TGGLinesPlus algorithm supplementary materials

The supplementary materials are prepared by Liping Yang, Joshua Driscol, and Ming Gong and proof-read by Katie Slack

This document provides some supplementary information to help our readers to understand the *TGGLinesPlus* algorithm.

- Section A below provides a more detailed elaboration with implementation details for the *TGGLinesPlus* algorithm compared with the concise version introduced in the *TGGLinesPlus* paper.
- Section B provides benchmark data source and experiment specifications.

A. Elaboration with implementation details for the *TGGLinesPlus* algorithm

This section provides an elaboration on specific details of our implementation of *TGGLinesPlus*. We hope this will be of use to our readers/users who would like to take advantage of our open-sourced *TGGLinesPlus* algorithm and Python implementation for those who would like to further improve our *TGGLinesPlus* algorithm and its implementation in Python, and/or for researchers looking to port *TGGLinesPlus* to other programming languages. The repository URL for our *TGGLinesPlus* algorithm can be found in the **DATA AND CODES AVAILABILITY STATEMENT** above.

Step 0. Preprocessing: create an image skeleton from input image

Our method expects an image skeleton. That means that it is up to the user to binarize and then skeletonize the image before using the *TGGLinesPlus()* method. As we show here (https://github.com/GeoAIR-lab/TGGLinesPlus/blob/main/notebooks/1_test_skeleton_methods.ipynb), there are many different libraries and algorithms to binarize and skeletonize code that will have a direct effect on how well our *TGGLinesPlus* algorithm works.

Step 1. From the image skeleton, produce an image graph

Once an image skeleton is input to our algorithm (as a 2D *NumPy* array), we convert the image skeleton to a *SciPy* sparse array using the *pixel_graph()* method from scikit-image. At the same time, we extract the position of the pixels of the skeleton in the image for plotting later on. From there, we convert the sparse array into a graph using *NetworkX* and its *from_scipy_sparse_array()* method.

Step 2. Split the main graph into subgraphs

From the main graph, we can call *NetworkX's connected_components()* method to find a list of subgraphs (if any) that are present in the main graph. If there are no subgraphs, then this list of subgraphs has a length of 1, meaning that it contains only the main graph. Each subgraph has "speckle" removed from it, meaning that we remove small subgraphs that are 2 nodes or less in length by default as we assume that this is noise in the image skeleton and resulting main graph. This may not be an appropriate assumption for every use case, however.

Step 3. For each subgraph, do procedures specified in 3.1 and 3.2 sections below

Note that in the description of the *TGGLinesPlus* algorithm in the main body text above, we separate 3.1 in Algorithm 2 and 3.2 in Algorithm 3 to allow users easy to grasp the major overflow of our *TGGLinesPlus* algorithm. Here we put the algorithm back to its sequential order, to make it user-friendly for those who would like to look into the implementation details.

Step 3.1 Simplify the subgraph:
We can now iterate over the subgraphs, for each one performing the following steps:

3.1.1 Find cliques from junction subgraphs

From the degree of each node in the subgraph (how many connections a node has), we can find the node "type" so that we can isolate junctions. We define junctions to be any node with 3+ connections (not a turning node). See Section 3.3 for node types in the *TGGLinesPlus* paper. We create a subgraph from the subgraph that we are iterating over that includes only the junctions contained in it (if any) using *NetworkX* by calling *graph.subgraph()*. Then we get a list of cliques that form triangles in the subgraph using the *find_cliques()* method, also using *NetworkX*, that would complicate the path segmentation algorithm. This helps us isolate edges that are part of cliques so that we can remove those junctions that would complicate path segmentation.

3.1.2 Remove edges

Next we remove the diagonal edges from the cliques because they would cause our algorithm to "double back" on itself during path segmentation. We remove these edges (see Step 6 in Figure 2 and Step 7 in Figure 3 in Section 3.4 for illustration in the *TGGLinesPlus* paper).

3.1.3 Identify the path segmentation endpoints for the subgraph

Since we potentially removed some edges from the subgraph, some junction nodes may no longer have 3+ connections (i.e., are no longer junctions), so should not be in our resulting list of graph segmentation points. After getting a final set of junctions and terminal nodes (where connections = 1), we combine these points and call this list path segmentation points. In essence, these are all the potential starting and ending points that we consider for path segmentation in the simplified subgraph.

Step 3.2 . Segment simplified graph into paths

3.2.1 Create a copy of the simplified graph for path segmentation

We create a copy of the simplified graph to segment paths because we keep track of found paths in each subgraph by removing them from the subgraph copy (more on this below) and we do not want to touch the initial simplified graph. We store the simplified graph in the final returned dictionary, as some users might want to analyze this graph further aside from segmenting paths.

3.2.2 Compute initial paths list

Using the copy of the simplified subgraph that we created, we start by finding the shortest path in a graph between two of our path segmentation points using *NetworkX's shortest_path()* algorithm. When we find it, we add it to the list of segmented paths and then delete it from the copy of the graph we made. We do this until there are no more paths in the copied graph. Note: we originally designed this algorithm using *shortest_simple_paths()* in *NetworkX*. While this worked for small examples (and did not require making a copy of the subgraph for segmenting), it was slow to the point of not completing for medium to large graphs. Now, by iterating over the shortest paths in each subgraph and then deleting them one by one, most small to medium examples complete under one second or within a few seconds. Larger examples still complete on the order of minutes on a laptop computer using a Jupyter Notebook.

3.2.3 Check for cycles in the graph

Cycles begin and end at the same endpoint or endpoints (multiple endpoints in a "string" or line are possible in the same loop), which is different from other paths (that start and end at a separate terminal and/or junction). *NetworkX* does not find these using the shortest_path() algorithm, which finds a path between two nodes, not a node and itself. However, we can check for cycles in our undirected graph using *NetworkX's* cycle_basis() method. These cycles are then added to the full list of segmented paths, if they exist (which they may not). Note: cycles (loops that are part of a graph) are treated the same as perfect loops (loops that are the graph or subgraph itself, think of a number 0). We cover both of these cases with this method.

3.2.4 Split initial paths list into lists containing only two path segmentation endpoints

This initial paths list and the list of cycles often contain paths that contain more than two segmentation points. We want paths that start and end at exactly one path segmentation point, with none in between. Therefore, we split each path (should it need it) into smaller paths.

Step 4. Merge sublists, check to see whether paths span the graph

We have collected lists of junction and terminal nodes, cliques, what edges we removed from each subgraph, the paths included in each subgraph, etc. It is much easier to plot these items when they are in the same list, so we merge them with list comprehensions in Python. However, we keep a list of the subgraphs with their respective variables in a separate list so if a user is interested in indexing through and inspecting subgraphs, they can do so. We created the processing pipeline such that every plotting method will work on both on the main graph or an individual subgraph. If a user is interested in plotting multiple subgraphs, they can simply combine them using list comprehensions like we have done at the end of the *TGGLinesPlus* method. After we have merged the lists of important variables together, we check to see whether the paths we found span the entire graph. We do this by comparing a set of nodes for the entire graph, subtracting the set of "speckle" or noisy nodes in the image skeleton, and then comparing a set of nodes covered in the paths list to those remaining nodes. {nodes in graph} - {noisy nodes} - {nodes in paths} = {}. With this check and the checks we have done during the graph segmentation stage (Step 3.2 in Appendix B), we assert that:

- every node in the main graph is covered either by a path or is classified as noise
- every path starts and ends with exactly 1 path segmentation endpoint
- no path will include a path segmentation point between its starting and end point

Step 5. Return a dictionary that contains useful information to our user

For the convenience of our users, we include useful information at each step in our returned dictionary, detailed below.

- Simplified graph with removed edges
- Path segmentation endpoints (primary junctions and terminals)
- All segmented paths
- Which edges were removed
- A list of subgraphs with each of these variables contained in a subgraph dictionary that can be indexed and then used with each plotting method

B. Benchmark data source and experiments specifications

The experiments and computational environment specifications (including data sources, implementation, and runtime) for the *TGGLinesPlus* algorithm introduced in the *TGGLinesPlus* paper are provided below.

B.1 Benchmark dataset source specifications

Table 1 below provides the data source and data format for the benchmark results (detailed in Section 4.2 in the *TGGLinesPlus* paper). Our well-considered and carefully chosen benchmark datasets range from small to medium and large image sizes and cover various domains, from character isolation and segmentation in documents, medical image segmentation, and satellite imagery (line segmentation) analysis.

Table 1. Benchmark data source and data format. The image # corresponds to the image # in Figures 4, 5, 6, 7. The original MNIST and CMNIST datasets are stored in CSV format. In order to use these datasets for our benchmark comparison, we keep the exact dimension from the CSV and save each input image as a NumPy array in Python.

Image #	**File dimensions**	**File format**	**File size**	**Data source**	**Notes**
01	28, 28	CSV	0.784 KB	https://www.kaggle.com/datasets/oddrationale/mnist-in-csv	Each MNIST image is a 28 x 28 (784 pixels) handwritten digit from "0" to "9." Each pixel value is a grayscale integer between 0 and 255.
02	64, 64	CSV	4.096 KB	https://www.kaggle.com/datasets/fedesoriano/chinese-mnist-digit-recognizer	CMNIST, index 6000
03	64, 64	CSV	4.096 KB	https://www.kaggle.com/datasets/fedesoriano/chinese-mnist-digit-recognizer	CMNIST, index 10,000
04	1500, 1500	TIF	2.3 MB	https://www.kaggle.com/datasets/balraj98/massachusetts-roads-dataset	Massachusetts Roads Dataset, specific TIF: 11278840_15.tif
05	1411, 1411	uint8 ndarray	1.990921 MB	https://scikit-image.org/docs/stable/api/skimage.data.html#skimage	Scikit-image retina

				.data.retina	
06	384, 544	PNG	210 KB	https://github.com/yhlleo/DeepCrack	DeepCrack dataset, specific image: 11215-5.png
07	177, 191	PNG	52 KB	USGS EarthExplorer data portal (https://earthexplorer.usgs.gov)	A zoom-in sub-image from Landsat-8 OLI panchromatic image acquired on 7 December 2018 over the Amery Ice Shelf, which was downloaded from USGS EarthExplorer data portal by Wang (fourth author of this paper). The Amery Ice Shelf is a broad ice shelf in Antarctica at the head of Prydz Bay between the Lars Christensen Coast and Ingrid Christensen Coast.
08	277, 266	PNG	105 KB	USGS EarthExplorer data portal (https://earthexplorer.usgs.gov/)	Another zoom-in sub-image from Landsat-8 OLI panchromatic image acquired on 7 December 2018 over the Amery Ice Shelf, which was downloaded from USGS EarthExplorer data portal by Wang (fourth author of this paper)
09	500, 500	PNG	1.5 MB	https://www.cabq.gov/gis/geographic-information-systems-data	Contours: original shapefile of contours was overlaid on a basemap of Albuquerque, NM, and then exported as a PNG
10	191, 384	uint8 ndarray	73.344 KB	https://scikit-image.org/docs/stable/api/skimage.data.html#skimage.data.page	Scikit-Image document page

B.2 Computation configuration, benchmark runtime, and benchmark implementation specifications

To guarantee the reproducibility and replicability of our algorithms and implementation, here we provide the computation configuration, benchmark runtime, and benchmark implementation specifications for references for our readers and users.

- All of the benchmark figures 4 - 7 in the *TGGLinesPlus* paper for the following four methods (i.e., *TGGLinesPlus* EDLines, LSD, and PPHT) were timed (detailed in Table 2 below) and created using a 2019 MacBook Pro with the following specifications: **Processor**: 2.6 GHz 6-Core Intel Core i7; **Graphics**: Intel UHD Graphics 630 1536 MB; **Memory**: 16 GB 2667 MHz DDR4; **OS**: macOS Ventura 13.2.1. (This part of the benchmark was carried out by Driscol (2nd author of this *TGGLinesPlus* paper.)
- All of the benchmark figures in the *TGGLinesPlus* paper for the following two methods (i.e., TGGLines, and Linelet) were timed (detailed in Table 3 below) and created using the following computer: **Processor**: Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz ((6 cores and 12 logical processors); **Graphics**: NVIDIA 2080 TI; **Memory**: 32.0 GB RAM; **OS**: Windows 10 Desktop 64 bit. (This part of the benchmark was carried out by Gong (3rd author of this *TGGLinesPlus* paper; Gong is also the lead author of our previous work TGGLines (Gong et al., 2020), who did most of the TGGLines implementation with Yang's guidance and supervision.)
- In addition, benchmark implementation specifications are provided in Table 4 below.

Table 2. Benchmark runtime recorded for methods *TGGLinesPlus*, EDLines, LSD and PPHT (time in seconds).

Image #	**Filename**	***TGGLinesPlus***	**EDLines**	**LSD**	**PPHT**
01	MNIST	0.01487s	0.00546s	0.00096s	0.00181s
02	CMNIST 1	0.01377s	0.00402s	0.00100s	0.00142s
03	CMNIST 2	0.01068s	0.00581s	0.00086s	0.00123s
04	Roads	14.04825s	0.84893s	0.06589s	0.03751s
05	Retina	27.69127s	0.95755s	0.06135s	0.03772s
06	Cement	1.19188s	0.05973s	0.00989s	0.00661s
07	Landsat 1	0.23510s	0.12720s	0.00336s	0.00335s
08	Landsat 2	0.69258s	0.23047s	0.00587s	0.00824s
09	Contours	260.70512s	1.00062s	0.02429s	0.02428s
10	Page	0.55721s	0.09208s	0.00472s	0.00950s

Table 3. Benchmark runtime recorded for methods TGGLines and Linelet (time in seconds).

Image #	**Filename**	**TGGLines**	**Linelet**
01	MNIST	0.09s	2.39s
02	CMNIST 1	0.12s	2.3s
03	CMNIST 2	0.09s	2.16s
04	Roads	91.96s	39.54s
05	Retina	179.92s	42.34s
06	Cement	19.09s	4.18
07	Landsat 1	3.01s	5.58s
08	Landsat 2	7.02s	6.83s
09	Contours	162.88s	24.17s
10	Page	55.89s	5.08s

Table 4. Benchmark implementation specifications (we used the same parameter setting for each benchmark algorithm from our previous work TGGLines (Gong et al., 2020))

Methods	**Parameters**	**Implementation used for benchmark**
PPHT	*threshold: 10; line length: 5; line gap: 3*	*probabilistic_hough_line* function (Scikit-image) in Python
LSD	*scale: 0.8; sigma scale: 0.6, quant: 2.0, ang th: 22.5, density th: 0.7*	*createLineSegmentDetector* function (OpenCV) in *Python*
EDLines	**Internal parameters:** *{ratio: 50, angle turn: 67.5*np.pi/180, step: 3};* **Parameters for Edge Drawing:** *{ksize: 3, sigma: 1, gradientThreshold: 25, anchorThreshold: 10, scanIntervals: 4 };* **Parameters for EDLine:**	EDLines *Python* implementation (GitHub code)

	{minLineLen: 40, lineFitErrThreshold: 1.0}	
Linelet	*param.thres angle diff: pi/8; param.thres log eps: 0.0; param.est aggregation: Kurtosis*	*Matlab* code by the linelet authors (Cho et al., 2018)
TGGLines	TGGLines requires only one parameter, and it is adaptive (see line 7 in Algorithm 2) in (Gong et al., 2020)	Implemented in *Python* by the first author (Gong) and 2nd author (Yang, corresponding author), along with 3rd author (Potts) in the *TGGLines* paper (Gong et al., 2020)
TGGLinesPlus	**No need to tune any parameters**	Implemented in *Python (Jupyter Notebook)* by first two authors *(Yang & Driscol; where Driscol did most of the implementation with guidance and supervision from Yang)* of this *TGGLinesPlus* paper