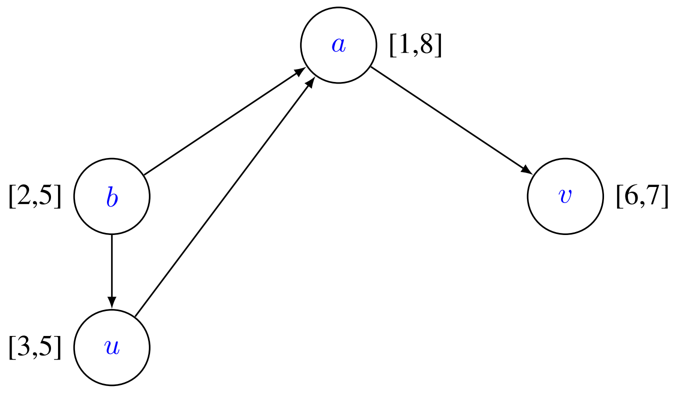
Homework #5 (Solutions)
Introduction to Algorithms/Algorithms 1
600.433/633
Spring 2018

Due on: Thursday, March 8, 5.00pm
Late submissions: will NOT be accepted
Format: Please start each problem on a new page.
Where to submit: Gradescope.
Please type your answers; handwritten assignments will not be accepted.
To get full credit, your answers must be explained clearly,
with enough details and rigorous proofs.

May 7, 2018

Problem 1

For both the problems, the following graph suffices. Run DFS from vertex a. Consider b as the first neighbor of a during DFS, and u as a neighbor of b. So, we would have $u.start = 3, u.finish = 4, v.start = 6$, and $v.finish = 7$.



Problem 2

The algorithm for finding the universal sink of the graph is as follows. Start with all vertices in the set of potential universal sinks S. Each time pick two different vertices x and y from this set and look at the cell $A(x, y)$. If $A(x, y) = 1$, remove x from S, and otherwise remove y from S. Therefore, in $n - 1$ steps, and by looking at one cell of the matrix at each step, you can removed $n - 1$ vertices from S and you only have one potential universal sink say vertex u. Check the whole column and row corresponding to vertex u in matrix A and declare u as a universal sink if the whole row is zero and the whole column is one. Otherwise, the graph does not have a universal sink.

In this algorithm, we check $n - 1$ cells of the matrix to end up with one vertex and then we check the whole row and column corresponding to that vertex for a total of $(n - 1) + (2n - 1)$ cells. Hence, the algorithm has $O(n)$ running time.

Correctness proof: Omitted.

Problem 3

Perform a modified topsort on the graph. Set the cost of every node to $-\infty$ and the start node to 0. Push the source onto the stack. While the stack is not empty, pop a node and examine all of its neighboring nodes. If the cost of the neighbor is less than the that of the node plus the weight of the edge between them, update the cost of the neighbor to this higher value. Delete the edge between the node and its neighbor. If the neighbor is now a source, push it on to the stack. Once the target node is popped from the stack, we know that its cost will not change since it must be a source to have been pushed on the stack. Thus, we can immediately return the target node's cost.

```
function longestPath(G, s, t):
    //input: a graph G with vertices and edges, start
      node s, target node t
    //output: length of the longest path between s and
       t
    stack = Stack()
    for v in G.vertices():
        v.cost = -infinity
        if v is source:
            stack.push(v)
    s.cost = 0
    while stack not empty:
        v = stack.pop()
        if v == t:
            return v.cost
        for each edge (v, w):
            if w.cost < v.cost + weight(v, w):
                w.cost = v.cost + weight(v, w)
            delete edge (v, w)
            if w is source:
                stack.push(w)
    return infinity
```

Running time will be the same as topsort's.

Problem 4

First, if we want to determine whether a graph G is a bipartite graph or not, we may try to make use of a classic result for bipartite graph

Lemma 0.1. *Every bipartite graph has chromatic number ≤ 2 (a.k.a 2-colorable). Any 2-colorable graph is bipartite.*

Proof. By definition, the chromatic number of a graph G is the smallest number of colors needed to color the vertices of G, so that there is no two adjacent vertices share the same color. From the definition of bipartite graph, V can be divided into two disjoint sets V_1 and V_2 and there are no edges between the vertices in V_1 and between the vertices in V_2. So if there is no edge (a, b) where $a \in V_1$ and $b \in V_2$, we can just use one color to color all the vertices. If there is at least one such edge as (a, b), two colors are needs. This is because we need one color for V_1 and the other color for V_2 to avoid that two adjacent vertices share the same color.
If a graph is colored by only one color, vertex set can be divided into any two subsets that meet the definition. If a graph is colored by two colors, divide the vertices into two subsets based on color. By the coloring method mentioned above, any pair of vertices in the subsets are not adjacent .

Let's provide a BFS coloring algorithm to decide whether G is bipartite.

1. Choose any uncolored vertex s and color s RED.
2. Color the neighbors of s BLACK. (use adjacency list)
3. Color the neighbors of all neighbor's RED. (use adjacency list)

4. If all vertices are colored, return TRUE, else repeat the steps starting from 1. While all the coloring steps, if a neighbor of the current vertex is already colored the same color as the current vertex, return FALSE. In this way, a proper 2-coloring is failed and graph G is not bipartite.

5. return TRUE.

Running Time:
The procedures take the same number of vertex search as BFS, which takes $O(|V|+|E|)$ time by using adjacency list.
Correctness:
By running the algorithm, we can use at most two colors to color all the vertices. If the coloring succeed finally without returning FALSE, by lemma 0.1, the graph is bipartite.