Brandon Adamson-Rakidzich
Warren Goodson

CS 454 THEORY OF COMPUTATION
Final Project Report

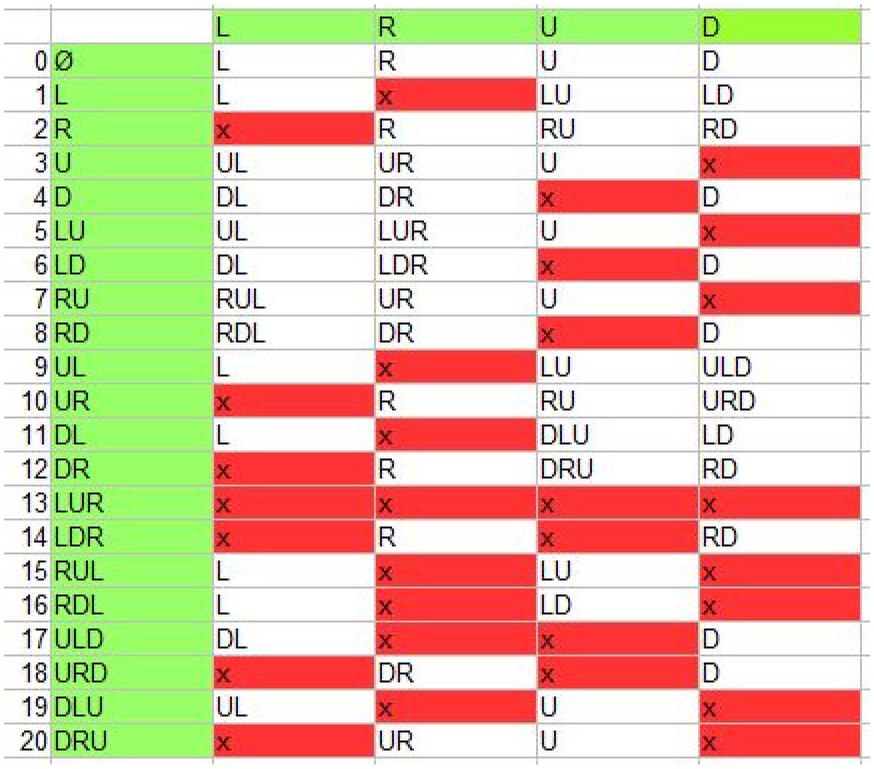
Counting 2-Dimensional Self-Avoiding Walks

Problem Statement:

A self-avoiding walk is a walk of length n through a 2d plane that has no cycles equal to or smaller than some length k. Counting the number of such walks for any given n and k is an interesting problem because the difficulty of the counting increases exponentially in k. Our goal was to take the paper “Improved Upper Bounds for Self-Avoiding Walks in Zd” and use the information in it in order to efficiently count the number of self-avoiding walks in various k’s.

Methodology:

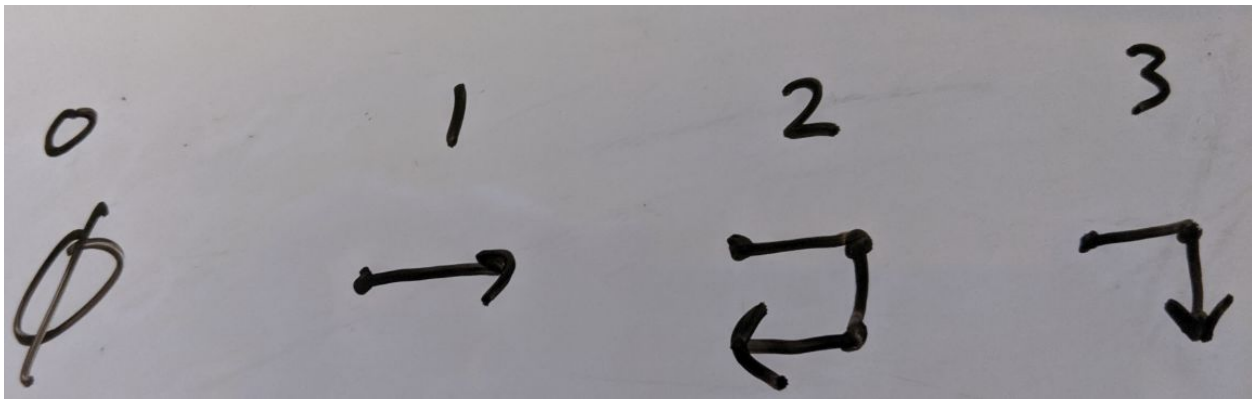
We started with counting walks for k = 4. To get our feet wet, we created a DFA to count the number of walks using the standard type of DFA construction that we’ve used in class so far. We encoded movements as L, R, U, and D for Left, Right, Up, and Down, and we used the states of the DFA to encode the last 3 moves. To reduce the number of states, we didn’t include states like LL or ULU, because the the first moves in both of these states don’t contribute towards a loop of length 4. We only tracked the last 3 *relevant* moves: Moves that will produce a unique loop within k steps. The transition table for this DFA follows



		L	R	U	D
0	Ø	L	R	U	D
1	L	L	x	LU	LD
2	R	x	R	RU	RD
3	U	UL	UR	U	x
4	D	DL	DR	x	D
5	LU	UL	LUR	U	x
6	LD	DL	LDR	x	D
7	RU	RUL	UR	U	x
8	RD	RDL	DR	x	D
9	UL	L	x	LU	ULD
10	UR	x	R	RU	URD
11	DL	L	x	DLU	LD
12	DR	x	R	DRU	RD
13	LUR	x	x	x	x
14	LDR	x	R	x	RD
15	RUL	L	x	LU	x
16	RDL	L	x	LD	x
17	ULD	DL	x	x	D
18	URD	x	DR	x	D
19	DLU	UL	x	U	x
20	DRU	x	UR	U	x

Using this transition table, we can use dynamic programming to write a function that will give us the number of walks length n that don't have cycles of length 4 or less.

Next we used the method described in the paper to create a more efficient DFA. The key concept in the approach from the paper is recognizing that there is a lot of room to make cuts to the number of states. Walks like URD and DRU are functionally identical when it comes to counting self-avoiding walks, so there's no need to track them separately in our DFA. Instead, we can recognize that there are certain shapes that will be produced as walks are formed, and keep track of those shapes which can lead to a loop of length 4 or less. This gives us generalized states. These states are encoded as shown in the image below:

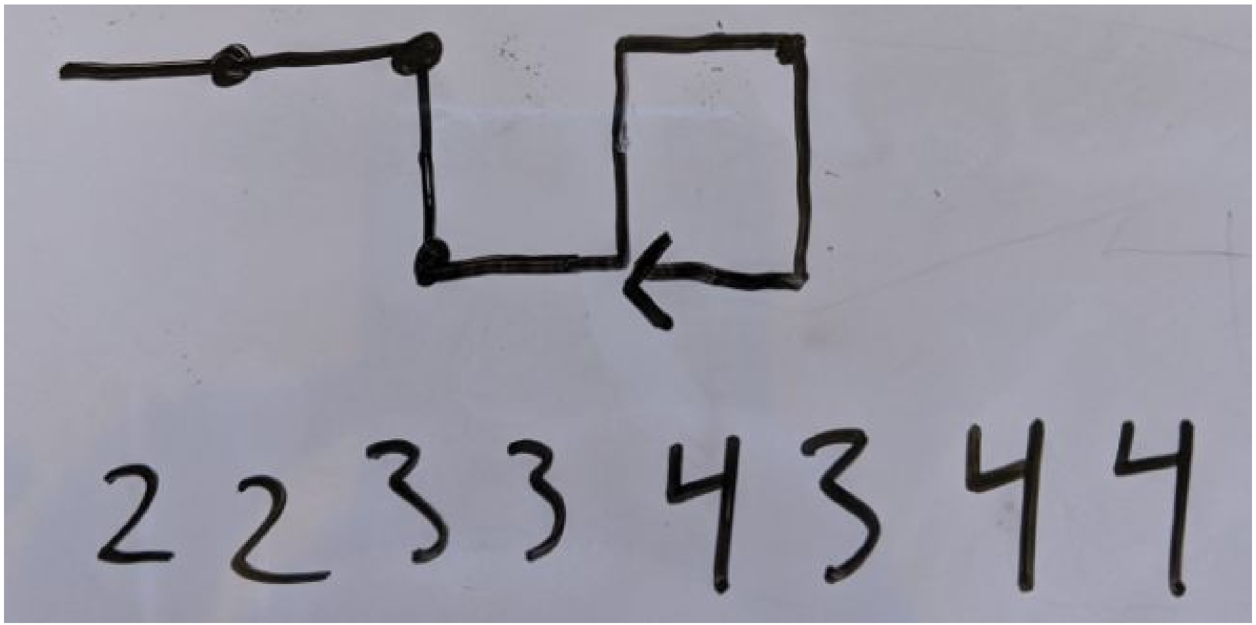


To then generalize the moves, we divided them into 4 different types.

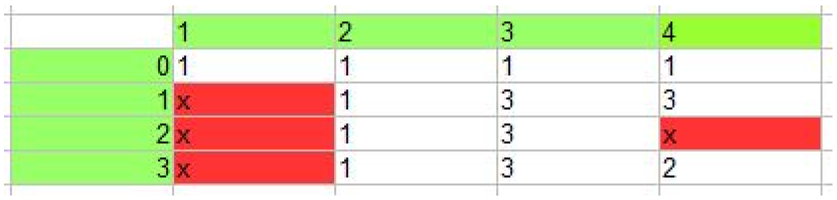
1. Moves that go opposite the previous move
2. Moves that repeat the previous move
3. Moves that turn into a new axis
4. Moves that turn into the a new axis along the opposite direction last turned (loop closers)

1 is a move in any direction at the start of the walk, and 3 and 4 are equivalent before a new axis has been turned into during the walk.

Here's an image showing an example string and the walk it produces



Using this encoding for the states and input gives us a much simpler DFA for k = 4



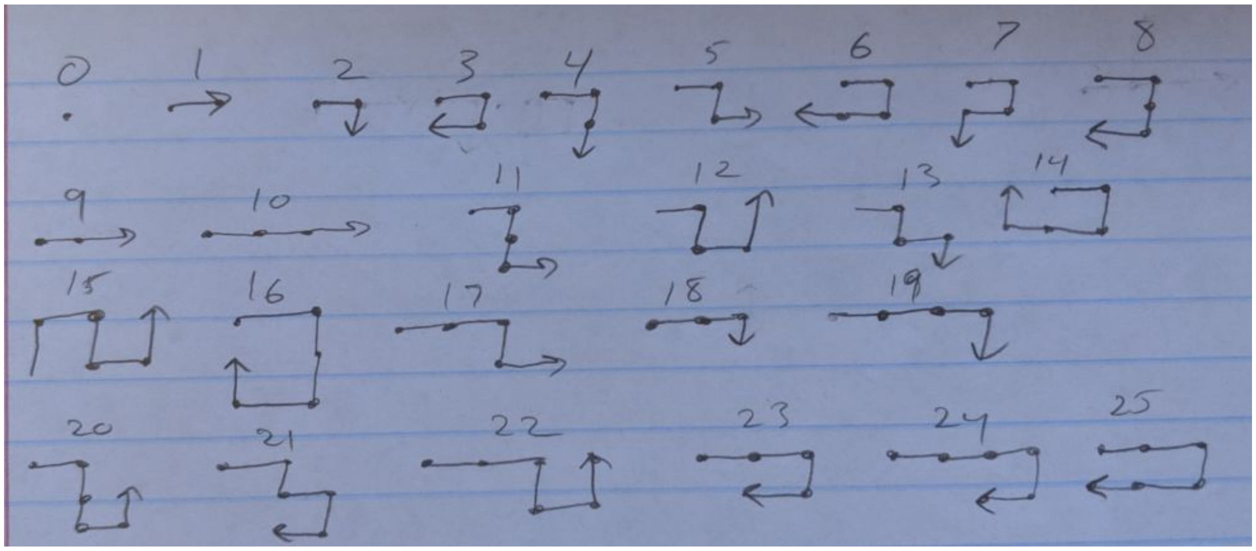
	1	2	3	4
0	1	1	1	1
1	x	1	3	3
2	x	1	3	x
3	x	1	3	2

We implemented a function for counting the number of self-avoiding walks for any n using this DFA as well, and confirmed that the numbers matched the previous DFA.

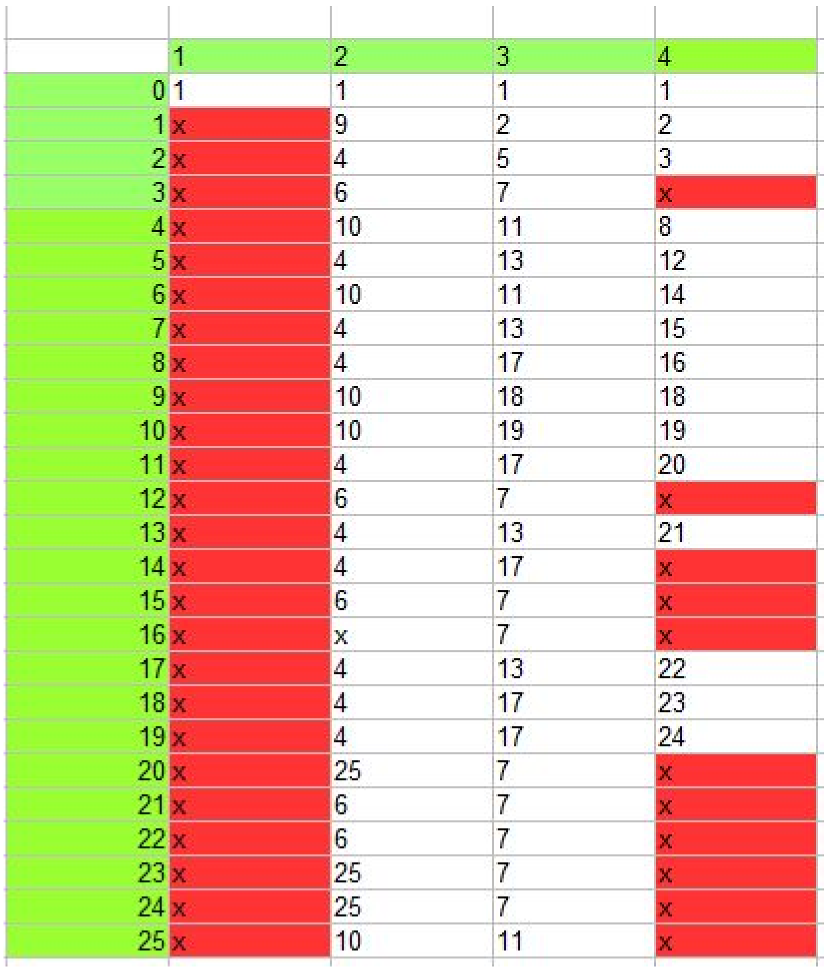
To go beyond the DFA given to us by the paper we decided to expand this algorithm to k = 6. The same principles apply, but it requires a bit more construction. The states needed aren't as obvious as they are for k = 4, and so require some thinking. Pönitz and Tittmann describe the method to do this:

1. Create a list of states to consider, initially empty except for state 0 (the empty walk)
2. Take a random state s from the list
3. Try every input from s and see if it produces a walk that could lead to a loop in 6 or fewer steps
4. If it doesn't, then remove steps from the beginning of s until we get to such a walk
5. If we now have a walk shape we haven't seen yet, create a new state representative of this walk and add it to our list of states to to consider
6. Go back to 2 and repeat until we have no more states to consider

This algorithm is somewhat complicated at first glance and took us a deal of time to understand, but once we understood it we found that implementing it is straightforward. The result was these states:



And the resulting transition table:



	1	2	3	4
0	1	1	1	1
1	x	9	2	2
2	x	4	5	3
3	x	6	7	x
4	x	10	11	8
5	x	4	13	12
6	x	10	11	14
7	x	4	13	15
8	x	4	17	16
9	x	10	18	18
10	x	10	19	19
11	x	4	17	20
12	x	6	7	x
13	x	4	13	21
14	x	4	17	x
15	x	6	7	x
16	x	x	7	x
17	x	4	13	22
18	x	4	17	23
19	x	4	17	24
20	x	25	7	x
21	x	6	7	x
22	x	6	7	x
23	x	25	7	x
24	x	25	7	x
25	x	10	11	x

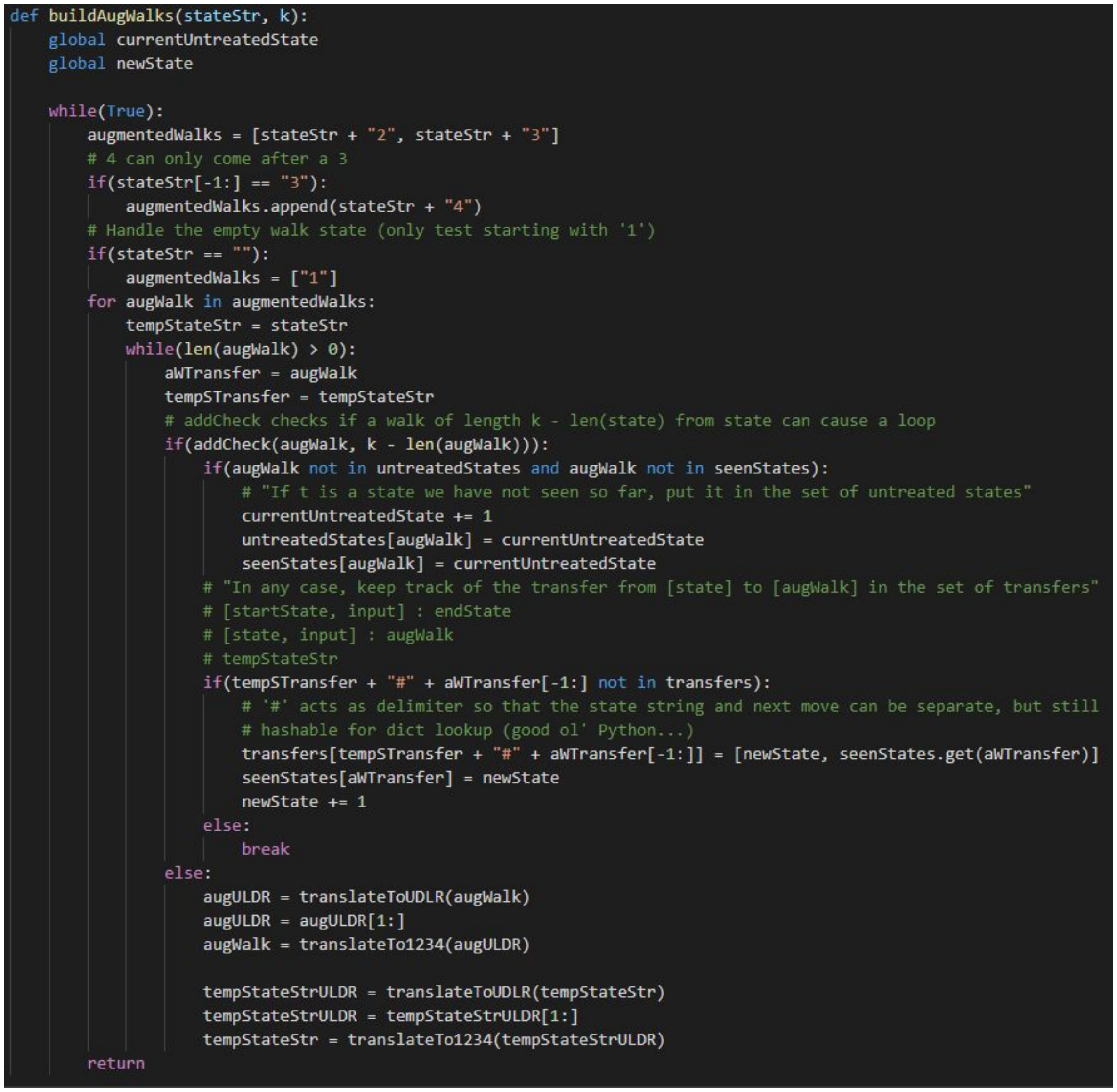
This table is a lot bigger than for k = 4, but much smaller than it would have been if we used states to encode specific walk patterns instead of generalized ones.

We made a function for this DFA as well and found that the count it gave was consistent with our expectations. It was the same as the k = 4 DFA for an n < 6, and it was less than the k = 4 DFA for all other n. This is to be expected, because for n < 6 there will be no cycles of length 6, so only cycles of 4 or less will exist. And for n >= 6, the number should be smaller because as k increases we become more restrictive with what we count.

Implementation of DFA Generation Algorithm

We then began to implement the aforementioned algorithm in order to generate DFA’s for n > 6. This was done using standard Python 3.7. The details of the implementation are best gathered by examining the code, but we will share some of the more notable choices here.

We thought that it would be best to stay with our step encoding of 1, 2, 3, and 4 because it would eliminate many iterations of the algorithm by utilizing the symmetry of these kinds of walks. In order to do this while still examining all possible future walks of up to a certain length, translation functions were formulated. One translates from the absolute notation (using symbols ‘U’, ‘D’, ‘L’, and ‘R’ for up, down, left, and right, respectively) to our 1234 encoding. The other does the opposite, translating from 1234 encoding to absolute encoding. Since translating from 1234 to UDLR has four possible orientations, we chose to always have the first step be an R. The following code represents steps 2 through 4 of the algorithm as described by Pönitz and Tittmann:

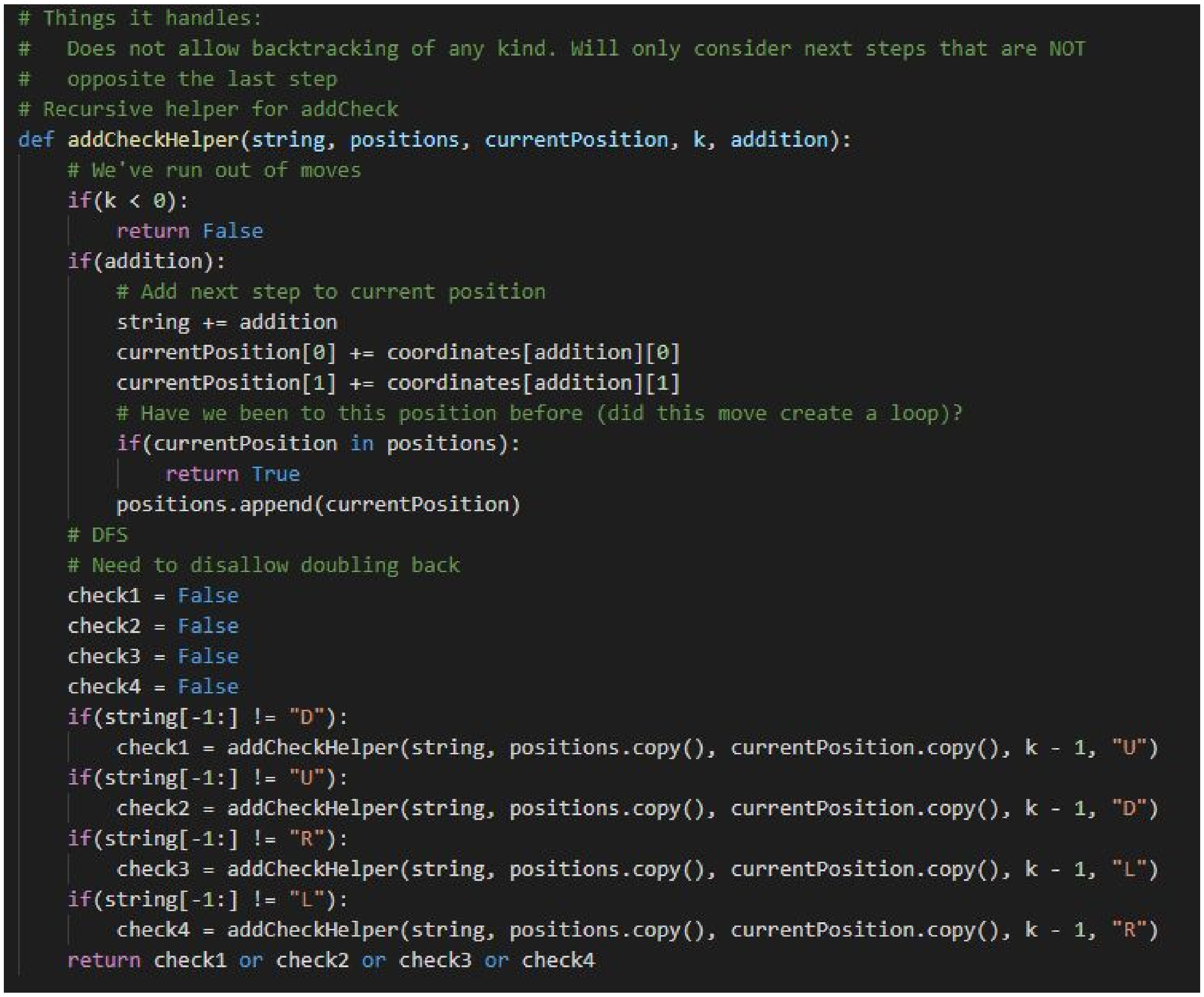


```python
def buildAugWalks(stateStr, k):
    global currentUntreatedState
    global newState

    while(True):
        augmentedWalks = [stateStr + "2", stateStr + "3"]
        # 4 can only come after a 3
        if(stateStr[-1:] == "3"):
            augmentedWalks.append(stateStr + "4")
        # Handle the empty walk state (only test starting with '1')
        if(stateStr == ""):
            augmentedWalks = ["1"]
        for augWalk in augmentedWalks:
            tempStateStr = stateStr
            while(len(augWalk) > 0):
                aWTransfer = augWalk
                tempSTransfer = tempStateStr
                # addCheck checks if a walk of length k - len(state) from state can cause a loop
                if(addCheck(augWalk, k - len(augWalk))):
                    if(augWalk not in untreatedStates and augWalk not in seenStates):
                        # "If t is a state we have not seen so far, put it in the set of untreated states"
                        currentUntreatedState += 1
                        untreatedStates[augWalk] = currentUntreatedState
                        seenStates[augWalk] = currentUntreatedState
                    # "In any case, keep track of the transfer from [state] to [augWalk] in the set of transfers"
                    # [startState, input] : endState
                    # [state, input] : augWalk
                    # tempStateStr
                    if(tempSTransfer + "#" + aWTransfer[-1:] not in transfers):
                        # '#' acts as delimiter so that the state string and next move can be separate, but still
                        # hashable for dict lookup (good ol' Python...)
                        transfers[tempSTransfer + "#" + aWTransfer[-1:]] = [newState, seenStates.get(aWTransfer)]
                        seenStates[aWTransfer] = newState
                        newState += 1
                    else:
                        break
                else:
                    augULDR = translateToUDLR(augWalk)
                    augULDR = augULDR[1:]
                    augWalk = translateTo1234(augULDR)

                    tempStateStrULDR = translateToUDLR(tempStateStr)
                    tempStateStrULDR = tempStateStrULDR[1:]
                    tempStateStr = translateTo1234(tempStateStrULDR)
        return
```

In order to check whether a walk could loop in the right amount of steps, the steps were encoded in ordered pairs corresponding to their coordinate changes in a two-dimensional grid (‘U’ is equivalent to [0, 1], ‘R’ to [1, 0], etc). We used a recursive depth-first search approach in order to enumerate all possible walks, decreasing the allowed length by one at each recursive level. The search returns true if any such walk successfully loops, false otherwise. Here is the code:



```python
# Things it handles:
#   Does not allow backtracking of any kind. Will only consider next steps that are NOT
#   opposite the last step
# Recursive helper for addCheck
def addCheckHelper(string, positions, currentPosition, k, addition):
    # We've run out of moves
    if(k < 0):
        return False
    if(addition):
        # Add next step to current position
        string += addition
        currentPosition[0] += coordinates[addition][0]
        currentPosition[1] += coordinates[addition][1]
        # Have we been to this position before (did this move create a loop)?
        if(currentPosition in positions):
            return True
        positions.append(currentPosition)
    # DFS
    # Need to disallow doubling back
    check1 = False
    check2 = False
    check3 = False
    check4 = False
    if(string[-1:] != "D"):
        check1 = addCheckHelper(string, positions.copy(), currentPosition.copy(), k - 1, "U")
    if(string[-1:] != "U"):
        check2 = addCheckHelper(string, positions.copy(), currentPosition.copy(), k - 1, "D")
    if(string[-1:] != "R"):
        check3 = addCheckHelper(string, positions.copy(), currentPosition.copy(), k - 1, "L")
    if(string[-1:] != "L"):
        check4 = addCheckHelper(string, positions.copy(), currentPosition.copy(), k - 1, "R")
    return check1 or check2 or check3 or check4
```

Pre-defining relationships between steps was also extremely useful. In particular, dictionaries of both opposite steps and right turns were pre-defined in global scope for quick reference.

In the end, we were not able to successfully create a DFA from this algorithm. However, the algorithm as we implemented it does generate correct state transitions between those states that successfully loop in k (supplied by the user) total steps, and will enumerate every possible unique sequence of our 1234 encoding corresponding to these states, and therefore also count them. This correctness was verified via comparison with our tables for k = 4 and k = 6. We included the option to output these strings to a text file, because the count grows exponentially in k, and having 190,630 walks of length 16 or less fill your terminal can be somewhat inconvenient.

References

Pönitz, André and Peter Tittmann: *Improved Upper Bounds for Self-Avoiding Walks in Zd,* The Electronic Journal of Combinatorics, 2000