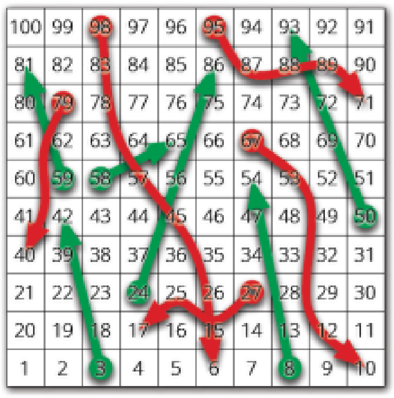
CPSC 320 2020S2: Tutorial 3

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.

Tutorial 3

Ruolin Li 31764160

1. Create a graph $G = (V, E)$

Create vertices: for each ~~vertice~~ spot i, create vertex V_i.

Create edges: Suppose the vertex has the value j, which takes us to V_i

- If V_i is not bottom of a ladder, or tail snake,
  Create edge (V_{i-j}, V_i).
- Otherwise, suppose this goes to V_l. Create edge (V_{i-j}, V_l)

// Create edge

for each vertex $i \geq 2$ and each $1 \leq j \leq \min\{k, i-1\}$, 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 V_{i-j} to V_l.

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's possible that the distance from 1 to n is infinite.

2. Intepreters and Graphs

Create a graph $G = (V, E)$

For the ith assignment, create a vertex labeled i.

If the right-hand side R_i contains variable V_j, we create a directed edge (i, j).

If the graph is DAG, we output no temporary variable needed, the algorithm produces a topological ordering. This algorithm performs the parallel assignment.

Suppose the graph is not DAG. so it has a directed circle C. Suppose we were to execute the assignment in some order. Let j be the vertex on cycle C that is the first to be executed under this 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.

Tutorial 3

Ruolin Li 31764160

1. Create a graph $G=(V,E)$

Create vertices: for each ~~vertice~~ spot i, create vertex V_i.

Create edges: Suppose the vertex has the value j, which takes us to V_i

- If V_i is not bottom of a ladder, or tail snake,
  Create edge (V_{i-j}, V_i).
- Otherwise, suppose this goes to V_l. Create edge (V_{i-j}, V_l)

// Create edge

for each vertex $i \geq 2$ and each $1 \leq j \leq \min\{k, i-1\}$, 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 V_{i-j} to V_l.

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's possible that the distance from 1 to n is infinite.

2. Intepreters and Graphs

Create a graph $G=(V,E)$

For the i^{th} assignment, create a vertex labeled i.

If the right-hand side R_i contains variable V_j, we create a directed edge (i,j).

If the graph is DAG, we output no temporary variable needed, the algorithm produces a topological ordering. This algorithm performs the parallel assignment.

Suppose the graph is not DAG. so it has a directed circle C. Suppose we were to execute the assignment in some order. Let j be the vertex on cycle C that is the first to be executed under this 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.

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.