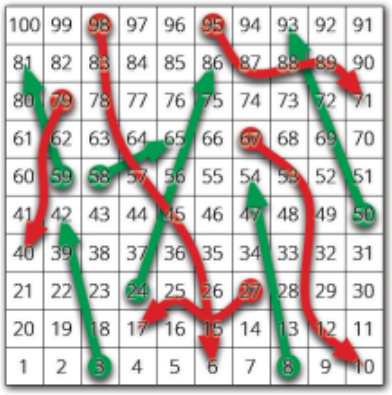
CPSC 320 2023S2: Tutorial 3 Solutions

1 Snakes and Ladders

This classic board game consists of a $g \times g$ 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 (like a dice roll). 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:

Given a Snakes and Ladders board, we will turn it into a *directed* graph, and then run breadth-first search (BFS). A snag is that we haven't defined BFS in directed graphs, so first we will do that.

Recall that for a BFS on an undirected graph, we essentially start at some node, traverse all the adjacent nodes at the present depth, and then move on to traverse nodes at the next depth. To make it work in directed graphs, we simply need to define BFS such that we can only traverse an adjacent node if there is a directed edge towards it from the current node. **In short, this version of BFS can only traverse in the direction of the directed edges.**

Returning to snakes and ladders, we will create a graph has $n = g^2$ vertices. Vertex i corresponds to the square on the board labeled 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.

  For example, suppose there is a snake from grid 10 to grid 1 and we have 3-sided dice so $k = 3$. Then, we will add a directed edge from grid 9, 8, and 7 to grid 1. That is, it is possible to go from grid 9, 8, or 7 to grid 1 within one move depending on what they roll their dice (to reach grid 10).

  Note that we don't add edges from the top to the bottom of a snake. In the game, traversing the snake doesn't count as a "move". It happens automatically when you land at the top of a snake. In other words, you will never start or stay at grid 10. [1]

- 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.

  For each grid, we can advance up to k different grids depending on the dice roll, so we add an edge for each possible roll.

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 Interpreters and Graphs

Several modern programming languages, including JavaScript, Python, Perl, and Ruby, include a feature called **parallel assignment**, which allows multiple assignment operations to be encoded in a single line of code. For example, the Python code `x,y = 0,1` simultaneously sets x to 0 and y to 1. The values of the right-hand side of the assignment are all determined by the **old** values of the variables. Thus, the Python code $a, b = b, a$ swaps the values of a and b, and the following Python code computes the nth Fibonacci number:

```
def fib(n):
    prev, curr = 1, 0
    while n > 0:
        prev, curr, n = curr, prev+curr, n−1
    return curr
```

Suppose the interpreter you are writing needs to convert every parallel assignment into an equivalent sequence of individual assignments. For example, the parallel assignment `a,b = 0,1` can be serialized in either order – either `a=0; b=1` or `b=1;a=0` – but the parallel assignment `x,y = x+1,x+y` can only be serialized as `y=x+y; x=x+1`. Serialization may require one (or even more) additional temporary variables. For example, serializing `a,b = b,a` requires a temporary variable.

Your task is to design an algorithm to determine whether a given parallel assignment can be serialized without additional temporary variables.

To formalize this, say there are n parallel assignments. The i^{th} assignment is of the form "$v_i \leftarrow R_i$", where v_i is a single variable and R_i is an expression involving many variables. For simplicity, you may assume the following:

- Each variable can only appear on the left-hand side of one assignment.
- Computing a right-hand side R_i doesn't have any "side effects". That is, computing R_i does not involve functions that modify variables.

SOLUTION: Create a directed graph as follows.

- For the i^{th} assignment, we create a vertex labeled i.
- If the right-hand side R_i contains variable v_j, we create a directed edge (i, j).

Then we run the topological ordering algorithm on this graph. If the graph is a DAG, we output "no temporary variable needed". If it is not, we output "temporary variable(s) needed". Let us now see why this is correct.

Case 1: Suppose the graph is a DAG, so the algorithm produces a topological ordering. Considering performing the assignments in that order. The parallel assignment would fail only if the old value of some variable v_j gets destroyed before it is used in some right-hand side, say R_i. But because the ordering is produced from the directed graph, assignment i happens *before* assignment j. So the old values of variables are always used before they are destroyed. This ordering correctly executes the parallel assignment.

Case 2: Suppose the graph is *not* a DAG, so it has a directed cycle C. Suppose we were to execute the assignments in some order. Let j be the vertex on cycle C that is the first to be executed under

that ordering. Then its old value is destroyed before any other assignment in C takes place. However, since C is a cycle, an edge (i, j) exists where i is also on the cycle. By choice of j, assignment i is executed after assignment j. Because the edge (i, j) exists, the right-hand side R_i depends on v_j, which has already been destroyed at the time of executing R_i. Since this ordering was arbitrary, *no* ordering correctly executes the parallel assignment.