Crossing-free paths in the square grid

POST-PRINT

Article published on
Computers & Graphics, Volume 114, August 2023, Pages 296-305

https://www.sciencedirect.com/science/article/pii/S0097849323001115?via%3Dihub

Lidija Comic,
Faculty of Technical Sciences, University of Novi Sad, Novi Sad, Serbia
comic@uns.ac.rs

Paola Magillo
DIBRIS, University of Genova, Genova, Italy
magillo@dibris.unige.it

Abstract

We consider paths in the 2D square grid, composed of grid edges, given as a sequence of moves in the four cardinal compass directions, without U-turns, but possibly passing several times through the same vertex or the same edge (if the path is open, it cannot pass twice through its starting vertex). We propose an algorithm which reports a self-crossing if there is one, or otherwise draws the path without self-crossings. The algorithm follows the intuitive idea naturally applied by humans to draw a curve: at each vertex that has already been visited, it tries to insert two new segments in such a way that they do not cross the existing ones. If this is not possible, a self-crossing is reported. This procedure is supported by a data structure combining a doubly-linked circular list and a skip list. The time and space complexity is linear in the length of the path.

1 Introduction

We consider the plane as covered by the infinite square grid with cell edge of unit length. In the infinite grid graph, composed of the vertices and edges in the square grid, a digital path of length n (with $n \geq 1$) is a sequence $\pi = e_1, e_2, ..., e_n$ of edges such that two consecutive edges are distinct and mutually adjacent. The same path could also be expressed as a sequence of vertices $P_0, P_1, \dots P_n$ such that P_i is the common endpoint of e_i and e_{i+1} for $1 \leq i \leq n-1$, P_0 is the endpoint of e_1 different from P_1, and P_n is the endpoint of e_n different from P_{n-1}.

The path may be open or closed (i.e., the last vertex P_n may or may not be coincident with the first one P_0) and it may present repetitions of both vertices and of edges. Note that two consecutive edges e_i, e_{i+1} of the path are adjacent in the grid graph: since an edge is not adjacent to itself, then $e_i \neq e_{i+1}$ and $P_{i-1} \neq P_{i+1}$. For closed paths, we also require that $e_n \neq e_1$ (or, equivalently, that $P_1 \neq P_{n-1}$).

We encode a digital path π as a starting vertex P_0 and a word $w = w_1...w_n$ of (finite) length $n \geq 1$ over the alphabet $\Sigma = \{\text{N,E,W,S}\}$, i.e., $w_i \in \Sigma$, $1 \leq i \leq n$. Each letter w_i in the word w corresponds to the edge e_i in the path π considered as directed from P_{i-1} to P_i. The letters (directions, or moves) denote the four cardinal directions of a compass. For a letter

d in Σ, we denote the opposite direction as $-d$ in the obvious manner, e.g. if d =E then $-d$ =W. The considered words do not contain consecutive pairs of opposite moves, i.e., EW, WE, NS or SN (a.k.a. U-turns). In other words, $w_{i+1} \neq -w_i$, $1 \leq i \leq n-1$. Additionally, for closed paths, $w_n \neq -w_1$.

We say that a digital path π is crossing-free if the edges incident with each vertex P in π can be arranged in a radial sequence so that for any two indexes $1 \leq i, j \leq n-1$, the edges e_i, e_{i+1}, e_j, e_{j+1} do not appear in the alternating order $e_i, e_j, e_{i+1}, e_{j+1}$ in the sequence. For example, the path in Figure 1 (a) is crossing-free, while the one in Figure 1 (b) is self-crossing at the vertex $P_1 = P_5$.

We address two related questions:

1. Is the path π self-crossing?
2. If not, how can we draw π without crossings?

We propose an algorithm to answer these questions by tracing a (directed) curve C, which draws the path π edge by edge, inserting each current edge e_i at its correct place in the cycle of the edges incident with its endpoint P_i and, if possible, inserting the next edge e_{i+1} of π at its correct place in the same cycle, or reporting a self-crossing otherwise.

An open crossing-free path is homeomorphic to a segment, and a closed one is homeomorphic to a circle. Therefore our method to check this property can help in curve classification. Moreover, thanks to the incremental nature of our algorithm, it could be used during interactive drawing, to ensure crossing-free digital curves. By extending it to other types of grids, provided that they have a finite and discrete number of possible moves from each vertex, it can find application to path planning and navigation on a polygonal mesh.

2 Related work

A word on the alphabet of the four cardinal compass directions describes a digital path, composed of edges in the square grid. When the path is the boundary of a digital object, i.e., simple and closed, such encoding is known as the Freeman code [9, 10, 11].

Freeman codes have been used to represent paths in robot path planning [15], in image retrieval and registration [12, 13], and for text recognition in manuscripts or images [3, 8, 5].

Brlek et al. [6] considered the problem of testing whether a path, represented by its Freeman code, passes several times through the same vertex. This question can be answered in $O(n \log n)$ time by using sorting or an AVL tree, but the authors provide an efficient solution working in linear time, based on a quad-tree combined with a radix tree. This is a related but different problem with respect to the one we address, since we admit several passages through the same vertex, provided that the path does not cross itself at that vertex.

Brlek et al. [7] described how to detect a self-crossing in the specific case of digital paths that are closed and pass through the same vertex at most twice. Such paths are intended as the boundary of a non-connected object (i.e., a set of pixels) in the square grid, where tunnels (sequences of edges traversed two times in opposite directions) are used to bound different connected components of the object with a unique contour. Here, we address a more general problem, admitting multiple passages through the same vertex, and open paths as well.

Abbott et al. [1] classify paths into self-crossing, non-touching, and self-touching. The last ones correspond to our idea of passing multiple times through a vertex or edge, but without moving to the other side. Self-touching paths can be unfolded to non-touching ones by moving their vertices continuously. Banerjee and Chandrasekaran [4] employ non-touching paths for motion planning. Some interest for non-touching paths also exists in manufacturing, where a linear piece of wire must be deformed to achieve a desired configuration, if possible without crossing [2].

3 Algorithm for testing whether a path is self-crossing

In our algorithm, we apply the same approach that a human would naturally use to draw a digital path.

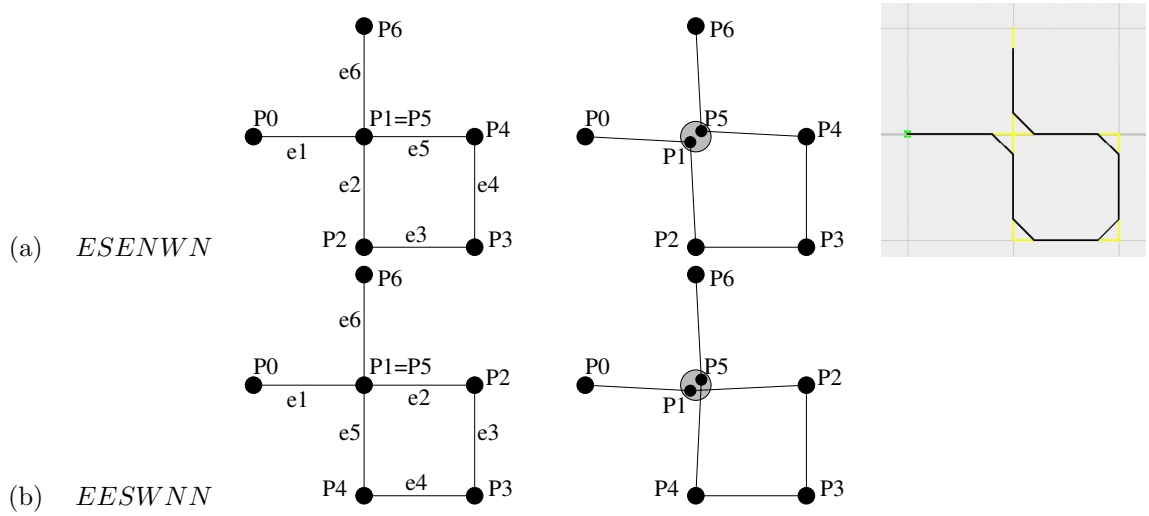

Figure 1: The first column shows the words encoding two paths, where (a) the first can be drawn without crossings, and (b) the second is self-crossing. The second column shows the traversed vertices, and, without letters, this drawing is ambiguous. The third column draws the path $ESENWN$ without crossings, and the last one shows the output of our demo.

Given the word w, we follow the digital path from the starting vertex P_0 and, for each move w_i, we try to draw a unit segment exiting from the current vertex in the direction w_i. If one such segment has been drawn already, we draw the new segment above/below (for w_i =E or W) or left/right (for w_i =N or S) to the old one, with a small offset, in such a way to avoid crossing. If it is not possible, we report self-crossing.

We assume that the path passes only once through its starting vertex P_0 (in case of a closed path, $P_0 \equiv P_n$ is allowed). In Section 7.3, we will discuss the rationale of this requirement and possible ways to relax it partially.

We define a half segment as an oriented segment exiting from a source point. In general, a half segment has a length and a direction. Here, the source point is a vertex of the grid, directions are in Σ, and all half segments have length equal to one. Every half segment has an opposite half segment, corresponding to the same segment with the opposite orientation and the other endpoint as source.

Starting from the vertex P_0, the path will traverse a sequence of vertices $P_0, P_1, \ldots, P_n$. These vertices are not necessarily distinct (see Figure 1). Every pair of consecutive moves $w_i w_{i+1}$ in the input word defines a passage through the vertex P_i. Three half segments are involved (see Figure 2):

1. the half segment exiting from P_{i-1} in the direction w_i,
2. its opposite half segment, having the source P_i and the direction $-w_i$; we call it the entering half segment for P_i,
3. the half segment exiting from P_i in the direction w_{i+1}; we call it the exiting half segment for P_i.

We say that the entering and the exiting half segments of P_i are mates of each other. Note that the two mate half segments are directed consistently with the fact that their source is P_i. So if w_i =E and w_{i+1} =S, then the entering half segment has direction W= −E. Since U-turns are not allowed, the two mate half segments must have distinct directions.

The path may pass several times through the same vertex P, i.e., $P = P_i \equiv P_j \equiv P_k \ldots$ for $i \neq j \neq$

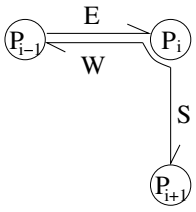

Figure 2: The step from P_{i-1} to P_i through the move w_i =E and from P_i to P_{i+1} through the move w_{i+1} =S. The three involved half segments are shown with their directions. Two half segments with source P_i (linked with an arc) are mates of each other and define a passage through P_i.

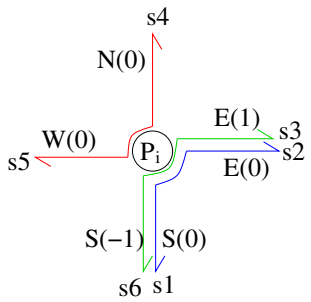

Figure 3: The radial list $s_1, s_2, s_3, s_4, s_5, s_6$ of half segments at a traversed vertex P_i. For each half segment, we show the direction and the offset. Half segments drawn in the same color, and linked by an arc, are mates.

$k \ldots$. We store all half segments generated by all passages through P in a list, radially sorted around P in counterclockwise order.

Half segments referring to different passages through P may have the same direction. In order to disambiguate the counterclockwise order in such case, each half segment has an offset: a small quantity to be added to the x coordinate of the segment (if the segment is vertical) or to the y coordinate (if it is horizontal). When two (consecutive) half segments have equal direction, they have different offset. If the direction is E or S then the lower offset precedes the higher one; if the direction is W or N then the higher offset precedes the lower one.

An example of the radial list around a vertex P_i is shown in Figure 3. P_i has three passages, i.e., three pairs of mate half segments (each shown in the same color and connected by an arc). The direction and the offset of each half segment is also shown. The sorted list is $s_1, s_2, s_3, s_4, s_5, s_6$ (as the list is circular, the first element is arbitrarily chosen).

It is important to notice that offset values are symbolic, and not related with the edge length of the grid. We use numbers, but we could use character strings, or any other type that supports a total order and such that an intermediate element always exists between any two given elements. In Figure 3, the values 1 and -1 are used to distinguish the relative order of half segments with equal direction. These values will be replaced by appropriate ones when drawing the curve (as we will explain in details in Section 5).

3.1 Processing the first vertex

The first vertex P_0 has just an exiting half segment s_{out} in direction w_1, so this half segment would have no mate (it can be conventionally set as the mate of itself).

This half segment is inserted in the radial list of the first vertex P_0 (now containing only it) with offset equal to 0. For the paths in Figure 1 (a) and (b), the radial list of P_0 will contain the exiting half segment s_{out}, the first move of the path, in direction E.

3.2 Processing the next vertices

For $i > 0$, the entering half segment s_{in} of P_i is the opposite of the exiting half segment of P_{i-1}, which has been processed in the previous step. The offset of s_{in} is the same as its opposite half segment, and therefore it is known.

Based on the offset of s_{in}, there is a unique position for inserting it in the radial list of P_i. We insert s_{in} in the list at its position, and we try to find the position for its mate, the exiting half segment s_{out} in direction w_{i+1}. This may or may not be possible, depending on the position of the existing half segments around the vertex P_i. Let us first consider some examples.

Consider the crossing-free path ESENWN in Figure 1 (a). For vertices P_i, $i = 1, 2, 3, 4$, the radial

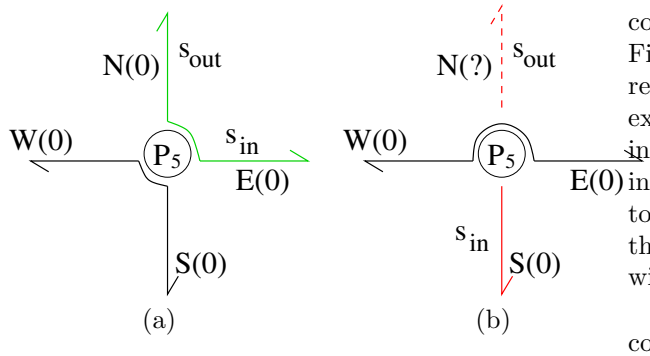

Figure 4: Processing the vertex P_5 in the paths of Figure 1 (a) and (b), respectively. In (a) we can add s_{out} without crossings. In (b) we find a self-crossing when trying to add s_{out}.

list will contain just the entering and the exiting half segments, both with offset equal to 0. When processing $P_5 \equiv P_1$ (see Figure 4(a)), the radial list contains one half segment in direction W and one in direction S. The entering half segment s_{in}, having direction E, is inserted between the one in direction S and the one in direction W. The exiting half segment s_{out} has direction N, and it is inserted just after its mate s_{in}.

Consider the self-crossing path EESWNN in Figure 1 (b). The vertices P_i, $i = 1, 2, 3, 4$ are processed in the same way as in the previous example. When processing $P_5 \equiv P_1$ (see Figure 4(b)), the radial list contains one half segment in direction W and one in direction E. The entering half segment s_{in}, having direction S, is inserted between the one in direction W and the one in direction E. The exiting half segment s_{out} has direction N. It cannot be inserted because s_{in} and s_{out} lie in the two opposite sectors defined by the existing pair of mate half segments.

In the previous examples, for no vertex there are two different half segments with the same direction, therefore all half segments have offset equal to 0. A more complex example is shown in Figure 5.

Up to P_8, each traversed vertex has just two half segments. At P_9, the vertex has two mate half segments in directions S and E. The new passage inserts two more half segments in directions W and N. All half segments have offset 0.

When processing $P_{10} \equiv P_6$, the radial sequence contains two half segments in directions W, E (see Figure 6(a)). The entering half segment s_{in} has direction S and offset 0. It is inserted between the two existing half segments. The exiting half segment s_{out} in direction E is inserted between s_{in} and the existing half segment in direction E, and its offset is set to a value lower than 0, for example -1. Similarly, the exiting half segments at P_{11} and P_{12} are inserted with offsets different from 0.

When arriving at $P_{13} \equiv P_9 \equiv P_3$, the radial list contains four half segments in directions S,E,N,W, where the pairs in directions S,E and N,W are mates, and all offsets are equal to 0 (see Figure 6(b)). The new entering half segment s_{in} in direction E with offset 1 is inserted after the existing half segment in direction E. Its mate exiting half segment s_{out} in direction S is inserted before the other one in direction S, and its offset is set to -1.

When processing $P_{14} \equiv P_2$, the radial list contains the half segments of one passage, with directions W and N and offsets 0 (see Figure 6(c)). The entering half segment s_{in} in direction N with offset -1 (it is the opposite half segment of the one exiting P_{13}) is inserted between the two half segments in directions N and W. The exiting half segment s_{out} in direction E cannot be inserted, because it and its mate lie in opposite sectors defined by the previous passage at P_{14}.

4 Implementation details

4.1 Managing the radial lists

Every traversed vertex P_i has an associated list of half segments, radially sorted around P_i in counterclockwise order. The radial order takes offsets into account, when two consecutive half segments have the same direction: for directions N and W, higher offset precedes lower offset, for direction S and E, lower offset precedes higher offset.

Such list is implemented as a standard doubly-linked (circular) list, i.e., a chain of nodes, each pointing to the next node and to the previous node. In addition, each node, containing a half-segment s, has a pointer to the node containing the mate half-segment

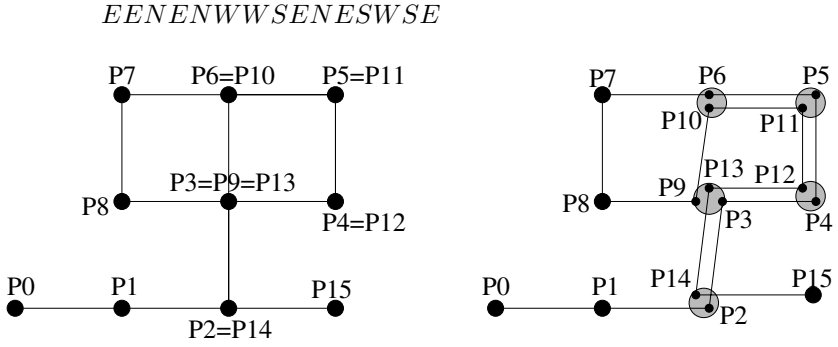

Figure 5: A path, the raw way to draw it, and a drawing highlighting the passages of the curve through the vertices. The path can be drawn without self-crossings until the last segment $P_{14}P_{15}$.

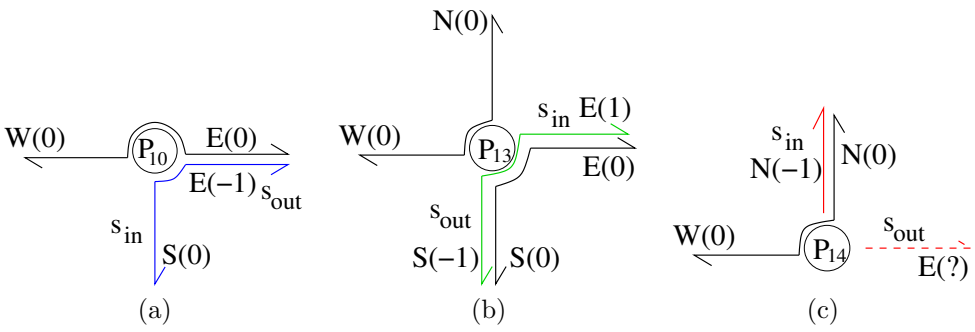

Figure 6: Processing the vertices $P_{10} \equiv P_6$, $P_{13} \equiv P_9 \equiv P_3$, and $P_{14} \equiv P_2$ in the path of Figure 5. A self-crossing is detected when processing P_{14}.

of s. In other words, the pair of mate half-segments, representing the same passage through the vertex P_i, are mutually connected with pointers. Our implementation will use such mate pointers, to skip entire radial sectors during a search.

The idea of using additional pointers leading elsewhere in a linked list, to exclude portions of the list from a search, comes from skip lists. A skip list [14] is a linked list storing a sorted sequence of values, where each node has an additional pointer, leading to another node located $K > 1$ positions forward, with K randomly chosen. Such skip pointers are used to speed up the search for a given value x in the sorted list. The algorithm starts from the first node and loops while the current node N contains an element $y < x$. If the skip pointer of N brings to a node M containing $z < x$, then the search goes directly to M, skipping the part in between. As skip pointers in a skip list, our mate pointers can bypass portions of a list, that are not relevant. In particular, we use them during the search for locating the correct position of the half-segment corresponding to a new move from P_i (see details later).

Referring to the radially sorted list of half segments associated with a vertex P, we define the following predicates (we recall that the offset of a half segment may be undefined, and this happens temporarily for the exiting half segment to be inserted in the list):

- s `strict_between`(s_1, s_2) holds if s is between s_1 and s_2 in the radial order. If the direction of s is the same as that of s_1 and/or s_2, the value of the predicate depends on the offset of s and, if the offset of s is still undefined, it is false.

- s `loose_between`(s_1, s_2) is equal to the previous predicate in all cases except when the direction of s is the same as that of s_1 and/or s_2 and the offset of s is undefined. In that case, `loose_between`(s_1, s_2) returns true, and the offset of s will be set in such a way that `strict_between`(s_1, s_2) will hold.

The insertion of the entering half segment s_{in} in the list is done by the function `add_entering_seg`, whose pseudocode is shown in Figure 7. If the radial list is empty, we simply add s_{in}. Otherwise, we loop until we find two consecutive half segments such that s_{in} lies strictly between them.

```
add_entering_segment(s_in)
   if list is empty
     add s_in as first element
     return
   curr = first element
   while true
      if s_in strict_between(curr.prev, curr)
          add s_in before curr
          return true
      curr = curr.next
```

Figure 7: Pseudocode of the function which adds the entering half segment s_{in} to the skip list associated with the traversed vertex.

Function `add_exiting_seg`, whose pseudocode is shown in Figure 8, searches for a correct position to insert the exiting segment s_{out} (mate of s_{in}, in direction of the move w_{i+1}), starting from the position of s_{in}. This is not always possible, and the function return true (false) in case of success (failure). In case of success, the function also sets the offset of s_{out} (initially undefined).

If the radial list contains just s_{in} (i.e, this is first passage through P_i), we insert s_{out} in any position and set its offset to zero.

Otherwise, the radial list contains at least three segments (s_{in} and a pair of mates from a previous passage). We search for a pair of consecutive half segments s_{prev}, s_{curr}, such that s_{out} can be inserted between them, by setting its offset as necessary. At the beginning, $s_{prev} = s_{in}$ and s_{curr} is the next half segment of s_{in}, so we try to put s_{out} just after its mate.

If the direction of s_{out} lies strictly in the radial sector between s_{prev} and s_{curr}, or it is equal to the direction of one of them, then we insert s_{out} between s_{prev} and s_{curr}, and set its offset: zero if s_{out} is strictly in the radial sector, or non-zero if its direction coincides with that of s_{prev} and/or s_{curr}.

Otherwise, we consider the mate segment s_{mate} of s_{curr}. If s_{out} has a direction strictly between s_{curr} and s_{mate}, then s_{out} cannot be inserted (a self-

```
add_exiting_seg(s_out)
   if list has one element
     // it is the mate of s_out
     s_out.offset = 0
     add s_out to list
     return true
   s_curr = s_out.mate.next
   while true
      if s_out loose_between(s_curr.prev, s_curr)
          update_offset(s_out, s_curr.prev, s_curr)
          add s_out before s_curr
          return true // added
      // if s_curr is mate,
      // we tested the last possible position
      if s_curr == s_out.mate
          return false // not added
      // otherwise,
      // the mate of s_curr is in the list
      if s_out strict_between(s_curr, s_curr.mate)
          return false // cannot add
      // skip the sector between s_curr
      // and its mate
      s_curr = s_curr.mate.next
```

Figure 8: Pseudocode of the function which adds the exiting half segment s_{out} to the skip list associated with the traversed vertex, when the mate half segment s_{in} of s_{out} has already been added. This operation may not be possible. The function returns true if the half segment has been inserted and false otherwise.

crossing has been detected). Otherwise, we skip the whole radial sector from s_{curr} to s_{mate}: for the next iteration s_{prev} will be s_{mate} and s_{curr} will be the half segment following it.

If we have completed a turn around the vertex (s_{curr} is now s_{in}), this means that all possible positions have been tested, and thus s_{out} cannot be inserted (a self-crossing has been detected).

4.2 Retrieving already visited vertices

When, following the path, we arrive at the next vertex P_i, we have to determine whether this is a new vertex (first passage through P_i, its radial list does not exist and it must be created), or an already visited one (its radial list exists, is not empty, and must be retrieved).

Our implementation relies on dictionaries. The dictionary is a data type consisting of a set of associations key $\longrightarrow$ value. Here the key consists of the pair of coordinates of a vertex and the associated value is the radial list of half segments. Dictionaries can retrieve the value associated with a given key in expected constant time, if implemented with hash tables. Some alternative implementations are discussed in Section 6.2.

4.3 The main procedure

The pseudocode of the overall process is given in Figure 9. The return value is true if the path is self-crossing and false if it is crossing-free.

Beside the already mentioned functions `add_entering_seg` and `add_exiting_seg`, the functions `add` and `add_first_pair` add one or two half segments in any order, respectively, into an empty skip list. Function `update_offset`, whose pseudocode is given in Figure 10, sets the offset of a half segment s, inserted between two half segments s_{prev} and s_{succ}, in such a way that s `strict_between`(s_{prev}, s_{succ}) holds.

If the direction of s is distinct from that of s_{prev} and s_{succ}, the offset will be 0; if it is equal to the direction of s_{prev} (s_{succ}), then the offset is set to the one of s_{prev} (of s_{succ}) plus or minus an offset depending on the direction; if the directions of the three segments are equal, the offset is set to the mean between the offsets of the other two segments.

5 Drawing

The usual way to draw a path w starts from P_0, applies the first move w_1 to compute the next vertex P_1, draws the segment P_0P_1, then it repeats the process for P_i and w_{i+1}, for $i = 1 \ldots n-1$. This drawing is ambiguous if the same vertex is traversed multiple times (compare Figures 1 (a) and (b)).

Our drawing without intersections works in the following way. When drawing a passage through a vertex P_i, P_i becomes two consecutive points, one as

```
self_crossing(P0, path)
  // create first segment
  first = new HalfSegment(P0, path[0])
  first.offset = 0
  // insert it into the radial list of P0
  listP0 = setListForPoint(P0,new SkipList())
  listP0.add(first)
  //the cycle will create all other half segments
  last = first
  for i=1 to n-1
    // compute next vertex
    P = move_vertex(last.start, last.dir)
    // create s_in, the entering half
    //segment of P, opposite of last
    s_in = new HalfSegment(P, -last.dir)
    s_in.offset = last.offset
    // create s_out, the exiting half
    // segment of P, mate of s_in,
    // with undefined offset
    s_out = new HalfSegment(P, path[i])
    make_mates(s_in,s_out)
    // insert the two mate half segments
    // into the radial list of P
    listP = retrieveListForPoint(P)
    if listP does not exist:
         // first passage through P
         listP=setListForPoint(P,new SkipList())
         listP.add_first_pair(s_in,s_out)
    else // a subsequent passage through P
         listP.add_entering_seg(s_in)
         success=list[P].add_exiting_seg(s_out)
         if not success // self-intersection
                return true
    // prepare for next iteration
    last = s_out
  return false // end of cycle, no intersection
```

Figure 9: Pseudocode of the self-crossing test. The function returns true if the path is self-crossing and false if it is crossing-free. The elements of the word w, encoding the input path, are stored in an array `path` where indexes start from 0 and `path[i]` contains w_{i+1}, for $0 \leq i \leq n-1$.

endpoint of the entering half segment, and one as endpoint of the exiting half segment. The positions of the two points depend on the direction and offset of the entering / exiting half segments. The two points lie on a virtual square centered at $P_i = (x_i, y_i)$ with edge $= 2\,\Delta$, where $0 < \Delta < \frac{1}{2}$, assuming that the grid edge is 1. The coordinates (x, y) of the point that replaces P_i as endpoint of each half segment are:

- for half segment oriented E, $x = x_i + \Delta$
- for half segment oriented W, $x = x_i - \Delta$
- for half segment oriented N, $y = y_i + \Delta$
- for half segment oriented S, $y = y_i - \Delta$

The other coordinate depends on the offset and will be $y_i + \varepsilon$ *offset* if the half segment is horizontal (direction E or W) and $x_i + \varepsilon$ *offset* if the half segment is vertical (direction N or S). The values of Δ and ε must be chosen in such a way that $\Delta < 1/2$ and $\varepsilon < \Delta / max\_offset$, where *max_offset* is the maximum absolute value among all offsets. By connecting such points, we draw a curve representing the given path without intersections. Three examples are shown in Figure 11.

```
update_offset(s, s_prec, s_succ)
    // two Booleans indicate whether the
    // adjacent half segments give a
    // constraint for the offset of s
    noprec= (s_prec==null || s_prec.dir!=s.dir)
    nosucc= (s_succ==null || s_succ.dir!=s.dir)
    if noprec and nosucc // no constraint
      s.offset = 0
      return
    // incr gives the sign of the in offset
    if (s.dir==E) or (s.dir==S): incr = 1
    else incr = -1 // W or N
    if noprec
       s.offset = s_succ.offset-incr
    else if nosucc
       s.offset = s_prec.offset+incr
    else
       s.offset =
         0.5*(s_prec.offset+s_succ.offset)
```

Figure 10: Pseudocode of the procedure which sets the offset of a new exiting half segment s_{out}, based on the ones preceding and following it in the radial order.

Choosing $\varepsilon < \Delta/max\_offset$ guarantees that, whatever the number of passages through a given grid edge PQ, the segments, drawn to represent them, remain distinct and their endpoints lie within the squares centered at P and Q with edge length $= 2\Delta$. They are distinct because the offsets of such segments are all distinct and so are the quantities $\varepsilon\, offset$ added to the coordinates of P and Q. They lie within the square because $|\varepsilon\, offset| < \varepsilon\, max\_offset < \Delta$. Let us consider the example in Figure 11 (c) and the grid edge PQ. This edge is traversed four times, and the offsets of the four segments are $0, 1, 1.5, 2$, respectively (from left to right). The segments to be drawn have $x = x_P, x_P + \varepsilon, x_P + \frac{3}{2}\varepsilon, x_P + 2\varepsilon$ (here, $max\_offset = 2$).

If the offset difference of two consecutive segments traversing the same edge is too small, the coordinates assigned to them, even if distinct in the continuum, may fall on the same pixel when dicretized on screen. For example, let us consider a path of the form $E(NNNESWSESW)^k$ where the exponent denotes a subpath repeating k times. Referring to Figure 12, edge PQ is traversed $2k$ times. For $k = 1$, the two segments have offsets $= 0, 1$. For $k = 2$ (see Figure 12(a)), the two new segments are located between the segments of the previous traversal, with offsets $= 0.5, 0.75$ (the distance between them is 0.25). For $k = 3$ (see Figure 12(b)), the offsets of the two new segments, located between the previous pair, are $= 0.625, 0.7$ (with distance 0.075). The segments at PQ, although not many, become too close, and cannot be drawn distinctly on the screen.

To avoid this problem, we can reconfigure the offsets of the segments occurring at a given grid edge, here PQ. This operation affects only the radial lists of the two points P and Q (containing the two opposite half segments of such segments). In the radial list, we count the half segments with each direction (E, S, W, N). If the minimum offset difference is < 1, we redistribute the offsets evenly between $-\lceil \frac{m}{2} \rceil$ and $\lfloor \frac{m}{2} \rfloor$, where m is their number. Figure 12(c) shows the path $E(NNNESWSESW)^3$ drawn after reconfiguration (the offsets of the six segments are now $-3, -2, -1, 0, 1, 2$). Figure 12(d) shows the reconfigured path $E(NNNESWSESW)^5$.

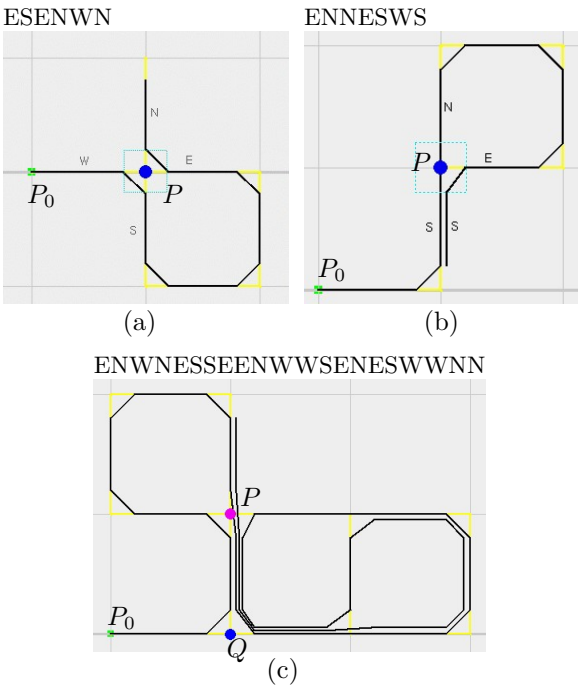

Figure 11: Examples of paths and drawings. In (a) and (b), the blue vertex P is traversed by two passages. The virtual square centered at P is shown dotted. The letters show the directions of the half segments incident with P. In (b) the two segments with direction S have offsets 0 and 1 (from left to right). The path in (c) traverses four times the grid edge connecting the marked vertices P and Q.

6 Computational complexity

The size of the input is the length n of the path. Let $\mathcal{Q}$ be the set of traversed vertices and $q = |\mathcal{Q}|$. Then $q \leq n + 1$, because the path may pass several times through the same vertex.

6.1 Cost for managing the radial lists

Let $\delta(P)$ be the number of passages through a vertex $P \in \mathcal{Q}$. We have that $\sum_{P \in \mathcal{Q}} \delta(P) = n + 1$ and $\delta(P) \leq \lceil n/4 \rceil$ for all $P \in \mathcal{Q}$, because, as U-turns are not allowed, the path needs at least four moves (e.g.,

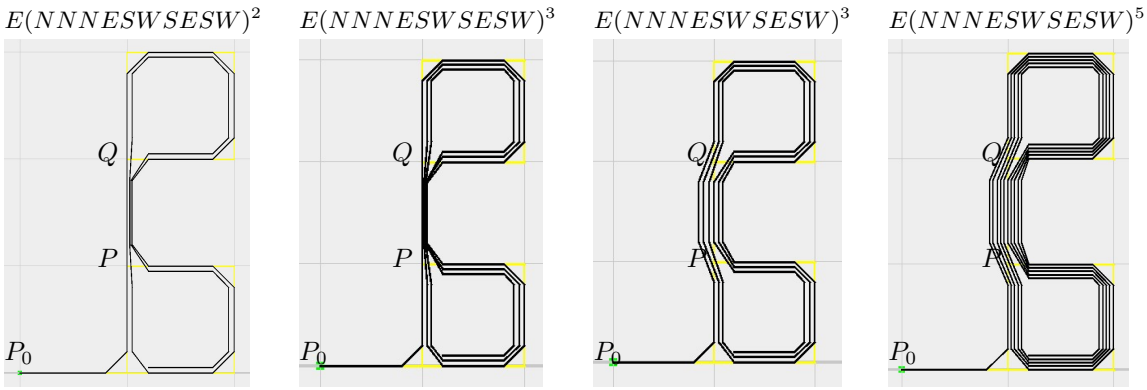

Figure 12: Drawing the path $E(NNNESWSES)^k$ for (a) $k = 2$, (b,c) $k = 3$, (d) $k = 5$. In (c,d), the offsets have been reconfigured.

ENWS) to return to the same vertex.

When a passage through P is processed, we have to locate the position for the entering half segment s_{in} in the radial list around P, and then the position for the exiting half segment s_{out}. The first operation costs at most $\delta(P)$ steps. The second one costs four steps thanks to the use of skip pointers, as shown below.

The half segments present in the radial list have at most four different directions. The first checked position is the one between s_{in} and its next half segment s_{curr}. If s_{out} cannot be placed there, we skip the whole sector between s_{curr} and its mate, which must have a different direction from s_{curr} (because they belong to the same passage, and U-turns are not allowed). So, each step changes the direction, and in at most four steps we return to s_{in}, if neither the correct place for inserting s_{out} nor a crossing have been detected in the meantime.

6.2 Cost for retrieving repeated vertices

Many approaches can be used to support the retrieval of the radial list of an already visited vertex.

- With a preliminary sorting of all vertices, we pay an $O(n \log n)$ preprocessing time, allowing for an $O(1)$ query time during the main cycle. This approach needs to know all vertices in advance, so it is not suitable for drawing paths in an interactive way.
- With an AVL tree based on the vertex coordinates, we have an $O(\log n)$ cost at each iteration of the cycle, leading to an overall $O(n \log n)$ time complexity.
- With a dictionary based on the vertex coordinates, we have an expected $O(1)$ query time during the main cycle, but the (very unlikely) worst-case cost at a single vertex could be up to $O(n)$.
- We could use the data structure proposed in [6] to detect whether a digital path passes twice through the same vertex. The structure can be easily modified to support the retrieval of the first visited instance of a vertex, in the following way. In the original data structure, a node is marked with a Boolean value: visited or not visited. If a new vertex falls into a visited node, then the algorithm stops and reports the existence of a multiple passage. We can mark each node with an integer number: -1 for unvisited, or the index (≥ 0) of the first found vertex falling in that node otherwise. In case of another passage through the vertex, the corresponding node will provide the index of the first encountered copy of the same vertex. The radial lists can be

stored separately in an array. This alternative will provide an $O(n)$ time and space complexity, as shown in [6].

6.3 Overall time complexity

The time complexity for retrieving already visited vertices can be $O(n)$ as described above. The time complexity for managing all insertions in all radial lists is expressed by

$$T(n) = \sum_{i=1}^{n} (\delta(P_i) + 4) = 4n + \sum_{i=1}^{n} \delta(P_i)$$

In the worst case, $\delta(P_i) = n/4$ and thus $T(n) = O(n^2)$. The worst case is a spiral-like path, e.g., move E followed by an arbitrary numbers of loops ENWS. The path $E(NNNESWSESW)^k$, shown in Figure 12, is another one with quadratic time complexity. All vertices different from P_0 have degree $2k$ or k, and $n = 1 + 10k$, i.e., the path length n and the number k of repetitions have the same order of magnitude.

For realistic paths, $\delta(P_i)$ is likely to be bounded by a constant, and thus the overall time complexity is expected to be linear in the path length.

As an example, let us consider a fractal path generated by the recursive expansions of a single segment, following the rules (shown in Figure 13):

- N expands to NENWSWNEN
- S expands to SWSENESWS
- E expands to ESENWNESE
- W expands to WNWSESWNW

Three successive expansions of a segment directed S are shown in Figure 14. In this path, $\delta(P) \leq 2$ for all vertices, and therefore $T(n) = O(n)$.

For simulating an average case, we generated random paths with no U-turns of lengths n from 100 to 54000 (note that such random paths are generally not crossing free). Figure 15 shows the maximum vertex degree $\max\{\delta(P_i)\}$ and the sum $\sum_{i=1}^{n} \delta(P_i)$ of all vertex degrees in such paths, as a function of n. Both quantities have large fluctuations, but the overall trend of the vertex degree is sublinear, and that of the sum of vertex degrees (which gives the overall time complexity) is linear in n.

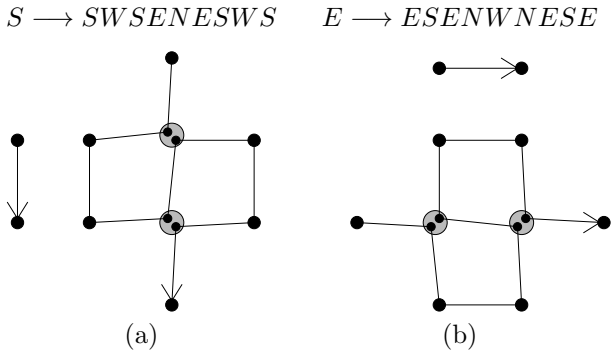

Figure 13: (a) Expansion of a segment directed S and (b) expansion of a segment directed E. Expansions of segments directed W or N are symmetric to these ones, just exchanging the pairs of directions E,W and S,N.

7 Extensions and limitations

We posed two requirements for our algorithm: the input path must not contain U-turns, and it must not traverse the starting vertex P_0 twice (it may only return at P_0 as the last vertex, i.e., $P_n = P_0$, in case of a closed path). Both such conditions can be checked in linear time over the word $w = w_1...w_n$ encoding the path. For U-turns, w must not contain subsequences EW, WE, NS or SN. For another vertex P_k, with $0 < k < n$, to have the same coordinates as P_0, the prefix $w_1...w_k$ must have an equal number of E and W, and an equal number of N and S. In the following, we discuss the reasons for such requirements and whether they can be relaxed.

7.1 Paths with U-turns

Our algorithm relies on the fact that the position for inserting s_{out} in the radial order at a traversed vertex P is uniquely determined by the known offset of s_{in} and the local situation at P. In the presence of U-turns, this is no longer true.

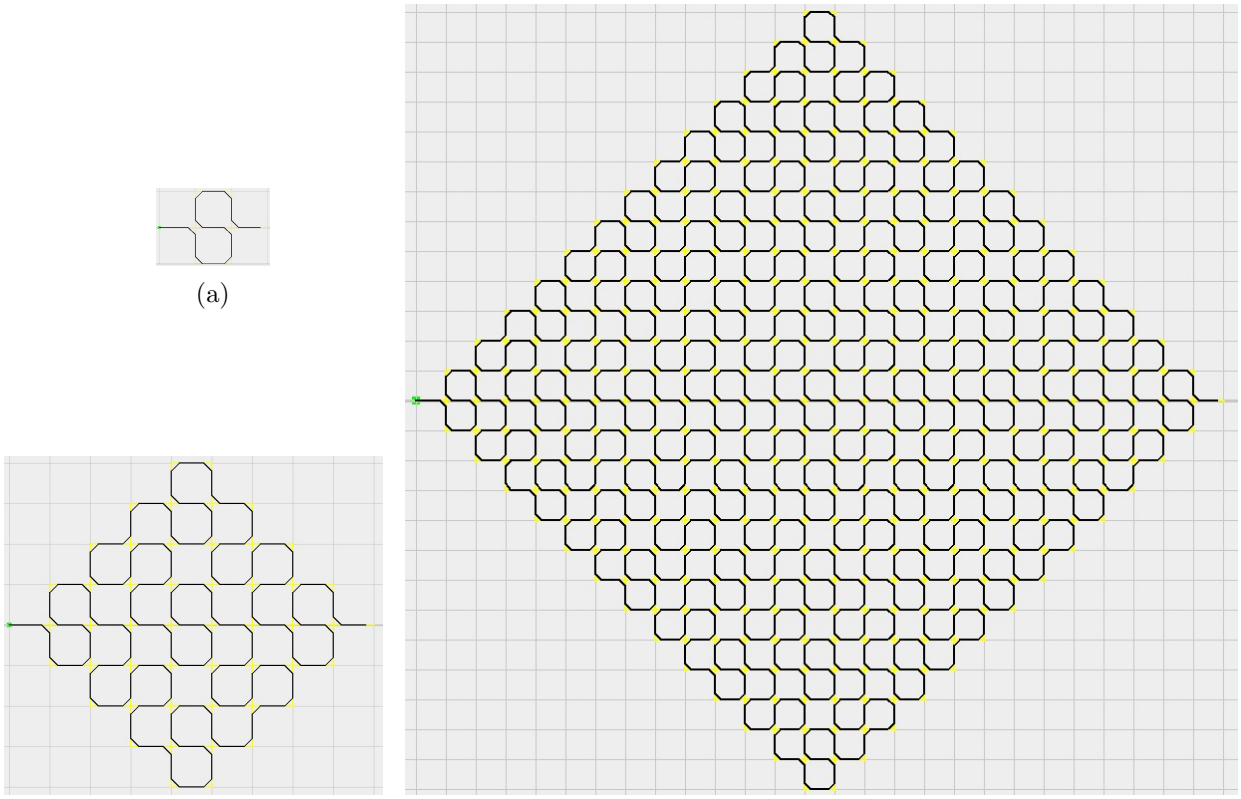

Figure 14: A fractal path: (a) first, (b) second, and (c) third expansion of a segment directed E.

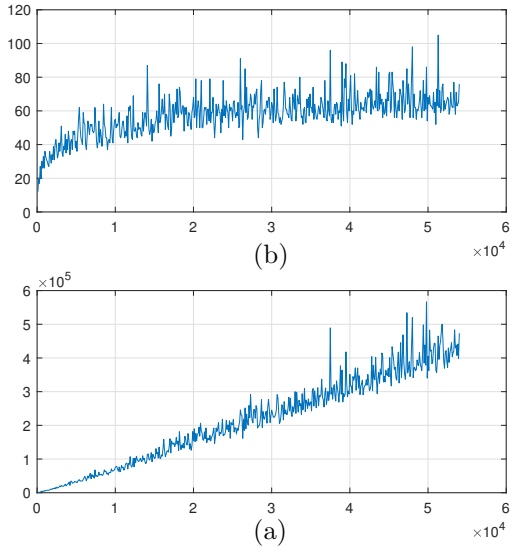

Figure 15: (a) Maximum vertex degree and (b) maximum sum of vertex degrees, computed for random paths of length n from 100 to 54000. For each length, 10 random paths were considered.

U-turns introduce an ambiguity which cannot be solved locally, if it can be solved at all. At a U-turn, the exiting segment s_{out} has the same direction (as seen from the current vertex P_i) as the entering segment s_{in}. Locally, we cannot decide where to put s_{out} in the radial list: it can be equivalently placed just before or just after s_{in} (with appropriate offset). A solution could be taking one of the two possibilities, and backtrack if it does not lead to a crossing-free drawing. But this would give an exponential time complexity in the number of U-turns.

7.2 A relaxed condition for the first vertex

A second passage through the first vertex P_0 may cause an ambiguity similar to that arising with U-turns. In the example of Figure 16, when arriving at $P_5 \equiv P_0$, the drawing may continue either above or below the existing segment exiting from P_0 in the direction E.

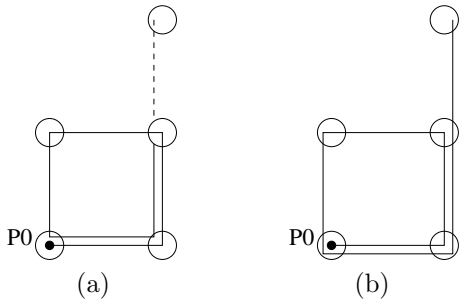

Figure 16: The possibility of drawing the path ENWSENN without self-crossings depends on whether we pass (a) above or (b) below the first segment.

This ambiguity may arise only when another time we pass through P_0, the new exiting half segment s_{out} has the same direction as the first move w_1 of the path. Since the first half-segment (corresponding to the first move w_1 from P_0) has no mate, s_{out} could be placed either before or after it in the radial order, and we cannot determine locally which of the two choices (if any) will allow to draw the path with no intersections.

Instead, there is no ambiguity when the segment s_{out} has a direction different from w_1 or when the entering half segment has direction equal to w_1 (as entering half segments have their offset already set).

Thus, the condition on the starting vertex can be relaxed, as follows. The input path must not pass through a vertex P_i, with $0 < i < n$, such that $P_i \equiv P_0$ and $w_{i+1} = w_1$. This condition can also be checked in linear time in the length n of the word encoding the path.

Let π be a closed path satisfying this relaxed condition. If the word representing π has been processed until the last move w_n without finding a self-crossing, we can conclude that π is crossing-free only if it is possible to join the first and the last edge without creating a self-crossing at the vertex $P = P_n \equiv P_0$. In other words, in the radial list of the vertex $P = P_0 \equiv P_n$, the entering half segment s_{in} (opposite half segment of the last move w_n from P_{n-1} to P_n) must be feasible as the mate of the half segment s_0 exiting from P_0 (created when processing the first move w_1).

After adding s_{in} to the radial list of P, we try to insert a copy of s_0 with undefined offset in the same radial list (we mimic the re-insertion of the already present half segment s_0). If a feasible position for it is found, and this position is immediately before or immediately after s_0, then the path is crossing-free. Otherwise, the path is self crossing.

7.3 Paths with arbitrary first vertex

Let the open path $w = w_1 w_2 \dots w_n$ not satisfy the relaxed condition of Section 7.2 on its first vertex P_0. If the reversed path, described by the word $w' = -w_n - w_{n-1} \dots - w_1$ and starting from P_n, satisfies the same condition, then we can draw w' instead of w. However, digital paths exist for which this is not possible.

For a closed path w, we can find a vertex of the path that is traversed only once, and consider a cyclic permutation of the path, such that that vertex is the first one. However, closed digital paths exist which traverse each vertex multiple times. Moreover, checking how many times a vertex is traversed has a linear cost in the path length n, and we may need to check all vertices, which brings to a quadratic time complexity.

8 Concluding remarks

We proposed an intuitive algorithm to detect self-crossing digital paths in the 2D square grid, and to draw without crossings a path that is not self-crossing.

This extends the works of Brlek et al., who considered the problem of testing whether a digital path passes twice through the same vertex [6] and whether a closed digital path, that passes no more than twice through a vertex, is self-crossing [7]. For our algorithm, paths can be open or closed, and they may traverse the same vertex an arbitrary number of times, with the only limitation that they must not contain U-turns, and, if open, they must pass only once through one of their endpoints.

An interactive demo, implemented in Java, is available at `https://github.com/pmagillo/PathChecker`. The demo allows the user to draw a path without crossings, or to detect that it is self-crossing. The drawings in Figures 1, 11, 12 and 14 are produced with it.

The same approach can be extended to digital paths in other grids (e.g., regular triangular or hexagonal grids) by just considering a different number of possible directions for the half segments around a vertex. It can also be extended to any situation where a path has a predefined set of possible directions to move, for example paths in a graph, or following the edges of a polygonal surface. This opens to way to applications in mesh navigation and path planning.

9 Acknowledgments

This research (paper) has been supported by the Ministry of Science, Technological Development and Innovation through project no. 451-03-47/2023-01/200156 "Innovative scientific and artistic research from the FTS (activity) domain".

References

[1] T. G. Abbott, E. D. Demaine, and B. Gassend. A Generalized Carpenter's Rule Theorem for Self-Touching Linkages. *CoRR*, abs/0901.1322, 2009.

[2] E. M. Arkin, S. P. Fekete, and J. S. B. Mitchell. An algorithmic study of manufacturing paperclips and other folded structures. *Comput. Geom.*, 25(1-2):117–138, 2003.

[3] A.N. Azmi, D. Nasien, and F.S. Omar. Biometric signature verification system based on freeman chain code and k-nearest neighbor. *Multimedia tools and applications*, 76(14):15341–15355, 2017.

[4] B. Banerjee and B. Chandrasekaran. A framework of Voronoi diagram for planning multiple paths in free space. *J. Exp. Theor. Artif. Intell.*, 25(4):457–475, 2013.

[5] Abdelhak Boukharouba and Abdelhak Bennia. Novel feature extraction technique for the recognition of handwritten digits. *Applied Computing and Informatics*, 13(1):19–26, 2017.

[6] S. Brlek, M. Koskas, and X. Provençal. A Linear Time and Space Algorithm for Detecting Path Intersection. In *Discrete Geometry for Computer Imagery, 15th IAPR International Conference, DGCI 2009*, pages 397–408, 2009.

[7] S. Brlek, X. Provençal, and J.-M. Fedou. On the tiling by translation problem. *Discrete Applied Mathematics*, 157(3):464–475, 2009.

[8] Alexiei Dingli, Mark Bugeja, Dylan Seychell, and Simon Mercieca. Recognition of handwritten characters using google fonts and freeman chain codes. In Andreas Holzinger, Peter Kieseberg, A Min Tjoa, and Edgar Weippl, editors, *Lecture Notes in Computer Science*, volume 11015, pages 65–78. Switzerland: Springer International Publishing AG, 2018.

[9] H. Freeman. On the Encoding of Arbitrary Geometric Configurations. *IRE Trans. Electronic Computers*, 10(2):260–268, 1961.

[10] H. Freeman. Boundary encoding and processing. In *Picture Processing and Psychopictorics*, pages 241–266. Academic Press, New York, 1970.

[11] H. Freeman. Computer Processing of Line-Drawing Images. *ACM Comput. Surv.*, 6(1):57–97, 1974.

[12] R. Jana and C. Ray. Image registration using object shape's chain code. In *2nd International Congress on Image and Signal Processing*, pages 1–5. IEEE, 2009.

[13] Jongan Park, Khaled Mohammad Mohiuddin Chisty, Jimin Lee, Youngeun An, and Youngil Choi. Image retrieval technique using rearranged freeman chain code. In *2011 First International Conference on Informatics and Computational Intelligence*, pages 283–286. IEEE, 2011.

[14] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[15] M. Zhao and H. Lu. Robot Path Planning Based on Freeman Direction Chain Code. In *IEEE Symposium Series on Computational Intelligence, SSCI*, pages 1967–1973. IEEE, 2019.