TopoGO: Knot Detection of Images

Yicheng Zhang
Department of Electronic and Electrical Engineering
Southern University of Science and Technology
Shenzhen, China
12210714@mail.sustech.edu.cn
GitHub: @RUSRUSHB

Zhengyang Cao
Department of Electronic and Electrical Engineering
Southern University of Science and Technology
Shenzhen, China
12110623@mail.sustech.edu.cn
GitHub: @drinktoomuchsax

***Abstract*—This project implements recognition of knot pictures with two approaches, getting their Alexander polynomials to uniquely describe their property. The first approach using sliding window achieved 95.4% success rate, and the second approach using thinning achieved 96.7%. Both are capable of automatically detecting errors and can achieve 100% success rate for regulated inputs. Through the fusion of image processing techniques and topological transformation, our project stands as a testament to the efficacy of computational methods knot analysis.**

I. Introduction

Knot theory is a branch of mathematics that plays a pivotal role in understanding complex structures prevalent in science fields such as physics, chemistry, biology, as well as cultural fields like art and archaeology. Central to knot theory is the notion of the Alexander polynomial, a powerful invariant that encapsulates the essential characteristics of knots. Traditionally, the determination of a knot's Alexander polynomial has relied heavily on manual techniques, which are laborious and prone to errors.

In this paper, we present two approaches to knot analysis that leverage advanced image processing techniques and mathematical modeling to automatically extract the Alexander polynomial from knot images. Furthermore, we have developed a robust visualization method to step-by-step showcase the processing steps. This provides a powerful tool for educational and outreach purposes, enabling clear and accessible explanations of the intricacies involved in knot analysis.

II. Related Work Review

Knot invariants are essential tools for classifying and distinguishing between different knots. Various methods exist to obtain these invariants, each utilizing different mathematical techniques and properties. The key methods include the Alexander polynomial, Jones polynomial, HOMFLY-PT polynomial, and Vassiliev invariants, among others. Notably, the Alexander polynomial can be derived from intuitive graphical operations that require only high-school-level knowledge, in contrast to the more abstract topological transformations required by other methods. It also behaves nicely for properties like palindromes[3].

Considering the lack of prior research on knot image recognition, we chose to explore the Alexander polynomial due to its foundational nature. In addition, this method is also more accessible for public education and outreach, thereby enhancing the broader impact of our work.

A. Procedure of Obtaining Alexander Polynomial

The following steps outline the procedure for obtaining the Alexander polynomial:

- Construct a planar diagram of the knot.
- Number the line segments and crossings in the sequence of the direction of the line. (See Fig. 1)
- Construct the Alexander matrix in accordance with the property of each crossing. (See Fig. 2)
- Calculate the determinant of a submatrix of the Alexander matrix, and normalize the polynomial.

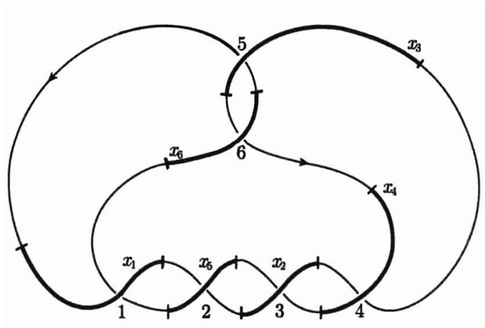


Fig. 1: Example Numbered Diagram[1]

$$\begin{pmatrix} 1-t & -1 & 0 & 0 & -1 & 0 \\ 0 & t & 1-t & -1 & 0 & 0 \\ 0 & 0 & 0 & t & 1-t & t \\ 0 & 0 & -1 & 1-t & 0 & -1 \\ -1 & 1-t & t & 0 & 0 & 0 \\ t & 0 & 0 & 0 & t & 1-t \end{pmatrix}$$

Fig. 2: Example Alexander Matrix[1]

Detailed procedures and explanations can be found in most topology textbooks.

B. Approach of Implementation

We chose double-line hand-drawn pictures as input images[2] (see Fig. 3) which are more complex than the single-line ones. We will also show that our algorithm can handle single-line images.

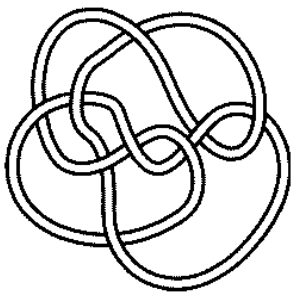


Fig. 3: Binarized Image

Our task is characterized by a fundamental dualism: At each step, mathematics guarantees a clear distinction between correctness and error, with any misstep resulting in failure. As a consequence, we favor deterministic methodologies over probabilistic ones. Additionally, the absence of labeled knot images and that knots with the same topological structure can appear very differently (see Fig. 4) further precludes the feasibility of employing deep learning approaches.

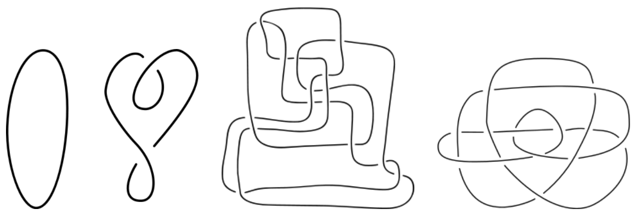


Fig. 4: Same knot with Different Appearances

The challenges of the task lie in two parts. The first part is to identify the components correctly: Where are the line segments? Where are the crossings? The second part is to understand the topological relations among the components: Which line is covering other lines in a crossing? How to sort the line segments and crossings along the direction of the line correctly?

For the first challenge, it is intuitive and feasible to address the problem using morphological methods, just as humans do. For the second challenge, imitating human actions may not be as easy as we thought because imagination and holistic cognition are needed, which is hard for computers to achieve. In the following parts, we will show the methods we devised to overcome these challenges.

III. Methodology

The fig. 5 shows the overall process of our algorithm. First, we start with the original picture, which is pre-processed into a binary image. This binary image is then segmented to produce a segmented image.

Next, we differentiate lines from the background, resulting in a lines image. From here, we have two approaches: the sliding window technique and thinning.

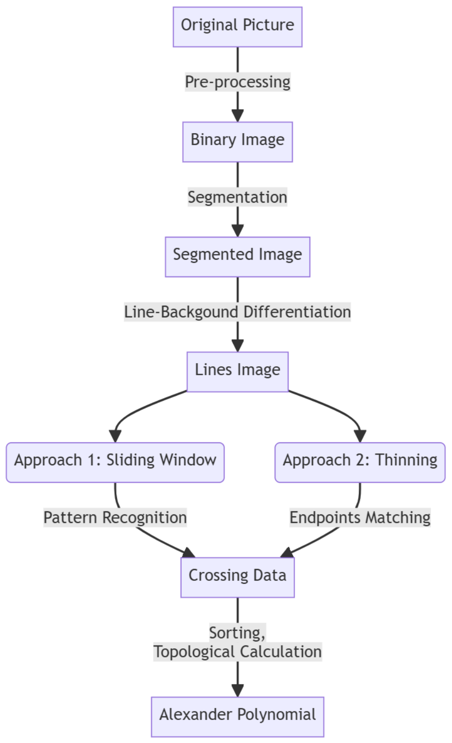


Fig. 5: Diagram of Overall Process

The sliding window approach leads to pattern recognition, while the thinning approach involves endpoints matching to derive crossing data. Finally, the results from both approaches undergo sorting and topological calculations to compute the Alexander polynomial.

A. *Common Processing Methods*

- Binarization

B. *Detailed Steps*

1) Union-Find Labeling: (Contributor: Zhengyang Cao) We need to transform the binarized image into a more usable image that distinguishes which pixels belong to lines and which pixels are part of the background. To achieve this, we first label the connected components in the binarized image using a method called Union-Find Labeling.

This method involves using a data structure known as Union-Find[6] (or Disjoint Set Union) to efficiently manage and merge connected components. The Union-Find data structure allows us to dynamically group and track sets of connected pixels. Each set represents a connected component in the image. The process can be broken down into several key steps, as described below.

a) Key Steps of the Algorithm:

- **Initialization**: Create a label matrix to store labels for each pixel and initialize the Union-Find data structure to manage the connected components.
- **First Pass: Label Assignment and Union Operation**: Traverse each pixel in the image. For each foreground

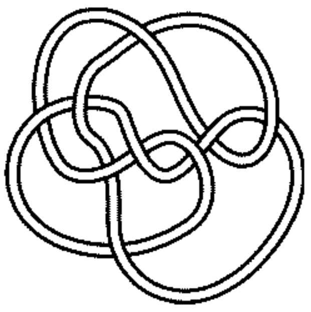
pixel (value 255), check its left and top neighbors. Assign a new label if no labeled neighbors are found. If labeled neighbors are found, assign the smallest label and union the sets.
- **Second Pass: Path Compression**: Traverse the image again to update each pixel's label to its root label.
- **Reassign Labels for Consistency**: Ensure labels are consecutive starting from 1 for better readability.

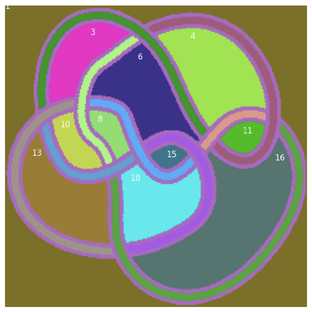
Listing 1: First pass: Label assignment and union operation

```
for y in range(height):
    for x in range(width):
        if img[y, x] == 255:  # Process
        ↪ foreground pixels
            neighbors = []
            if x > 0 and labels[y, x - 1] >
            ↪ 0:  # Left neighbor
                neighbors.append(labels[y, x
                ↪ - 1])
            if y > 0 and labels[y - 1, x] >
            ↪ 0:  # Top neighbor
                neighbors.append(labels[y -
                ↪ 1, x])

            if not neighbors:  # No labeled
            ↪ neighbors
                labels[y, x] = current_label
                parent[current_label] =
                ↪ current_label
                rank[current_label] = 0
                current_label += 1
            else:  # Labeled neighbors exist
                smallest_label =
                ↪ min(neighbors)
                labels[y, x] = smallest_label
                for neighbor in neighbors:
                    union(parent, rank,
                    ↪ smallest_label,
                    ↪ neighbor)
```

The result of this method is showed in fig. 6.


(a) Original Line Image

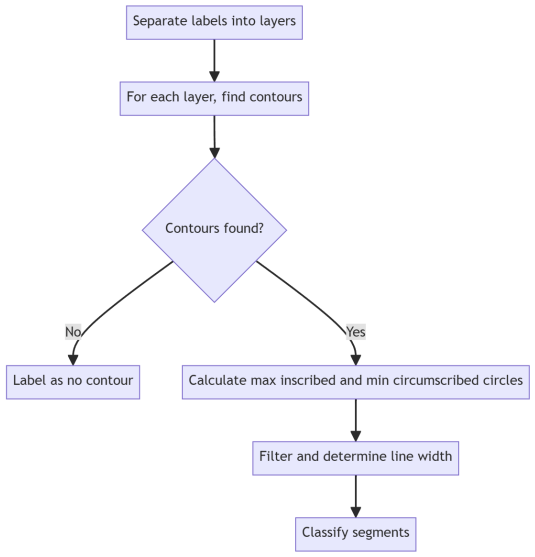


(b) Connected Components

Fig. 6: Segmentation of the binarized image. (a) shows the original line image. (b) displays the connected components after applying the Union-Find Labeling method.

2) Maximum Inscribed Circle: (Contributor: Zhengyang Cao)
After obtaining the labeled image, it is essential to distinguish between the line segments and the background. This distinction is achieved by analyzing the geometric properties of the contours within each layer.

The process is implemented in the Python file `is_line.py`, which follows the procedure outlined in fig. 7



Separate labels into layers
For each layer, find contours
Contours found?
No
Yes
Label as no contour
Calculate max inscribed and min circumscribed circles
Filter and determine line width
Classify segments

Fig. 7: Process of distinguishing line segments from the background in the Python file `is_line.py`.

To find the Maximum Inscribed Circle in an image layer, follow these steps:

- **Convert the Layer to a Binary Mask**: Convert the image layer to an 8-bit unsigned integer mask, where the pixels belonging to the contour are set to 255 (foreground) and all other pixels are set to 0 (background).
- **Find Contours**: Use the OpenCV function `cv2.findContours` to detect contours in the binary mask. Contours are the boundaries of the objects in the image.
- **Select the Primary Contour**: If multiple contours are found, select the primary contour, usually the largest one, for analysis.
- **Distance Transform**: Create an empty image of the same size as the mask, and draw the contour onto this image. Perform a distance transform on the image using the function `cv2.distanceTransform`. The distance transform computes the distance of each pixel to the nearest zero pixel (boundary).
- **Find the Maximum Distance**: Use the function `cv2.minMaxLoc` to find the maximum value in the distance-transformed image. This value represents the radius of the largest circle that can be inscribed within the contour.

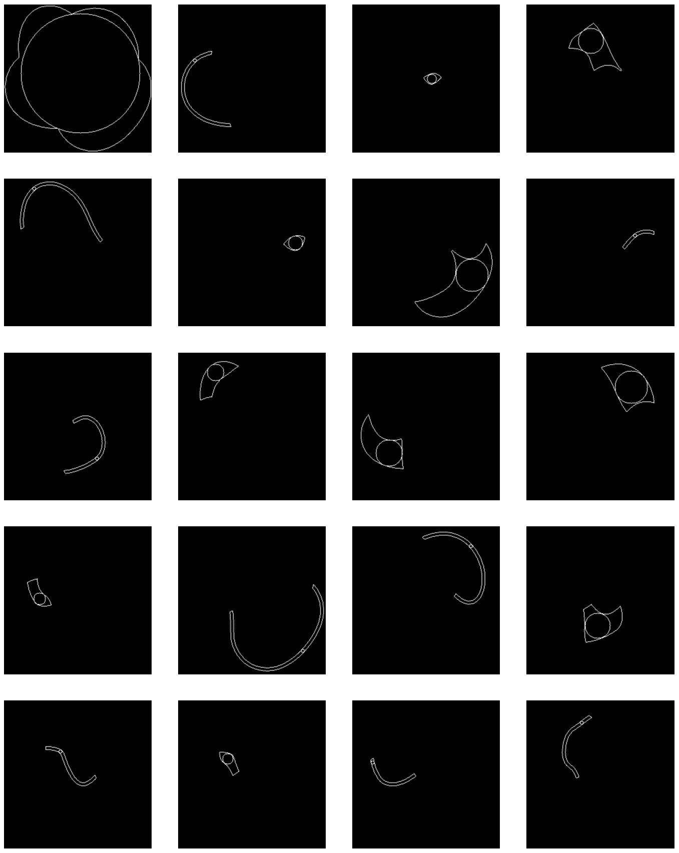


Fig. 8: For each layer in the labeled image we find the Maximum Inscribed Circle

3) Crossing Detection of Thick Lines: (Contributor: Yicheng Zhang)

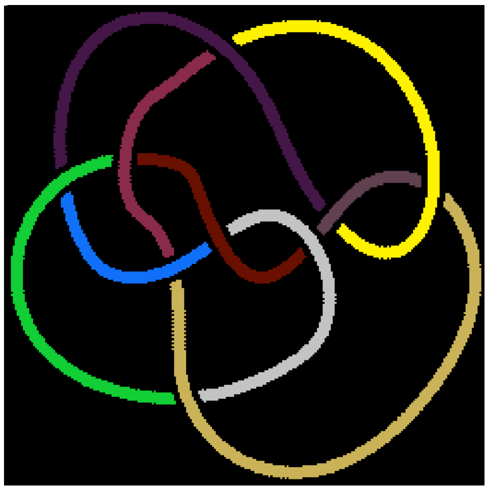


Fig. 9: Thick Lines

After extracting lines(see Fig. 9), we need to identify the crossings and understand the top-bottom relationship there. An easy method is to check whether there are three colors in a window. This method works for many images, but if the crossings are pictured too close to each other, it will fail (see Fig. 10).

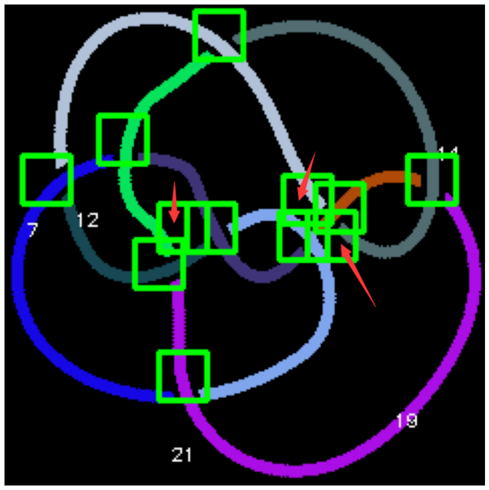


Fig. 10: Three-Color Method: Failed at red arrows

Examinations show that this failure cannot be revised by adding restrictions on area or using adaptive window. We must add restrictions on the pattern.

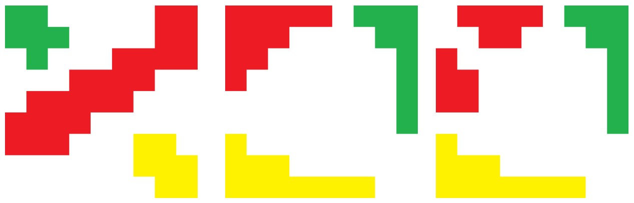


Fig. 11: Crossing Patterns

It is viable to identify the correct pattern with the connectivity test, which requires $O(n^2)$ time and space complexity (n stands for the size of the window). However, we propose a method with $O(n)$ complexity:

Consider the boundary only and ignore the background (see Fig. 12). In correct patterns, the over line and lower lines will appear alternatively, where the over line appears twice and lower lines appear once respectively. Construct a boundary array surrounding the window clockwise from the left-top point (counterclockwise is the same, and the start point can be anywhere), merge the neighboring same color as one, there shall be patterns like [up, down1, up, down2] or [down1, up, down2, up], where down1 and down2 are symmetric.

Listing 2: Boundary Pattern Recognition

```
boundary_labels =
    ↪ np.concatenate([window[0:-1, 0],
    ↪ window[-1, :], window[-2:0:-1, -1],
    ↪ window[0, -1:0:-1]])
# constructed boundary labels

boundary_labels = boundary_labels[edge_labels
    ↪ != background]
```

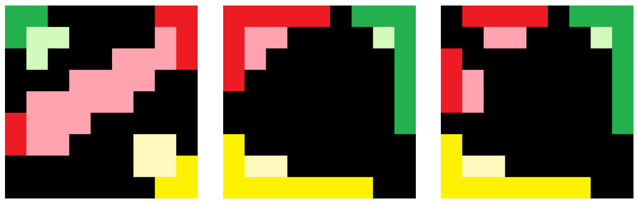

(a) Correct Pattern (b) Fake Pattern 1 (c) Fake Pattern 2

Fig. 12: Crossing Pattern Boundaries: Black and light color region is ignored

```
i = 0
while i < len(edge_labels) - 1:
  if edge_labels[i] == edge_labels[i + 1]:
    edge_labels = np.delete(edge_labels, i)
  else:
    i += 1
# merged same colors

if edge_labels[0] == edge_labels[-1]:
  edge_labels = edge_labels[:-1]
  # deleted redundant color

for label in boundary_labels:
  if boundary_labels.count(label) == 2
    up_label = label
    # up line appears twice
    break

[down_label_1, down_label_2] = [x for x in
    ↪ unique_labels if x != up_label]
# down lines are the others

u, d1, d2 = up_label, down_label_1,
    ↪ down_label_2
# rename for simplicity

boundary_labels = boundary_labels.tolist()
possible_labels = [[u,d1,u,d2],[u,d2,u,d1],
    ↪ [d1,u,d2,u],[d2,u,d1,u]]
for label in possible_labels:
  if edge_labels == label:
    is_true_alternating = True
  if not is_true_alternating:
    print('Fake pattern.')
    return False
}
```

For example, in Fig. 12(a), the array is [Green, Red, Yellow, Red(, Green)], which correspond with [down1, up, down2, up]. You have noticed that due to the disconnectedness between the first and last element in an array, there might exist one redundant color which should be removed.

The improved algorithm succeeded in preventing fake crossings (see Fig. 13). The success rate rises to 95.3% from 66.2%, and the remaining failed cases are not about fake crossing.

4) Crossing Detection of Thin Lines: (Contributor: Yicheng Zhang)

For images of lines with one pixel radius (see Fig. 14), the procedure is easier. Get the endpoints with neighborhood

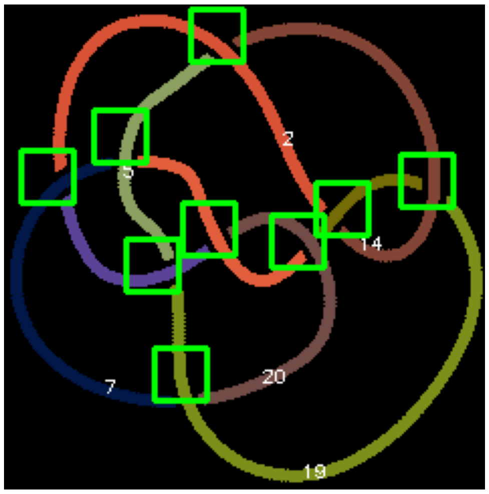


Fig. 13: Correct Crossing: With alternating pattern test

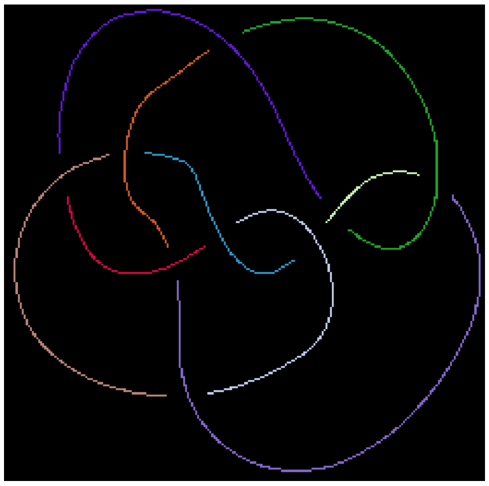


Fig. 14: Thin Lines

detection. A point is an endpoint if one and only one point in its 8-neighborhood are the same with it.

Listing 3: Endpoints Extraction

```
def get_endpoints(line):
  endpoints = []
  for y in range(1, line.shape[0] - 1):
    for x in range(1, line.shape[1] - 1):
      if line[y, x] > 0:
        neighborhood = line[y-1:y+2, x-1:x+2]
        if np.sum(neighborhood) == 2:
          endpoints.append((y, x))
  return endpoints
```

Then match the endpoints of different line segments. We prioritize matching endpoints which are near, and confirm the matching only if the connection between them overlaps another line. Fig. 17(b) shows why it is needed to check overlapping: red lines are matched with smallest location, but only the green ones are correct. KDTree[4], a fast way of distance sorting is used to accelerate the matching.

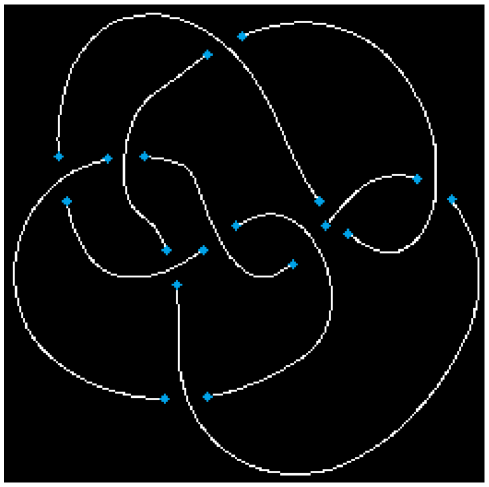


Fig. 15: Endpoints

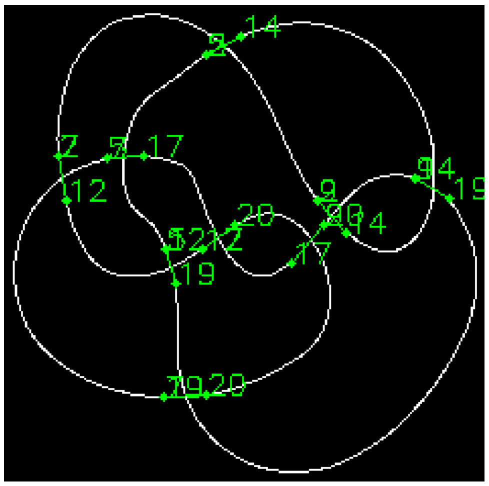


Fig. 16: Endpoints Matching

Listing 4: Endpoints Matching

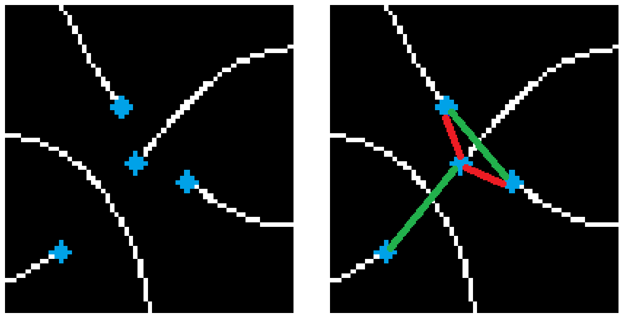
```
tree = KDTree([(y, x) for y, x, label in
    ↪ all_endpoints])
crossings = []
# store recorded matchings
matched_endpoints = set()

for y, x, label in all_endpoints:
  distances, indices = tree.query((y, x), k=6)
  # if the 6 nearest points all failed,
      ↪ abandon search
  # k can be larger if is needed

  for idx in indices[1:]:
    ny, nx, nlabel = all_endpoints[idx]
    if nlabel == label:
      continue  # ignore itself
    if (y, x) in matched_endpoints or (ny,
        ↪ nx) in matched_endpoints:
      continue # ignore if the destination
          ↪ point is matched

    test_image = np.zeros_like(centerlines,
        ↪ dtype=bool)
    test_image = dilate_line(test_image, (y,
        ↪ x), (ny, nx), width=3)
    # check the lines between them
    overlap_labels =
        ↪ set(centerlines[test_image])
    overlap_labels.discard(background)
    overlap_labels.discard(label)
    overlap_labels.discard(nlabel)
    # discard background, itself and the
        ↪ destination label
    # if there is one and only one label,
        ↪ then it's a crossing
    if len(overlap_labels) == 1:
      crossing = [list(overlap_labels)[0],
          ↪ label, nlabel]
      crossing[1], crossing[2] =
          ↪ sorted([label, nlabel])
      # start point and destination point are
          ↪ symmetric

      if crossing not in crossings:
        crossings.append(crossing)
        matched_endpoints.update([(y, x),
            ↪ (ny, nx)])
```


(a) Endpoints: Regional

(b) Endpoints Matching:
Red is wrong, green is right

Fig. 17: Endpoints and Matching: Regional

It is noteworthy that checking overlapping on the connection line may encounter mistake (see Fig. 18(a)). This is because lines extend in an 8-neighboring way, so two intersecting lines may not overlap if they cross diagonally. The revision is to check for overlap on a dilated line (see Fig. 18(b)).

5) *Crossing Sorting:* (Contributor: Yicheng Zhang)

(To professor: This part exceeds the requirement of ”each person explains two algorithms in maximum”, so you have to ignore it in evaluation. For the integrity of the overall logic of this paper, I find it impossible to remove it from our paper, and that’s why there it is.)

After crossing detection, we have the information of all crossings, but not in the correct sequence with the direction of the line (see Fig. 19). If mirror human action, computers need $O(n)$ and even $O(n^2)$ complexity of operation to ”move along the line”, using algorithms like flood fill. However, we propose an $O(1)$ operation as follows.

(See Fig. 20.) Consider the fact that all line segments have two and only two endpoints, which appear in two different

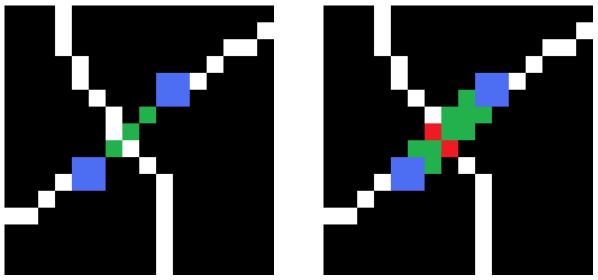

(a) Overlap Check Failed (b) Check with Dilated Lines: Red part is overlapping

Fig. 18: Overlap Check

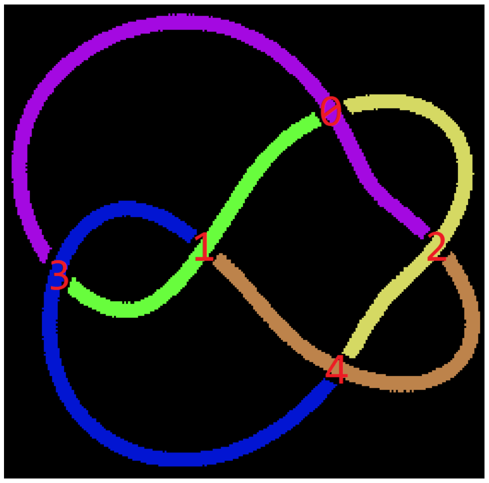


Fig. 19: Unsorted Crossings

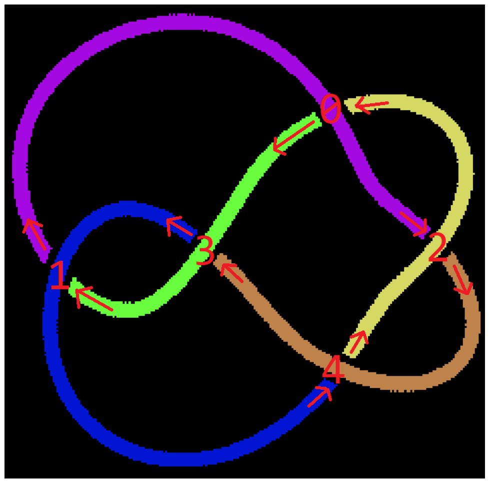


Fig. 20: Sorted Crossings

crossings as down line. Set one crossing as the first crossing (crossing_0), and set one of the down line as the starting current line (green line). Then, the next crossing is another crossing with the current line as a down line (in crossing_1, green line is a down line). So, now we have the information that the green line moves from crossing_0 to crossing_1. Then we change the current line to the other down line of crossing_1 (which is purple). Repeat this procedure, we will sort the crossing in an organized way in the direction of the line. In the figure, the sequence is: green, purple, brown, blue, yellow.

Listing 5: Crossing Sorting

```
raw_uplines = raw_data[:, 0]
raw_downlines = raw_data[:, 1:]

crossing_list = [0]
# start from the first crossing
start_line_list = [raw_downlines[0, 0]]
# start from the first down line

crossing_num = len(raw_data)
target = raw_downlines[0, 0]

for i in range(crossing_num):
  for j in range(crossing_num):
    if target in raw_downlines[j] and j not
        ↪ in crossing_list:
      # current down line appears in another
          ↪ crossing,
      # then this crossing is the next
          ↪ crossing
      crossing_list.append(j)

      target = [x for x in raw_downlines[j]
          ↪ if x != target][0]
      # change the current line to the other
          ↪ down line in the crossing
      start_line_list.append(target)
      break

uplines = raw_uplines[crossing_list[i]]
downlines_out = start_line_list[i]
downlines_in = [x for x in
    ↪ raw_downlines[crossing_list[i]] if x !=
    ↪ sorted_data[i, 2]][0]
```

This sorting preserves the information of the direction of line, ensuring correctness of the crossings and line segments numbering. And this requires $O(1)$ complexity only, much better than the $O(n^2)$ operations like flood fill.

C. *Steps Using Existing Code*

- Thinning: ‘cv.ximgproc.THINNING_ZHANGSUEN‘. In ZHANGSUEN thinning, end points and pixel connectivity are preserved[5]. It is used to obtain one-pixel-line images.
- Other OpenCV code: threshold, findContours, distanceTransform, minMaxLoc, minEnclosingCircle

IV. Results and Discussion

A. *Comparison with Other Results*

We did not find other algorithms for Knot Detection in images, but we have developed two approaches for crossing detection, each with a few minor variants. Therefore, we can compare these methods here. In addition, we list some other possible methods to show the efficiency of our algorithms.

To conduct the test, we crawled all the images from the Rolfsen Knot Table [2]. We then manually filtered out the images with obvious errors, resulting in a total of 151 knot images. We ran our tests on these 151 images. The results are summarized in Table I. Lower methods are always superior than the uppers ones in accuracy.

TABLE I: Comparison of Knot Detection Methods

Method	Accuracy	Runtime	Failure Cases
Three-Color	37.1%	110s	(1)(2)(3)(4)
Alternating Pattern	42.4%	217s	(2)(3)(4)
Adaptive Window	66.2%	356s	(3)(4)
Adap. Win. & Alt. Pat.	95.4%	552s	(3)(4)
Thinning	96.7%	207s	(4)

(1) Close crossings
(2) Very close crossings
(3) Large crossings
(4) Background errors

Fig. 21 shows the structures of possible approaches. Three-Color-Method requires $O(m^2)$ complexity, Alternating Pattern requires $O(m^2n)$ complexity (we can consider it $O(m^3)$), and the thinning approach is $O(m^3)$. Note that m is the size of the image, and n is size of the window. Three-Color method is fast and can deal with regulated images (no too near crossings, e.t.c,) perfectly, but always fails with bad inputs that results in 37.1% overall accuracy. Improving Three-Color method with adaptive window and alternating pattern can achieve 95.4% accuracy, while the cost is much higher. Thinning method is a little bit slower than Three-Color method, but is robust to bad inputs with very close crossings or too large ones, achieving 96.7% overall success rate.

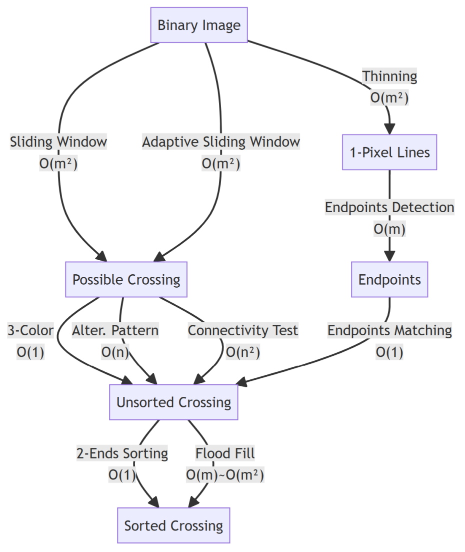


Fig. 21: Complexity of Approaches. m: size of the image. n: size of the window.

B. Results Analysis Summary

The Thinning method proved to be the best, with the highest accuracy of 96.7% and a reasonable runtime of 207 seconds. It is particularly robust against background errors and large crossings, making it a reliable choice for a variety of scenarios.

In contrast, the Three-Color method was the worst performer, with the lowest accuracy of 37.1%. However, this low accuracy is specific to the manually drawn images from the Rolfsen Knot Table [2], which often have closely positioned crossings. Despite its poor performance in this test, the Three-Color method has potential under different conditions. If the dataset were expanded with more images and the crossings were sufficiently spaced apart, the Three-Color method could achieve high or even perfect accuracy while maintaining the shortest runtime. This makes it a viable option for applications where speed is crucial and images are well-regulated.

V. Conclusion

In this paper, we presented two advanced approaches for knot detection in images, utilizing sliding window and thinning techniques. Our primary goal was to automate the extraction of Alexander polynomials from knot images, thus improving accuracy and efficiency compared to traditional manual methods.

The results indicate that the Thinning method is the most effective, achieving an impressive accuracy of 96.7% with a reasonable runtime of 207 seconds. This method's robustness against background errors and large crossings makes it highly reliable for various scenarios.

On the other hand, the Three-Color method, while the fastest, showed the lowest accuracy at 37.1%. However, its performance could significantly improve with more regulated images where crossings are adequately spaced apart, making it a viable option for certain applications where speed is critical.

Our findings underscore the importance of choosing the right method based on the specific requirements of the task at hand. The methodologies developed here not only advance the field of knot detection but also provide a strong foundation for further research and development. Future work could explore enhancing the adaptability of these algorithms to diverse image conditions and expanding their applications in educational and scientific domains.

References

[1] M. A. Armstrong, *Basic Topology*, Springer, 1997.
[2] Knot Atlas, *The Rolfsen Knot Table*, https://katlas.org/wiki/The_Rolfsen_Knot_Table, Accessed: April 1, 2024.
[3] Iva Halacheva, *A foray into knot theory: the Alexander polynomial*, March 27, 2014, p. 41.
[4] Gieseke F., Oancea C., Igel C., *bufferkdtree: A Python library for massive nearest neighbor queries on multi-many-core devices*, Knowledge-Based Systems, 2017.
[5] T. Y. Zhang and C. Y. Suen, *A Fast Parallel Algorithm for Thinning Digital Patterns*, Image Processing and Computer Vision, March 1984.
[6] C. Lee, "Union find algorithm," Medium, https://yuminlee2.medium.com/union-find-algorithm-ffa9cd7d2dba.