. This is true because the only time that BFS and DFS both return the same graph is when G consists solely of tree edges.

If there were any cross edge $e = (n, n')$ in G after running BFS, then in DFS n' would have become a direct child of n, and $T_{BFS} \neq T_{DFS} \neq G$.

If there were any forward edge $e = (n, n')$ in G after running DFS, then in BFS n' would be a direct child of n, and $T_{BFS} \neq T_{DFS} \neq G$.

If there were any back edge $e = (n, n')$ in G after running DFS, then in BFS n' would be a direct child of n, and $T_{BFS} \neq T_{DFS} \neq G$.

3

3.1 i.

The needed graph G would be a graph containing all of the values (x, y, z) with x being the amount of water in the 10-pint jug, y the amount in the 7-pint jug, and z the amount in the 4-pint jug, i.e. $x \leq 10, y \leq 7, z \leq 4$. The edges of G would be the valid transitions in the puzzle between all possible valid puzzle states. A valid transition is a valid move according to the rules of the puzzle, specifically, that in any given move exactly one jug must be emptied or one must be filled, and the total volume of water must be 11.

To solve the puzzle, we must determine if there is a path p from $(0, 7, 4)$ to $(x, 2, z)$ or $(x, y, 2)$.

3.2 ii.

Using this configuration, a DFS on G should determine whether such a path exists.

3.3 iii.

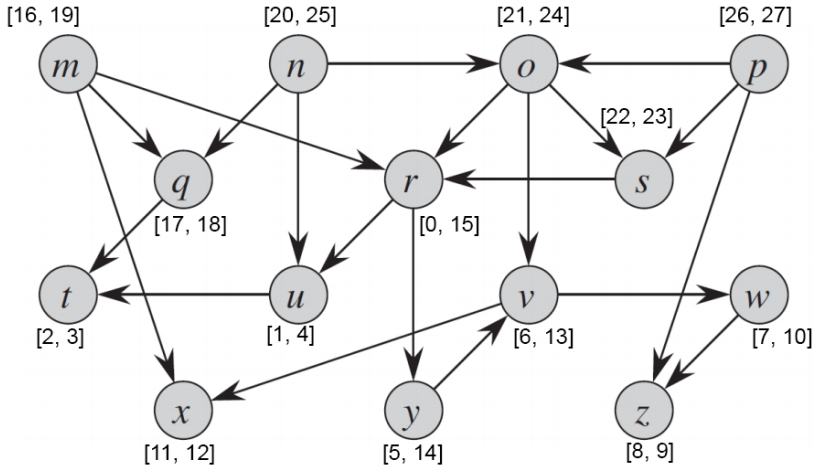
DFS by following valid edges with the greatest x, then y, then z values.

```
(0, 7, 4)
Out edges: (7, 0, 4), (4, 7, 0)
(7, 0, 4)
Out edges: (0, 7, 4), (7, 4, 0), (10, 0, 1)
(10, 0, 1)
Out edges: (3, 7, 1), (7, 0, 4), (10, 1, 0)
(10, 1, 0)
Out edges: (6, 1, 4), (10, 0, 1), (4, 7, 0)
Already saw (10, 0, 1)
(6, 1, 4)
Out edges: (0, 7, 4), (10, 1, 0), (6, 5, 0)
Already saw (10, 1, 0)
(6, 5, 0)
Out edges: (2, 5, 4), (4, 7, 0), (6, 1, 4)
Already saw (6, 1, 4)
(4, 7, 0)
Out edges: (0, 7, 4), (10, 1, 0), (4, 3, 4)
Already saw (10, 1, 0)
(4, 3, 4)
Out edges: (7, 0, 4), (8, 3, 0), (0, 7, 4)
```

```
Already saw (7, 0, 4)
(8, 3, 0)
Out edges: (10, 1, 0), (4, 3, 4), (4, 7, 0)
Already saw (10, 1, 0), (4, 3, 4), (4, 7, 0)
Back to (4, 3, 4)
Already saw (8, 3, 0), (0, 7, 4)
Back to (4, 7, 0)
Already saw (0, 7, 4), (4, 3, 4)
Back to (6, 5, 0)
Already saw (4, 7, 0), (6, 1, 4)
(2, 5, 4)
Out edges: (2, 7, 2), (6, 5, 0), (0, 7, 4)
Already saw (6, 5, 0)
(2, 7, 2)
```

The result is $(0, 7, 4) \to (7, 0, 4) \to (10, 0, 1) \to (10, 1, 0) \to (6, 1, 4) \to (6, 5, 0) \to (2, 5, 4) \to (2, 7, 2)$

4



Exit times: $27, 25, 24, 23, 19, 18, 15, 14, 13, 12, 10, 9, 4, 3$

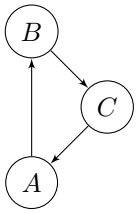
Topological order: $p, n, o, s, m, q, r, y, v, x, w, z, u, t$

5 Connectivity

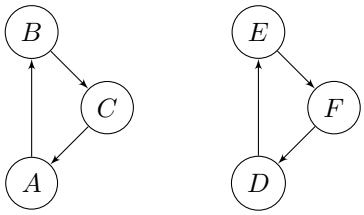
5.1 i.

Doing BFS on G from node u produces tree T containing every vertex in G with exactly one path $u \rightsquigarrow v' \ \forall \ v' \in T$. Removing any vertex $v \in T$ with no descendants ensures that T remains connected, which means that G is still connected.

5.2 ii.

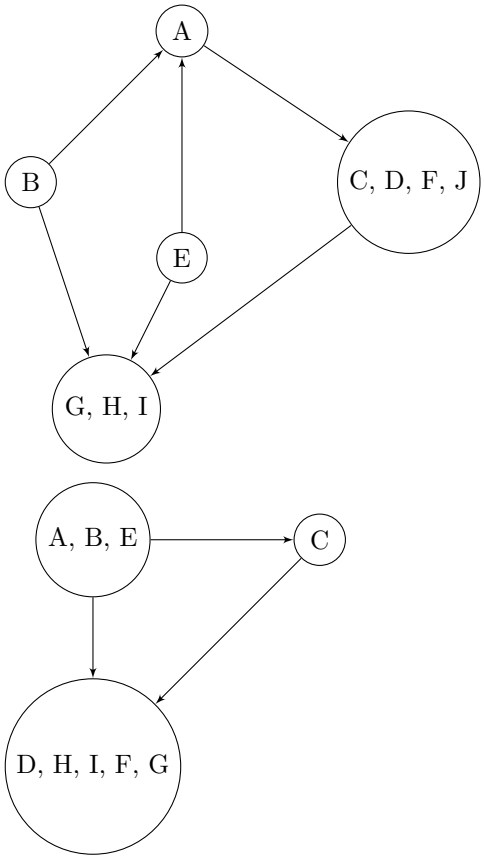


5.3 iii.



6

6.1 i.



6.2 ii.

E, B, A, (G, H, I), (C, J, F, D)

(A, E, B), C, (F, H, I, D, G)

7

Algorithm 1 Explore every vertex connected to v in graph G and set their discover and exit times

```
1  def explore(G, v):
2      counter = 0
3      stack = Stack()
4      stack.put_on_top(v)
5
6      while stack.is_not_empty():
7          current = stack.peek_top()
8          if not current.visited:
9              # set current.visited, set discovery time,
                   and increment counter
10             pre_visit(counter, current)
11
12             for w in current.reachable:
13                 if not w.visited
14                 and not stack.contains(w):
15                     stack.put_on_top(w)
16
17         else:
18             # set exit time, and increment counter
19             post_visit(counter, current)
20             stack.pop_top()
```