Recognizing Self-Avoiding Walks in a $3 \times n$ Square Lattice

Sean Robbins, Jacob Combs, Matthew Hardwick

December 9, 2013

Contents

1 Introduction

Counting the number of self-avoiding walks in a hypercubic lattice is an open problem in combinatorics. Previous work in this area utilized finite automata to count walks beginning at the origin in a d-dimensional hypercubic lattice $\mathbb{Z}^d$, an infinite space [4].

In this report, we solve the simpler problem of identifying self-avoiding walks in a restricted (finite) lattice of size $3 \times n$ for finite n. We develop a method for encoding the lattice as input to a finite automaton, and design the automaton that recognizes self-avoiding walks over this input. We illustrate the correctness of the automaton with several test cases.

2 Problem statement

In graph theory, a *walk* is a sequence of edges connected by vertices. A *self-avoiding walk* is one that can be traced from start to finish without tracing over the same vertex more than once. Put

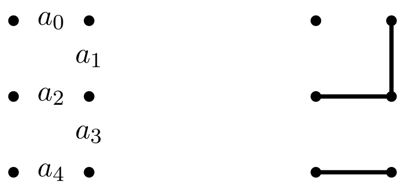

Figure 1: *Left:* Positions of each of 5 bits in the binary string $a = a_0a_1a_2a_3a_4$, a symbol in the alphabet; $a_i = 1$ if there is an edge in position a_i, and 0 otherwise. *Right:* Example input symbol $a = 01101$ as it appears on the lattice.

another way, the a walk w is self-avoiding if w does not have any self loops.

A finite automaton can be used to process the set of edges $3 \times n$ square lattice. We develop a set of symbols that can be used to encode any such lattice; this symbol set constitutes the input alphabet of the finite automaton that recognizes lattices that contain a self-avoiding walk.

This finite automata accepts $3 \times n$ square lattices that meet the following criteria:

1. The edges of the lattice represent a self-avoiding walk.
   (a) Walks in the lattice do not *jump*. That is, there are exactly two ends, or vertices of degree 1.
   (b) Walks in the lattice do not *branch*. That is, no vertex has degree greater than 2.
   (c) There is no *closed loop* in the lattice.
2. A walk begins at the origin (the leftmost, bottom point) of the lattice. This condition not only simplifies the design of the automaton, but also ensures that if the automaton is used to count self-avoiding walks, double-counting the horizontal translations or vertical reflections of the same walk is avoided.

3 Solution overview

Ultimately, we will design a deterministic finite automaton (DFA) $M = \langle Q, \Sigma, \delta, q_0, F \rangle$, where Q is the state set, Σ is the input alphabet, $\delta : Q \times \Sigma \to Q$ is the transition function, q_0 is the initial state, and $F \subseteq Q$ is the set of final states. We begin by discussing our encoding of the lattice, follow that with a discussion of our concept of state, and conclude this section with an overview of the transition function δ.

3.1 Encoding: the alphabet

First, we define the alphabet Σ. A finite lattice is represented as a sequence of 5-bit binary strings as described in Figure 1. We encode the lattice left-to-right. Note that this means for the first (leftmost) symbol $a = a_0a_1a_2a_3a_4$, it must be the case that $a_0 = a_2 = a_4 = 0$, because edges that correspond to these three symbols do not appear on the lattice.

3.2 State

The finite automaton reads input symbols over the alphabet described in Section 3.1, and determines if they represent a valid self-avoiding walk as defined in Section 2. A state in this automaton is represented by a 4-tuple $q = \langle t, m, b, c \rangle$:

- Elements t, m, and b represent the degrees of the top, middle, and bottom (respectively) of the rightmost vertices of the input so far. (Notice that these values depend only on the most recent input symbol.) Because self-avoiding walks may not have branches, we are only interested in states for which $t, m, b \in \{0, 1, 2\}$. Provided the input is a valid walk, the number of ones among t, m, and b represents the number of vertices that the next input symbol may connect to. We categorize states by this characteristic:
  - If a state has a single 1-degree vertex, then we refer to it as *single-ended.*
  - If a state has exactly two 1-degree vertices, then we refer to it as *double-ended.* Notice that if we are in a double-ended state, we must have already seen both ends of the valid walk (because one end is at the origin and to induce a second would otherwise require a branch, which is not allowed).
  - If a state has three 1-degree vertices, then we refer to it as *triple-ended.* For the purpose of identifying closed loops, we require an additional bit of state in the case of triple-ended states because two possible pairs of states may have already been connected; see the description of the c element.
  - If a state has *no* 1-degree vertices, then we have finished processing the walk and any further input should be trivial in order for the lattice to be valid.
- Element c represents closure information. If the state is single- or double-ended, then c is not needed (so let $c = 0$ in those cases); if the state is triple-ended, c will be either 1 or 2 depending on which pair of vertices is connected:
  - If the middle and bottom vertices are connected, then let $c = 1$.
  - If the top and middle vertices are connected, then let $c = 2$.

In addition to the states that can be described in the manner outlined above, we will use three auxilliary states: *'start'*, *'done'*, and *'dead'*. We present the complete state set Q in Table 1.

3.3 Transition function

In this section, we present an overview of the algorithm behind the transition function δ. For the implementation details, see Section 4.

There are only a four valid inputs for the first input symbol (because only two edges represented actually appear on the lattice); furthermore, because we require that walks start at the origin, we must ensure that there is no more than one edge connected to that vertex. Therefore, we first address the case that the input state is *'start'* by handling each of the four inputs directly.

At this point, we should decide where to transition on trivial input (i.e. the input is $a = 00000$). From an accepting state (*'done'* or any single-ended state), there should be a transition to the

Single-ended	Double-ended	Triple-ended	Auxilliary
1000	1010	1111	***start***
1200	1210	1112	***done***
1220	1100		*dead*
0100	1120		
2100	0110		
0120	2110		
0010			
0210			
2210			

Table 1: The state set Q, categorized by how many 1-degree vertices there are in the set of rightmost vertices of the input so far. Final states are represented in bold, and include all the single-ended states as well as the *'start'* (accept walks of length 0), and *'done'* states. (The delta function outlined in Section 3.3 may be used to verify that every state listed here is reachable from the start state.)

'done' state; on trivial input from any non-accepting state, there should be a transition to the *'dead'* state.

The *'dead'* state is just that: dead; so it has a self-loop on any input.

Now consider the would-be next state $q' = \langle t', m', b', c' \rangle$. Calculating t', m', and b' is straightforward, as these are only dependent on the input symbol. If these values indicate that the next state will *not* be a triple-ended state, then we may set $c' = 0$. However, if $t' = m' = b' = 1$, then we could be in a triple-ended state, so we need to set c' to 1 or 2 based on the criterion described in Section 3.2. If q is a triple-ended state, then the connectedness of q' is the same as that of q, so $c' = c$ (this is because connectedness cannot switch between two triple-ended states without an intermediate single-ended state). If q is a single-ended state, then the connectedness of q' depends on which vertex is connected to the origin (this information is represented by c).

Once the possible next state q' has been calculated, we need to check that the lattice could still contain a valid self-avoiding walk. Conditions that indicate a transition to the *'dead'* state are enumerated in the order that we check them below.

1. Check for a *branch.* Calculate the new degree for vertices of q, considering any degree added by the current input symbol. If any of these vertices, or any vertex of q', has degree higher than 2, then transition to the *'dead'* state.
2. Check for a *horizontal jump.* Given input $a = a_0a_1a_2a_3a_4$, if $a_0 = a_2 = a_4 = 0$, but at least one of a_1 and a_3 is 1, then transition to the *'dead'* state.
3. At any point while processing a valid input lattice (if we have not yet reached the *'done'* state), there will be a single degree-1 vertex (an "end") that is connected to the origin. The *vertical jump* case is characterized by an input that renders this running path unreachable. To check for this, we consider three separate cases:
    - If q is single-ended, then check that its degree-1 vertex connects to the input symbol. If it does not, then transition to the *'dead'* state.
    - If q is double-ended, then check that *both* of the "ends" of q are connected to the input

symbol. If this is not the case, then transition to the *'dead'* state.

- If q is triple-ended, ensure that if a vertex of q is not connected to the input symbol, that it is *not* the vertex connected to the origin. If the vertex connected to the origin is not connected to the input symbol, then transition to the *'dead'* state.

4. Check for a *closed loop*. If q is a triple-ended state and bits of the input connect two vertices that are marked (by c) as already connected, transition to the *'dead'* state.

At this point, we know the input lattice could represent a valid self-avoiding walk. Therefore, if $q' \in Q$ then we transition to q'. If $q' \notin Q$, then it must be the case that we have finished processing the walk (if not the entire lattice) because the rejection conditions have not been met, but there is no available vertex to connect to; from this point, we should see only trivial input so we transition to the *'done'* state.

4 Implementation details

4.1 Transition function

In Listing 1, we present the python module that contains documented code for the transition function outlined in Section 3.3. As the module documentation explains, this code is intended for use with David Eppstein's Python Algorithms and Data Structures (PADS) library [2], which we used to test it.

This code is also available with additional test code and documentation as a public git repository [1].

Listing 1: Implementation of transition function δ in Python.

```
"""
This module contains the transition function and state lists for the DFA that
finds self-avoiding walks in a 3xN Lattice. It can be used on it's own but it's
intention is to be used with the PADS Library by David Eppstein as the
transition function for the DFA Automata Library where the repository can be
found here:

    http://www.ics.uci.edu/~eppstein/PADS/.git

This library is hosted, along with our report and sample inputs here:

    https://github.com/matthewhardwick/self-avoding-walks

An example, a working demo of this library can be found at the follow website:

    Demo: http://cs454-final-project.herokuapp.com/
    Repo: https://github.com/matthewhardwick/cs454-final-project-heroku

Valid State Definitions:
    Single Ended - Where one possible valid connection point exists in a given
        state.
    Double Ended - Where two possible valid connection points exist in a given
        state.
    Triple Ended - Where three possible valid connection points exist in a
        given state.

    Each state is defined in a 4-char notation.

    1st Char - Top vertex
    2nd Char - Middle vertex
    3rd Char - Bottom vertex
    4th Char - Closure state for a Triple-ended state

    Valid char set ['0', '1', '2'], where that number signifies the current
    degree of that positions vertex for the first three characters in the
    string. The fourth charater signifies a vertical connection for a triple
    ended state. If Middle-Bottom then a 1 is used, else if a Top-Middle, then
    a 2 used.

    Short hand notation for possible connection points in a given state:
        [T]op connection possible
        [M]iddle connection possible
        [B]ottom connection possible
"""


SINGLE_ENDED = [
    '1000', '1200', '1220',  # T
    '0100', '2100', '0120',  # M
    '0010', '0210', '2210'   # B
    ]

DOUBLE_ENDED = [
    '1010', '1210',  # T, B
    '1100', '1120',  # T, M
    '0110', '2110'   # M, B
    ]

TRIPLE_ENDED = [
    '1111',  # M is connected to B
    '1112'   # T is connected to M
    ]

OTHER_STATES = ['start', 'dead', 'done']

STATE_LIST = SINGLE_ENDED + DOUBLE_ENDED + TRIPLE_ENDED + OTHER_STATES


def delta(current_state, symbol):
    """Given a state and input symbol, find the next state.

    Arguments:
    * current_state is a 4-char string over ['0', '1', '2']; a valid state.
        The initial 3 symbols indicate the degree for possible connecting
        vertices. The fourth symbol indicates the position for a triple-ended
        connection vertical bar where 1 indicates a M-B connection, and 2
        indicates a T-M connection.
    * symbol is a 5-char string over ['0', '1'].
        Each position of the string represents if an edge connects two
        vertices. 0 represents no connection, and 1 represents a connection.

    Expected Output:
        If a valid transition to a state exists, then the 4-char string
        representing that state will be returned, 'done' represents if that
        state transitions to an accepting state. If the transition is not
        valid, then the state of 'dead' is returned, and the entire input is
        not valid, and the input is rejected.
    """

    # Handle valid transitions from the start state, or return the dead state
    if current_state == 'start':
        if symbol == '00000':
            return '0010'  # Single: B
        if symbol == '00010':
            return '0120'  # Single: M
        if symbol == '01000':
            return '1112'  # Triple: T-M
        if symbol == '01010':
            return '1220'  # Single: T
        return 'dead'


    # When already in the 'done' state, stay in that state if the symbol is
    # trivial, or empty, otherwise, reject and transition to the 'dead' state.
    if current_state == 'done':
        if symbol == '00000':
            return 'done'
        else:
            return 'dead'

    # When the current state is dead, or rejected, there is no possible way to
    # recover, but it is possible to self loop on a dead state transition.
    if current_state == 'dead':
        return 'dead'

    # Handle initial trivial input, meaning if our symbol has no connections
    # and our current state is that we are in a single-ended connection then we
    # have a valid ending, and we can return 'done'. If we are in any other
    # state then it is an invalid transition, and we transition to the dead
    # state.
    if symbol == '00000':
        if current_state in SINGLE_ENDED:
            return 'done'
        else:
            return 'dead'

    # Generate the next state by calculating the degree of connection for each
    # vertex with a connection point based on the current input symbol.
    t = sum(int(symbol[i]) for i in [0, 1])
    m = sum(int(symbol[i]) for i in [1, 2, 3])
    b = sum(int(symbol[i]) for i in [3, 4])
    next_state = ''.join(str(i) for i in [t, m, b]) # Helper next state string

    # Handle triple-ended connection, and set closure position if necessary.
    # Triples in general can only either connect to another triple,
    # or a single-ended state, and will be rejected otherwise.
    if next_state == '111':
        if current_state in TRIPLE_ENDED:  # Triple -> triple.
            next_state += current_state[3]
        elif current_state in SINGLE_ENDED:  # Single -> triple.
            if current_state in ['1000', '1200', '1220']:  # Bottom closure.
                next_state += '1'
            elif current_state in ['0010', '0210', '2210']:  # Top closure.
```

```
                next_state += '2'
            else:  # Invalid.
                next_state += '0'
        else:  # Invalid.
            next_state += '0'
    else:  # Not a triple-ended state.
        next_state += '0'

    # Handle Branch condition. If a vertex has a degree of 3 or higher, then
    # it is connected to by more than two edges, and is therefore rejected.
    if ('3' in next_state[0:3] or
        int(current_state[0]) + int(symbol[0]) > 2 or
        int(current_state[1]) + int(symbol[2]) > 2 or
        int(current_state[2]) + int(symbol[4]) > 2):
        return 'dead'

    # Handles the Horizontal Jump Condition, where a gap of a horizontal
    # nature is created, and is therefore rejected.
    if (symbol[0] == '0' and symbol[2] == '0' and symbol[4] == '0' and
        (symbol[1] == '1' or symbol[3] == '1')):
        return 'dead'

    # Handles the vertical jump condition in a single-ended state, where
    # a gap is created in a way such that a vertical separation is created
    # between two vertices where only one connection is possible.
    # Additionally, this means we have left the walk which starts
    # at the origin, which is a necessary requirement for this DFA.
    # We in turn reject that input.
    if (current_state in SINGLE_ENDED and
        (current_state[0] == '1' and symbol[0] == '0' or
        current_state[1] == '1' and symbol[2] == '0' or
        current_state[2] == '1' and symbol[4] == '0')):
        return 'dead'

    # Handles the vertical jump condition in a double-ended state, where
    # a gap is created in such a way that a vertical separation is created
    # between two vertices where two possible vertices have a connection point.
    # Additionally, this means we have left the walk which starts
    # at the origin, which is a necessary requirement for this DFA.
    # We in turn reject that input.
    if (current_state in DOUBLE_ENDED and
        (int(current_state[0]) + int(symbol[0]) == 1 or
        int(current_state[1]) + int(symbol[2]) == 1 or
        int(current_state[2]) + int(symbol[4]) == 1)):
        return 'dead'

    # Handles the vertical jump condition in a triple-ended state, where
    # a gap is created in such a way that a vertical separation exists
    # between vertices where three possible connection points exist.
    # Additionally, this means we have left the walk which starts
    # at the origin, which is a necessary requirement for this DFA.
    # We in turn reject that input.
    if (current_state in TRIPLE_ENDED and
        (current_state[3] == '1' and
        int(current_state[0]) + int(symbol[0]) == 1 or
        current_state[3] == '2' and
        int(current_state[2]) + int(symbol[4]) == 1)):
        return 'dead'

    # Handles the case in which a loop is created, and as the name of this
    # project suggests must be avoided. Since, a loop is created, it
    # is rejected.
    if (current_state[3] == '1' and symbol[2:5] == '111' or
        current_state[3] == '2' and symbol[0:3] == '111'):  # Closed loop.
        return 'dead'

    # If we have made it through all of our checks, and the next state is
    # in fact in the state list then we either enter or loop on an accept
    # state.
    if next_state not in STATE_LIST:
        return 'done'

    # If we have made it through every other possible case, then that means
    # we have a valid transition to another state, and can therefore move
    # to that state and return that state.
    return next_state
```

4.2 Visualization

A tool to help visualize the use of the finite automaton presented in this paper is freely available online [3]. Put simply, it is nothing short of revolutionary.

5 Sample input and output

When creating test cases to check the validity of the transition function outlined in Section 3.3 we broke down our test cases to exploit the different types of failures on the lattice. Our first constraint of the walk was to begin at (0,0). It is tested by the value of the start state, if the walk could pass this it would move out to check for jumps, branches and closure properties. A jump on the lattice can happen in one of two ways: horizontal or vertical. A branch can happen in one of many ways depending on what is a valid input from the next, the idea is that whatever node the state is currently in can only transfer to one or the other but not both. Closure properties test for self-avoiding walks, so we created test cases that would demonstrate the delta function knew how to proceed. We ran over 50 test cases through the DFA. In the following paragraphs, we present 4 examples of these and discuss the sequence of states that leads to acceptance or not.

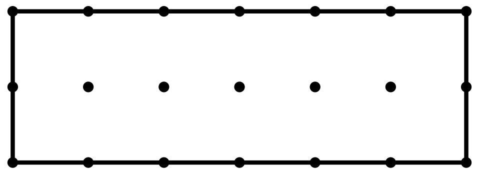
Figure 2: Test case to illustrate branch identification and rejection.

The input in Figure 2 illustrates how a branch condition results in rejecting an invalid state. After the initial input is processed, for this to be a valid walk it must proceed from the top (walks must originate at the origin). After the next input is read this would transition to the dead state. Code to identify branches successfully detected that this input diverged into two paths. Once the DFA is in the dead state, we know the input is invalid.

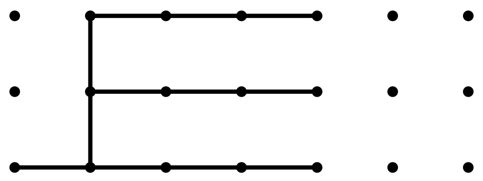

Figure 3: Test case to illustrate multiple branch identification.

The input in Figure 3 demonstrates the branch identification capabilities of the DFA. It is clear that there has been a branch after the third input symbol because several of the resulting vertices have degree 3. It is possible to transition to a triple-ended state from a single-ended one, but this example is not a valid way.

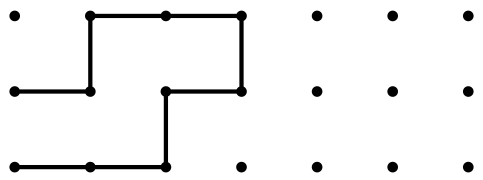

Figure 4: Test case to illustrate double-ended state resulting in a valid walk.

Execution of the DFA on the input presented in Figure 4 shows that once the DFA is in a double-ended state, it must remain in a double-ended state. The input continues in a double-ended state and is accepted upon being closed.

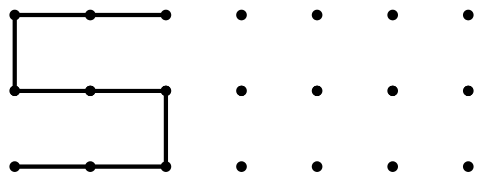

Figure 5: Test case to illustrate triple-ended state behavior.

The input in Figure 5 begins in a triple-ended state. The third input closes two of the paths (recall that valid input from triple-ended state must connect all the nodes to proceed). After the final nontrivial input, the DFA is in a single-ended, accepting state.

6 Conclusions and directions for further research

Designing the encoding and finite automaton to recognize self-avoiding walks was an iterative process. Our concept of state and alphabet underwent numerous revisions as we tested and re-tested each idea. The result is an algorithm, the transition function δ outlined in Section 3.3, that we believe is sound and complete. At each step of the algorithm, we reason that each case discussed is accounted for, and that the complete algorithm accounts for all possible inputs. In addition, we tested the automaton on more than 50 inputs. While the data provide strong evidence that our algorithm is good, this research would be more complete with a formal proof of the algorithm's correctness.

The DFA produced by this algorithm consists of 20 states, and 136 transitions to non-dead states. Using the tools available in PADS [2], we found that the minimized DFA consists of only 17 states and 126 transitions to non-dead transitions. In fact, evaluation of the minimal DFA reveals that we could simplify our concept of state (presented in Section 3.2) slightly by removing some "double-ended" states that were apparently redundant. It is encouraging that we were able to develop a DFA that is so close to minimal without using minimization tools. Furthermore, provided that the algorithm is correct, this slight revision to our concept of state would result in an "optimal" algorithm for the transition function in the sense that it would produce a minimal DFA.

It would be straightforward to use this DFA constrction to count the number of self-avoiding walks in a $3 \times n$ square lattice. Counting the number of walks of a given length k is less straightforward, because each input symbol may add a variable number of segments to a valid walk. However, considering the "weight" of each input symbol in terms of the length it adds to the self-avoiding walk is likely a good place to start.

References

[1] Combs, J., Hardwick, M., and Robbins, S. Repository for self-avoiding walk research report and software. `https://github.com/matthewhardwick/self-avoding-walks`. Accessed: 2013-12-10.

[2] Eppstein, D. Python algorithms and data structures. `http://www.ics.uci.edu/~eppstein/PADS/`. Accessed: 2013-12-10.

[3] Hardwick, M. Visualization tool for testing self-avoiding walks. `http://cs454-final-project.herokuapp.com`. Accessed: 2013-12-10.

[4] Pönitz, A., and Tittman, P. Improved upper bounds for self-avoiding walks in $\mathbb{Z}^d$. *The Electronic Journal of Combinatorics 7* (2000).