Introduction

Our project, Graphs and Machine Learning, held under the WXML, was inspired by a paper Dr. Billey co-wrote, entitled “Fingerprint Database for Theorems” with Bridgette Tenner. The initial goal of this project was to create a searchable database for graphs similar to the Online Encyclopedia of Integer Sequences (OEIS). The purpose of such a database is straightforward. Say a professor or researcher thinks they have made a new discovery and that said discovery produces a graph. Given the aforementioned database, the professor could then input a graph related to the theorem they produced and search if someone else has already discovered said theorem. If someone already has, then the professor could then find out who found the theorem and potentially other related theorems, via searching related graphs. Hence, our group aims to document certain forms of theorems from graph theory into an online searchable database, for which papers and references may be indexed by the graphs contained within them. In the process of creating this encyclopedia of graphs, or Graphlopedia, we developed a number of tools and made several design decisions for our database.

Database Creation

We elected to store the graphs we collected in a document-based database for several reasons. Chief among these was the flexibility offered by this style of database; with a document-based database, we could add different fields and amend previous entries anytime in the future with ease. Furthermore, we had already constructed several database entries by hand in a JSON format, and it is trivially easy to add data in this format to a document database. Of all the document-based databases available, we selected MongoDB for our purposes due primarily to its extensive documentation and proven track record. Apart from simply storing graphs, this database also stores families of graphs and users for the web app we constructed.

In order to share the data we collected with the world, and to allow others to contribute, we decided to construct a web application that would allow anyone to view our graphs and to propose new additions. This had always been the goal of the project, as described earlier**.**

We currently have a working system for the website but it is currently not yet available to the public. We have created an account on the UW computer system called Ovid which will host webpages with search capabilities. We still need to setup additional software on Ovid to support MongoDB. The website will appear at http://depts.washington.edu/graphlop/. Our plan is to purchase the domain name graphlopedia.org once our system is running publicly.

Invariant Selection and Database Input

We opted to include invariants that were both efficient for manual entry into the database and sufficient for the user to distinguish between entries. Hence selection for invariants was key to efficiency and efficacy of database input. In our selection process we found that there are a few invariants that we can use to differentiate different graphs, though not all are ideal for the job. An adjacency matrix is an obvious choice for a fingerprint candidate, however a single graph can have multiple equivalent adjacency matrices, and deciding on a canonical form for such matrices is a non-trivial task. We chose to fingerprint graphs by their degree sequence: a list of the number of edges connected to each vertex, entered in descending order. While a degree sequence isn't necessarily a unique fingerprint, since two different graphs can have the same degree sequence, a mathematician can easily search our database by degree sequence and narrow their search. There will be no false negatives, only false positives. Other useful invariants are the eigenvalues of the adjacency matrices, the chromatic number, and the clique number. The eigenvalues are a dominant trait in graph theory, but the values are complex numbers so they can't always be stored effectively in our database. Storing the characteristic polynomial is a more sound approach and it encodes the same data in principle. Within the coming months we hope to develop algorithms to automate calculations of said invariants from our existing entries. Currently, we have a number of methods to compute different invariants described below.

In order to maximize user search efficiency we decided to input invariants in canonical order. To store the data we collected from our graphs, we needed to format it consistently. We determined that JavaScript Object Notation (JSON) is an efficient and convenient way to enter the information. At this time, we have the following fields for each graph in the database: title, name, vertices, edges, degree sequence, links, comments, references, and contributors. At the current moment graphs and papers are searchable via keywords, degree sequence and graph index though we hope to extend our search mechanisms to allow for greater specificity soon.

More recently, we added a new type of entry to the database, graph families. So far we only have two families, the Petersen family, and the non-planar characterization family (K_5, $K_{3,3}$). It is our hope that allowing for graphs to be searched for by their corresponding families may elucidate related theorems and published materials, facilitating research within the field.

Graph Extracting Programs

We allow the users (including us) to contribute to our database. However the database stores numerical and text data while the users may only have a graph image from a journal article or website, hence there is a need for a tool to extract a graph from a digital image. To accomplish this goal, we designed two programs using Python. One of them can efficiently extract graphs drawn

using LaTex, which has a special way to encode its images. Making use of this fact, this program directly retrieves the graph data from a LaTex-written PDF file.

The other program can process a variety of image types (.jpg, .png, etc.) using OpenCV, a package dedicated to perform image processing. Before we jump into details about the procedure of the program, let's first discuss about how an image is stored in a computer.

A digital image is composed of pixels, the smallest unit representing a certain color. Therefore in a computer, an image is essentially a matrix of pixels. For example, if an image has resolution Error, then essentially this is a 768×1024 matrix. Moreover, just like in real life where any color is a mix of yellow, blue and red, a color in a computer is represented with R (red), G (green) and B (blue) values, all of which range from 0 to 255. Each pixel stores a tuple of these three values, so technically an image is a matrix of tuples, not numbers. However it is in general much more convenient to process a 2D matrix than a 3D cube. So most of the time when performing image analysis, we convert a color image into a monochrome one, which in computer vision is called a grayscale image. After the conversion, the image becomes a 2D matrix of integers in the range [0, 255]. Furthermore, it turns out that, at least in our case, the problem size can be further reduced. Based on the intuitive fact that a graph only needs two colors to represent, we can define the pixel representing the graph to have value 1 and the that representing the background to have value 0. The result is called a binary image, and many image processing algorithms are based on this version of the original image.

In what follows, we will discuss how the program can extract a graph step by step.

Preparation

When a human attempts to recognize a graph from an image, they will naturally first want to locate the vertices. This is also true for our program. The program first asks the user to provide a image file, and then it asks the user to crop a piece of the image that contains a vertex. In computer vision, this piece is usually called a *template*. Using this vertex example, the program then performs *template matching*, a technique which simply scans the whole image to find a piece that match the example the most. It only finds one piece per run so in general this should be executed multiple times. Unfortunately this technique is not perfect, meaning that it can make errors when detecting the vertices. We implemented some features to manually fix these errors. Overall this step is not so difficult.

Noise Reduction and Image Thinning

Earlier we mentioned that a graph only takes two colors to represent. However even if some graph images seem to contain only two colors, there are often more than two. Therefore in order to make use of the well-developed algorithms we need to convert the image into a binary version. Changing an image from a color version into a monochrome one is straightforward since there exists a standard method. But converting a monochrome image into a binary one can be tricky.

The method is simple: for a pixel, if the value is higher than or equal to a certain value, then we set it to 0 since this is a background pixel, otherwise we set it to 1 since it represents a content pixel. The problematic part is the threshold value, as there is no guarantee that a certain value can divide the background and the content perfectly. Since we do not want to lose any content pixel, it is safe to set the threshold to be the background value in the original image. As a result, in the binary image, many non-content pixels will become part of the content (they all have value 1 now). We call this group of pixels the *noise* of an image. There are two primary algorithms to reduce the noise: *erosion* and *dilation*. If we think the background as an ocean and the content as an island, then *erosion* is like the same term in real world: it makes the ocean (background) to take over the shore of the island (content). *Dilation* is exactly the opposite. These operations usually come together to reduce the noise of an image, as *erosion* cleans the outlying pixels while *dilation* fills the holes. Our program interacts with the user to perform these two operations in order to obtain a satisfied result where the noises are greatly reduced.

Usually a graph is drawn with curves of certain width for better display. However when recognizing a graph from an image, what really matters is the graph structure, hence the need to obtain the "skeleton" of the image. This step is called *image thinning*. There are a few well-developed algorithms that can do this work, here we implemented zhang-seun's algorithm and this step is automatically executed after the noise reduction.

Edge Extraction

Now that we have the locations of the vertices and the binary skeleton of the graph, we can now consider extracting the edges. The first task is to locate the starting points of each edge. There are more than one way to accomplish this goal. In our program we simply check the distance between each edge pixel and the center of each vertex. Next, let's first consider an easy case where there is no edges crossing each other. Starting from one end of an edge, we can trace the entire edge by following this algorithm:

1. Obtain the neighborhood of the current point. The neighborhood is defined to be the eight points surrounding the current one. For each of these neighbor points, if its pixel value is 1 and it has not yet been examined, then put it into the candidate set. Go to step 2.
2. If the candidate set is empty, go to step 3. Otherwise, for each point in the candidate group, go back to step 1.
3. Examine if the current point belongs to the starting points set that was obtained before the execution of the algorithm, or the distance between the current point and any vertex center is within some value. If so then we are done with the current edge since we now know which two vertices it is connecting. Otherwise we do nothing since this branch is caused by some noise.

In short, this algorithm can be summarized as follows: keep looking for an unexplored point till we reach the end. Now let's consider the hard situation where edge crossing exists. Recall that a binary image is a matrix of 0's and 1's, and that in the previous algorithm at each iteration we are

considering a neighborhood of the current pixel. Therefore ultimately we are trying to determine which direction to go when we encounter some cases like these:

1 0 0	1 1 1
0 1 0	0 1 0
0 1 1	0 1 0
(a)	(b)

In other words, when the candidate set has more than one element, we want to avoid brute-force-search and try to correctly determine which element to choose to proceed. In order to accomplish this goal, we need to modify the algorithm into the following:

1. Obtain the neighborhood of the current point. The neighborhood is defined to be the eight points surrounding the current one. For each of these neighbor points, if its pixel value is 1 and it has not yet been examined, then put it into the candidate set; **otherwise put it in a list called *trail***. Go to step 2.
2. If the candidate set is empty, go to step 3. Otherwise, **go to step 4**.
3. Examine if the current point belongs to the starting points set that was obtained before the execution of the algorithm, or the distance between the current point and any vertex center is within some value. If so then we are done with the current edge since we now know which two vertices it is connecting. Otherwise we do nothing since this.
4. Construct a list of vectors called, each of which is calculated using all the adjacent points in the list *trail*. If there are (n + 1) elements in *trail*, then there are n vectors in the vector list. Evaluate a weighted sum of these vectors using the following formula:

   Error

   Choose a candidate which has the direction closest to this sum, and then go to step 1.

In step 4, the most important modification, each a_i is a normalized weight, and these weights should be chosen such that if the corresponding v_i is closed to the current point, then a_i has a rather high value. The intuitive idea behind is that, we are evaluating a vector from the previous path in a reasonable way and use this vector as a reference to determine which direction to go at a cross. Note that the *image thinning* technique often creates a conjunction at a cross. So when recording the *trial*, any pixel in the conjunction should be excluded. However, the conjunction structures vary, and how to handle these conjunctions brings another challenge which remains unsolved.

Invariants Evaluation

When the essential graph information (the vertex set and the edge set) has been obtained, we want to evaluate some of its invariants and store them into our database for more efficient query. We have implemented methods to check the connectivity (using brute force), the completeness,

the chromatic polynomial and chromatic number (using Zykov's algorithm), the cliques (using Bron-Kerbosch algorithm), and the characteristic polynomial.

Another noticeable feature of this program is the error checking. Since user interaction is involved at multiple steps, we have implemented a number of exception handlers to ensure that the program can be executed with minimum interruption. In the future, we will try to improve the existing programs by attempting to solve the conjunction challenge in the edge extraction part and fine tuning the codes to make them more efficient. We are also planning to introduce another program to examine the graph information given by the users to determine if the input is valid.

Conclusion

We have numerous goals for the future. We will spell out a few direction for next year.

We currently have about 50 graphs entered into our database. We would like to see Graphlopeida include all of the journal publications and arXiv entries that contain interesting graphs. Ideally a majority, if not all, related publications of a certain theorem may be easily accessed from our database. However, it is unfeasible to enter hundreds upon thousands of graphs by hand, and since we only recently published our website we currently do not have many users to upload graphs either. Therefore, a potential solution to hit a "critical mass" of graphs would be to develop a web crawler that would find pdfs and images containing graphs and feeding them into the aforementioned graph recognition programs.

In the future, we would like Graphlopedia to include directed graphs, multigraphs, and edge-weighted graphs. It currently only supports undirected, unweighted, simple graphs. This is a relatively easy addition to include in terms of the database. We will need to think more about the appropriate fingerprint/invariant.

In order to facilitate automated or outside database input we hope to develop algorithms to "fact check" potential entries, with the end goal of minimizing the necessity for manually checking input accuracy.

A few graphs in our database contain pointers to other related graphs and graph families. How could we assist the users to find other graphs related to one they have just entered? Should we link together every graph that appears in one paper? Should we leave it to the author to suggest related graphs? Can this be automated in some way along the lines of the Netflix movie recommendations? How are related papers characterized by the graphs contained within them and how may we accommodate our database to allow for a related paper and or theorem search?

Finally, given more image data on graphs, we hope to apply machine learning techniques to both improve parts of our image recognition algorithm as well as to correlate and find new data about the graphs in our database.