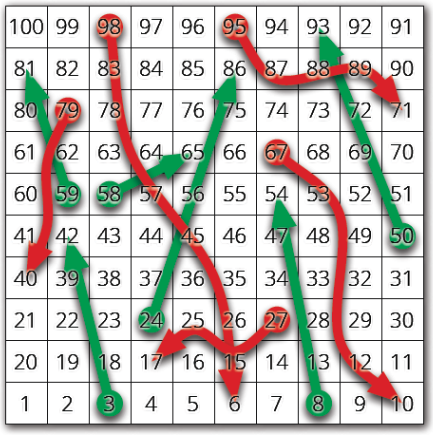
CPSC 320: Intermediate Algorithm Design 2016 Summer Term 1

Tutorial 3 Solutions

Prof. Nick Harvey *University of British Columbia*

1. **(Snakes and Ladders)** This is a classic board game, originating in India no later than the 16th century. The board consists of an $g \times g$ grid of squares, numbered consecutively from 1 to g^2, starting in the bottom left corner and proceeding row by row from bottom to top, with rows alternating to the left and right. Certain pairs of squares in this grid, always in different rows, are connected by either "snakes" (leading down) or "ladders" (leading up). Each square can be an endpoint of at most one snake or ladder.



A typical Snakes and Ladders board.
Upward straight arrows are ladders; downward wavy arrows are snakes.

You start with a token in cell 1, in the bottom left corner. In each move, you advance your token *up to* k positions, for some fixed constant k. If the token ends the move at the top end of a snake, it slides down to the bottom of that snake. Similarly, if the token ends the move at the bottom end of a ladder, it climbs up to the top of that ladder. Describe and analyze an algorithm to compute the smallest number of moves required for the token to reach the last square of the grid.

SOLUTION:

"Where are the dice?" – Lisa Simpson
"Daddy says dice are wicked." – Todd Flanders
"We just move one space at a time. It's less fun that way." – Rod Flanders

Given a Snakes and Ladders board, we will turn it into a *directed* graph, then run Breadth-First Search.

In class we discussed BFS for undirected graphs. Only a few small tweaks are needed to make it work in directed graphs. Recall that there is a for loop that consider all neighbours of vertex u. For the case of directed graphs, we should instead loop over vertices v such that there is a directed edge from u to v.

The graph has $n = g^2$ vertices. Vertex i corresponds to the square on the board labels i. The edges correspond to the to the possible moves of the game.

- For each vertex $i \geq 2$ and each j with $1 \leq j \leq \min\{k, i-1\}$, if i is the top of a snake or bottom of a ladder whose other end is at vertex ℓ, we add an edge from vertex $i-j$ to ℓ.
- Otherwise, for each vertex $i \geq 1$ and each $1 \leq j \leq \min\{k, i-1\}$, we add an edge from vertex $i-j$ to vertex i.

The reason we don't just add edges from the bottom to the top of a ladder is that, in the game, traversing the ladder doesn't count as a "move". It happens automatically when you land at the bottom of the ladder.[1]

We now run a BFS in this graph starting at vertex 1. This computes the distance from 1 to all other vertices. The distance from vertex 1 to vertex n is the smallest number of moves in the game to arrive at vertex n.

It is possible that the distance from 1 to n is infinite, if the game board is pathological and there is no sequence of moves that allows you to arrive at board entry n. This can happen even in the version of the game with dice.

[1]Technically we didn't handle the case that vertex 1 is the bottom of a ladder, but it's not clear how that should be counted. Also, we did not specify what should happen if a square on the board is *both* the bottom of a ladder *and* the top of a snake. Also, what happens if there is a snake and a ladder with the same starting and ending squares. Is the player supposed to enter an infinite loop? I guess the game designers would rule out these complicating circumstances.

2. **(Testing Bipartiteness)** In lecture we talked about bipartite graphs, which are useful for modeling situations like stable matching, matching clients to servers, etc. Recall that a bipartite graph is one whose vertices can be colored red and blue, such that there are no blue-blue edges and no red-red edges.

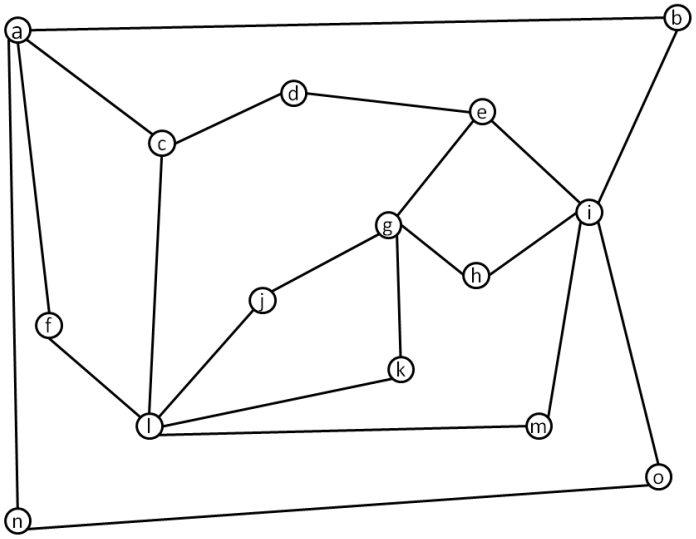
   We observed that an odd cycle (a cycle with an odd number of vertices) cannot possible be bipartite. This is because its colors must alternate red-blue-red-blue-... until at the end we have two consecutive vertices of the same color. For the same reason, any graph containing an odd cycle cannot be bipartite. And, surprisingly, that is the only possible reason for being not bipartite.

   Claim 0.1. A graph is bipartite if and only if it has no odd cycle.

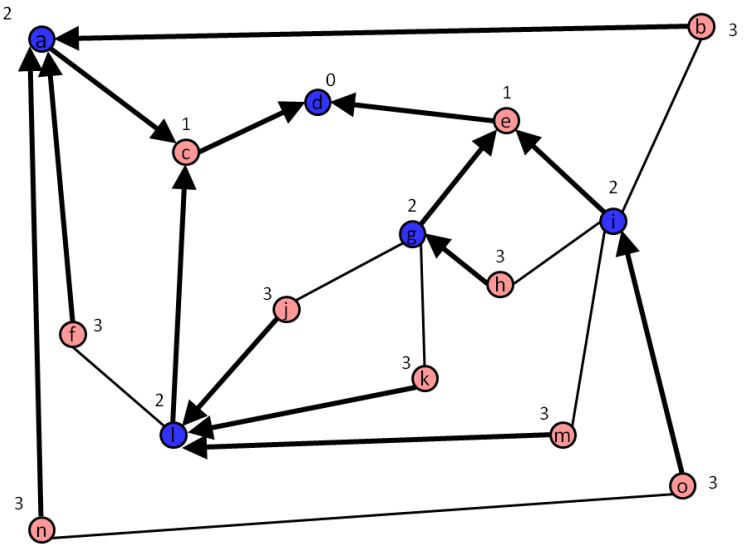
   We claim that the following algorithm proves this claim.

   Run BFS starting from any vertex.
   Let $dist[v]$ denote the distance to vertex v computed by BFS.
   Let $B = \{ v : dist[v] \text{ is even} \}$.
   Let $R = \{ v : dist[v] \text{ is odd} \}$.
   If there is an edge between two vertices in B, or two vertices in R return ''non−bipartite''.
   Otherwise, let all vertices in B be blue, let all vertices in R be red, and return ''bipartite''.

   (a): As an example, run that algorithm on the following graph, where the BFS starts from the labeled "root" (vertex d).



   SOLUTION:



The edge $n - o$ has both endpoints in R, so the output is "non-bipartite".

Let us now discuss correctness of the algorithm.

Case 1: If the algorithm returns "bipartite" then that is obviously correct because the algorithm checked that there are no blue-blue or red-red edges.

Case 2: If the algorithm returns "non-bipartite" then we will find an odd cycle using the BFS tree. Suppose the algorithm finds an edge u-v with both $u, v \in R$. Let w be the least-common ancestor in the BFS tree of u and v. (Meaning that, if you follow the path of parent pointers from u to s and follow the path from v to s, then w is the first vertex where these paths meet.)

Then

$$(dist[u] - dist[w]) \; + \; (dist[v] - dist[w])$$

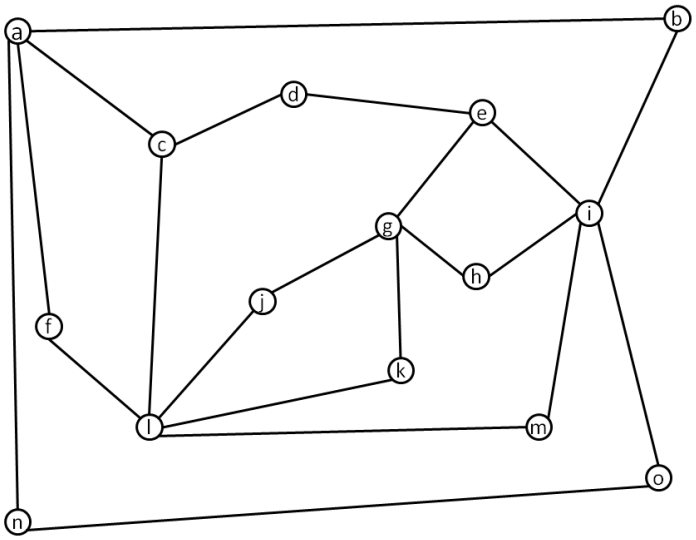
is always even because it equals

$$\underbrace{(dist[u] + dist[v])}_{\text{always even}} \; - \; \underbrace{2 \cdot dist[w]}_{\text{always even}}$$

which is even, because $dist[u]$ and $dist[v]$ are either both even or both odd.

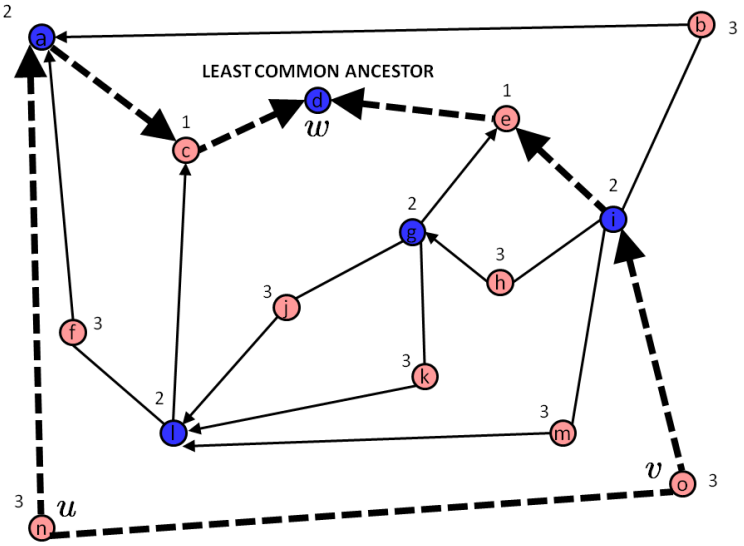
So we obtain an odd cycle by combining the path from u to w, the path from v to w, and the edge u-v.

(b): Let us continue the same example from before:



Look at the edge you found for which both endpoints have the same color. Find the least-common ancestor of its endpoints, and the corresponding odd cycle. How long is your odd cycle?

SOLUTION:



The least-common ancestor w happens to be the root vertex, d. The u-w and v-w paths both happen to have length 3. The odd cycle has length 7.