Zachary Robinson

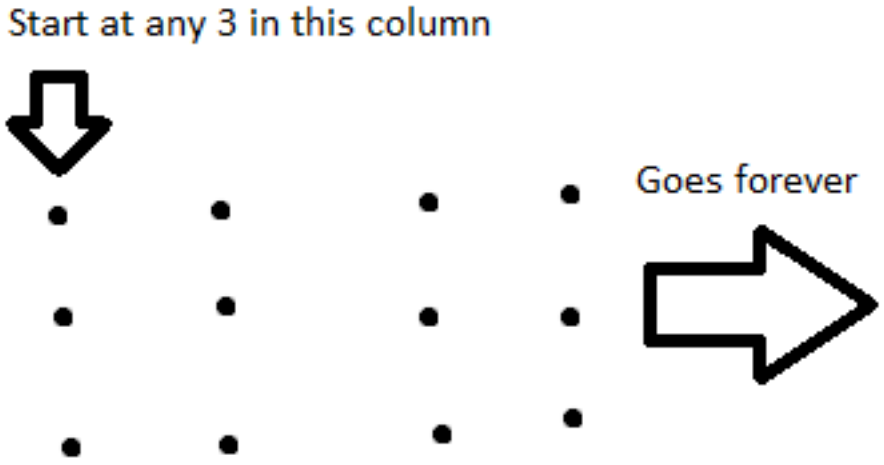
Maximillian vonBlankenburg

CS454

17 May 2022

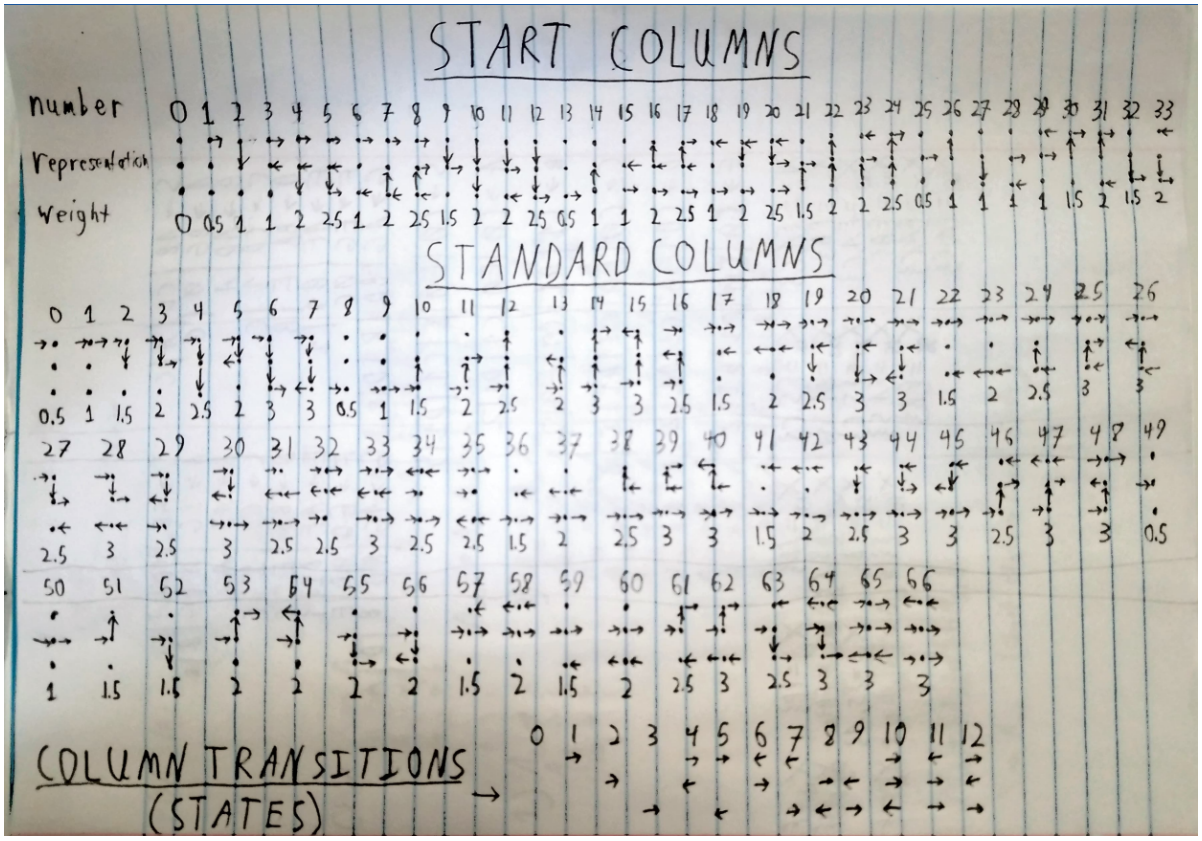
Counting Self Avoiding Walks in a Coordinate Grid

Counting self avoiding walks in an expanding XY coordinate system is a very well known coding exercise for several different variations but a quick overview is that the walks are paths through the grid that must be of a set length and no node may be visited twice in the same walk. Lets modify the problem to be a 3 node tall grid that reaches infinitely only towards the positive x direction with any particular walk able to start from any of the 3 leftmost nodes. So, we have a grid that consists of 3 vertically stacked nodes and we want to count the number of length n walks in our short grid as it expands to the right.



It is important to note that at length n = 0 we trivially say that we have 3 walks of length n. When we go to length n = 1, these 3 starting walks that we know exist will begin expanding outwards with the top and bottom corners having 2 possible movements each and the middle node having 3 possible movements. Therefore, for length n = 1 we can clearly see that there are 7 possible walks. The problem grows exponentially and by n = 4 were already looking at 57 different possible walks.

To count the number of length N walks in linear time in our grid of length N, we will build a DFA based on all the possible "slices" that can possibly occur in each rightward step in our grid. We separated these slices into two categories: beginning slices (each of which our DFA can start at) and regular slices (which make up every slice after the first slice). Each of our "walks" then consisted of a beginning slice followed by a number of regular slices up to N. There were many possible ways for us to encode these columns, like encoding each possible line as a bit to represent if it was there or not. We ended up enumerating over all possible columns and throwing out the columns that were impossible to encounter given our rules (for instance, columns containing nodes with more than two lines connected to them are impossible). Once we found the set of all valid columns, we assigned each one a decimal index and encoded them that way. We ended up counting 34 possible beginning slices and 67 possible regular slices (columns = slices). The full table we used for reference is shown below.



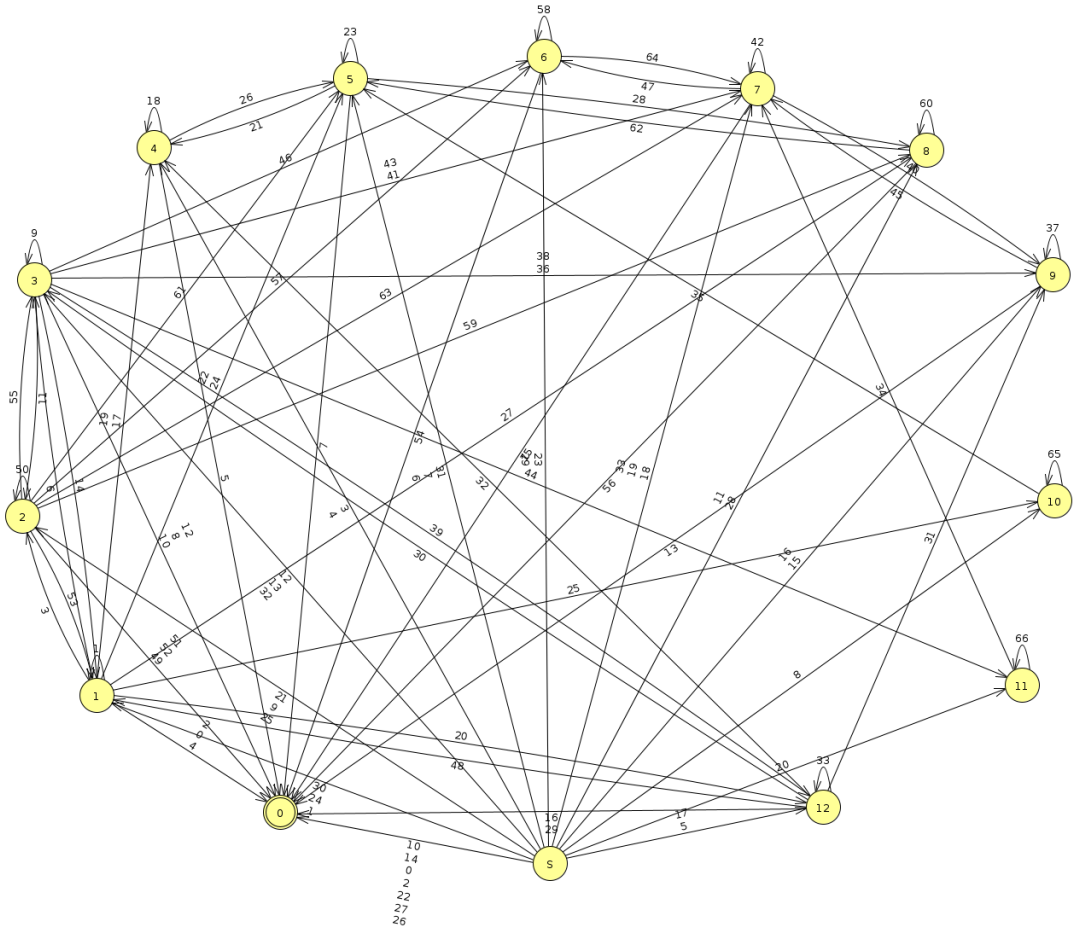
In the above image, the arrows indicate the direction a line is being drawn. We also made a list of all valid “transition sets” between slices, for later use when building our DFA. As an example, the transition between start column 7 and standard column 7 is 5. This is the logical process we used to build our DFA.

In terms of time complexity, our program runs in linear time or O(n). No matter the input value for n, we will always be looking at n + 1 slices and the transitions between these slices. Therefore, the number of computations per loop of the count algorithm is always the same. Although the coefficient for how many operations are made in linear time is over 100, we can still reach values like n > 200 since we are running in linear time.

Also listed under each column is a number between 0 and 3, which is the line weight of that column. Since we’re counting the number of walks, not the number of slices, we need to keep track of how many line segments each slice has. Line segments that go between slices have

a weight of 0.5, while line segments going between nodes have a weight of 1. For example, column 42 has a weight of 2 because it has four line segments of weight 0.5. This information is not used in our DFA, but it's vital for our program to keep track of the length of each walk it counts.

For our DFA we defined each column as a transition, and each state as a possible transition between columns, as shown in our table. Our start state would read in a beginning slice, and move to one of 13 states. Then each subsequent state would read in a regular slice and move to another one of the 13 states. The accepting state of the DFA represents the 0 transition, which has no lines between the columns and indicated an end of input. Our full DFA is shown below.



Once we had this DFA, writing the counting algorithm was trivial. We just had to made some additions to the transition table to account for the weights. Our startTable and startTransitions arrays handled every transition from state S to one of the other states. mainTransitions mapped each state to a transition, columnTable gave the weight of each transition, and endTransitions mapped each transition to its new state. In hindsight these data structures could have been combined in a way that was more straightforward, but we wanted to break them up to ensure that we entered all the data correctly. In this way, the structure of our program could be improved, but this does not take away from our efficiency or final result, so we did not consider it a priority.

There does not seem to be an existing valid solution that is widely public to this computation problem so we set up a simpler method with a much higher time complexity that should brute force every possible walk with a flood fill algorithm and then check if each walk is self avoiding to compare against our DFA method. The two methods match inputs and outputs up until n = 5 where the DFA method returns 117 walks and the flood fill method returns 115 walks. We spent a good amount of time trying to resolve this discrepancy, but could not find a solution. The flood fill algorithm does not seem to have any bugs, and our more efficient solution has been triple-checked for errors in the tables. Because of this, we are unsure as to whether the flood fill is undercounting, or the DFA method is overcounting. The only way to know for sure to arrive at a concrete solution seems to be writing out 115-117 walks by hand which could introduce human errors in of itself. We also did not have time to hand write the walks by the time that it was understood that we would not have test cases to run our output against.

In conclusion, we are unsure if our solution is 100% accurate to the statement of the problem but we are confident that the methodology used can answer the question accurately. Our theory and algorithms are sound and if executed correctly should produce the correct output. With that, we can conclude, with a high degree of certainty, that the problem of counting the number of walks of length n on a 3 by infinity grid can be solved in O(n) time.

Sample inputs/outputs:

N = 0-9

[3, 7, 13 ,29, 57, 117, 225, 453, 873, 1721]

N = 50

[915939268538589]

N = 100

[177922618601041086378744713465]

N = 1000

[27592508662441206911477857470262359295875689216790314805624573243833707004835110308604265581926636223380928716914670953365916443078073859908839030435084638292342431963478845305196982215915917182655966293049027709670653152410548791532085062731056443094225139374875558181546912109561135881]

Source code:

```
# Author(s)
#   Max vonBlankenburg
#   Zachary Robinson
import sys

# Each index is a unique starting column, stores the weight of each
```

```
startTable = [0, 0.5, 1, 1, 2, 2.5, 1, 2, 2.5, 1.5, 2, 2, 2.5, 0.5, 1, 1,
2, 2.5, 1, 2, 2.5, 1.5, 2, 2, 2.5, 0.5, 1, 1, 1, 1, 1.5, 2, 1.5, 2]
# Transition table for each starting column (there are 13 valid states)
startTransitions = [0, 1, 0, 4, 4, 12, 5, 5, 10, 2, 0, 8, 3, 3, 0, 9, 9,
12, 7, 7, 11, 2, 0, 6, 1, 2, 0, 0, 8, 6, 1, 5, 3, 7]
# Transition from startTransitions to columns other than the starting
column, incomplete
mainTransitions =  [[], [0,1,2,3,4,6,17,19,20,22,24,25,27],
[49,50,51,52,53,55,57,59,61,63], [8,9,10,11,12,14,36,38,39,41,43,44,46],
                    [5,18,26], [7,21,23,28], [54,58,64], [15,40,42,47],
[56,60,62], [13,37,45], [35,65], [34,66], [16,29,30,31,32,33,48]]
# Stores the weight of each standard column (0-66)
columnTable       =
[0.5,1,1.5,2,2.5,2,3,3,0.5,1,1.5,2,2.5,2,3,3,2.5,1.5,2,2.5,3,3,1.5,2,2.5,3
,3,2.5,3,2.5,3,2.5,2.5,3,2.5,2.5,1.5,2,

2.5,3,3,1.5,2,2.5,3,3,2.5,3,3,0.5,1,1.5,1.5,2,2,2,2,1.5,2,1.5,2,2.5,3,2.5,
3,3,3]
# Transition table for each standard column (there are 13 valid states)
endTransitions =
[0,1,0,2,0,0,3,0,0,3,0,2,0,0,1,0,0,4,4,4,12,4,5,5,5,10,5,8,8,0,3,9,4,12,7,
5,9,9,9,12,9,7,7,7,11,7,6,6,1,0,2,0,0,1,0,3,0,6,6,8,8,5,5,7,7,10,11]



# We define our coordinate system as a 3 by infinity grid,
# with starting point p at any point farthest left (0,-1) (0,0) (0,1)
# grid goes from (0,-1) to (inf,1)
# We will define a character of our language as a unique arrangement
# of lines in a single column of the coordinate system. The set of
# characters in the starting column is separate from the rest of the
columns,
# and is dealt with separately.
# The first transition of our "DFA" involves reading in the start
# character and choosing a state based on the "output arrows".
# Every transition following that moves to a different one of these
# states.
# Every transition will also have a weight, depending on the number of
# arrows in the character involved in the transition.
class gridwalk:
    length = 0
```

```
    validSet = [] # In progress
    newSet = [] # In progress

    def __init__(self, length):
        # Each index in validSet represents a DFA state.
        # There are 13 states (not counting the start state), each
representing a possible set of paths between each column.
        # Each value of validSet is an array.
        # Each index of this array represents a weight.
        # Each value of this array represents the number of possible paths
resulting in that weight.
        if self.length < 0:
            return "Invalid Length"
        else:
            self.length = length
            for i in range(13):
                self.validSet.append([0 for it in range((length * 2) +
1)]) # Max weight we care about = length of target walk
            for i in range(len(startTable)):
                if startTable[i] <= self.length:
                    self.validSet[startTransitions[i]][int(startTable[i] *
2)] += 1

    # Calculates the number of valid walks of length "length"
    def walk(self):
        if self.length == 0:
            return 3
        runningTotal = 0
        # Our oracle here is validSet[0], which is the final state of the
DFA
        # Everything ending at validSet[0] with length "length" is a walk
we want to count
        runningTotal += self.validSet[0][self.length * 2]
        # We continue this until there is no weight <= length * 2 in the
final state.
        while True:
            self.updateSet()
            flag = False
            for i in self.validSet[0]:
                if i != 0:
```

```
                    flag = True
            if flag == False:
                break
            runningTotal += self.validSet[0][self.length * 2]

        return runningTotal

    # Simulates the reading in of a new character, and updates validSet.
    def updateSet(self):
        self.nextSet = []
        for i in range(13):
            self.nextSet.append([0 for it in range((self.length * 2) +
1)])
        for i in range(len(self.validSet)):
            for j in range(len(self.validSet[i])):
                for k in range(len(mainTransitions[i])):
                    weight = columnTable[mainTransitions[i][k]] # The
weight of the transition
                    if (j / 2) + weight <= self.length:           # We
only care about counting weights up to the desired walk length
                        index = endTransitions[mainTransitions[i][k]] #
The state index that the transition leads to
                        self.nextSet[index][int(j + (weight * 2))] +=
self.validSet[i][j]
        self.validSet = self.nextSet[:]



def main():
    args = sys.argv[1:]
    if len(args) != 1:
        print("Format: python3 avoiding_my_responsibilities.py <length>")
    walker = gridwalk(int(args[0]))
    count = walker.walk()
    print(count)

if __name__ == "__main__":
    main()
```