Optical Graph Recognition and Analysis (OGRA)

Kevin Jacob, Parv Mahajan, Seohyun Park

Georgia Institute of Technology

MATH 3012 Combinatorics

Yaofeng Su

November 9 2023

Abstract

Graph theory provides a powerful framework for understanding complex relationships in various fields, but analyzing analog graphs remains challenging for non-experts. This paper introduces a script that utilizes Canny Edge Detection, Hough Line Transforms, Hough Circle Transforms, and Euclidean Distance metrics to first convert analog images into computer-readable graph formats, before analyzing qualities such as the components, Euler circuits, and chromatic numbers within the graph through the use of algorithms. Applications for this tool range from digitizing paper-based graphs and simplifying infrastructure evaluations to enhancing educational settings and interdisciplinary research. Additionally, this tool is aimed at democratizing graph theory to benefit a wide audience across disciplines, making graph analysis accessible and practical.

Introduction and Background

The field of graph theory in mathematics forms a framework for understanding complex relationships in various domains ranging from bioinformatics to subway systems. However, being able to obtain critical information from visual graphs quickly poses a challenge to individuals without a background in mathematics or computer science. The methods to prove or disprove a graph's qualities can be arduous for large graphs, as several algorithms are necessary to identify various characteristics such as planarity and the existence of a Hamiltonian path (Keller & Trotter, 2016). Furthermore, translating analog, handwritten graphs to a computer-readable format can be especially difficult with larger graphs, costing time and money (Mahmud et. al., 2019).

To bridge this gap between visual representation and analysis, we introduce a script in which any user can upload and obtain valuable information about any image of a graph. The script has the capability to automatically identify the number of components, check for the existence of an Euler circuit, and determine the graph's chromatic number by use of optical graph recognition and machine learning to identify vertices and edges within the image.

This tool can be utilized in several applications, ranging from digitizing paper-based graphs and simplifying infrastructure diagram evaluation to educational purposes in visualizing abstract mathematical concepts. In a broader scope, the app has the potential to democratize the field of graph theory by making it accessible and applicable to a broader audience across various disciplines, from virology to civil engineering. This research paper presents the development, functionality, and future applications of this script through practical examples.

Novelty and Contribution 1 - Digitization of Graphs

To digitize an image of a graph is a crucial problem at the forefront of graph analysis; it allows for easy analysis and conclusions from a pictorial representation. The format of the text file easiest for analysis is structured as follows: the first row should contain the number of nodes and each consecutive row should show the connections between nodes by referencing two nodes per line.

To accomplish the digitization of a graph image to this text file format, we propose a novel combination of Canny Edge Detection, Hough Line Transforms, Hough Circle Transforms, and Euclidean Distance metrics. Canny edge detection is a multi-step algorithm to outline the presence of edges in an image (OpenCV, n.d.). It begins with noise reduction on the image using a 5x5 Gaussian filter; using this filter helps the image to more easily define the edges. The next step of this algorithm defines edge gradients and directions of each pixel; edges are defined as gradient directions normal to an edge. Next, hysteresis thresholding involves setting minimum and maximum threshold parameters to define the intensity of the edges that should be selected. This allows for easy application of the Hough Line Detection algorithm which allows for the actual edges (in the sense of nodes and edges to be defined). The Hough Line Transform analyzes the intersection of curves between images to define straight lines above a threshold value (OpenCV, n.d.). The Hough Circle Transform allows for digitization to easily determine the number of nodes in an image (OpenCV, n.d.).

The novel algorithm that this paper contributes first utilizes the Hough Circle Transform to define the nodes of the image. Then, Canny Edge Detection is applied to the image to pre-process it for the Hough Line Transform. The Hough Line Transform defines the edges of the image, and then that is when the Euclidean distance metrics are applied. The Euclidean

distance works to determine if the distance from the endpoints of an edge to the center of a node is equal to the node's radius; if so, the node is determined to be attached to that edge. If two nodes are attached to opposite endpoints of an edge then those nodes are neighbors of one another.

Digitization Results

The following figures below represent simple inputs and outputs of this algorithm.

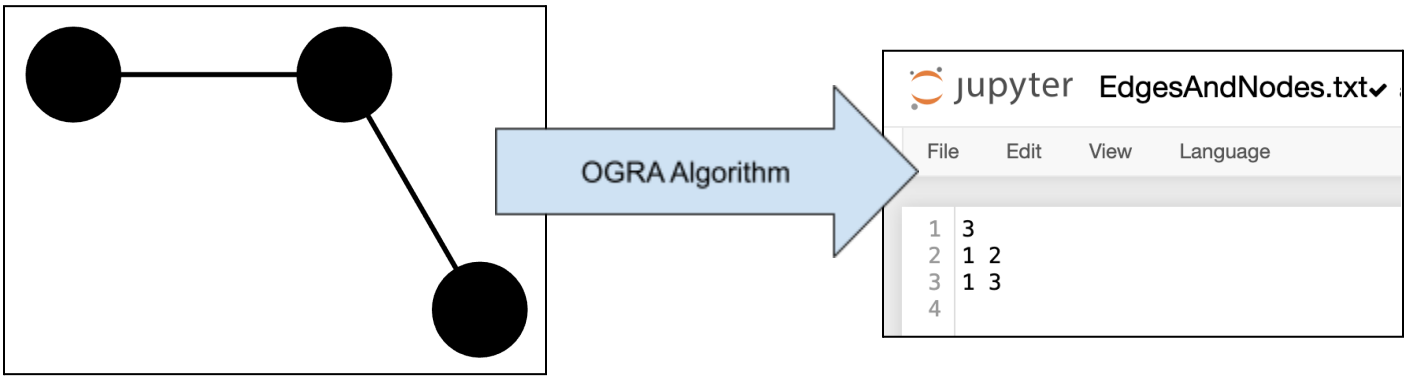


Fig. A: Shows the digitization of a linear graph with 3 nodes and 2 edges. Nodes are arbitrarily assigned a number from 1 to n where n equals the number of nodes present in the image—represented by the first line of the code. In this text file, we see that there are 3 nodes total (first row) and each consecutive row shows a connection between nodes: 1 and 2 as well as 1 and 3 respectively.

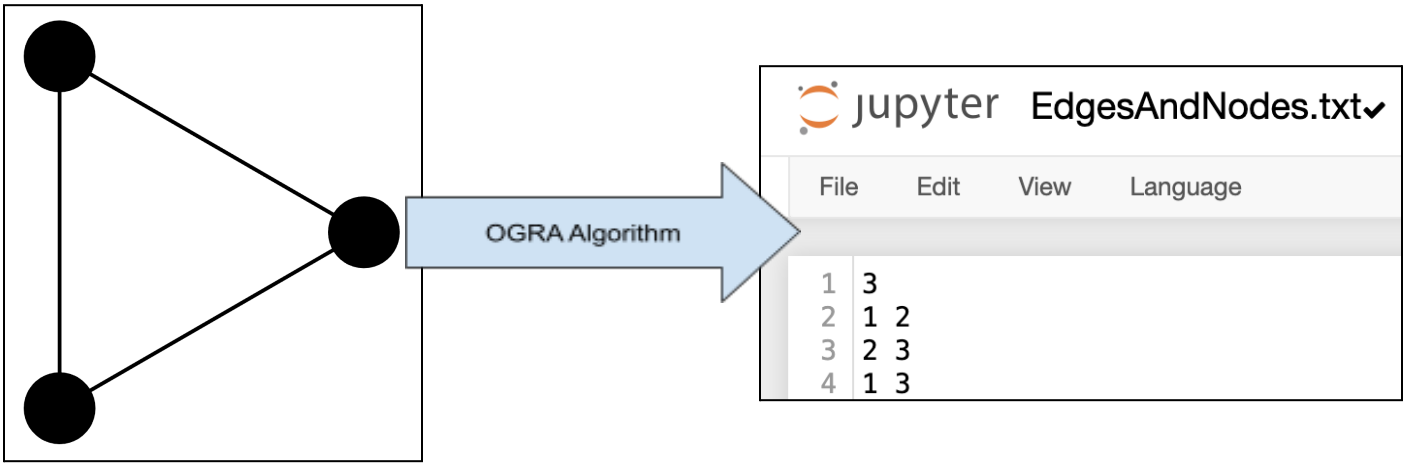


Fig B: Shows the digitization of a cycle with 3 nodes (triangle). Once again, the first row shows the number of nodes, and the consecutive rows show the connections between nodes. Note contrast from the 2 edges and subsequently 2 rows that are present in Fig A.

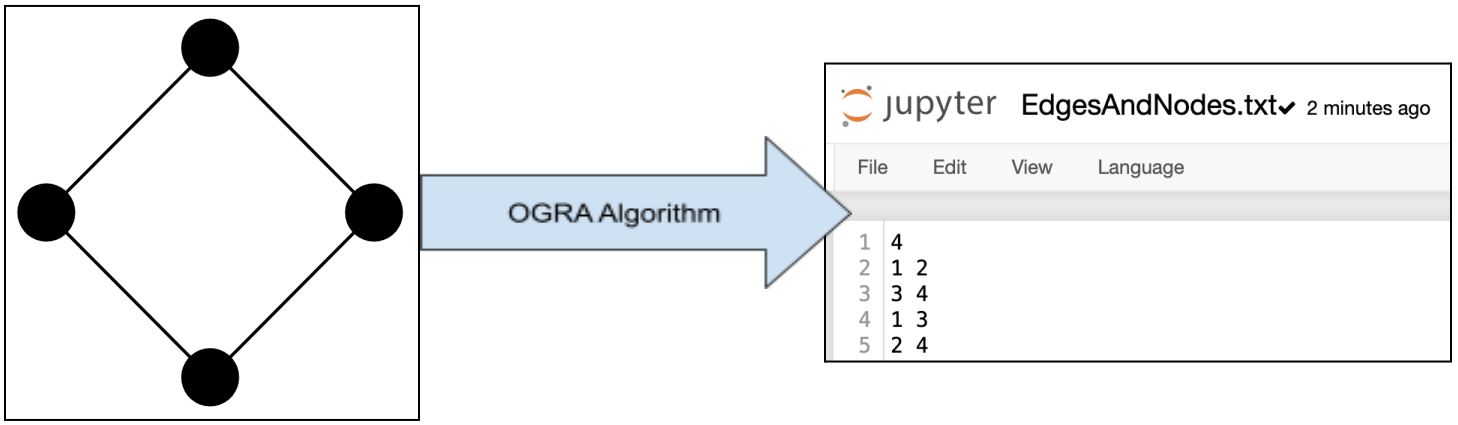


Fig C: Shows the digitization of a cycle with 4 nodes (square). Once again, the first row shows the number of nodes, and the consecutive rows show the connections between nodes.

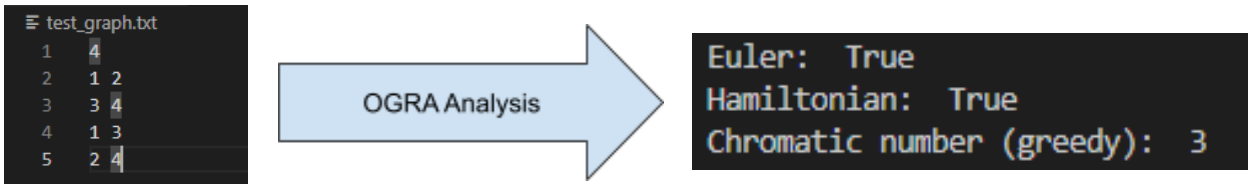
These images demonstrate the effectiveness of our algorithm in digitizing an unlabeled graph into a simplistic, easy-to-analyze format.

Concepts and Contribution 2 - Analysis of Graphs

Three properties of the given graph are tested: the presence of an Euler cycle, the presence of a Hamiltonian cycle, and the chromatic number. After receiving a data file in the format below, an adjacency matrix is created, recording the edges in the graph. This, of course, depends on correct formatting in the data file - if the number of nodes is incorrect, the algorithms will fail.

To determine the existence of an Eulerian cycle, all vertices with non-zero degrees must be connected (Keller & Trotter, 2016). Loose points can be ignored because they cannot be "dead ends" - nothing connects to them (Keller & Trotter, 2016). Additionally, the sum of all vertices must be an even number, as each vertex must have an equal number of "exit" and "entry" nodes (Acosta & Tomescu, 2023). So, the implemented algorithm, after checking that all non-zero vertices were visited through a depth-first search, thus checking connectedness, counted the number of odd-degree vertices. If it is two or zero, then an Euler circuit must exist (Keller & Trotter, 2016).

Fig D (Analysis Results): Shows the analysis of the square graph in Figure C. Note the inefficiency of the greedy algorithm.



Unlike certifying the existence or non-existence of an Eulerian circuit, which can be done in polynomial time, certifying a Hamiltonian cycle, where all vertices are visited exactly once, is NP-complete (Mazzuoccolo & Ruini, 2021). After checking for total connectivity, all possible

paths from a given vertex must be evaluated (Keller & Trotter, 2016), and so a recursive depth-first backtracking algorithm was implemented.

Similarly, calculating chromatic number is NP-hard, and so a greedy algorithm was implemented; for each point, the minimum possible number was used. This algorithm is a useful approximation (Keller & Trotter, 2016) and works in polynomial time.

Conclusion

Our process can be divided into two main portions: Identification and Analysis. Within the identification process, we learned about several aspects of optical graph detection, foremost of which required mastery of several concepts: Canny Edge Detection, Hough Line Transforms, Hough Circle Transforms, and Euclidean Distance metrics. Each step of our algorithm incorporated these aspects, from creating edges using a Gaussian filter in Canny edge detection to defining the edge gradients and directions of each pixel.

In terms of Analysis, rather than learning new concepts, we had to utilize what we had learned in Combinatorics to create coding pathways to reach computer algorithms. For example, we knew that Eulerian cycles had to have even vertices and only one component, but to check those properties we had to create a computer algorithm to check that the sum of all the vertices was even and that all non-zero vertices were visited through a depth-first search, and so on. Rather than simply learning about and incorporating new concepts, we had to devise an algorithm that fulfilled the conditions of concepts already known.

With this script, we provide an exploration space for students new to graph theory, and hope to help build quick intuitive understanding, democratizing the field. In the future, we would like this script to work for more complex types of graphs. Currently, identification is limited to planar graphs without intersections, and larger, more complex graphs have not been tested.

Despite these limitations, however, the core identification process can be expanded, allowing for more complex analysis. Furthermore, we wish to adopt a more efficient chromatic number algorithm, as the Greedy Algorithm will not necessarily assign the lowest chromatic number for every graph.

By leveraging more testing we hope to further optimize our algorithms and make the program useful for full analog to digital conversion, and eventually offer the presented script in web-app format for end-user use.

References

Acosta, N. O., & Tomescu., A. I. (2023). Simplicity in Eulerian circuits: Uniqueness and safety. *Information Processing Letters, 183*.

Auer, C., Bachmaier, C., Brandenburg, F. J., Gleißner, A., & Reislhuber, J. (2013). Optical graph recognition. In W. Didimo & M. Patrignani (Eds.), *Graph Drawing* (pp. 529–540). Berlin, Heidelberg: Springer Berlin Heidelberg.

Keller, M.T., & Trotter, W.T. (2016). *Applied combinatorics*.

Mahmud, B. U., Shuva, R. D., Bose, S. S., Rahman Majumder, M. M., Jahan, B., & Alam, M. M. (2019). A proposed method for recognizing complex hand drawn graphs using digital geometric techniques. *2019 International Conference on Sustainable Technologies for Industry 4.0 (STI)*, 1–6. doi:10.1109/STI47673.2019.9068093

Mazzuoccolo, G., & Ruini., B. (2010). The NP-Completeness of automorphic drawings. *Discussiones Mathematicae, 30*(705-710).

OpenCV. (n.d.). *Canny edge detection.* https://docs.opencv.org/4.x/da/d22/tutorial_py_canny.html

OpenCV. (n.d.). *Hough circle transform*. https://docs.opencv.org/3.4/d4/d70/tutorial_hough_circle.html

OpenCV. (n.d.) *Hough line transform.* https://docs.opencv.org/4.x/d9/db0/tutorial_hough_lines.html

Appendix:

All code can be found on Github under https://github.com/parviam/3012ogr.

Appendix A - Code for Digitization of Graph

Python

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

def digitizeGraph(image):
  # Load the image
  img = cv2.imread(image, cv2.IMREAD_COLOR)

  plt.imshow(img)

  # Convert the image to grayscale
  gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)


  # Apply Canny edge detection
  edges = cv2.Canny(gray, threshold1=100, threshold2=150, apertureSize=7)

  # Apply Hough Circle Transform
  circles = cv2.HoughCircles(gray,cv2.HOUGH_GRADIENT, dp=1, minDist=20, param1=50, param2=30,
minRadius=0, maxRadius=0)

  # Identify lines using Hough Line Transform
  lines = cv2.HoughLinesP(edges, 1, np.pi / 180, threshold=50, minLineLength=100, maxLineGap=0)



  #lines are given in xy coordinates
  # Determine which circles are connected by lines
  connected_circles = []
  for line_id, line in enumerate(lines):
    x1, y1, x2, y2 = line[0]

    two_circles = []
    for circle_id, circle in enumerate(circles[0]):
      cx, cy, _ = circle
      distance1 = np.sqrt((cx - x1)**2 + (cy - y1)**2) #distance formula
      distance2 = np.sqrt((cx - x2)**2 + (cy - y2)**2)
      if distance1 <= circles[0][0][2] + 100 or distance2 <= circles[0][0][2] + 100:
        #if within a very small distance, intersection
        two_circles.append(circle_id + 1)
    connected_circles.append(two_circles)
```

```
  unique_connections = []
#remove duplicates
  for pair in range(len(connected_circles)):
    if connected_circles[pair] in unique_connections:
      continue
    if len(connected_circles[pair]) == 0:
      continue
    else:
      unique_connections.append(connected_circles[pair])

  print(unique_connections)

  # Write connections to a text file
  with open('EdgesAndNodes.txt', 'w') as file:
    file.write(f"{len(circles[0])}\n")
    for connection in unique_connections:
      file.write(f"{connection[0]} {connection[1]}\n"
)
```

Appendix B: Code for Analysis of Digitized Graphs (Contribution 2)

Python

```
from collections import defaultdict

#create a Graph class
class Graph:
  def __init__(self, vertices):
    self.V = vertices
    self.graph = defaultdict(list)

  def add_edge(self, u, v):  #basic data structure functions
    self.graph[u].append(v)
    self.graph[v].append(u)

  def remove_edge(self, u, v):
    self.graph[u].remove(v)
    self.graph[v].remove(u)

  def is_valid_next_edge(self, u, v):
    if len(self.graph[u]) == 1:
      return True
    else:
      visited = [False] * (self.V)
      count1 = self.dfs_count(u, visited)

      self.remove_edge(u, v)
```

```
        visited = [False] * (self.V)
        count2 = self.dfs_count(u, visited)

        self.add_edge(u, v)

        return False if count1 > count2 else True

    def dfs_count(self, v, visited):
      count = 1
      visited[v] = True

      for i in self.graph[v]:
        if not visited[i]:
          count = count + self.dfs_count(i, visited)

      return count

    def is_eulerian_circuit(self):   #all non-zero degree vertices connected?
      if not self.is_connected():
        return False

      odd = 0
      for i in range(self.V):        #count odd vertices - must be even number!
        if len(self.graph[i]) % 2 != 0:
          odd += 1
      if odd != 0 and odd != 2:
        return False

      return (odd == 0) or (odd == 2 and self.is_connected())  #check connected if odd ct. 2

    def is_connected(self):
      visited = [False] * (self.V)    #init matrix - all vetices unvisited

      for i in range(self.V):      #get first vertex with degree >0, return false if none found
        if len(self.graph[i]) > 1:
          break
      if i == self.V - 1:  #or if its the last one it fails anyway bc what's it connecting to?
        return True

      self.dfs(i, visited)    #depth-first search

      for i in range(self.V):    #was everything visited?
        if not visited[i] and len(self.graph[i]) > 0:
          return False

      return True
    def dfs(self, v, visited):   #recursive depth-first search
      visited[v] = True

      for i in self.graph[v]:
```

```
      if not visited[i]:
        self.dfs(i, visited)
  def is_hamiltonian(self):
    def hamiltonian_util(path, pos):
      if pos == V:
        return self.graph[path[pos - 1]][path[0]] == 1   #vertex between these two points?

      for v in range(1, V):
        if is_valid(v, pos, path):
          path[pos] = v

          if hamiltonian_util(path, pos + 1):    #recursive: check every possible path
            return True

          path[pos] = -1

      return False

    def is_valid(v, pos, path):           #is this path valid? (vertex between both points)
      if self.graph[path[pos - 1]][v] == 0:
        return False

      if v in path:              #already visited (breaks rules of hamiltonian circuit)
        return False

      return True

    V = len(self.graph)     #vertices
    path = [-1] * V     #init
    path[0] = 0

    if not hamiltonian_util(path, 1):   #check path from 1
      return False

    print("hamiltonian circuit found: ", end="") #print circuit if exists
    for vertex in path:
      print(vertex, end=", ")
    print(path[0])
    return True
  def chromatic(self):
    colors = [-1] * len(self.graph)
    used_colors = set()

    for vertex in range(len(self.graph)):
      available_colors = set(range(len(self.graph)))   #by using sets, avoid repitition
      for neighbor in self.graph[vertex]:
        if colors[neighbor] != -1:
          available_colors.discard(colors[neighbor])   #eliminate not-available
possibilities
```

```
      if available_colors:
        color = min(available_colors)     #find the smallest one you can use (greedy)
      else:
        color = max(used_colors) + 1

      colors[vertex] = color
      used_colors.add(color)

    chromatic_number = len(used_colors)
    return chromatic_number


# TESTING

#import file
FILENAME = 'test_graph.txt'
file = open(FILENAME)
file.seek(0)
g = Graph(int(file.readline())) #create blank graph

#add edges
l = file.readlines()
for line in l:
  g.add_edge(int(line.split()[0]) - 1, int(line.split()[1]) - 1)

euler = g.is_eulerian_circuit()
hamiltonian = g.is_hamiltonian()
chromatic = g.chromatic()
```