Hand-Drawn Graph Problems in Online Education

by

Katharine M. Daly

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2015

Author .....
Department of Electrical Engineering and Computer Science
May 22, 2015

Certified by .....
Isaac Chuang
Professor of Electrical Engineering and Computer Science; Professor of Physics; Senior Associate Dean of Digital Learning
Thesis Supervisor

Accepted by .....
Professor Albert R. Meyer
Chairman, Master of Engineering Thesis Committee

Hand-Drawn Graph Problems in Online Education

by

Katharine M. Daly

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2015, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Machine-gradable assessments in online education platforms are currently limited to questions that require only keyboard or mouse input, and grading efforts generally focus only on final answers. Some types of problems in the science, technology, engineering, and math (STEM) domain, however, are most naturally answered through sketches drawn with a pen. We introduce a simple graph problem type that accepts solutions drawn using a stylus as a proof-of-concept extension to online education platforms. Simple graphs have a small number of components (vertices, arrows, and edges only), and we describe a three-step recognition process consisting of segmentation, symbol classification, and domain interpretation for converting users' pen strokes into a simple graph object representation. An experiment run on Mechanical Turk demonstrates the usability of our trained, recognition-driven drawing interface, and examples of simple graph problems illustrate how course developers can not only check students' final answers but also provide students with intermediate feedback.

Thesis Supervisor: Isaac Chuang
Title: Professor of Electrical Engineering and Computer Science; Professor of Physics;
Senior Associate Dean of Digital Learning

Acknowledgments

I would like to thank the Office of Digital Learning at MIT for supporting this work and my advisor, Ike Chuang, for his guidance and encouragement at all stages of this thesis. Professor Chuang enthusiastically welcomed me into his group, introduced me to the local online education research community, and helped me choose a project that combined my interests in machine learning and online education. His many ideas and suggestions shaped key aspects of this thesis.

I am grateful for the teachers at all stages of my education who shared their knowledge and shaped my attitude towards learning. In particular, I would like to acknowledge my middle school and high school math club advisors, who helped me find one of my passions and led me to appreciate the value of hard work.

Last but not least, I am thankful for my family's support.

Contents

List of Figures

Chapter 1

Introduction

For many problems in the STEM domain, handwritten scratch work including equations, numerical calculations, and diagrams is the most natural form of scratch work. While this scratch work is a key component of grading in traditional classroom education, current online education platforms focus on students' final answers and permit a limited array of answer formats. For example, edX, a popular online learning platform, allows a small number of machine-gradable answer formats: text boxes, check boxes, dropdown menus, multiple choice, image selection, and highly constrained drag-and-drop diagram manipulations [edX15].

Problems that do involve diagram creation often allow very limited drawing freedom. The chemical molecule editor used by edX requires users to click and drop molecular components from a toolbar [BE13], and the INK-12 project for K-12 math and science education permits drawing only through a copy-and-paste stamp mechanism [KR13]. In order to match or exceed traditional evaluation methods for many math and science problems, online education platforms will need to improve their ability to collect, interpret, and evaluate scratch work and final answers as well as impose fewer drawing restrictions when diagram creation is involved.

The work presented here describes a system for accepting and grading simple graph problems where students draw their solutions with a stylus either directly on a screen or via a digitizer. We choose to focus on simple graphs because machine recognition of hand-drawn sketches is a challenging problem, and simple graphs consist of a well-

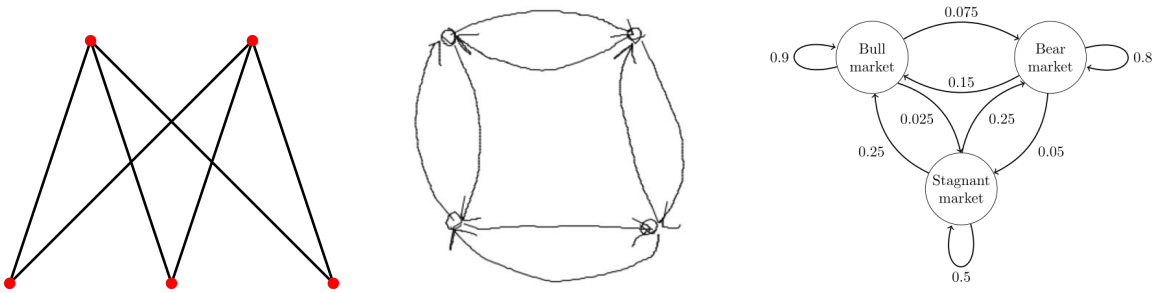


Figure 1-1: Simple graphs consist of a small number of component types that can be combined in an infinite number of ways. Here are examples of a bipartite graph, a hand-drawn directed graph, and a Markov chain.

defined and small set of components. These components, which can combine to form simple graphs in an infinite number of ways, are vertices, arrows, and edges. Drawings of components of the same type can vary widely, adding complexity to the recognition task. Self-loops and multi-edges are permitted in our definition of simple graphs, and labels can also be appended to vertices and edges. Figure 1-1 contains three examples of simple graphs including one hand-drawn graph that is representative of the sketches we aim to recognize.[1,2]

We also choose to create a simple graph problem type because topics involving simple graphs appear across the standard undergraduate computer science curriculum, and sketches play a major role in communicating information when these concepts are covered. A selection of topics involving simple graphs include search algorithms (e.g. Breadth First Search, Depth First Search, A*), data structures (e.g. Binary Search Trees, AVL trees, heaps), shortest path algorithms (e.g. Dijkstra, Bellman-Ford), max flow, topological sort, matchings, colorings, scheduling, connectivity, minimum spanning trees, planar graphs, state machines, and Markov chains. The graphs in each of these topics are constructed using only vertices, arrows, edges, vertex labels, and edge labels.

In general, there are two different types of recognition of hand-drawn input. The first type, which is known in the literature as online recognition, receives as in-

[1] http://upload.wikimedia.org/wikipedia/commons/thumb/e/e2/Complete_bipartite_graph_K3%2C2.svg/1187px-Complete_bipartite_graph_K3%2C2.svg.png

[2] http://upload.wikimedia.org/wikipedia/commons/thumb/9/95/Finance_Markov_chain_example_state_space.svg/492px-Finance_Markov_chain_example_state_space.svg.png

put strokes that contain temporal information about the trajectory a pen tip takes through a set of points. The second type, commonly known as offline recognition, essentially treats the hand-drawn input as an image, so no time-related information is available. Note that the use of the words online and offline in this context signify that the recognition occurs in realtime or does not occur in realtime, respectively. No association with the internet or browsers is intended in this context. In decades of research on handwriting recognition, recognition rates for the online case have been much higher than the offline case [PS00]. This is not a surprising result, for the input provided to offline recognizers is a strict subset of the input provided to online recognizers. Since it is straightforward in our use case to track the trajectories of students' styluses and to record time-related information as they sketch their graphs, we decided to develop an online recognition algorithm, which incidentally runs in a browser.

Online recognition of hand-drawn sketches is a field of ongoing research with a wide range of applications. Recognition systems for pen-based input have been successfully constructed for chemical diagrams [OD07], circuits [GKSS05], and flow charts [MEFC09]. Much like simple graphs, the sketches in each of these domains contain a limited number of basic components that can be combined in an infinite number of ways. While Ouyang et al were were able to recognize letters representing chemical symbols as well as recognize basic shapes in their chemical diagram recognition work, labels in simple graphs have the potential to be much more complicated. Labels in state machines are often phrases, and labels in other types of simple graphs can contain mathematical symbols and various mathematical expressions. Since the state-of-the-art technology for recognition of handwritten mathematical expressions is not yet very accurate [ZB12], in this work, we perform machine recognition on vertices, arrows, and edges, but we require users to enter labels using a keyboard.

The bulk of the work presented here describes our three-part approach to recognizing users' hand-drawn sketches composed of vertices, arrows, and edges while maintaining reasonable levels of drawing freedom. The few assumptions we do make about the sketching process are presented in Chapter 2 along with an overview of the

recognition process, which consists of segmentation, component classification, and domain recognition. Chapters 3 through 5 go into detail about each of the recognition steps, describing the methods other researchers have used to accomplish similar tasks, the motivation behind the design decisions we made, and the process of training and testing parameters using a Mechanical Turk experiment.

Since this project is intended to show how sketching might be incorporated into an online education platform, we have also built a user interface that attempts to recognize users' strokes as they draw and to communicate the current interpretation to users. Chapter 6 covers the functionality of this interface and the results of a second Mechanical Turk experiment designed to test the interface's usability. Finally, the ability to interpret in realtime students' constructions of their solutions provides an opportunity to give intermediate feedback to students. Traditional online education answer formats such as text boxes for final numerical answers simply do not afford this opportunity since students perform their scratch work off-screen. In Chapter 7, we provide examples of how intermediate feedback might be presented to a student using several sample problems involving simple graph concepts.

Chapter 2

Recognition Overview

The input to the simple graph online sketch recognition problem is a set of pen strokes, and the desired output is a collection of correctly positioned and connected components. A pen stroke represents the trajectory of a user's pen tip from a single pen down event to the next pen up event, and it consists of a sequence of 2D pixel coordinates with associated timestamp information. Since users are required to enter vertex labels and edge labels using a keyboard, a decision affecting problem scope that was motivated in Chapter 1, there are only three different components (vertices, arrows, and edges) that need to be detected amongst the pen strokes.

This chapter presents an overview of the process of recognizing these different graph components. Section 2.1 describes the challenges we faced, and Section 2.2 explains the assumptions made regarding drawing technique. The roles of each step in our three-part recognition algorithm are discussed in Section 2.3, and finally, Section 2.4 describes our approach to training and testing the parameters in our algorithm.

2.1 Challenges

Current graphical user interfaces for sketching applications vary widely in their ease of use. Those that require lots of interaction with menus and toolbars can have perfect interpretation capabilities, but their drawing experience is often unnatural and

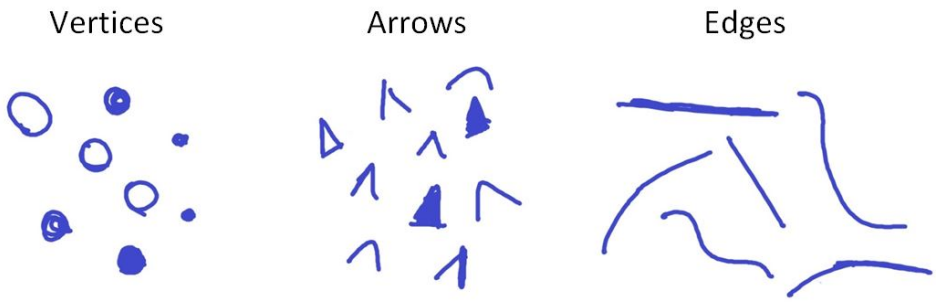


Figure 2-1: Each of the three simple graph components can be drawn in a large variety of acceptable ways, which complicates the recognition process.

contains many interruptions. On the other hand, sketching applications that permit lots of drawing freedom in order to closely simulate the paper sketching experience can suffer from poor recognition and interpretation capabilities. The restrictions and freedoms chosen for this project's sketch recognition task attempt to strike a balance between these two extremes and are motivated by a combination of the assumptions commonly made in the field of hand-drawn sketch recognition and the unique properties of the simple graph domain.

The large variation in how these components can be drawn, as demonstrated in Figure 2-1, provides much of the complexity of the recognition task. For example, vertices are sometimes drawn as filled-in circles and other times only the circle perimeter is drawn. Arrows can be two sides of a triangle, an arc-like shape, a complete outline of a triangle, or a poorly filled-in triangle. Edges can be straight or curvy, composed of one stroke or multiple strokes, and sometimes portions of edges are traced multiple times. The size and orientation among components of the same type can also be highly variable.

On many occasions, graph components or parts of components cannot be understood in isolation even by humans. In these situations, such as the ones illustrated in Figure 2-2, surrounding portions of the graph must be visible in order to correctly identify a component type. For example, a short, straight pen stroke might be one side of an arrow, but it could also be an edge connecting two nearby vertices. Similarly, a medium-sized circle might be a vertex, but it could also be an edge functioning

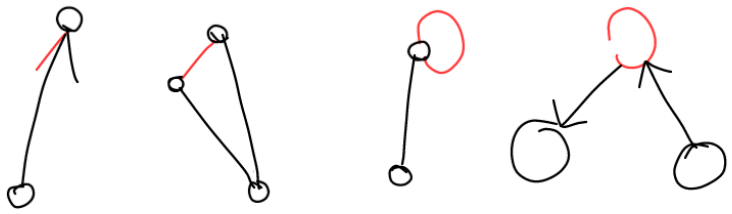


Figure 2-2: Components or parts of components cannot be reliably interpreted when they are viewed in isolation. On the left, one stroke can become part of an arrow or an entire edge. On the right, one stroke can become either an edge functioning as a self-loop or a vertex.

as a self-loop where both edge endpoints are connected to a vertex with a smaller radius.

2.2 Assumptions

We designed our graph recognition algorithm to have the potential to recognize each of the components shown in Figure 2-1 as their correct type. Thus, in an effort to avoid imposing unnatural restrictions on component construction, we make no assumptions regarding the number or pattern of strokes used to draw any component. At the same time, however, we do not expect our algorithm to work equally well across the infinite number of drawings which could reasonably represent the same component type. Rather, we have created an algorithm that, through training, adjusts to the ways in which components are most commonly drawn.

Several existing hand-drawn sketch recognition systems also ignore the stroke composition of specific symbols. For example, in their online scribble recognizer for common multi-stroke geometric shapes, Fonseca et al recognize that geometric features are more reliable than counting the number of strokes for shape recognition tasks. They do, however, assume that edges consist of just one stroke [FPJ02]. Gennari et al's analog circuit sketch parsing, recognition, and error correction (AC-SPARC) system for analyzing hand-drawn circuits makes no assumptions about the number of strokes used to create a circuit element such as a resistor or inductor [GKSS05].

While we ignore stroke-related information when attempting to classify individual components, we do make the stroke-based assumption that one component is always drawn in its entirety before a second component is started. To make this assumption fail, a user could draw a portion of an arrow, draw an edge segment, and then return to draw the rest of the arrow. This type of discontinuous drawing behavior is unexpected, however. Additionally, in a user interface where recognition occurs periodically and users' strokes are replaced with computer-rendered versions of the interpreted components, users will have no reason to add additional strokes to components that have already undergone the recognition process.

Numerous existing sketch recognition systems have made the same assumption regarding non-overlapping generation of symbols. For example, the AC-SPARC circuit system requires one symbol to be completed before the next [GKSS05], Ouyang et al's chemical diagram recognizer assumes that symbols consist of up to seven sequential strokes [OD07], and Lank et al's Unified Modeling Language (UML) diagram recognizer assumes that UML glyphs consist only of sequential, intersecting strokes [LTC00]. The temporal information encoded in the stroke data provided to online recognition systems is well suited for selection of consecutive strokes, a task which would be very difficult given an offline representation of the sketch.

Two final assumptions directly tied to time are made. Firstly, we assume that each component is drawn within a specified amount of time. This maximum time parameter can be increased as desired to minimize its effect, and the parameter does not directly restrict the number of strokes a user can make while drawing a single component. As the recognition algorithm is incorporated into user interfaces and is expected to provide realtime interpretations to users, a method for triggering the recognition process is needed. We choose to trigger recognition in our user interface after a specified amount of time has elapsed since the end of a user's last stroke, a decision that is described in more detail in Chapter 6. Whenever this condition is met and the recognition process runs, we assume that the current sketch consists only of complete components. Thus, in addition to the total time restriction on drawing individual components, there is also a restriction on the amount of uninterrupted

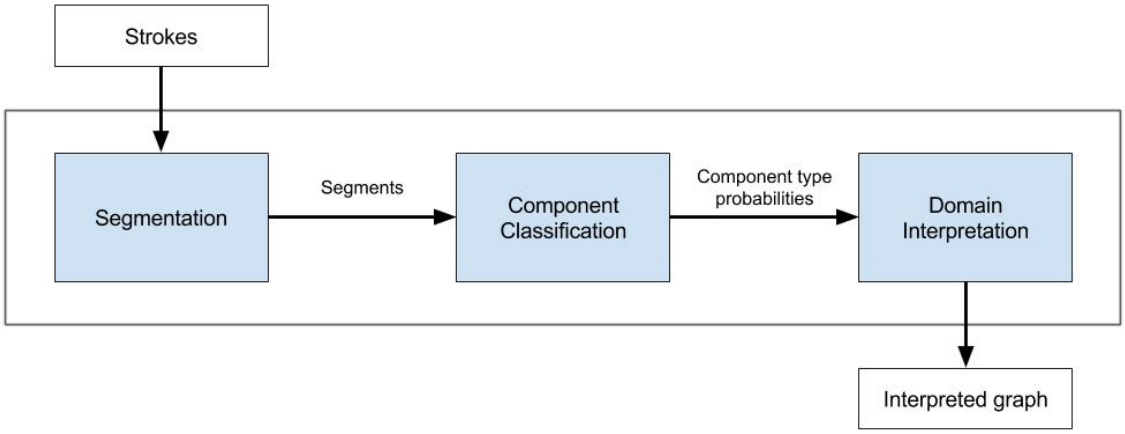


Figure 2-3: The recognition process has three steps. The segmentation step divides strokes into segments, which the component classification step uses to compute component type probabilities. The domain interpretation step analyzes component interactions and returns the final interpreted graph.

time a user's stylus can remain inactive while a single component is being drawn.

2.3 Algorithm Design

The literature on hand-drawn sketch recognition describes a wide range of approaches to online recognition algorithm design that are often highly influenced by the intended sketch domain and by the assumptions made regarding drawing style. Often, however, the basic structure of a recognizer can be thought of as a three-step process consisting of segmentation, component classification, and domain interpretation. Our algorithm also follows this structure, which is shown graphically with the expected inputs and outputs for each step in Figure 2-3. The basic roles of each step are explained below, and more details are discussed in Chapters 3 through 5.

The segmentation step takes strokes as input and produces segments, which are effectively smaller strokes, as output. Since no assumptions are made regarding the stroke composition of individual simple graph components, a single component can include any number of complete strokes and may even include portions of strokes. For example, as illustrated in Figure 2-4, consider sketching a vertex, creating an edge extending from that vertex, and then adding an arrow to the end of the edge,

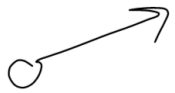


Figure 2-4: Components need not contain a whole number of strokes. Here, a vertex, edge, and arrow are drawn with a single stroke.

all without lifting up the pen. Here, each component consists of a portion of a single stroke. Note that this drawing sequence obeys the assumption that components are drawn one after the other.

Since the subsequent component classification step is only capable of determining which single component a set of points best represents, the segmenter must divide the given strokes into a set of segments such that each component consists of one or more whole segments. Ideally, the strokes would be broken up to produce a single segment per component. For the example above, this would mean dividing the stroke into one segment for the vertex, one segment for the edge, and one segment for the arrow. This precise division of strokes is often not practical, however, so in general, the segmenter should aim to create the smallest number of segments possible while meeting the requirement that each segment is part of only one component with high probability.

In component classification, the segments produced by the segmenter are given scores that reflect their resemblance to each of the graph component types. Resemblance measures are calculated as probabilities by combining several features calculated from the input points. Due to the end condition of the segmentation step and the assumption made earlier that each component is drawn in its entirety before another component is begun, the component classification step can assume that each component consists of one or more temporally consecutive, whole segments. Thus, given n segments, probabilities for each component type are assigned to $O(n^2)$ sequences of consecutive segments. Without the ordering assumption, a much larger number of segment sequences would need to be classified.

The final step in the recognition process, domain interpretation, takes the proba-

bilities calculated by the component classifier and returns a final interpretation of the sketch that is influenced by rules and/or conventions of the domain. As an example of such rules, in the simple graph domain, edges in completed graphs are attached to a vertex on either end, and arrows are found pointing towards vertices at the ends of edges. Thus, our domain interpretation step involves iterating through different partitions of segments, establishing connections between graph components, and computing scores that are a combination of classifier probabilities and the extent to which the components connect to form a typical graph.

This three-part recognition structure consisting of segmentation, component classification, and domain interpretation incorporates a large amount of error recovery potential from step to step. If the segmenter produces more segments than necessary, causing some components to consist of more than one segment, the classification and domain interpretation steps still have the potential to produce the correct interpretation. Similarly, the classification step need not always assign a high probability to the correct component type for a sequence of segments or even assign the correct type a higher probability than what it assigns to other component types. The domain interpretation step's attempts to establish connections and adjust the classification step's scores based on these connections can cause the correct final interpretation to be generated even in the event of misleading classifier probabilities.

2.4 Training and Testing

Each step in our recognition algorithm contains multiple parameters that control the step's behavior, so completing our recognition system required finding optimal settings of these parameters. For each step, we followed the standard procedure of training parameters by first creating a set of data containing pairs of inputs and the corresponding desired outputs and then dividing this set into a training set and a test set. We aimed for a 70-30 split of the data between the training and test sets. With the help of a cost function, we found the parameters that produced the smallest cost when applied to the training set data, and these ideal parameters were in turn

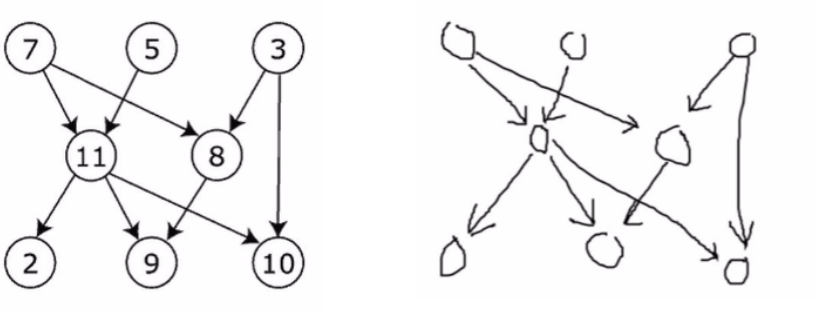


Figure 2-5: Sketches of graphs were collected using a Mechanical Turk experiment. One of the 15 graph images that workers were asked to draw is on the left, and one worker's submission for this graph is on the right. (Workers were instructed to ignore labels in the graph images.)

evaluated by determining the cost when they were applied to the test set data.

To collect data from which training and test sets could be formed, we conducted a Mechanical Turk experiment. Mechanical Turk is an online marketplace created by Amazon where requesters can post HITs (Human Intelligence Tasks) and receive responses from workers. Our experiment presented workers with images of 15 simple graphs and asked them to draw a copy of each graph using a stylus. Since our goal with this experiment was to collect stroke information from natural sketches of graphs, the task contained zero drawing guidelines or restrictions and did not inform workers that the experiment was part of an effort to create a hand-drawn graph recognition algorithm. The workers performed the task entirely within a web browser, and JavaScript was used to capture the drawing process.

Stroke data was gathered from 10 workers for a total of 150 simple graph sketches. Figure 2-5 provides an example of an original graph image accompanied by a worker's submission, and Appendix A contains all 15 original simple graph images along with a subset of the sketches we collected. Different hardware combinations were represented across the users, and as expected, the workers' sketches varied widely in neatness while maintaining resemblance to the original graphs. We strongly encouraged workers to use a stylus to complete the task since our primary use case involves styluses, but as there is no way to distinguish between all the different input methods (mouse, touchpad, finger, stylus, etc) within a browser, it is possible that some workers used an

alternative input method. This is not a significant issue, however, since our algorithm was designed to work independently of the input method.

Post-processing of the collected stroke data was performed to prepare separate training data for each of the three recognition steps. Details about the post-processing for a specific step can be found in Section 3.3 for the segmentation step, in Section 4.3 for the classification step, and in Section 5.3 for the final domain interpretation step. To summarize the post-processing briefly, transitions from one component to the next component were marked for segmentation training. Sequences of temporally consecutive points were labeled as their correct graph component type for component classification training. Lastly, the evolution of graph connectivity for each sketch was recorded for domain interpretation training.

A main objective of training and testing the algorithm parameters was to show that accurate recognition can occur for sketches drawn by an individual who is not represented in the training data. This is important for our online education use case, for we want new users to be able to sketch graphs and to view the recognizer's interpretation without having to first provide sketches to a training procedure. Thus, for each recognition step, we created instances of test data consisting of a single individual's sketches and compared the performance of parameters trained on the remainder of that individual's sketches to the performance of parameters trained on an equivalent amount of data from other individuals' sketches. The results of these tests are presented in the next three chapters.

Chapter 3

Segmentation

As the first step in the online graph recognition algorithm, the segmenter divides the input strokes into smaller segments such that each component consists of some whole number of segments. Section 3.1 describes the role of the segmenter, Section 3.2 both reviews existing approaches to segmentation and presents our design, and Section 3.3 discusses the training of the segmenter's three parameters.

3.1 Role

It is necessary to find the segmentation points that mark the transitions from one component to the next because the subsequent classification step is capable of classifying a given set of points only as a single component and not as a combination of components. In some recognition applications, drawing restrictions guarantee that each component consists of exactly one stroke or some whole number of strokes. In these cases, segmentation can be skipped. Our application, however, permits multiple components to be drawn with a single stroke, so segmentation is necessary.

Ideally, segmentation points would only be identified at the points along strokes where there is actually a transition from one component to another. These types of points will be referred to as true segmentation points. Often, however, other points along a stroke have features that make them indistinguishable from true segmentation points. This issue is illustrated in Figure 3-1, where a stroke consisting of an edge

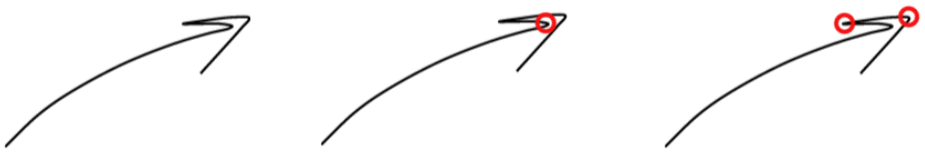


Figure 3-1: The arrow and edge on the left are drawn with a single stroke. There is one true segmentation point that occurs at the transition between these two components, as indicated by the circle in the middle image. The stroke has two other points, which are circled in the right image, that appear to have properties similar to those of the true segmentation point.

and an arrow has only one true segmentation point but contains two other points with similar geometric properties.

There are two types of errors a segmenter can make: false positives occur when superfluous segmentation points are identified, and false negatives occur when true segmentation points are not identified. False positives are minor errors from which subsequent steps in the recognition process can typically recover. The algorithm's overall efficiency suffers, however, as the number of false positives increases, so these errors should be minimized when possible. On the other hand, false negatives are severe errors and are generally non-recoverable. The relative cost of the two error types was reflected in the training of our segmenter in Section 3.3.

3.2 Design

Various approaches to detecting segmentation points are used across the literature of online recognition algorithms. In the AC-SPARC circuit recognition system, strokes are divided into arcs and lines by detecting points where the stylus speed is at a minimum, where there is high curvature, or where the sign of the curvature changes [GKSS05]. The VizDraw platform, which recognizes mechanical, electrical, thermal, and flow-chart symbols, uses the maxima and minima from horizontal, vertical, and diagonal projections of the stroke trajectory to place segmentation points [MEFC09].

Our segmentation algorithm divides strokes at points where there are abrupt and

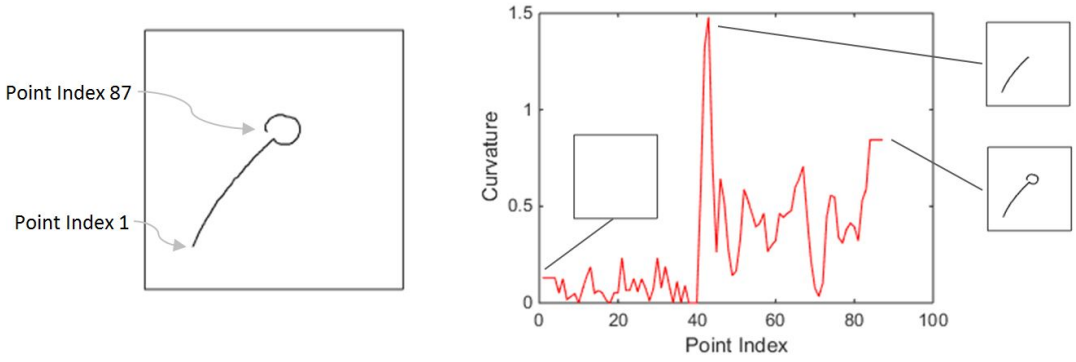


Figure 3-2: When a stroke transitions from an edge to a vertex, the curvature of the stroke increases. The graph on the right shows the curvature at each stroke point, and the insets indicate how much of the sketch has been completed at the associated point.

significant changes in curvature, a design decision that was influenced by the properties of transitions that can be made between the three types of graph components. Figure 3-2 and Figure 3-3 show two examples of how the curvature of a stroke changes during a transition between a pair of components. As displayed in Figure 3-2, as a user finishes drawing an edge and begins drawing a vertex with the same stroke, the curvature of the stroke becomes much higher. If an arrow is drawn immediately following an edge, as seen in Figure 3-3, the region of transition has much higher curvature than the regions immediately preceding and following the transition. While these are two different types of curvature changes, with the first persisting and the second fleeting, the curvature of a stroke transitioning from any graph component to another graph component will typically show one of these patterns.

Before curvatures are calculated, some initial processing of the input stroke data is needed. For our online education application, stroke data is collected using JavaScript, and the current position of the stylus is added to an array whenever a move event occurs. The distance between stylus positions extracted from consecutive move events can be highly variable. In general, the distance between positions from consecutive events is related to the speed of the stylus, although move events are not triggered at regular time intervals. This results in the granularity of the data being unnecessarily fine in regions where the stylus was moved slowly. Iterating through the array of

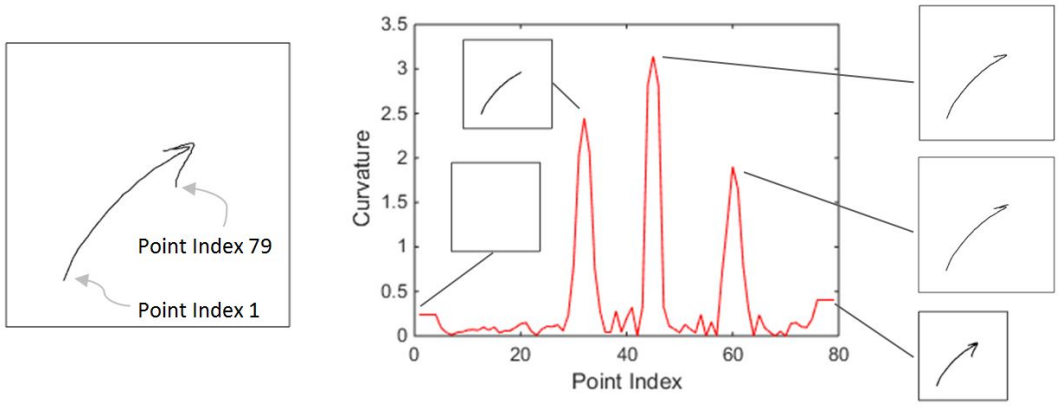


Figure 3-3: When a stroke transitions from an edge to an arrow, the curvature of the stroke momentarily increases. This increase is also seen at the other points throughout the stroke where abrupt changes in direction occur. The graph on the right shows the curvature at each stroke point, and the insets indicate how much of the sketch has been completed at the associated point.

collected points, we remove points that are within a distance threshold, d_t, from their neighbors. Let $P_i = (x_i, y_i)$ represent the position of the i-th point on the stroke, and let $\overrightarrow{P_iP_{i+1}}$ be the 2-dimensional vector from point P_i to point P_{i+1}. For each stroke, this removal process results in a sequence, $S = [P_1, P_2, P_3, \ldots, P_n]$, of n points where $\left\|\overrightarrow{P_iP_{i+1}}\right\| < d_t$ for $1 \leq i < n$. We have set $d_t = 2$ where the units are screen pixels.

Using the refined strokes, the curvature of a point P_i is computed as the angle formed by three points where the middle point is P_i. These three points are not necessarily consecutive, but rather their indices are separated by the skip parameter, s. Mathematically, the curvature of a point P_i is

$$C_s(P_i) = \arccos\left(\frac{\overrightarrow{P_{i-s}P_i} \cdot \overrightarrow{P_iP_{i+s}}}{\left\|\overrightarrow{P_{i-s}P_i}\right\| \left\|\overrightarrow{P_iP_{i+s}}\right\|}\right) \text{ for } s+1 \leq i \leq n-s.$$

The curvatures near the stroke endpoints are calculated as

$$C_s(P_i) = C_s(P_{s+1}) \text{ for } 1 \leq i \leq s$$

and

$$C_s(P_i) = C_s(P_{n-s}) \text{ for } n-s+1 \leq i \leq n.$$

The skip parameter, s, couples with d_t to influence the separation between the three points used in the curvature computation. A separation that is too small can cause the computations to be affected by small wiggles or noise in the stroke, and a separation that is too big can fail to capture meaningful changes in curvature. The skip parameter is one of three parameters in the segmenter that will be trained in Section 3.3.

Given the curvature values at each point in a stroke, abrupt and significant changes in curvature must next be detected. The extent to which a point's curvature is abnormal is calculated as the difference between the point's curvature and the average of the surrounding curvatures. A window parameter, w, determines the number of surrounding points used in the computation. Mathematically, the abnormality of the curvature of point P_i is

$$A_{s,w}(P_i) = C_s(P_i) - \left(\frac{\sum_{j=a}^{i-1} C_s(P_j) + \sum_{j=i+1}^{b} C_s(P_j)}{b-a} \right)$$

where

$$a = \max(1, i-w) \text{ and } b = \min(n, i+w) \text{ for } 1 \leq i \leq n.$$

The final step in determining where to place segmentation points is to find the ranges of consecutive points where $k \cdot A_{s,w}(P_i) > 1$ for each point in the range and then to select the point with the maximum abnormality value within each range. The multiplicative factor k represents the third and final parameter in the segmentation step, and its effects can be easily understood. As k increases, false positives become more common, and false negatives become less common. Precisely, P_i is a segmentation point if

$$\begin{aligned} \exists a, b \text{ s.t. } (a < i < b) &\wedge (k \cdot A_{s,w}(P_j) > 1 \text{ for } a < j < b) \\ &\wedge (k \cdot A_{s,w}(P_a) \leq 1) \wedge (k \cdot A_{s,w}(P_b) \leq 1) \\ &\wedge (A_{s,w}(P_i) > A_{s,w}(P_j) \text{ for } a < j < b, j \neq i). \end{aligned}$$

Note that the above method is different from locating the points where $k \cdot A(P_i)$

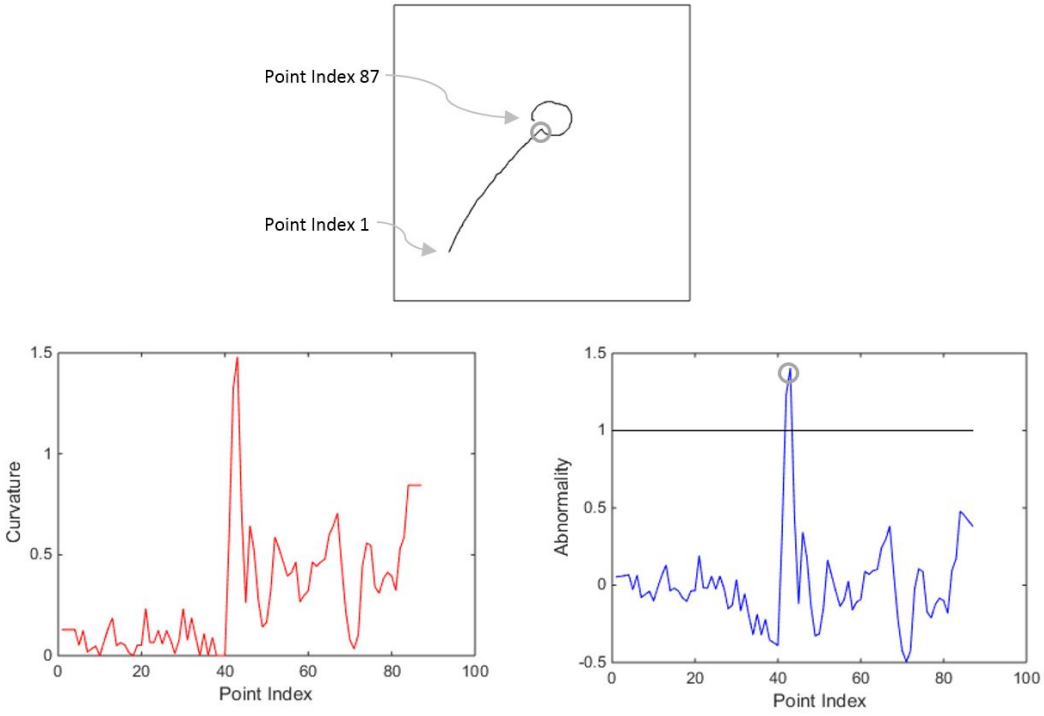


Figure 3-4: The curvature and scaled abnormality values are plotted for each point in this stroke ($s = 3$, $w = 14$, $k = 1.2$). This is an instance where the segmenter made no errors, and the circled regions refer to the same point on the original stroke.

is both a local maxima and greater than 1. This alternative approach was rejected because it would lead to excessive false positive errors. As seen in Figure 3-4, with a specific setting of the parameters s, w, and k, the complete segmentation procedure correctly identifies the segmentation points for a sample stroke.

Before settling on the segmentation method presented above, we experimented with time-based segmentation because transitions from one graph component to the next are often also accompanied by a momentary drop in stylus speed. Using a method parallel to the one described above, the points with speeds abnormally lower than than the speeds of their neighbors were selected as segmentation points. This approach placed unnecessary segmentation points at the locations where users momentarily paused while drawing individual components. Switching to the curvature-based approach to segmentation prevented these types of pauses from generating false positives while continuing to reliably detect true segmentation points.

3.3 Training and Testing

The three parameters that control the behavior of the segmenter required training. The skip parameter s affects the curvature calculations, the window parameter w affects the abnormality calculations, and the multiplicative factor k affects the threshold for choosing segmentation points. As mentioned in Chapter 2, for training purposes, we collected stroke data from sketches by 10 individuals of the same 15 graphs using a Mechanical Turk experiment. No drawing guidelines were given to users in an effort to capture a natural sketching experience, so naturally, various drawing styles were represented among the submissions. Some individuals used whole numbers of strokes for the vast majority of components, but others routinely used a single stroke to draw multiple components.

The stroke data by itself isn't helpful for training the segmenter, however, so we did some post-processing by hand to mark the ranges of points for each stroke where true segmentation points should be placed. This process was performed after the granularity of the stroke data was adjusted as explained above, and our segmentation post-processing interface is shown in Figure 3-5. Marking ranges for true segmentation points was chosen over marking individual points because the granularity of the data was still fine enough to allow a margin of error for segmentation point placement and not adversely affect the subsequent component classification step.

Given these ranges, which are used to decide if a segmentation point generated by the segmenter has been correctly placed, we created a cost function for training. Suppose $Q = \{p_1, p_2, p_3, \ldots, p_m\}$ is the set of m segmentation points produced by the segmenter for a refined stroke S and $R = \{(a_1, b_1), (a_2, b_2), (a_3, b_3), \ldots, (a_t, b_t)\}$ is the set of t ranges from the post-processing step for S. (Note that p_i, a_i, and b_i do not represent actual points with 2D coordinates, but rather they are indices into the refined stroke data.) With u and v as weights, the cost for S is then calculated as

$$cost(S) = u * error_1 + v * error_2$$

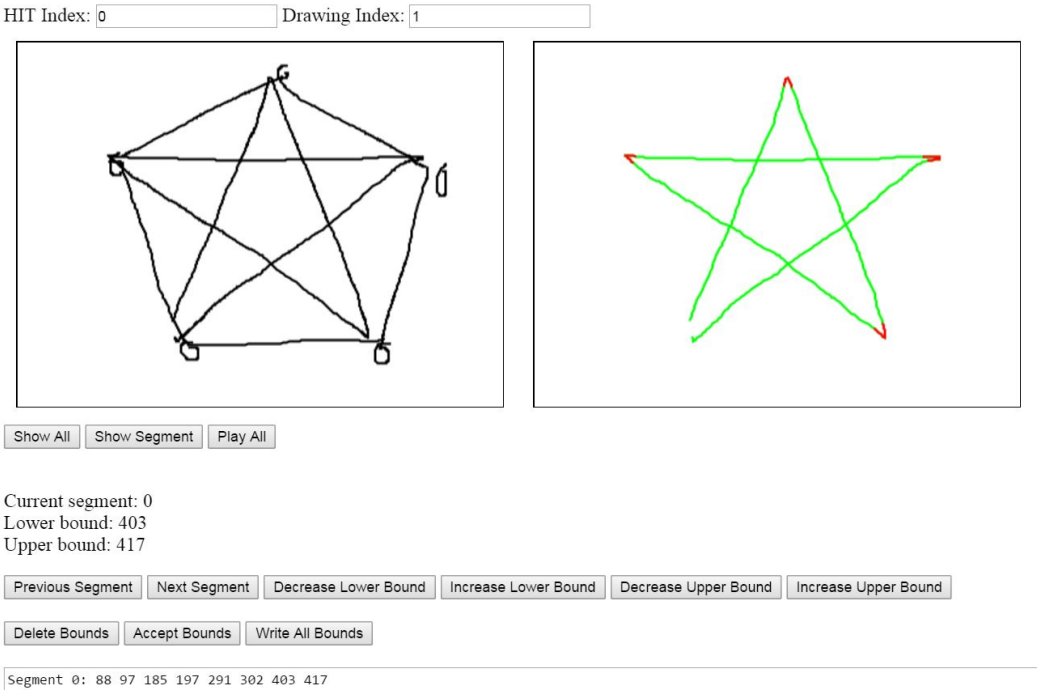


Figure 3-5: In the post-processing interface, the ranges in which true segmentation points should be placed are marked. The stroke displayed on the right contains 5 edges, so there are 4 ranges marked. The data generated by this interface is used to train the segmenter.

where

$$error_1 = m - \sum_{i=1}^{t} I_i$$

$$error_2 = t - \sum_{i=1}^{t} I_i$$

and for $1 \leq i \leq t$,

$$I_i = \begin{cases} 1 & \text{if } \exists p_j \in Q \text{ s.t. } a_i \leq p_j \leq b_i \\ 0 & \text{otherwise.} \end{cases}$$

Here, $error_1$ represents the number of false positives, and $error_2$ represents the number of false negatives. The indicator variable I_i reflects whether or not at least one segmentation point was placed within the i-th range. Finally, we have set $u = 0.02$ and $v = 1$ to reflect the fact that false negative errors are much more costly than false

positive errors.

As explained in Chapter 2, a major goal during the training and testing of each recognition algorithm step was to show that the step being analyzed could perform well when tested on an individual not included in the training data. To set up an experiment capable of making this claim, we created a collection of (training set 1, training set 2, test set) triplets. Training set 1 in a given triplet contained the refined strokes from 10 graphs drawn by a single individual. Training set 2 in the same triplet contained the refined strokes from the same 10 graphs with each graph drawn by any of the remaining nine individuals. Finally, the test set in the same triplet contained the refined strokes from the remaining 5 graphs drawn by the same individual who drew the training set 1 graphs.

Fifty triplets of training and test data were created, with each individual represented in 5 different test sets. Because edge-to-arrow transitions are the most common location for true segmentation points, the 10 graphs selected for each triplet's training set 1 were chosen randomly but were also required to include at least six graphs containing arrow components. (As can be seen in Appendix A, of the 15 graphs presented to the Mechanical Turk workers, 5 contained only vertices and edges while the other 10 contained vertices, arrows, and edges.)

For each triplet, we optimized one set of parameters based on training set 1, optimized a second set of parameters based on training set 2, and then compared the costs of running the segmenter on the test data using both sets of trained parameters. The optimal set of parameters for a given training set is the set of parameters that produces the minimum cost, and the parameter space explored during training was $(s, w, k) \in \{2, 3, 4\} \times \{8, 9, 10, \ldots, 16\} \times \{0.7, 0.8, 0.9, \ldots, 1.5\}$. Instances where the training set 1 parameters gave a lower test cost suggest that it is helpful to include the individual being tested in the training data. On the other hand, instances where the training set 2 parameters gave a lower test cost suggest that it is not necessary to include the individual being tested in the training data.

The outcomes for the 50 trials are listed in Table 3.1. It is not surprising that including the test individual in the training data produced lower costs in a majority

	Individual										Total
	1	2	3	4	5	6	7	8	9	10	
$cost_1 < cost_2$	2	5	2	3	3	2	5	3	1	2	28
$cost_1 = cost_2$	0	0	1	0	0	0	0	1	0	1	3
$cost_1 > cost_2$	3	0	2	2	2	3	0	1	4	2	19

Table 3.1: Segmentation test set cost comparison. The test cost was computed for two sets of trained parameters for 50 trials divided equally amongst the 10 individuals. The test cost from training on the individual used in the test is $cost_1$, and the test cost from training on the other individuals is $cost_2$. This table shows the breakdown of the cost comparisons for the 5 trials involving each individual.

of the trials, since sometimes individuals do have idiosyncrasies in their sketching habits. In fact, this was likely the case for the trials conducted for individuals 2 and 8, which all went in favor of the parameters optimized for training set 1. On average, across all 100 tests (50 trials with 2 parameter settings), 7.15 false positive errors and 0.17 false negative errors were made per graph.

The test costs produced by training set 1 parameters were lower in only 56% of the trials, however, indicating that it is reasonable to train a segmenter on one set of individuals and expect it to perform well on other individuals. With this in mind, we used the strokes from all 150 collected graphs to produce the final trained segmentation parameters used throughout the rest of this paper: $s = 3$, $w = 14$, and $k = 1.2$. These optimal parameters caused on average 8.07 false positive errors and 0.09 false negative errors per graph across all 150 collected sketches. The parameter combinations $(s, w, k) = (3, 15, 1.2)$ and $(s, w, k) = (3, 16, 1.2)$ produced results of nearly identical quality and in practice would likely work just as well.

Chapter 4

Component Classification

Component classification is the middle step in our three part recognition algorithm. This step receives a list of segments from the previous segmentation step, and it outputs a collection of probabilities, each of which reflects the likelihood that a portion of the sketch forms a specific component type. Section 4.1 describes the role of the classification step, Section 4.2 both reviews existing approaches to classification of symbols and presents our design, and Section 4.3 discusses the training of the classification step's many parameters.

4.1 Role

As noted previously in Chapter 2 and illustrated in Figure 2-1, there is a large amount of acceptable drawing variety among components of each type, yet the classification step must be able to distinguish between different component types. Edges in particular can have much variety, with self-loops (edges whose endpoints connect to the same vertex) appearing quite visually distinct from standard edges (edges whose endpoints connect to different vertices). To allow for more robust categorization, we created four classifiers that each assign probabilities for one of four component types: vertices, arrows, standard edges (which will be referred to henceforth simply as edges), and self-loops.

Three assumptions about the input to the classification step govern which portions

of the sketch are assigned component type probabilities. Two of these are from the original drawing technique assumptions described in Chapter 2, namely that each component is drawn within some maximum amount of time and that each component is drawn in its entirety before another is started. The third assumption is that the segmenter works correctly, producing a list of segments where each segment is part of only a single component. Combining these assumptions together, it becomes necessary for the classifiers to assign probabilities only to consecutive sequences of whole segments that were drawn within the allotted time parameter, which was set generously to 5 seconds.

4.2 Design

A large range of approaches for symbol classification are described in the literature on hand-drawn sketch recognition. Ouyang's chemical diagram recognizer features a discriminative classifier based on Support Vector Machines (SVMs) [OD07], Mishra uses a Hidden Markov Model (HMM) to recognize symbols in graphs from several engineering domains [MEFC09], Gennari's AC-SPARC circuit recognizer features a naive bayes classifier, and Fonseca uses fuzzy sets combined with rules to identify common multi-stroke shapes [FPJ02]. While their methods vary widely, each of these applications makes use of multiple geometric features such as perimeter, area, angles, intersection points, and curvature. Lee et al present a fairly different approach, using attributed relational graphs (ARGs) to describe the geometry and topology of arbitrary multi-stroke symbols and then performing graph matching during classification [LKS07].

We chose a logistic regression approach for our component classifiers. In this binary classification machine learning model, a consecutive sequence of segments is represented by a list of features $[x_0, x_1, x_2, \ldots, x_n]$, and a trained hypothesis predicts whether the segments produce a component of a specific type ($y = 1$) or not ($y = 0$). Each of the four component types (vertices, arrows, edges, and self-loops) has its own

classifier and thus hypothesis of the form

$$h_\theta(x) = g(\theta^T x) = g(\theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_n x_n) \tag{4.1}$$

where θ is a vector of parameters, x is the feature vector, and g is the sigmoid function $g(z) = 1/(1+e^{-z})$. By convention, $x_0 = 1$, while the remainder of the x_i are features calculated for the sequence of segments being classified. A convenient characteristic of the logistic regression model is the probabilistic interpretation of its output [Ng12]:

$$P(y = 1 \mid x;\ \theta) = h_\theta(x)$$
$$P(y = 0 \mid x;\ \theta) = 1 - h_\theta(x)$$

The success of machine learning techniques depends heavily on feature selection. Instead of creating a separate set of features for each of the four hypotheses, we decided to use the same set of five features across all classifiers. The differences among component types are thus reflected in the trained θ parameter vectors, which are determined separately for each component type's hypothesis. Given this design choice, features are most useful when they represent characteristics that are typically found in components of only one type. Much like the features used in most of the existing recognizers introduced at the beginning of this section, our features capture geometric properties. Fonseca's work in particular was a source of inspiration for a few of our features as will be detailed in Section 4.2.2 [FPJ02]. Convex hulls and alpha shapes form the basis of our features; these geometric shape calculations are introduced below, followed by a detailed explanation of each of the five features.

4.2.1 Geometric Computations

Convex Hull

As illustrated in Figure 4-1, a convex hull can be used to roughly describe the shape of a set of points.[1] Mathematically, the convex hull of a set S of points is the smallest

[1] http://mathworld.wolfram.com/ConvexHull.html

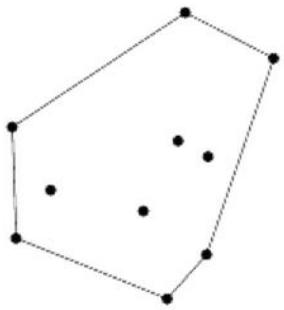


Figure 4-1: The convex hull of a set of points S is the smallest convex subset of a plane that contains S. The area within the lines represents the convex hull of these points.

convex set containing S. A subset R of a plane is convex if for every pair of points $P, Q \in R$, the line segment $\overline{PQ}$ is contained completely in R [DBVKOS00]. The convex hull of S can also be intuitively understood by imagining that each point in S is a pin stuck in a pinboard. If a rubber band were placed around all of the pins, the area inside the rubber band would represent the convex hull. The convex hull is often computed using Graham's scan [Gra72, And79].

Convex hulls are well suited to estimating the shape of vertices, arrows, and self-loops, but they very poorly capture the shape of curved edges. Alpha shapes, which are described below, can often provide a finer interpretation of the shape of a set of points.

Alpha Shape

Alpha shapes, which were first defined by Edelsbrunner, also describe the shape of a set of points, but unlike convex hulls, alpha shapes are capable of having concave regions [EKS83]. The convex hull of a set of points is uniquely defined, but as shown in Figure 4-2, many concave hulls can be drawn for a single set of points.[2] An alpha shape with parameter α for a set of points S is a uniquely defined shape, often a concave hull, that is created by removing segments from the Delaunay triangulation of S.

A Delaunay triangulation, such as the one in Figure 4-3, is a uniquely defined

[2] https://alastaira.wordpress.com/2011/03/22/alpha-shapes-and-concave-hulls/

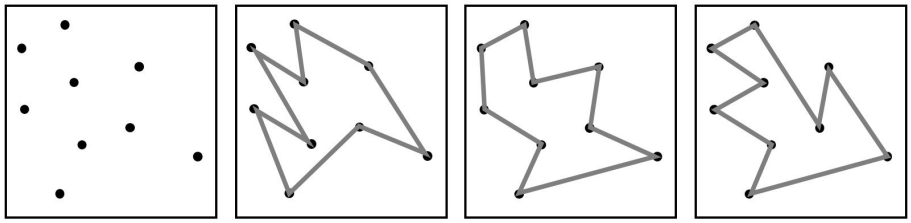


Figure 4-2: The three concave hulls on the right are drawn for the same set of points, displayed separately on the left. Given a specific parameter setting, alpha shapes define a unique and often concave shape for a set of points.

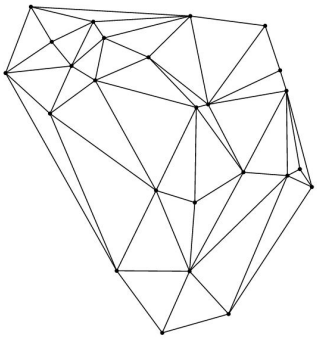


Figure 4-3: The Delaunay triangulation of a set of points is the triangulation where the smallest angle across all triangles in the triangulation is maximized.

triangulation for a set of points that exists when no three points are collinear and no four points lie on the same circle.[3] In comparison with all other possible triangulations of the same set of points, the Delaunay triangulation has the property that the that the minimum angle across all triangles in the triangulation is maximized. The Delaunay triangulation is incidentally also the dual graph of the popular Voronoi diagram [DBVKOS00]. Many methods for computing the Delaunay triangulation exist; we use the iterative Bowyer-Watson algorithm [Bow81, Wat81].

To arrive at an alpha shape from a Delaunay triangulation, segments are removed according to a parameter α. Any triangle with an edge length greater than 2α is removed from the triangulation, resulting in a shape that may be concave, may have holes, and may consist of discontinuous parts. If $\alpha = \infty$, the alpha shape is precisely the convex hull, and if $\alpha = 0$, the alpha shape is the union of the original points [Fis00].

[3] http://upload.wikimedia.org/wikipedia/commons/3/38/Delaunay_Triangulation_%2825_Points%29.svg

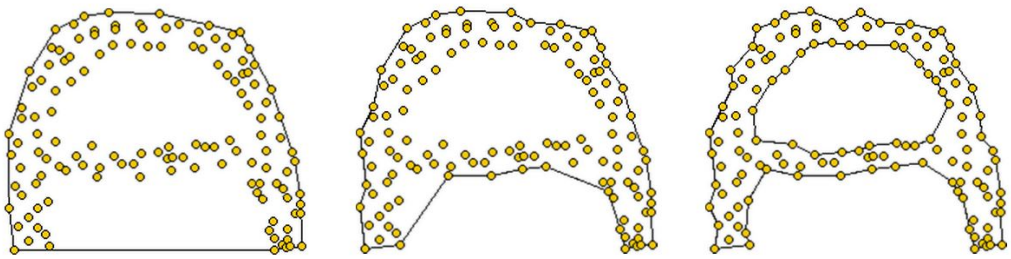


Figure 4-4: The α parameter decreases from left to right in these alpha shapes computed for the same set of points. In the leftmost image, $\alpha = \infty$, effectively producing the convex hull.

Figure 4-4 displays the alpha shape for a single set of points for three different α values, clearly demonstrating that an alpha shape with an appropriately chosen parameter is better suited than a convex hull for representing the shape of a complex set of points.[4]

An analogy similar to the one presented by Edelsbrunner and Mücke provides a helpful way to understand alpha shapes in 2D [EM94]. Imagine that the points from the original set S are each infinitely small rocks in a field of uncut grass and that you have a circular lawnmower of radius α. You use this lawnmower to cut as much of the grass as possible without hitting any of the rocks (picking the lawnmower up and placing it down again is acceptable). The final alpha shape is generated by replacing any arcs between two rocks on the boundary of the uncut grass by straight lines. This process, illustrated in Figure 4-5, produces the same result as the segment removal method from the previous paragraph due to the property that a circle circumscribing any triangle in a Delaunay triangulation does not contain any other points in the original set.[5]

4.2.2 Features

Prior to computing the convex hulls and alpha shapes that are then manipulated to form features, a small refinement of the segments is needed. Before the segmentation step occurred, each stroke was adjusted such that consecutive points were at least a

[4] http://cgm.cs.mcgill.ca/~godfried/teaching/projects97/belair/alpha.html
[5] http://doc.cgal.org/latest/Alpha_shapes_2/index.html

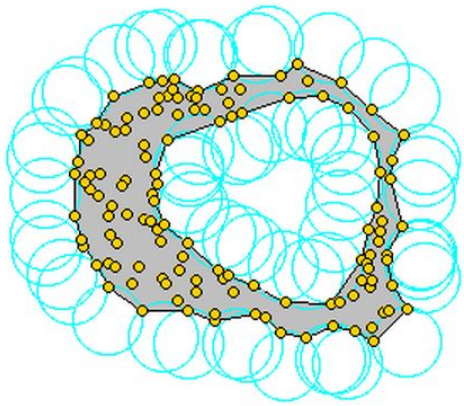


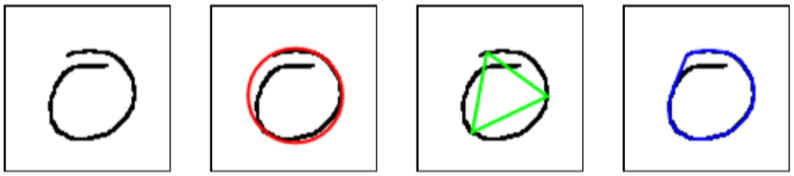
Figure 4-5: An alpha shape can be created by envisioning a circular lawnmower of radius α cutting grass around infinitely small rocks. Any remaining arcs are replaced by straight lines.

minimum distance apart. This removed unnecessarily fine granularity caused by a slow-moving stylus, but in regions where the stylus was moved quickly, the distance between adjacent points in the strokes (and now in the segments) can be large. When the alpha shape is computed, these large distances can cause individual segments to be undesirably separated into disjoint components. To prevent this, we perform piecewise linear interpolation of each segment, adding in new points such that adjacent points are within a maximum distance of each other. The minimum distance used for refining strokes before the segmentation step was 2; the maximum distance we use for refining segments before the classification step is 10.

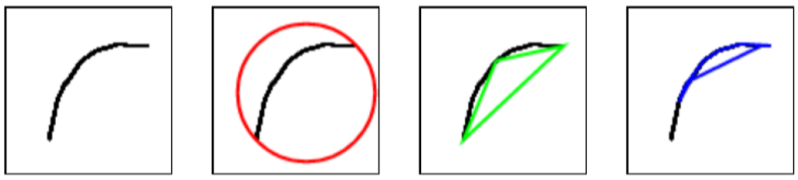
For each sequence of consecutive segments analyzed by the component classifier, a set of five geometric features is computed. These features, which use the geometric shape concepts discussed above, are described in the next few sections. Figure 4-6 displays the set of feature values computed for a few different sets of points.

Circumscribed Circle

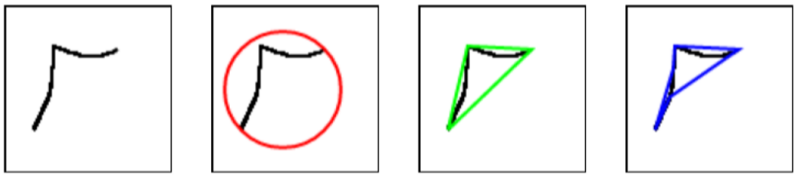
The circular shape of most vertices and some self-loops helps distinguish components of these types from arrows and standard edges. To capture the extent to which a set of points S is circular, the first feature, which ranges between 0 and 1, is calculated



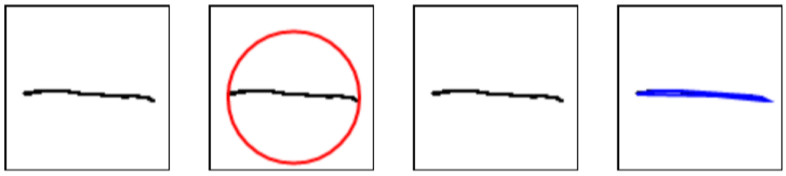
$x_1 = 0.84,\ x_2 = 0.47,\ x_3 = 0.08,\ x_4 = 0.34,\ x_5 = 0$



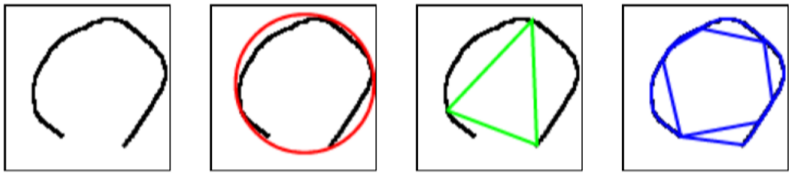
$x_1 = 0.21,\ x_2 = 0.77,\ x_3 = 0.48,\ x_4 = 0.36,\ x_5 = 0$



$x_1 = 0.26,\ x_2 = 0.99,\ x_3 = 0.25,\ x_4 = 0.32,\ x_5 = 0$



$x_1 = 0.04,\ x_2 = 0,\ x_3 = 1.10,\ x_4 = 0.33,\ x_5 = 0$



$x_1 = 0.82,\ x_2 = 0.45,\ x_3 = 0.06,\ x_4 = 0.49,\ x_5 = 0$

Figure 4-6: The five features are calculated for several example strokes in this diagram. The first image in each row shows the original stroke, the second image shows the circumscribed circle used to compute x_1, the third image shows the maximum inner triangle (if one satisfying the angle requirement exists) used to compute x_2, and the fourth image shows the alpha shape with parameter $\alpha = 25$ used to compute x_3.

as

$$x_1 = \frac{\text{Area of } CH(S)}{\text{Area of circumscribed circle of } CH(S)}$$

where $CH(S)$ represents the convex hull of S. This ratio is 0 for a perfectly straight line segment, ≈ 0.41 for an equilateral triangle, and 1 for a perfect circle. Although it is true that an alpha shape can capture the shape of a set of points more accurately than a convex hull can, for this feature, we are only interested in the points' convex outer boundary. This boundary is determined by the convex hull without any complications stemming from points either populating or not populating the central region. In the denominator, the circumscribed circle of the convex hull was chosen over the inscribed circle of the convex hull simply because it is easier to compute.

Inscribed Triangle

Arrows are best differentiated from the other component types by their triangular shape. The second feature calculates the extent to which a set of points S is triangular:

$$x_2 = \frac{f(\text{maximum inner triangle of } CH(S))}{\text{Area of } CH(S)}.$$

Here, $CH(S)$ is the convex hull of S, the maximum inner triangle of $CH(S)$ is the triangle with vertices in $CH(S)$ with the largest perimeter, and

$$f(\triangle ABC) = \begin{cases} \text{area of } \triangle ABC & \text{if } \angle A, \angle B, \angle C > 20^\circ \\ 0 & \text{otherwise.} \end{cases}$$

Similar to the first feature, the range of x_2 is $0 \leq x_2 \leq 1$. The ratio is 0 for a straight line, ≈ 0.41 for a perfect circle, and 1 for a triangle with at least two sides drawn and each angle greater than 20°. While the reasons for selecting a convex hull over an alpha shape were subtle for the first feature, here a convex hull is needed to capture the triangular shape of an arrow that is drawn with two sides as opposed to three. The 20° requirement on the angles of the maximum inscribed triangle is essential and was added to prevent very slightly curved segments, which are often edges, from

being assigned high triangularity scores. Finally, in the numerator, the maximum inner triangle was used instead of the minimum outer triangle to make computation simpler.

This inscribed triangle feature and the previous circumscribed circle feature were inspired by Fonseca et al's features in their application for recognizing twelve common geometric shapes. They inscribed triangles and quadrilaterals inside convex hulls and then constructed features based on ratios between the perimeter and area of these special polygons. While we ignore perimeter and instead use area ratios between the convex hull and a special polygon in these first two features, the next feature does use a perimeter to area ratio to capture thinness [FPJ02].

Alpha Shape Ratio

The third feature aims to assign high scores to all standard edges and low scores to vertices, arrows, and self-loops by representing the thinness of the set of points S where $A(S, 25)$ denotes the alpha shape of S computed with parameter $\alpha = 25$:

$$x_3 = \frac{1}{500} \cdot \frac{\text{perimeter of } A(S,25)}{\text{area of } A(S,25)}.$$

When a constant perimeter is maintained, the area of a shape decreases as the shape becomes more narrow, so edges will have higher x_3 values than other components, provided the alpha parameter is selected appropriately. Using the analogy from earlier, the alpha parameter must be large enough to leave significant amounts of uncut grass at the tip of a a 2-sided arrow and in the center of vertices, edge-loops, and 3-sided arrows. At the same time, the alpha parameter must be small enough to allow the lawnmower to cut the vast majority of the grass away from a softly curved edge. We found that $\alpha = 25$, with units in screen pixels, generally satisfied these requirements. In order to keep x_3 in roughly the same range as x_1 and x_2, an adjustment that benefits the gradient descent method used during training, we divide the original perimeter to area ratio by 500.

Perimeter

As the simplest of the features, the fourth feature uses perimeter to represent the general size of the set of points S with convex hull $CH(S)$:

$$x_4 = \frac{1}{500} \cdot \text{perimeter of } CH(S).$$

In general, vertices and arrows have the smallest perimeter, self-loops have an intermediate perimeter, and edges have the largest perimeter. Again, the division by 500 attempts to keep x_4 in the same range as the other features.

Disjoint Shapes

The fifth and final feature attempts to differentiate actual graph components from the large number of consecutive segment sequences that do not represent a graph component but will nevertheless be processed by the classifier. In many of these non-component sequences, though not all, the corresponding set of points consists of multiple clearly discontinuous regions. Actual components, in contrast, should not have multiple clearly discontinuous regions of points. Thus, we have, for a set of points S,

$$x_5 = (\text{number of regions in } A(S, 25) \text{ more than 50 units apart}) - 1$$

where again, $A(S, 25)$ represents the alpha shape of S with $\alpha = 25$. We only count two regions as being clearly discontinuous if they are more than 50 units apart. The reason for decrementing the number of clearly discontinuous regions by one will become apparent at the very end of this chapter.

4.2.3 Earlier Design

Prior to considering this simple set of five features, we attempted to build classifiers that would determine whether or not a set of points represented a complete component. A complete component was defined to consist of all the segments used to draw

that symbol; a subset of these segments that still resembled the symbol did not form a complete component. In this earlier iteration of the recognition algorithm, there was no extensive domain interpretation step, so complete components were part of an effort to incorporate an understanding of the surrounding graph into the classification step.

Along with the sequence of temporally consecutive segments to be classified, these earlier classifiers also received as input the immediately preceding segment and the immediately following segment. Finding a simple set of features that produced accurate classifications for complete components proved difficult, and it became obvious that observing a handful of segments temporally surrounding the sequence being classified did not provide very good graph understanding. The final classification step design does not attempt any graph understanding. Rather, sequences of segments are classified without regard to temporally or spatially surrounding segments, and the domain interpretation step uses connections between potential components to form an understanding of the graph.

4.3 Training and Testing

To complete the component classifier, we trained the θ parameter vectors that multiply the feature vectors in each of the four hypotheses, which follow the form in (4.1). Training data was again produced by post-processing the sketch data collected from our Mechanical Turk experiment in which 10 individuals each drew the same 15 graphs. This time, the bounds of each component were recorded along with the component type (vertex, arrow, edge, or self-loop) during post-processing. In order to train the classifier parameters independently of the segmenter, bounds were marked using the original strokes rather than segments. The lower and upper bounds were each specified as a specific position on a specific stroke.

We allowed only a single lower and upper bound pair for each component during the post-processing step, implicitly assuming that each component was drawn in its entirety before another was begun. Although this is one of the drawing technique

assumptions we made in Chapter 2, we did not specifically ask Mechanical Turk workers to follow this rule or any other rules while drawing. Nevertheless, the assumption was met for all but a handful of the roughly 2800 components that were drawn across all 10 individuals. Thus, the component training process was not negatively affected, and we also learned that our temporally contiguous component sketching requirement does not unnecessarily restrict users' drawing freedom.

To separate these components with their accompanying bounds and types into preliminary training sets and test sets, we followed a protocol similar to the one used in segmenter training. Again, a collection of (training set 1, training set 2, test set) triplets were created. Training set 1 in a given triplet consisted of 70% of the roughly 280 components drawn by a single individual. Training set 2 in the same triplet contained an equivalent number of components drawn by any of the remaining nine individuals. Finally, the test set in the same triplet contained the remaining 30% of the components drawn by the same individual who drew the training set 1 components. Fifty triplets of training and test data were created, with each individual represented in 5 different test sets. Because self-loops were so rare, with only eight appearing across all the original graph images, the components selected for each triplet's training sets 1 and 2 were selected randomly but were also required to include at least four self-loops.

The preliminary training and test sets from the triplets were then converted into separate lists of (x, y) pairs for each of the four classifiers to form the final training and test sets. In each pair, which corresponded to a single component from the preliminary sets, x was the vector of features calculated for the component's set of points, which could be determined from the original strokes and the component's identified bounds. The y variable in each pair represented whether the component was a positive example for the specific classifier ($y = 1$) or a negative example ($y = 0$).

Since all the examples, positive or negative, were components of some type and not non-components, they all theoretically consisted of a single continuous region ($x_5 = 0$). Constant features are useless during training, so we reduced the x vector in each pair to $x = [1, x_1, x_2, x_3, x_4]$ (recall that by convention $x_0 = 1$). As will be

explained later, the fifth feature was reinstated in the hypothesis after training the similarly reduced parameter vector $\theta = [\theta_0, \theta_1, \theta_2, \theta_3, \theta_4]$. Recall from (4.1) that each θ_i multiplies the corresponding feature value, x_i, in the relevant hypothesis, and thus the θ_i can be interpreted as weights.

Typically, for a multiclass classification problem such as ours, the classifier corresponding to a specific class uses examples of that class as positive examples ($y = 1$) and all examples from other classes as negative examples ($y = 0$). Due to the similarities between vertices and self-loops, however, the final training and test sets for both our vertex and self-loop classifiers only included arrows and standard edges as negative examples. The arrow and edge classifiers did follow the standard procedure of including all examples from the other three classes as negative examples. Differentiation between vertices and self-loops occurs primarily in the domain interpretation step when graph connectivity is analyzed rather than in the classification step.

Given a training set of m (x, y) pairs for a specific classifier, the optimal θ is found by minimizing the logistic regression cost function $J(\theta)$ [Ng11]:

$$J(\theta) = -\sum_{i=1}^{m} (y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))) \tag{4.2}$$

where $(x^{(i)}, y^{(i)})$ is the i-th training example and the hypothesis $h_\theta(x)$ is defined as in (4.1). Since $J(\theta)$ is a convex function, it can be minimized using batch gradient descent, where on each step, all θ_j are simultaneously updated according to

$$\begin{aligned} \theta_j &:= \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \\ &:= \theta_j + \alpha \sum_{i=1}^{m} (y^{(i)} - h_\theta(x^{(i)})) x_j^{(i)}. \end{aligned}$$

Continuing the probabilistic interpretation of logistic regression introduced previously, minimizing $J(\theta)$ is also equivalent to maximizing the likelihood of the parameters, $L(\theta)$ [Ng12]:

$$L(\theta) = \prod_{i=1}^{m} P(y^{(i)} \mid x^{(i)};\ \theta).$$

	Individual										Total
	1	2	3	4	5	6	7	8	9	10	
$\sum J(\{\theta^{(1)}\}) < \sum J(\{\theta^{(2)}\})$	1	5	5	3	5	4	3	4	2	5	37
$\sum J(\{\theta^{(2)}\}) < \sum J(\{\theta^{(1)}\})$	4	0	0	2	0	1	2	1	3	0	13

Table 4.1: Classification test set cost comparison. The test cost was computed for two sets of trained parameters for 50 trials divided equally amongst the 10 individuals. The test cost from training on the individual used in the test is $\sum J(\{\theta^{(1)}\})$, and the test cost from training on the other individuals is $\sum J(\{\theta^{(2)}\})$. This table shows the breakdown of the cost comparisons for the 5 trials involving each individual.

For each of the fifty triplets, using the above method, we optimized one set of parameters, $\{\theta^{(1)}\}$, based on training set 1 and optimized a second set of parameters $\{\theta^{(2)}\}$ based on training set 2. The set notation indicates that there are four separate parameter vectors, one for each of the classifiers. We then compared the costs of both sets of trained parameters for the test set using the cost function from (4.2). Instances where $\sum J(\{\theta^{(1)}\}) < \sum J(\{\theta^{(2)}\})$ suggest that it is helpful to include the individual being tested in the training data, while instances where $\sum J(\{\theta^{(2)}\}) < \sum J(\{\theta^{(1)}\})$ suggest that this is not necessary. Here, the sums are over the costs produced by the four classifiers. Table 4.1 lists the outcomes for the 50 trials.

The parameters trained on training set 1 produced a lower test cost in 74% of the trials, indicating that including the individual being tested in the training data might be advantageous. It is still not unreasonable, however, to train on one set of individuals and expect accurate classification of components drawn by a different set of individuals. Since our application is based on this latter situation, we trained the four classifiers on the entire set of data collected from the Mechanical Turk experiment. At this point, we also reconsider the fifth feature regarding the number of disjoint shapes that was excluded from training. Actual components have a single continuous region ($x_5 = 0$) whereas non-components often have multiple disjoint regions ($x_5 > 0$). Therefore, a simple way to assign these non-components a low classification score is to multiply this feature by a large negative parameter weight such as $\theta_5 = -1000$. The final trained component classification parameters used throughout the rest of

this paper are as follows:

$$\theta_{vertex} = [1.20, 21.20, -14.89, -8.61, -5.88, -1000]$$
$$\theta_{arrow} = [-4.13, -8.45, 21.40, -3.98, -27.19, -1000]$$
$$\theta_{edge} = [0.67, -18.85, -3.07, 2.44, 17.78, -1000]$$
$$\theta_{self-loop} = [-0.28, 23.74, -16.93, -25.10, 6.40, -1000].$$

Some of these trained parameters can be easily understood. The first calculated feature, x_1, represents the extent to which a set of points is circular; as expected, θ_1 is large and positive for the vertex and self-loop classifiers and is negative for the arrow and edge classifiers. The triangularity feature, x_2, has a large positive θ_2 for the arrow classifier, and the alpha shape feature, x_3, has a positive θ_3 only for the edge classifier. Finally, the perimeter feature, x_4, receives negative weights for the vertex and arrow classifiers, a small positive weight for the self-loop classifier, and a large positive weight for the edge classifier, as intended.

Chapter 5

Domain Interpretation

The final step in our graph recognition algorithm is domain interpretation. With the probabilities generated by the component classifiers as its input, this step produces a final interpretation of the users' sketched graph. Section 5.1 describes the role of the domain interpretation step, Section 5.2 both reviews final steps of existing recognition algorithms and presents our design, and Section 5.3 discusses the training of the domain interpretation step's three parameters.

5.1 Role

To allow the output to be easily processed by solution-checking code written by course developers, the final interpretation produced by the domain interpretation step is represented as a list of vertices, arrows, and edges with accompanying arrays that identify connections between vertices and edges and connections between arrows and edges. This representation fully defines a graph, and for problems that have a single correct answer, it can be used to determine if the user's sketched graph is isomorphic to the correct graph.

Perhaps the most trivial method of selecting a best final interpretation is to find the partition of segments that produces the largest sum of classification probabilities.

First, let us define the set P of possible partitions for a graph with n segments as

$$\begin{aligned} P = &\{[(a_1, b_1, t_1), (a_2, b_2, t_2), \ldots, (a_m, b_m, t_m)] \mid \\ &(a_1 = 1) \wedge (b_m = n) \wedge (t_i \in T \text{ for } 1 \leq i \leq n) \wedge (a_i = b_{i-1} + 1 \text{ for } 1 < i \leq n)\} \end{aligned} \tag{5.1}$$

where a_i and b_i are segment indices and $T = \{\text{vertex, arrow, edge, self-loop}\}$. Then, this ideal, largest sum partition would be

$$p_{max} = \underset{p \in P}{\text{argmax}} \sum_{(a_i, b_i, t_i) \in p} f(a_i, b_i, t_i). \tag{5.2}$$

where the argument in the summation is defined as

$$f(a_i, b_i, t_i) = \begin{cases} h_{\theta_{vertex}}(x(a_i, b_i)) & \text{if } t_i = \text{vertex} \\ h_{\theta_{arrow}}(x(a_i, b_i)) & \text{if } t_i = \text{arrow} \\ h_{\theta_{edge}}(x(a_i, b_i)) & \text{if } t_i = \text{edge} \\ h_{\theta_{self-loop}}(x(a_i, b_i)) & \text{if } t_i = \text{self-loop} \end{cases} \tag{5.3}$$

and $x(a_i, b_i)$ produces the feature vector for the sequence of segments starting at segment a_i and ending at segment b_i, inclusive.

This simple maximum partition score approach would succeed if there were little to no interaction between components. The simple graph domain, however, involves much interaction between components, and as noted previously in Figure 2-2, sometimes humans must even observe the components surrounding a sequence of segments in order to correctly identify the segments' component type. Common mistakes resulting from the approach described above include interpreting an arrow as two edges and creating more components than necessary when strokes are traced multiple times. By reasoning about connections between adjacent components and the role a component plays in the graph as a whole, humans are able to avoid many of these errors during interpretation.

Since connections between components need to be established anyway to complete the final interpretation representation discussed above, the selection of the best interpretation might as well be informed by these detected connections. Several rules and conventions in the graph domain provide ideas for how the presence or absence of connections might be used to guide interpretation. For one, in completed graphs, the endpoints of edges should each be connected to exactly one vertex (though vertices can connect to more than one edge). Also, in completed graphs, edge endpoints should be connected to at most one arrow, and every arrow should be connected to exactly one edge endpoint. While these rules can be applied strictly to completed graphs, they do not necessarily apply at all stages of the drawing process and are thus better treated as guidelines when considering intermediary versions of graphs.

5.2 Design

Some existing handwriting recognition applications stop at the classification step, but often when there is some notion of connection between individual symbols, a step involving interpretation of the sketch follows classification. The behavior of these steps is typically highly influenced by the application domain. For example, Gennari's AC-SPARC system attempts to identify missed components by detecting long sequences of segments that were each identified as wires since this would be an uncommon method of sketching a circuit were it intentional [GKSS05]. In Ouyang's chemical diagram recognition application, hypotheses are checked against a large list of chemical structure rules involving the bonding capabilities of various elements [OD07].

We chose to select the best graph interpretation by ranking candidate graphs with scores that are a function of the classifier probabilities, the total number of components, the number of connections made, and the number of missing connections. As in the trivial approach, candidate graphs are represented using the partition notation described in (5.1). For a candidate graph p, the total number of components, $C(p)$, is simply the combined number of vertices, arrows, edges, and self-loops. The num-

ber of connections made, $D(p)$, is twice the number of vertex-edge and arrow-edge connections (the factor of two recognizes each component in the connected pair).

The number of missing connections, $M(p)$, has a slightly more complex definition and is computed as $M(p) = c+d+1000e$ where c is the number of edge endpoints not connected to a vertex, d is the number of arrows not connected to an edge, and e is the number of self-loops where both endpoints are not connected to the same vertex. Since vertices and self-loops are so geometrically similar, as noted in Chapter 4, their differences in forming connections must be emphasized in the domain interpretation step. The third term in the missing connections sum is weighted heavily to ensure that in the interpretation ultimately selected, sequences of segments identified as self-loops have the connection properties associated with self-loops. Note that there is no penalty for vertices without incident edges or for endpoints without arrows because these are both acceptable in completed graphs.

To calculate the score for a candidate graph, $V_{A,B,C}(p)$, we combine each of these characteristics of the graph using the parameters A, B, and C as weights:

$$V_{A,B,C}(p) = \begin{cases} g(p) + A \cdot C(p) + B \cdot D(p) + C \cdot M(p) & \text{if } \forall i, f(a_i, b_i, t_i) > 0.2 \\ 0 & \text{otherwise} \end{cases} \tag{5.4}$$

where

$$g(p) = \sum_{(a_i,b_i,t_i)\in p} f(a_i, b_i, t_i).$$

The three parameters A, B, and C are constants that determine the relative importance of the number of components, the number of connections made, and the number of missing connections, respectively. Section 5.3 details the training of these parameters. To prevent any components with exceptionally low classification probabilities from being in the top-ranking candidate graph, a score of zero is assigned whenever a component has a probability below 0.2. The function f was defined in (5.3).

Before vertex-edge connections and arrow-edge connections can be detected, machine specifications for each component in the candidate graph must be created. A

component's machine specification contains information about the geometric properties of the component, and the need for these specifications is best illustrated through an example. Suppose that one sequence of segments is given a high score by the edge classifier, another sequence is given a high score by the vertex classifier, and we want to determine if there is a connection between the supposed edge and the supposed vertex. Proximity of the two sets of points is not sufficient; we also need to know the locations of the endpoints of the edge, which is not trivial if multiple segments are involved in creating the edge. Edge endpoints can be easily derived from the specification of an edge.

Machine specifications are also used to communicate to the user the recognizer's interpretation of the current sketch. As Chapter 6 describes in more detail, the user interface we have built automatically triggers the recognition process when there are pauses in sketching activity and replaces a user's strokes with a machine-drawn version of the interpreted graph. This interpreted view is constructed from the specifications of the components and presents vertices as perfect circles, arrows as triangles, and edges as smooth curves. The machine specifications for each of the component types are described next, followed by the methods for establishing connections between components.

5.2.1 Component Specifications

Vertex

A set of points classified as a vertex is specified by a center point and a radius. The circle constructed in the computation of the first feature in Chapter 4, which involved circumscribing a circle around the convex hull of a set of points, defines these two properties. Recording the radius of this circumcircle helps set an appropriate threshold when attempting to connect edges to this vertex.

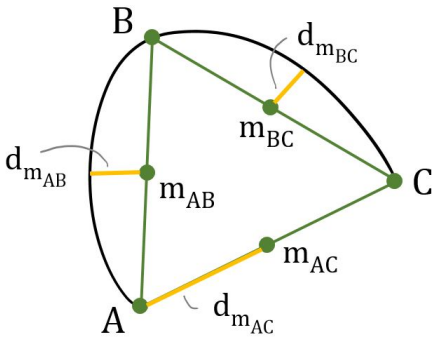


Figure 5-1: The distances to the midpoints of the inner triangle edges are computed to determine whether the arrow is 2-sided or 3-sided.

Arrow

The maximum inner triangle computed in the second feature in Chapter 4 forms part of the specification of a set of points classified as an arrow. We also attempt to capture whether the arrow has two sides or three, since there are additional restrictions for connecting edges to two sided arrows. To determine which one, if any, of the edges in a triangle $\triangle ABC$ is not represented in the user's arrow, we first partition the arrow's set of constituent points, S, into sets S_{AB}, S_{AC}, and S_{BC} such that $S = S_{AB} \cup S_{AC} \cup S_{BC}$, and point $P \in S_{edge}$ if $P \in S$ and P is closest to this edge. (The points corresponding to A, B, and C appear in the two sets representing the incident edges.) Then, we calculate the following distances where m_{edge} denotes the midpoint of that edge:

$$d_{m_{AB}} = \min_{P \in S_{AB}} dist(m_{AB}, P)$$

$$d_{m_{AC}} = \min_{P \in S_{AC}} dist(m_{AC}, P)$$

$$d_{m_{BC}} = \min_{P \in S_{BC}} dist(m_{BC}, P)$$

The Euclidean norm is represented by the $dist$ function, and each $d_{m_{edge}}$ is calculated to be the shortest distance from the midpoint of the edge to any point in the original set that is closest to this particular edge. Figure 5-1 provides a graphical view of these variables.

Given these shortest distances, we attempt to find a non-represented edge. Edge

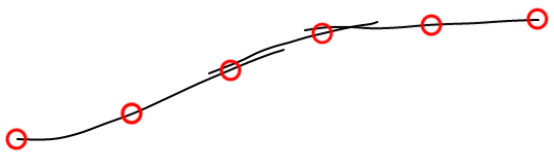


Figure 5-2: To create the specification for an edge consisting of multiple segments, target points that capture the edge's trajectory are sampled from the segments. Here, the circles represent the target points sampled from three overlapping segments.

AB is marked as the non-represented edge if

$$d_{m_{AB}} > \frac{length(AB)}{4} \wedge d_{m_{AC}} < \frac{length(AC)}{8} \wedge d_{m_{BC}} < \frac{length(BC)}{8}.$$

The criteria for marking the other two edges as the non-represented edge is completely analogous. If the criteria is not met for any of the three edges of the triangle, then the specification does not include a non-represented edge, and when connections are made, the arrow is interpreted as having three sides.

Edge

A set of points classified as an edge is specified by a sequence of points such that a curve traced out by these points is smooth, the first point represents one endpoint of the edge, and the last point represents the other endpoint. If an edge is constructed from a single segment, the segment itself essentially satisfies this classification criteria. Creating the edge specification becomes more complicated, however, when the set of points is derived from multiple overlapping segments. In general, we construct the sequence of points for an edge's specification by interpolating between target points selected from the edge's constituent segments. A visual representation of the desired target points for a multi-segment edge is included in Figure 5-2.

For any sequence of segments that has been classified as an edge, the target point sequence is constructed by considering one constituent segment at a time. To initialize the target point sequence, T, we sample points from the segment with the longest length such that adjacent sampled points are roughly 40px apart and the sampled

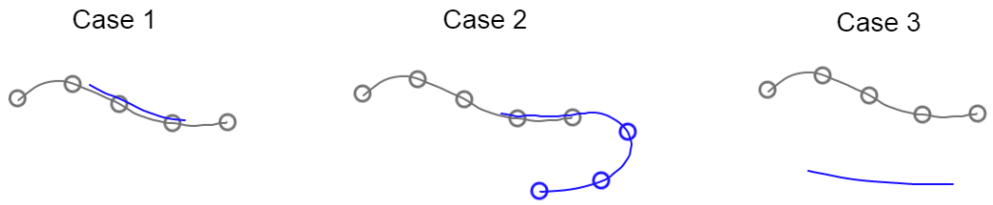


Figure 5-3: Three cases exist when a new constituent segment is considered. In the first case, the segment is contained within the target points, which are shown as circles with a connecting curve. In the second case, the segment partially overlaps with the existing target points, and new target points are added along the non-overlapping portion. In the third case, the segment does not overlap at all, and it is added to the back of the list of segments to consider.

points include the segment's starting and end points. Next, additional target points are added to T as the remaining constituent segments are considered. Let L be the list of remaining segments. While L is not empty, we continually extract the first segment, K, from L and follow the procedure in one of the cases below. Since T was initialized by sampling from the largest length segment, the three cases below cover all possible interactions between K and the current T. Figure 5-3 displays these cases graphically.

- *Case 1: K is encapsulated within T.* No changes to T are needed since K is already adequately represented by the target points.
- *Case 2: K partially overlaps with T.* Add new target points to the appropriate end of T by sampling from the portion of K that does not overlap with T. The non-overlapping endpoint of K should be one of the sampled points, and again, adjacent sampled points should be roughly 40px apart.
- *Case 3: K does not overlap with T.* Process K at a later time by appending it to the end of L.

Once the procedure above has been followed until L is empty, all segments that were used to create the edge are theoretically represented by the sequence of target points T. As the last step in creating the edge specification, we use Spencer Cohen's Smooth.js JavaScript library to interpolate between the target points with a

Catmull-Rom spline [Coh12]. This approach produces a sequence of points forming a smooth curve that passes through the target points and represents the edge's original trajectory.

Self-Loop

The specification for self-loops is nearly identical to the specification for standard edges because the sequence of points representing the edge's trajectory is again the only information needed. The only difference is that we aim for adjacent target points to be 20px apart rather than 40px apart because self-loops typically involve tighter curves.

5.2.2 Connections

In the following subsections, which describe the criteria for establishing connections, the word edge is used to refer to both standard edges and self-loops because the methods of assigning connections to these two types of edges are the same.

Vertex-Edge Connections

Given a candidate graph with machine specifications of each component, every vertex-edge pair is tested for a potential connection. Recall that the vertex is specified by a center point c and a radius r, while the edge is specified by a sequence of target points $E = [e_1, e_2, \ldots, e_n]$. Without loss of generality, let us try to connect the vertex to the edge endpoint specified by the first edge point, e_1. A connection is possible if $dist(c, e_1) - r < \delta$. The expression on the left of this inequality calculates the distance from the first edge point to the circumference of the circle included in the vertex specification. The threshold δ, which we have set to $\delta = 20$, determines whether this distance is sufficiently small. A graphical view is provided in Figure 5-4.

Each endpoint of an edge can only connect to a single vertex. Therefore, when a new vertex-edge pair is tested for a potential connection, we only establish the connection if the requirement from the previous paragraph is met and the connection would

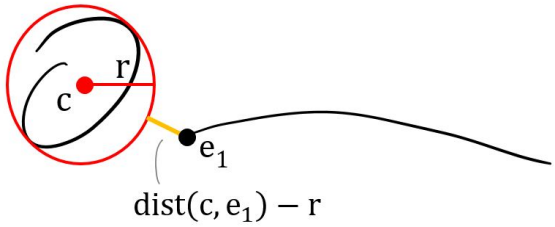


Figure 5-4: Determining whether a vertex and edge endpoint should be connected requires calculating the distance between the endpoint and the vertex's circumscribed circle.

be an improvement over any existing connection. To determine whether a connection is an improvement, for each edge endpoint, we store a score q that represents the current connection (and is initialized to $q = \infty$ before any connections are attempted). If, for a new vertex-edge pair, $dist(c, e_1) - r < \delta$ and $\max(dist(c, e_1) - r, 0) < q$, then any existing connection is dissolved and the pair being inspected forms a new connection. The edge endpoint score is also updated to $q = \max(dist(c, e_1) - r, 0)$.

Arrow-Edge Connections

Similarly, every arrow-edge pair is tested for a potential connection. The arrow is specified by a triangle $\triangle ABC$ and possibly also by a non-represented edge, with the latter being included if the arrow was interpreted during the machine specification step as being two-sided. Again, the edge is specified by a sequence of points $E = [e_1, e_2, \ldots, e_n]$, and without loss of generality, we will try to connect the arrow to the endpoint specified by the first edge point, e_1. First, we extend the edge a distance γ past e_1 to create $E' = [e_0, e_1, e_2, \ldots, e_n]$ where E' still describes a smooth curve and $dist(e_0, e_1) = \gamma$.

Starting from the pair (e_{n-1}, e_n) and moving towards (e_0, e_1), the segment between each consecutive pair of points in E' is tested for intersection with each of AB, AC, and BC. Let the first instance of intersection be at a point x, and without loss of generality, let the triangle edge involved in the intersection be AB. Two conditions must be satisfied for a connection between the arrow and edge to be possible. Firstly,

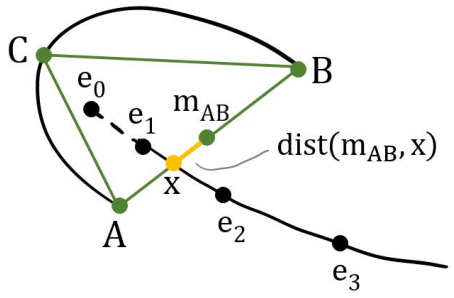


Figure 5-5: Determining whether an arrow and edge should be connected requires calculating the distance between the intersection point and the midpoint of the triangle edge involved in the intersection.

requiring $dist(e_1, x) < \gamma$ or $dist(e_1, C) < \gamma$ ensures that the arrow is positioned close enough to the edge. We have set the threshold γ, which was also used as a distance in the edge extension step, to $\gamma = 20$. Secondly, the edge involved in the intersection, which we have assumed is AB, must be the same edge as the non-represented edge in the arrow specification if one exists. This second requirement ensures that a two-sided arrow is oriented in the proper direction to connect to the edge. Figure 5-5 displays these computations graphically.

Each arrow can only connect to a single edge endpoint, so a connection is only established when it satisfies the two requirements above and would be an improvement over any existing connection. For each arrow, we store a score q for the current connection that is initialized to $q = \infty$. If, for a new arrow-edge pair, a connection is determined to be possible by satisfying the requirements above and $dist(m_{AB}, x) < q$, then a new connection is formed and any existing connection is dropped. The arrow score is also updated to $q = dist(m_{AB}, x)$. Here, we are again assuming that the triangle edge involved in the first intersection is AB, m_{AB} is the midpoint of AB, and x is the point of intersection.

5.3 Training and Testing

The domain interpretation step contains three parameters, A, B, and C, which, as shown in the score computation in (5.4), act as weights for the number of components,

the number of connections established, and the number of connections missing, respectively, in the candidate graph currently being scored. To train these parameters, we again produced training data by post-processing the 150 sketches collected from the Mechanical Turk experiment. Explaining our post-processing procedure requires first introducing the concept of locking in a graph.

Our application attempts to recognize a user's strokes throughout the drawing process, and as is discussed in more detail in Chapter 6, the user interface we have designed communicates the current interpretation periodically to the user by replacing the user's strokes with computer rendered components. Suppose the user has momentarily stopped drawing, causing a set of computer rendered components to appear on the screen. After the user sketches some more and then pauses again, an additional set of computer rendered components should appear, but no changes should be made to the original computer rendered components. Thus, each time the recognition process is triggered, the graph interpretation should be preserved or locked in.

For this chapter, locked in components must not change, and any connections between locked in components must be preserved when new strokes are considered. New connections can be made between newly drawn components and locked in components. The behavior of locked in components is changed slightly in Chapter 6 when eraser functionality in the user interface is discussed. We expected the recognition process to be triggered in the user interface after every few strokes, so our training procedure aimed to optimize the A, B, and C parameters to correctly add five new components at a time to the current locked in interpretation.

During post-processing, for each of the collected sketches, the connectivity of a sequence of progressively larger subgraphs was recorded. We began with a subgraph of the first 5 components the individual drew, added the next 5 components to produce the second subgraph, and continued to add 5 components at a time to produce new subgraphs until the entire graph had been drawn. The training of this domain interpretation step was necessarily tied to the segmentation and classification steps, so when producing the subgraphs, we had to include a whole number of strokes in

each. Thus, in the cases where it was impossible to add exactly 5 more components, we added in the smallest number of components larger than 5 that involved a whole number of strokes. The post-processing step thus produced training data for each sketched graph that consisted of sequences of pairs, where each pair represented a subgraph. The first element in each pair was a list of the strokes added since the previous subgraph, and the second element was the full representation of the subgraph that specified component types and connections.

When the recognizer was run on each of these sequences as part of the training process, the subgraphs were not considered in isolation. Let us define a single sequence as $Q = [(S_1, G_1), (S_2, G_2), \ldots, (S_m, G_m)]$, where S_i is the set of added strokes and G_i is the complete representation for a subgraph i. Each pair builds off the previous pair, so the recognizer must have been able to produce G_i to have had a chance at producing G_{i+1} when the segments in S_{i+1} were considered. Suppose that for the i-th pair, the domain recognition step produced a list of candidate graphs, Z_i, in descending order according to their scores. The final output of the recognizer would have been the top-ranking candidate graph in this list, $Z_i[1]$, but as long as G_i was isomorphic to $Z_i[j]$ for some j, then the recognizer was directed to treat $Z_i[j]$ as the locked in graph when attempting to interpret the new graph produced by adding S_{i+1}. While calculating whether two graphs are isomorphic is expensive in general, isomorphism calculations here were cheap because the order in which components were added to the graph was controlled by the original stroke order.

On some occasions, G_i was not isomorphic to any $Z_i[j]$. This occurred if the segmentation step made a false negative error, failing to detect a true segmentation point, or if a component in G_i was given a score lower than the threshold of 0.2 by its corresponding classifier. In these instances, we recorded all subsequent pairs in the sequence as failures. One might wonder why some $Z_i[j]$ must be used as the locked in graph instead of simply using G_i. When forming connections between newly drawn components and components from a locked in graph, the specifications of the components in the locked in graph must be known; the candidate graphs in Z_i contain this information but the G_i do not.

Given a set of sequences $Y = \{Q_1, Q_2, Q_3, \ldots, Q_n\}$, the cost of the parameters A, B, and C was calculated by a procedure described in pseudocode in Algorithm 1. The main function in this procedure is Cost, while CostSequence and CostPair are helper functions. In the two helper functions, L_i represents the locked in graph for some pair (S_i, G_i) from a specific sequence in Y. If no candidate graph in Z_i matches G_i, then $L_i = false$, the cost is automatically incremented for each of the remaining pairs in the sequence, and $L_j = false$ for $i < j \leq m$. Otherwise, the locked in graph is set to $L_i = Z_i[j]$ for some j.

To create the training and test sets for finding and then evaluating the optimal set of parameters, we again used the triplet method described in Chapters 3 and 4. Briefly, we created 50 (training set 1, training set 2, test set) triplets with each individual represented in 5 test sets. Training set 1 contained the sequences of (S, G) pairs from 10 graphs drawn by a single individual. Training set 2 in the same triplet contained the sequences of (S, G) pairs from the same 10 graphs with each graph drawn by any of the remaining nine individuals. Finally, the test set in the same triplet contained the sequences of (S, G) pairs from the remaining 5 graphs drawn by the same individual who drew the training set 1 graphs. The 10 graphs selected for training set 1 were selected randomly but were also required to include at least 5 graphs containing arrows and at least 2 graphs containing self-loops.

For each triplet, we optimized one set of parameters based on training set 1, optimized a second set of parameters based on training set 2, and then compared the results of running the recognition algorithm on the test data using both sets of trained parameters. The optimal set of parameters for a given training set is the set of parameters that produces the minimum cost, and the parameter space explored was $(A, B, C) \in \{-1.6, -1.5, -1.4, \ldots, -0.9\} \times \{0.2, 0.3, 0.4, 0.5, 0.6\} \times \{-0.6, -0.5, -0.4, -0.3, -0.2\}$. The chosen bounds were negative for A because of the recognition algorithm's tendency to produce extra components, positive for B to reward establishing connections, and negative for C to penalize candidate graphs for missing connections. The costs for the training and test sets were computed for each the different parameter combinations using the cost function defined in Algorithm 1.

Algorithm 1 Compute the cost of set of sequences for given parameters

```
1: function Cost(Y, A, B, C)
2:     totalcost ← 0
3:     for i = 1 to i = n do
4:         totalcost ← totalcost + CostSequence(Q_i, A, B, C)
5:     end for
6:     return totalcost
7: end function

8: function CostSequence(Q, A, B, C)
9:     cost ← 0
10:    for i = 1 to m do
11:        if i == 1 then
12:            cost ← cost + CostPair(S_i, G_i, {}, A, B, C)
13:        else
14:            if L_{i-1} is false then
15:                cost ← cost + 1
16:                L_i ← false
17:            else
18:                cost ← cost + CostPair(S_i, G_i, L_{i-1}, A, B, C)
19:            end if
20:        end if
21:    end for
22:    return cost
23: end function

24: function CostPair(S_i, G_i, L_{i-1}, A, B, C)
25:    Z_i ← recog(S_i, L_{i-1}, A, B, C)
26:    L_i ← LockIn(Z_i, G_i)
27:    if Z_i[1] not isomorphic to G_i then
28:        return 1
29:    else
30:        return 0
31:    end if
32: end function

33: function LockIn(Z_i, G_i)
34:    for j = 1 to length(Z_i) do
35:        if Z_i[j] is isomorphic to G_i then
36:            return Z_i[j]
37:        end if
38:    end for
39:    return false
40: end function
```

	Individual										Total
	1	2	3	4	5	6	7	8	9	10	
$cost_1 < cost_2$	1	4	1	2	3	1	2	4	0	3	21
$cost_1 = cost_2$	3	1	2	1	2	3	3	1	3	2	21
$cost_1 > cost_2$	1	0	2	2	0	1	0	0	2	0	8

Table 5.1: Domain interpretation test set cost comparison. The test cost was computed for two sets of trained parameters for 50 trials divided equally amongst the 10 individuals. The test cost from training on the individual used in the test is $cost_1$, and the test cost from training on the other individuals is $cost_2$. This table shows the breakdown of the cost comparisons for the 5 trials involving each individual.

Instances where the training set 1 parameters gave a lower test cost suggest that it is helpful to include the individual being tested in the training data. On the other hand, instances where the training set 2 parameters gave a lower test cost suggest that it is not necessary to include the individual being tested in the training data. Table 5.1 lists the outcomes for the 50 trials. The training set 1 parameters produced an equal or lower cost in 84% of the trials but a strictly lower cost in only 42% of the trials. It is thus very reasonable to train parameters in the domain interpretation step on one set of individuals and then expect these parameters to provide accurate interpretations on another set of individuals. Training on all 150 sequences produced the following optimal parameters: $A = -1.3$, $B = 0.2$, and $C = -0.6$. These optimal parameters are used throughout the rest of the paper. Several other parameter combinations such as $(A, B, C) = (-1.3, 0.3, -0.6)$ and $(A, B, C) = (-1.3, 0.2, -0.5)$ performed nearly as well as the optimal parameters on the 150 sequences and would likely produce similar quality interpretations in practice.

Chapter 6

User Interface

To demonstrate how students might interact with a simple graph problem type in an online education course, we designed a student interface that interprets sketches of graphs in realtime using our simple graph recognition algorithm. We aim for the drawing process to be natural and quick, the recognition output to be easily understood, and few errors to be made by the recognizer so that students can easily use the interface to create graphs. The functionality of the interface is described in Section 6.1, and Section 6.2 presents the results from a second Mechanical Turk experiment that tested the usability of our interface.

6.1 Functionality

Our user interface allows users to draw graphs, erase graph components, add labels, and move labels. Each of these four modes is activated by pressing the corresponding icon in a toolbar positioned above the drawing canvas as seen in Figure 6-1. Toolbars were mentioned earlier in Chapter 2 as a major drawback of many current drawing application interfaces, for the experience of continually moving between the toolbar and the drawing canvas is unnatural and causes many interruptions. While we allow users to draw components freehand instead of forcing users to drag and drop components from a toolbar, we did not avoid toolbars completely. Drawing, erasing, and moving are three separate uses of the stylus requiring three distinct modes that are

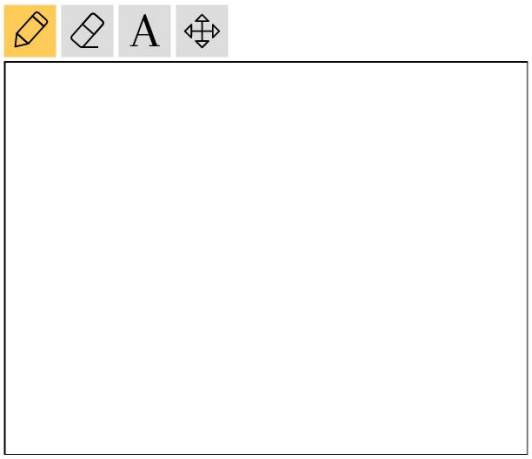


Figure 6-1: The user interface consists of a drawing canvas with a toolbar positioned above. The toolbar contains four modes for drawing, erasing, adding labels, and moving labels.

most easily enabled via a toolbar. The remaining mode for adding labels is needed as long as the recognizer does not interpret handwritten labels.

6.1.1 Creating and Modifying Components

When the mode for drawing graphs is active, users' strokes are periodically replaced by a computer rendered version of the interpreted graph. Instead of adding a fifth button to the toolbar for initiating the graph recognition process, recognition is triggered automatically when the time elapsed since the last drawing activity has exceeded a threshold of 0.5 seconds. Since the classifier and thus the recognition system are only capable of processing whole components, we assume that whenever the threshold condition is met and the recognizer is triggered, a whole number of components have been drawn. Equivalently, we assume that when a component is being drawn, the stylus remains inactive for no more than 0.5 seconds at a time. This was the second time-based assumption mentioned in Chapter 2.

The computer rendered version of the interpreted graph attempts to communicate both the interpreted components and the connections between components to the user. The machine specifications described in Chapter 5 for each component type are used to display vertices as red circles of uniform size, arrows as green triangles,

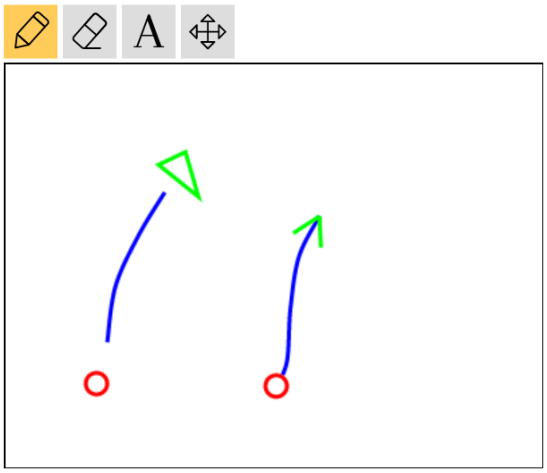


Figure 6-2: When an edge is connected to a vertex, the relevant edge endpoint is placed on the circumference of the vertex. A symmetrical, 2-sided arrow centered on the endpoint of an edge indicates a connection between the arrow and edge. On this sketching canvas, the components on the left are not connected while the components on the right are connected.

and edges as smooth blue curves. Connections between vertices and edges are communicated by placing the relevant endpoint of the edge at the circle circumference. When an arrow is connected to an edge, a 2-sided, symmetrical arrowhead is placed on the tip of the relevant endpoint. Figure 6-2 shows the difference between computer renderings of pairs of components that are not connected and pairs of components that are connected.

Our method of triggering the recognition process causes many best interpretations of the sketched graph to be presented to the user throughout the sketching process. Suppose the user draws some initial set of strokes and then pauses briefly, causing a set of computer rendered components to appear on the screen in place of the strokes. We expect the user to examine this initial set of computer rendered components and correct errors as needed using the eraser mode before starting to draw new strokes. Let G_1 be the computer rendered graph resulting from the first recognition process, and let G_2 be the computer rendered graph remaining after the eraser is used to correct any errors in G_1. If the user proceeded to draw more strokes and then paused again, the result of the second recognition process should be a graph G_3 such that G_2 is a subgraph of G_3.

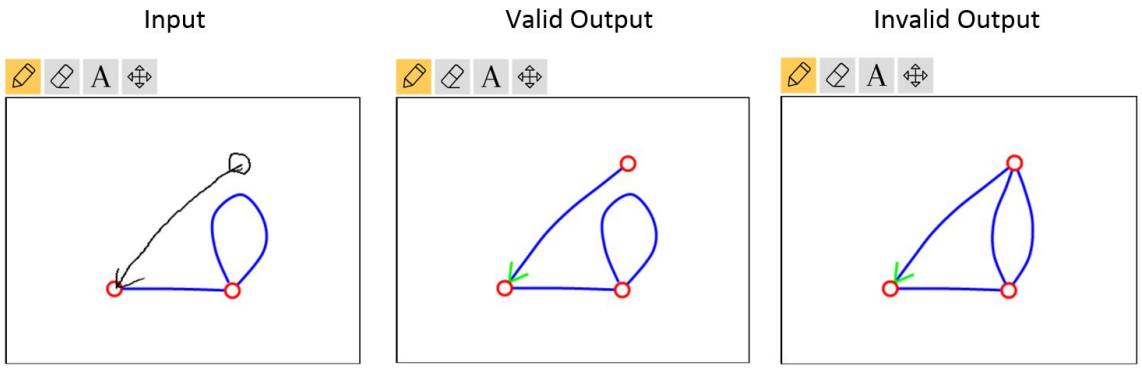


Figure 6-3: Suppose the recognition process is run on the leftmost image, which contains colored, computer rendered components that are part of a locked in graph and black strokes that were recently drawn and are now being recognized. The middle image shows a valid output. The output shown in the rightmost image is invalid because the self-loop from the locked in graph was modified to become two edges.

We achieve this preservation behavior by locking in the existing computer rendered graph each time the recognition process is triggered following the addition of new strokes. Let the locked in graph be L. The recognition process does not change any components in L nor does it break any connections established in L. Rather, the recognition process is only able to create components and establish connections between two newly drawn components or between a newly drawn component and a component from L. Examples of valid and invalid outputs of the recognition process given a locked in graph L and a set of newly drawn strokes are provided in Figure 6-3. This concept of locking in graphs was introduced previously in Section 5.3.

The only way to modify components or connections in a locked in graph is to delete them using the eraser mode. When the eraser mode is active, users can remove computer rendered components by dragging their stylus over a portion of the component. Removing a component causes any connections involving this component to be broken, but the components that formed connections with the removed component still remain in the graph. Any changes to the underlying graph representation caused by eraser usage are immediately communicated through an update of the computer rendered graph.

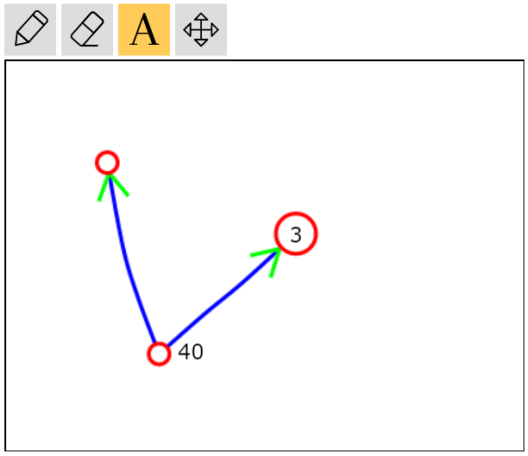


Figure 6-4: Labels associated with vertices are encapsulated by the vertex, while labels associated with edges are simply positioned nearby. In this image, 3 is a vertex label and 40 is an edge label.

6.1.2 Creating and Modifying Labels

Labels can be easily added by switching to the add label mode, clicking on the graph, and typing the desired characters on a keyboard. Once the user presses enter or clicks outside the newly created label, we attempt to associate the label with the closest component within a certain threshold distance. Labels can only be attached to vertices and edges, which we will call label-accepting components. A newly created non-empty label is attached to a label-accepting component if the center of the label is closer to this component than to any other label-accepting component and the distance between the label center and the component is within a threshold distance (currently 10px). If no characters are present in the label or there is no sufficiently close label-accepting component, the label is removed from the graph. Labels that become associated with a vertex are positioned inside the vertex, which enlarges to encapsulate the label as seen in Figure 6-4.

Mistakes associated with labels can be corrected in several ways. The contents of any existing label can be modified while the add label mode is active by clicking on the label and deleting and/or entering new characters using a keyboard. If a label is erroneously attached to a component, the mode for moving labels can be used. While this fourth mode is active, labels gain drag-and-drop functionality, allowing them to

be associated with different components. As a label is dragged around the canvas area, the closest label-accepting component within the threshold distance to the label center is highlighted yellow. To remove a label from the graph, the label can simply be erased (eraser mode), all the characters within the label can be deleted (add label mode), or the label can be moved to a position where no label-accepting component is within the threshold distance (move label mode).

6.1.3 Styluses

Our user interface was designed for use with two main classes of styluses, active and passive. Active styluses, such as the styluses paired with Wacom digitizers and Bluetooth styluses, contain circuitry, and applications can detect when these styluses are hovering over the screen. In addition, active styluses can often provide pressure sensitivity and palm rejection, and some have digital erasers on the side opposite the drawing tip. On the other hand, passive styluses, which are most often capacitive styluses, are essentially fingers with finer precision, and applications can only detect these styluses when they are touching the screen. These styluses are not capable of providing pressure sensitivity, palm rejection, or dual pen/eraser functionality [Hof13].

The only perceivable difference between using our interface with an active stylus versus a passive stylus involves hovering behavior. When an active stylus is used, a cursor tracks the position of a hovering stylus, and the cursor image is indicative of the current mode. In contrast, no cursor is displayed when a passive stylus hovers above the screen because these styluses have no detectable hover state. Lastly, we initially considered triggering the recognition process whenever an active stylus exited the hovering region, but in order to make recognition triggering behavior consistent across all styluses, we settled on the time-based criteria described earlier.

6.2 Testing

We designed the user interface described above to provide a natural drawing experience with minimal interruptions and to quickly and completely communicate the

current internal graph representation. Additionally, the user interface's backend, which was described in Chapters 3 through 5 and attempts to recognize the user's strokes, was designed to achieve high recognition accuracy and thus minimize eraser usage in the interface. Each of these goals relates to the usability of the graph creation process, and we ran a second Mechanical Turk experiment to evaluate these aspects of our user interface.

In the experiment, after workers read through a brief introduction about the functionality of the user interface, they were asked to draw in their browsers a sequence of fifteen graphs consisting only of vertices, arrows, edges, vertex labels, and edge labels. These graphs were different from the graphs used in the first Mechanical Turk experiment and are included in Appendix B alongside some of the submissions we received. Each of the fifteen graphs was presented on a different screen, and workers were instructed to click a button when they thought they had correctly drawn the provided graph image. If the graph was indeed drawn correctly, the screen would advance to show the next graph. If the graph was incorrect, however, the screen did not advance and workers were asked to modify the drawing canvas to correctly represent the provided graph image. Workers were given up to six attempts to get the graph correct before the next graph to be drawn was automatically shown.

In theory, a worker's graph is correct if it is isomorphic to the provided graph. The best known algorithm for computing whether two general graphs are isomorphic is expensive and runs in $2^{O(\sqrt{n \log n})}$ time where n is the number of vertices in each graph [BKL83]. We did not implement a true isomorphism checker for the Mechanical Turk experiment, but rather we used several heuristics to determine with high confidence whether or not the user's graph was identical to the provided graph image. These heuristics consisted of determining if the user's graph was valid, if the graph had the correct number of each component type, and if special orderings of the vertex and edge labels in the user's graph matched the orderings produced for the labels in the provided graph. Each of these heuristics is described in more detail below.

To pass the first isomorphism heuristic, a user's graph must be valid, which requires both endpoints of all edges to be connected to vertices and all arrows to be

attached to some edge. The second heuristic checks whether the numbers of vertices, arrows, edges, and self-loops are the same between the user's graph and the provided graph. Next, in the final heuristic, we create seven arrays for both the user's graph and the provided graph that were populated with vertex and edge labels:

- vertex_undirected: the i-th entry in this array is an array consisting of the edge labels of any undirected edges incident to the i-th vertex
- vertex_incoming: the i-th entry in this array is an array consisting of the edge labels of any incoming edges incident to the i-th vertex
- vertex_outgoing: the i-th entry in this array is an array consisting of the edge labels of any outgoing edges incident to the i-th vertex
- vertex_bidirected: the i-th entry in this array is an array consisting of the edge labels of any bidirected edges incident to the i-th vertex
- edge_undirected: if the i-th edge is undirected, the i-th entry in this array is an array consisting of the labels of the two vertices attached to either end of the i-th edge; otherwise the i-th entry is an empty array
- edge_directed: if the i-th edge is directed, the i-th entry in this array is an array consisting of the labels of the two vertices attached to either end of the i-th edge; otherwise the i-th entry is an empty array
- edge_bidirected: if the i-th edge is bidirected, the i-th entry in this array is an array consisting of the labels of the two vertices attached to either end of the i-th edge; otherwise the i-th entry is an empty array

All of these arrays were then sorted with JavaScript's default sorting function, which converts array elements to strings and orders them according to their Unicode encodings. This sorting procedure removes any variation caused by considering the vertices and edges in different orders when forming the arrays. To pass the third and final isomorphism heuristic, the sorted arrays in each of the seven pairs must be

identical (one array in each pair is constructed from the user's graph and the other is constructed from the provided graph). If all three heuristics pass, we assume that the user has drawn the graph correctly and the screen advances to the next graph.

Several statistics were collected from the workers' Mechanical Turk submissions. To understand how often the recognition process was being triggered, we tracked the number of new components added to the locked in graph following each recognition process. For each graph, we also recorded the number of components erased, the number of labels erased, the number of submissions (with a maximum of six possible due to the attempt limit), and whether or not the worker was ultimately able to correctly replicate the graph. We encountered some minor issues with browser incompatibility in our experiment, so the results below were generated from the submissions of the first ten workers who were able to draw on a fully functioning sketching interface.

Across all 150 graphs for which we collected data (10 workers each drawing the same 15 graphs), 148 graphs were ultimately drawn correctly after a maximum of six attempts. An average of only 1.28 attempts were needed to submit a correct graph. This indicates that the computer rendered presentation of components and their connections was very successful in communicating the system's interpretation of a user's sketch to the user. Rather than relying on the submit button to check for correctness, workers were able to correct errors by repeatedly examining the computer rendered graph. This is a very important result for our online education use case, for students must be able to know exactly what graph they are submitting if evaluation of their answers is to accurately measure concept knowledge. In the two cases where workers were unable to correctly draw the graph after six attempts, the final attempts were incorrect due to a small, unintentional edge obscured by another edge and an arrow attached to the wrong edge endpoint.

On average, 1.17 components were added to the computer rendered graph following a run of the recognition algorithm. The fifteen graphs contained anywhere between 10 and 24 components with an average of 15.3 components, so the recognition algorithm, which was triggered whenever 0.5 seconds had elapsed since the last pen stroke, ran more than 10 times for nearly all graphs. The number of components

added per recognition was significantly lower than we expected. In fact, in Chapter 5, we optimized the parameters in the domain interpretation step to recognize five additional components at a time. This low result is not necessarily problematic, but it does indicate that the workers constructed the graphs very deliberately, pausing often to see how the system would interpret their strokes. Perhaps with additional experience using the realtime recognition system, workers would become more comfortable drawing larger portions of the graphs between each recognition. It is worth noting that, as anticipated, edge-arrow pairs were frequently added simultaneously to directed graphs.

The final result from our second Mechanical Turk experiment quantifies eraser usage. Across all 15 graphs and all 10 workers, on average, the number of computer rendered components erased while drawing a graph was equal to 21% of the total number of components in that graph. This figure ranged widely when computed for individual workers and for individual graphs. One worker erased an average of 9% components across all graphs while another worker erased an average of 41%. When drawing one graph (Graph 11 in Appendix B) workers on average erased only 1% of components, but on another graph (Graph 10 in Appendix B) workers erased 86% of components. These wide ranges reflect the variety of drawing techniques, input hardware (most importantly stylus type), and graph complexity represented in our experiment.

Chapter 7

Intermediate Feedback

The student interface described in Chapter 6 continually updates a graph representation of the canvas area, providing course developers with the opportunity to offer intermediate feedback to students each time the graph recognition process is triggered. Many currently popular online education answer formats such as text boxes and multiple choice do not afford this ability, for these answer types capture only the students' final answers and not their process of arriving at a solution. Intermediate feedback is likely not appropriate for problems used to evaluate a student's understanding, but we imagine it could provide useful guidance to students as they work through practice problems after being introduced to a new concept. Note also that intermediate feedback need not preclude the existence of a submit button, which students should still use to determine if their final answer is correct.

While the type of intermediate feedback to provide is best determined by the course instructor on a problem-by-problem basis, to illustrate our vision, we have included four examples below of simple graph problems that incorporate intermediate feedback. Our goal in each of the example problems is to display cautionary messages when students have made mistakes and to display positive encouragement when key aspects of the questions have been correctly addressed. These messages are updated each time the recognition process is triggered so that they always reflect the current state of the computer rendered graph.

In addition to showing how intermediate feedback might be implemented, the

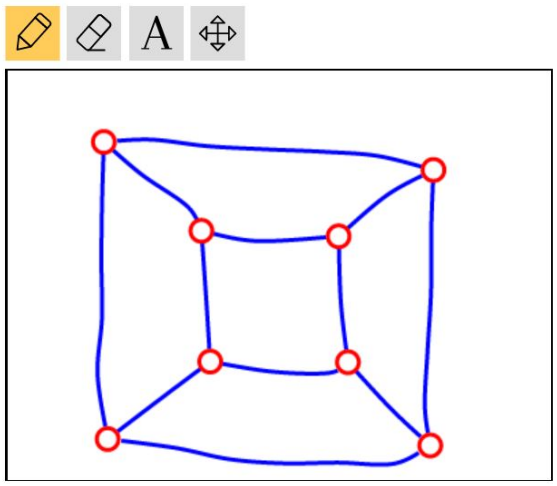


Figure 7-1: This graph is a potential solution to the planar graph example problem.

examples demonstrate the usefulness of a simple graph problem type, for each of the problems represents a simple graph concept from the standard undergraduate computer science curriculum that is most naturally tested by asking students to sketch a graph. The examples cover planar graphs (Section 7.1), binary search trees (Section 7.2), depth first search (Section 7.3), and partially ordered sets (Section 7.4).

7.1 Example 1: Planar Graphs

This first example tests the concept of planar graphs, which are undirected graphs that can be drawn in a plane with no edges intersecting.

Problem: Draw a planar embedding of a cube.

Solution: A correct solution can be seen in Figure 7-1.

Intermediate Feedback: Whenever a user's sketch contains intersecting edges, violating the properties of planar embeddings, the message "Remember that no edges can intersect in a planar graph" is displayed with a red background to the right of the canvas area as shown in Figure 7-2.

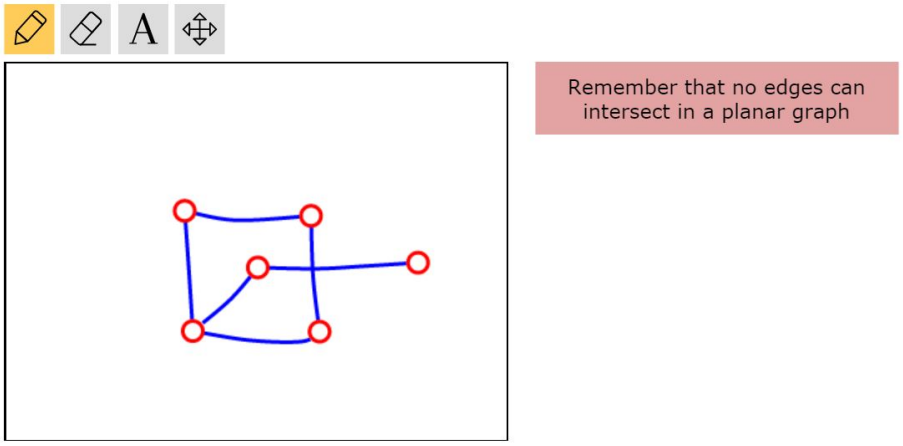


Figure 7-2: Drawing the graph on the left produces the intermediate feedback on the right for the planar graph example problem.

7.2 Example 2: Binary Search Trees

Binary Search Trees (BSTs) are a popular method for storing sorted information. When a node is added to a BST, it is assigned some value, and nodes can connect to at most one left child node and one right child node. For each node n, the BST invariant requires the values of the nodes in n's left subtree to be smaller than n's value and the values of the nodes in n's right subtree to be larger.

Problem: Draw the BST that would be formed by inserting the following numbers in order: $[7, 10, 4, 2, 8, 9, 5, 12]$.

Solution: A correct solution can be seen in Figure 7-3.

Intermediate Feedback: Three types of warnings are supported for this example. First, if the topmost node is labeled something other than 7, the message "Remember that the first number inserted into the BST should become the root" is shown. BSTs allow a node to have a maximum of one left child and one right child, so we display "Be careful! Each node should have at most one left (right) child" when this rule is violated. Finally, if nodes are labeled in a way that contradicts the BST invariant described above, we add "Watch out! The value of a node's left (right) child must be less (greater) than the value of the node itself" to the list of warnings. Figure 7-4 displays a state of the canvas that causes all three warning types to appear simultaneously.

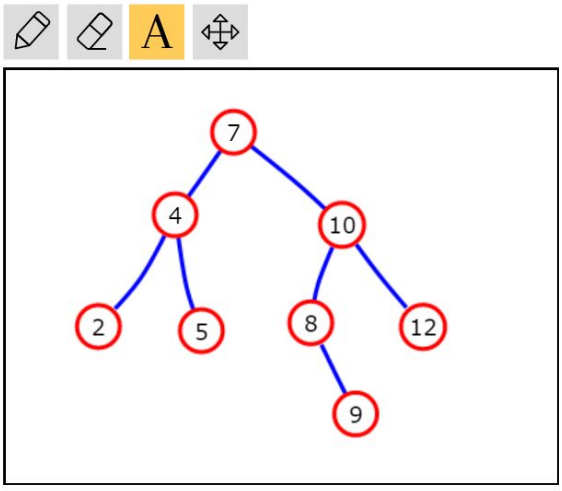


Figure 7-3: This graph is a potential solution to the binary search tree example problem.

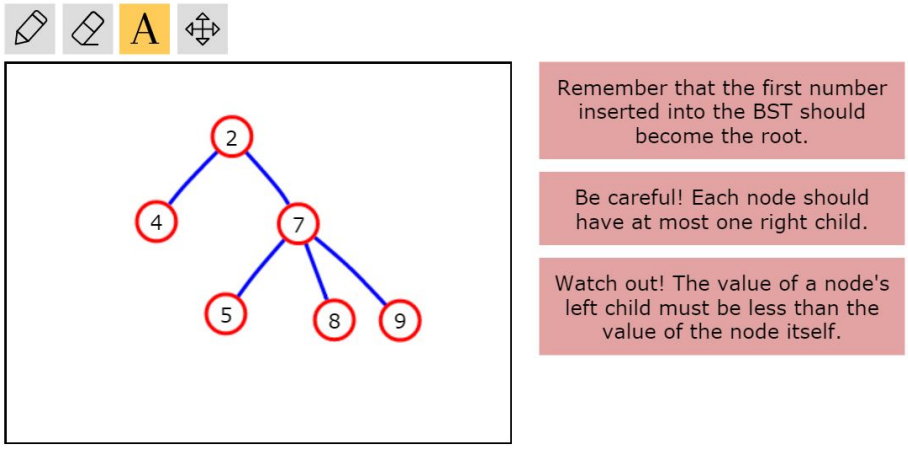


Figure 7-4: Drawing the graph on the left produces the intermediate feedback on the right for the binary search tree example problem.

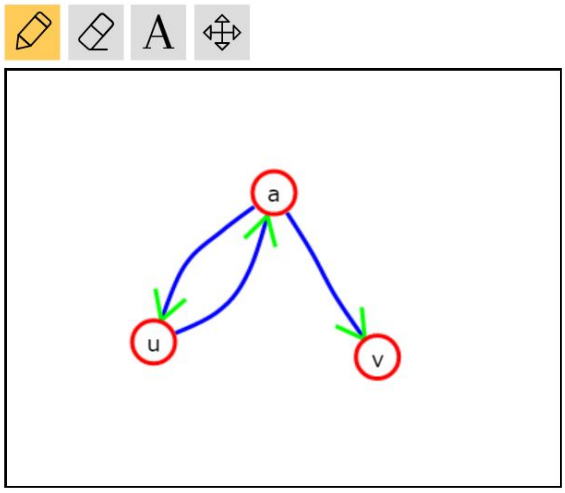


Figure 7-5: This graph is a potential solution to the depth first search example problem.

7.3 Example 3: Depth First Search

Depth First Search is a common algorithm for exploring an undirected or directed graph. The problem below is from the popular *Introduction to Algorithms* by Cormen et al, who use the notation $n.d$ to denote the discovery time of a node n and $n.f$ to denote the finishing time of n when the depth first search algorithm is run [CLRS09].

Problem: Give a counterexample to the conjecture that if a directed graph G contains a path from u to v, then any depth first search must result in $v.d \leq u.f$. Assume that depth first search begins at the topmost node in your sketch and that child nodes are considered left to right.[1]

Solution: A correct solution can be seen in Figure 7-5.

Intermediate Feedback: This example has two key conditions the sketch must meet to be a valid counterexample. If there is a path from a node u to a node v, we display the message "Great. There's a path from u to v" with a green background to the right of the canvas. If the second condition, namely that $v.d > u.f$, is met, we add the encouraging message "Good. Currently $v.d > u.f$". When there are labeled vertices u and v but v would be discovered before u is finished, we instead display the warning "Whoops. Your counterexample should have $v.d > u.f$". Two states of the canvas

[1] CLRS Problem 22.3-9

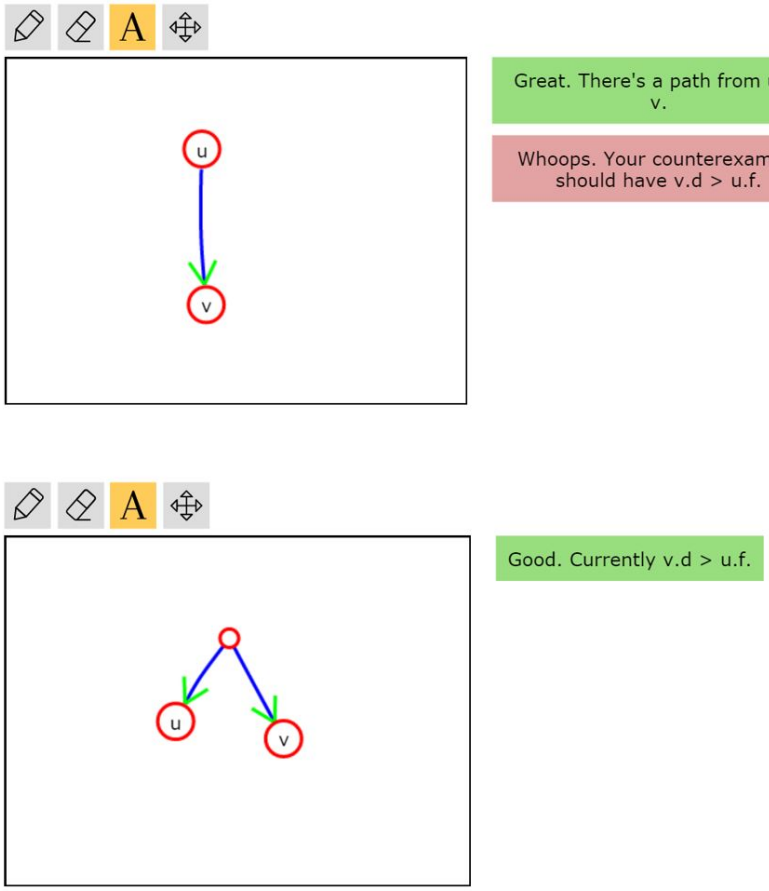


Figure 7-6: In each of these images, drawing the graph on the left produces the intermediate feedback on the right for the depth first search example problem.

and their corresponding messages are shown in Figure 7-6.

7.4 Example 4: Partially Ordered Sets

Our final example involves directed graph representations of partially ordered sets. A partial order is a reflexive, antisymmetric, and transitive binary relation, and it gives orderings to some pairs of elements in a set.

Problem: Draw the directed graph representing the divisibility relation on the set $\{2, 4, 5, 10, 12\}$. Hint: The divisibility relation is defined as $R : A \to B$ where aRb if and only if $a \mid b$.

Solution: A correct solution can be seen in Figure 7-7.

Intermediate Feedback: A tricky aspect of this problem involves recognizing that

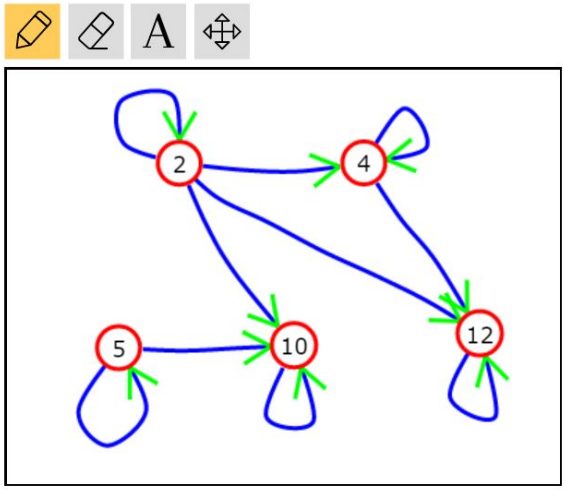


Figure 7-7: This graph is a potential solution to the partially ordered sets example problem.

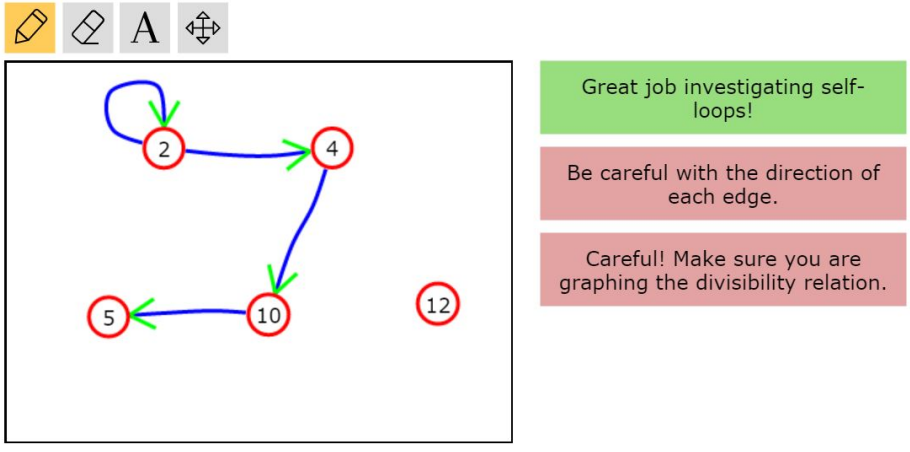


Figure 7-8: Drawing the graph on the left produces the intermediate feedback on the right for the partially ordered sets example problem.

each of the numbers in the set is divisible by itself and that each vertex should therefore have a self-loop. If a user draws a self-loop on a labeled vertex, we post the positive message "Great job investigating self-loops!". A common mistake is to draw arrows in the incorrect direction; if an arrow points from a given node to a node with a smaller value, we display "Be careful with the direction of each edge". Finally, to help users correct what is likely a careless mistake, if an edge is drawn between two nodes that should not be connected, the warning "Careful! Make sure you are graphing the divisibility relation" is added. Figure 7-8 displays a canvas state that causes all three messages to appear at the same time.

Chapter 8

Conclusion and Future Work

We have presented a simple graph problem type intended for online education applications that recognizes students' sketches drawn with a stylus. The growing presence of online education platforms coupled with the limited array of currently supported machine-gradable answer formats were the initial motivators for this work. In an effort to make our work meaningful for educators and to establish a reasonable problem scope, we decided to focus on simple graphs, which appear across the standard undergraduate computer science curriculum and are constructed from a small, well-defined set of components. These components are vertices, arrows, and edges, which users can draw freehand, as well as vertex and edge labels, which users must create using a keyboard.

Throughout the existing literature on hand-drawn sketch recognition, the assumptions made regarding drawing behavior strongly affect the development of recognition algorithms and their subsequent performance. To preserve a natural drawing experience, we made relatively few assumptions, the most significant of which requires users to finish drawing a component before they begin to draw another. Apart from two other adjustable time-based constraints, users are theoretically free to draw components in any manner they wish (i.e. any number or ordering of strokes is acceptable) as long as the sketched component represents one of the component types within reason.

The process of converting users' strokes into a graph representation that communi-

cates both the graph's constituent components and connections between components was divided into three steps: segmentation, classification, and domain interpretation. These steps occur in a sequential manner, and error recovery was built into the algorithm such that some miscues made by early parts of the recognition process can be recovered by latter parts. To find the optimal settings for the parameters in each of the steps, we used training data created by post-processing sketches of graphs submitted by ten individuals through a Mechanical Turk experiment. Each of the recognition algorithm steps is recapped briefly below.

Given a sequence of strokes, the segmentation step produces a sequence of smaller segments such that components consist only of a whole number of segments. This step is needed because the subsequent classification step is only capable of identifying single complete components and because we don't make any assumptions regarding either the number of strokes used to construct components or whether the stylus is lifted up during transitions between components. Segmentation points are located by examining changes in stroke curvature, and three parameters involved in the curvature calculations and threshold placement were trained to produce the final segmenter.

In the second step of the recognition algorithm, sequences of consecutive segments are assigned probabilities indicating the likelihood that the segments produce a specific component type. We used the machine learning logistic regression technique to create four classifiers that identify vertices, arrows, standard edges, and self-loops by combining the calculations of five geometric features involving convex hulls and alpha shapes. These features capture the extent to which a set of points forms a circular shape, a triangular shape, and a smooth curve, and they also determine the rough size of the space occupied by these points and the number of clearly distinct regions. Gradient descent was used to find the the optimal weight parameter vector that multiplies the feature vector in each of the classifiers.

As the final step in the recognition algorithm, the domain interpretation step attempts to output a best interpretation of the user's sketch by combining the scores from the classification step with knowledge specific to the graph domain. The components in a graph sketch are not independent of each other; rather, the components

connect in specific ways, with vertices and arrows both connecting to edge endpoints. This step generates partitions of the segments, sums together classification probabilities, establishes connections between hypothetical components, and then adjusts the probability sums based on the number of components, the number of missing connections, and the number of established connections. Three parameters that act as weights during the adjustment of sums process were trained to create the optimal domain interpretation step.

To demonstrate how students might interact with our graph recognition algorithm, we created a user interface that incorporated realtime recognition functionality. A toolbar above the sketching canvas allows users to switch between four modes for drawing, erasing, adding labels, and moving labels. Recognition of the users' strokes is triggered each time the user ceases drawing for 0.5 seconds, and hand-drawn strokes are replaced by a machine rendered representation of the resulting best graph interpretation in order to communicate the system's current graph model to the user. We demonstrated the usability of our interface through a second Mechanical Turk experiment.

Since we hope for our simple graph problem type to be incorporated into online education courses, we also composed four examples suggesting how instructors might capitalize on our algorithm's ability to perform realtime recognition by providing intermediate feedback to students. This feedback is customized to each problem and can either attempt to help students recognize and correct an error or provide students with encouragement if an aspect of the problem has been completed successfully. Our examples, which covered planar graphs, binary search trees, depth first search, and partially ordered sets, also served to illustrate the wide range of topics involving simple graphs.

Certainly, many aspects of the work presented here could be further explored. Starting with the graph recognition algorithm itself, it is worthwhile to examine whether or not online (realtime) training of the algorithm parameters could be used to refine the initial set of trained parameters according to each user's specific drawing style. While we showed that it was reasonable to train parameters on one set of

individuals and then expect these parameters to produce accurate recognition results for a second set of individuals, the costs for our test sets were consistently lower when the individual from the test set was represented in the training data.

While online training approaches for the segmentation and domain interpretation steps may not be immediately apparent, the batch gradient descent method we used to train our logistic regression classifiers does have a convenient online training analog in stochastic gradient descent [Ng12]. Problems that are correctly answered by only a single graph can be used to produce new training data for the classifiers, for each time a student submits a correct answer, additional positive examples of the graph's constituent components are available and can be used to update the classifier weights.

Our algorithm has the potential to recover from several types of miscues, such as false positive segmentation points and low classifier probabilities for the true component type, but it cannot currently recover from failing to detect a true segmentation point. Perhaps this deficiency could be addressed by exploring a recognition algorithm architecture different from our linear three step process. For example, if all classifier probabilities computed for segment sequences containing a specific segment were low, it is possible that a true segmentation point in this segment went undetected. The error could potentially be corrected by passing this segment back through a segmenter with a lower detection threshold and then recalculating the classifier probabilities for the applicable segment sequences.

An obvious potential point of expansion is to modify the recognition algorithm to correctly interpret a larger set of components and to also recognize handwritten labels. Dotted edges can be found in some more advanced graphs, and vertices can also sometimes be more complicated than the circular vertices we currently support (i.e. Van Emde Boas trees are frequently drawn with sequences of square cells acting as vertices). Provided it was accomplished with high accuracy, support for handwriting recognition of labels would help significantly in making the drawing experience more natural and removing interruptions caused by alternating between the drawing and labeling modes. Creating a well-defined domain of supported labels would be an important step in adding handwritten label recognition functionality.

It may also be worthwhile to consider adding additional assumptions regarding drawing technique if the simple graph problem type is to be used outside a research environment and brought into online courses. In an effort to preserve a natural drawing experience, we imposed a minimal number of assumptions. In particular, there was no assumption regarding the number or order of strokes used to construct each component. We did observe a wide range of approaches to drawing vertices, arrows, and edges in the sketches collected from our first Mechanical Turk experiment, but it is worth noting that it is possible to draw each of the graph components with a single stroke (as opposed to an equal sign, for example, which requires two strokes). If users were required to draw each component with a single stroke, recognition rates could likely be improved significantly. There is of course a trade-off between drawing freedom and recognition accuracy that affects the student experience.

Additional features could be added to the user interface to support responses for a wider range of simple graph problems. For example, allowing individual components to be highlighted would enable students to identify components with certain properties or to show a path that an algorithm would take through a graph. Some problems are best answered with a time series of simple graphs. Consider, for example, a problem that asks a student to perform Dijsktra's algorithm on a simple graph, requiring vertex labels to each be updated multiple times. Adding a feature enabling frames of the canvas to be captured and stored in a frame roll would allow students to submit a time sequence of graphs as their solution. As a third example of a possible user interface extension, a course developer might want to initialize the sketching area to display a graph that students could then modify.

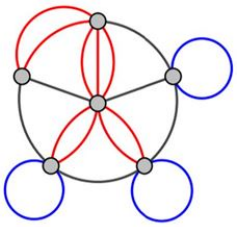
Lastly, the concept of providing intermediate feedback to students in online education courses could be further explored. In our examples, we displayed a mix of positive and negative messages on a side panel that reflected the current state of the computer rendered graph. It would be interesting to consider displaying negative messages with a time delay in order to give students the opportunity to recognize and correct mistakes before having these mistakes pointed out. While we depended on text to provide feedback, other indicators such as colors or symbols could display

information in a less obtrusive manner. Finally, the notion of feedback could easily expand to include time-released hints aimed at guiding students towards a solution.

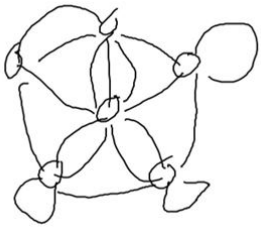
Appendix A

Mechanical Turk Experiment 1

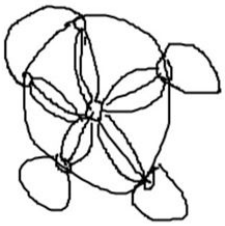
The first Mechanical Turk experiment was designed to collect natural sketches of graphs in order to produce training data for each of the parts of the recognition algorithm. Sketches of the same 15 graphs were collected from 10 individuals, and workers were directed to copy the graphs as accurately as possible while ignoring any labels. Workers did have the ability to clear graph canvases if needed. The 15 graphs from the experiment are displayed below along with two randomly chosen individuals' sketches of each graph.



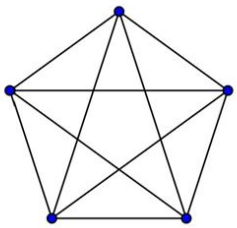
Graph 1



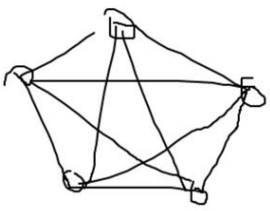
Individual 6



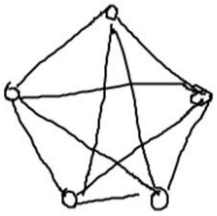
Individual 10



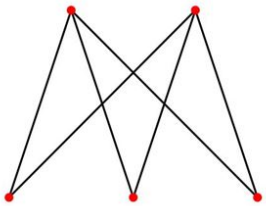
Graph 2



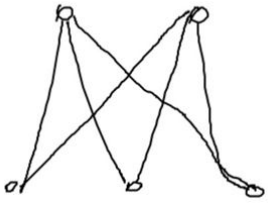
Individual 3



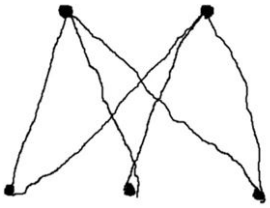
Individual 8



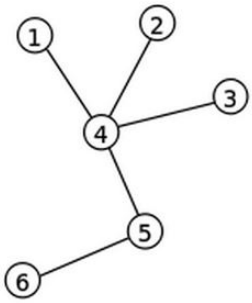
Graph 3



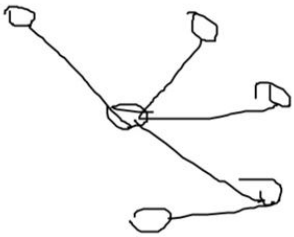
Individual 8



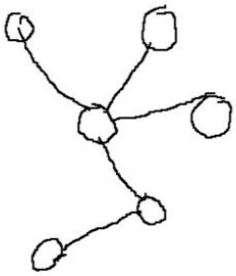
Individual 5



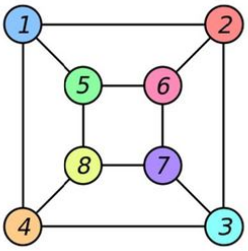
Graph 4



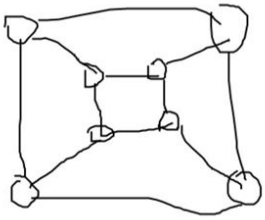
Individual 2



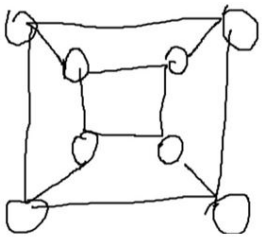
Individual 5



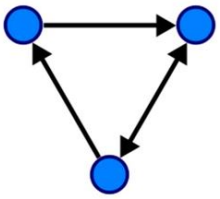
Graph 5



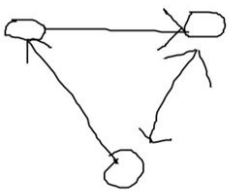
Individual 7



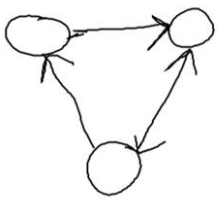
Individual 1



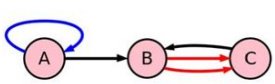
Graph 6



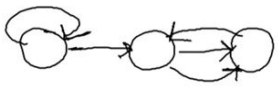
Individual 2



Individual 4

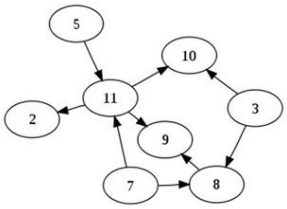


Graph 7

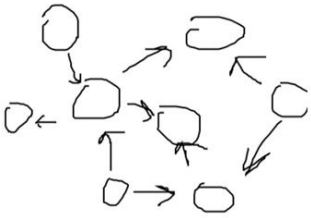


Individual 4

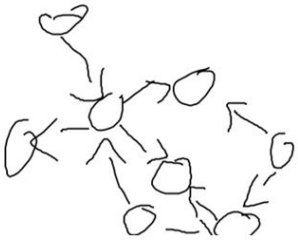
Individual 6



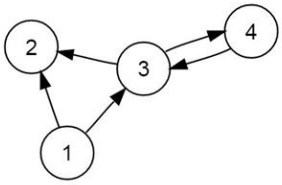
Graph 8



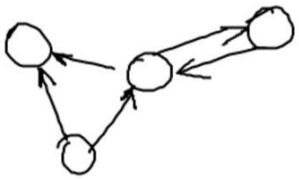
Individual 7



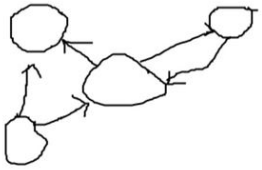
Individual 6



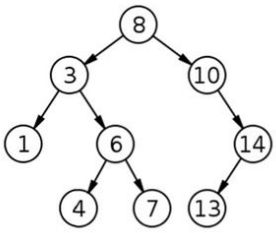
Graph 9



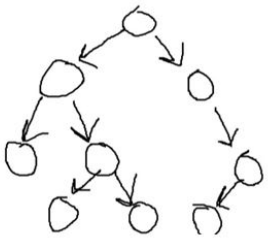
Individual 8



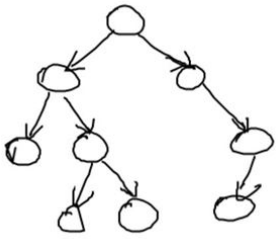
Individual 9



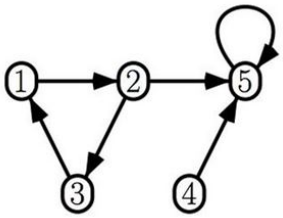
Graph 10



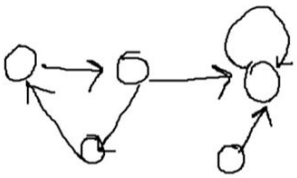
Individual 1



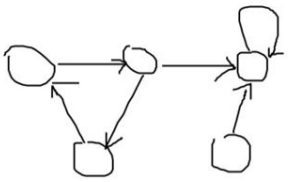
Individual 8



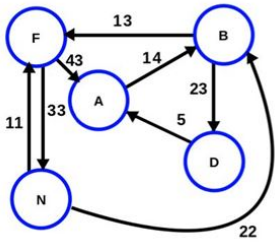
Graph 11



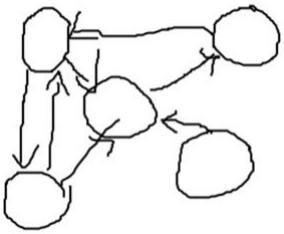
Individual 1



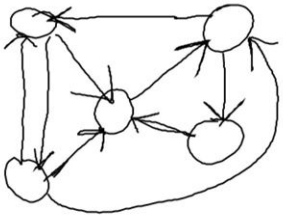
Individual 3



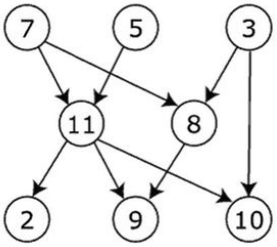
Graph 12



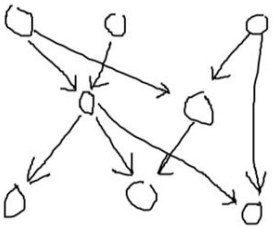
Individual 9



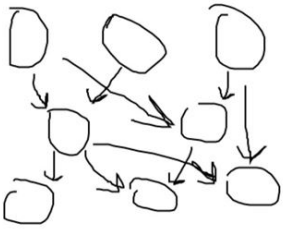
Individual 4



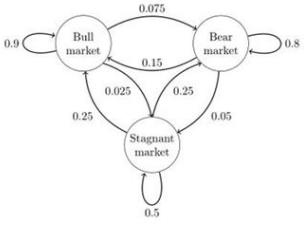
Graph 13



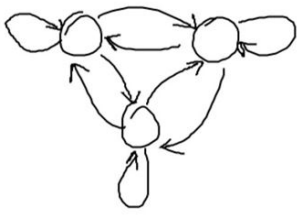
Individual 10



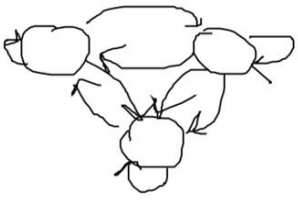
Individual 7



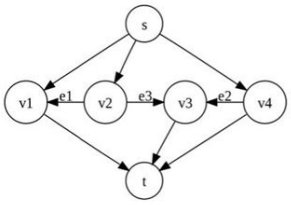
Graph 14



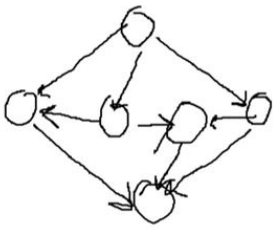
Individual 8



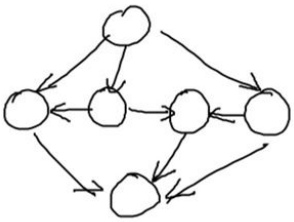
Individual 2



Graph 15



Individual 1



Individual 8

Graph 1: http://upload.wikimedia.org/wikipedia/commons/thumb/c/c9/Multi-pseudograph.svg/330px-Multi-pseudograph.svg.png

Graph 2: http://upload.wikimedia.org/wikipedia/commons/thumb/c/cf/Complete_graph_K5.svg/918px-Complete_graph_K5.svg.png

Graph 3: http://upload.wikimedia.org/wikipedia/commons/thumb/e/e2/Complete_bipartite_graph_K3%2C2.svg/1187px-Complete_bipartite_graph_K3%2C2.svg.png

Graph 4: http://upload.wikimedia.org/wikipedia/commons/thumb/2/24/Tree_graph.svg/162px-Tree_graph.svg.png

Graph 5: http://upload.wikimedia.org/wikipedia/commons/thumb/8/84/Graph_isomorphism_b.svg/315px-Graph_isomorphism_b.svg.png

Graph 6: http://upload.wikimedia.org/wikipedia/commons/thumb/a/a2/Directed.svg/250px-Directed.svg.png

Graph 7: http://upload.wikimedia.org/wikipedia/commons/thumb/d/da/3n_multigraph.svg/557px-3n_multigraph.svg.png

Graph 8: http://upload.wikimedia.org/wikipedia/commons/thumb/3/39/Directed_acyclic_graph_3.svg/356px-Directed_acyclic_graph_3.svg.png

Graph 9: http://upload.wikimedia.org/wikipedia/commons/5/51/Directed_graph.svg

Graph 10: http://upload.wikimedia.org/wikipedia/commons/thumb/d/da/Binary_search_tree.svg/300px-Binary_search_tree.svg.png

Graph 11: http://upload.wikimedia.org/wikipedia/commons/0/0c/Small_directed_graph.JPG

Graph 12: http://upload.wikimedia.org/wikipedia/commons/thumb/a/a0/CPT-Graphs-directed-weighted-ex2.svg/548px-CPT-Graphs-directed-weighted-ex2.svg.png

Graph 13: http://upload.wikimedia.org/wikipedia/commons/0/08/Directed_acyclic_graph.png

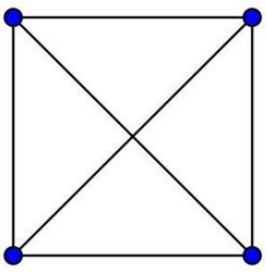
Graph 14: http://upload.wikimedia.org/wikipedia/commons/thumb/9/95/Finance_Markov_chain_example_state_space.svg/400px-Finance_Markov_chain_example_state_space.svg.png

Graph 15: http://upload.wikimedia.org/wikipedia/commons/thumb/6/6e/Ford-Fulkerson_forever.svg/533px-Ford-Fulkerson_forever.svg.png

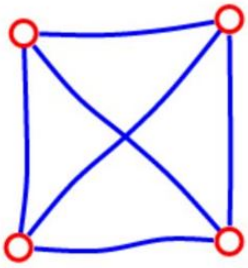
Appendix B

Mechanical Turk Experiment 2

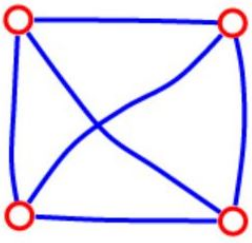
To evaluate the usability of the user interface described in Chapter 6, we ran a second Mechanical Turk experiment. Workers were presented with a sequence of 15 graphs and were asked to replicate these graphs on the provided canvas, which periodically replaced their strokes with machine rendered interpretations. The graphs were displayed one at a time, and workers were given six attempts to correctly draw each. As the workers interacted with the interface, for each graph, we collected data on the number of new components added each time the recognition process was triggered, the number of components erased, the number of labels erased, the number of attempts, and whether or not the worker ultimately drew the graph correctly. We also recorded the machine representation of each graph submitted. Two randomly chosen individuals' final submissions for each graph are displayed below along with the original graph images.



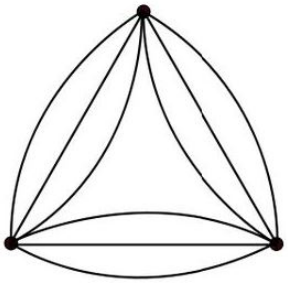
Graph 1



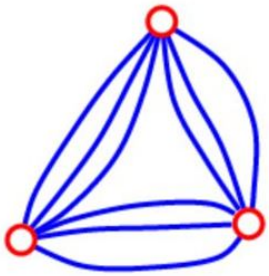
Individual 5 (Attempt 3)



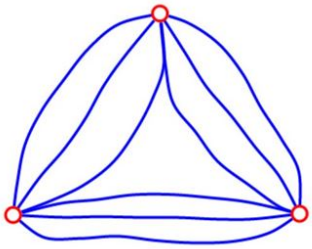
Individual 2 (Attempt 1)



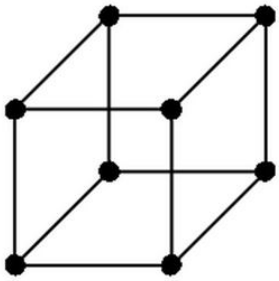
Graph 2



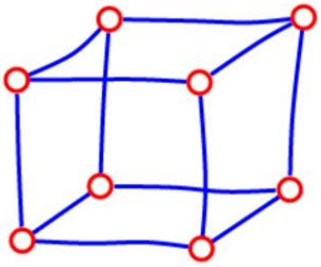
Individual 1 (Attempt 1)



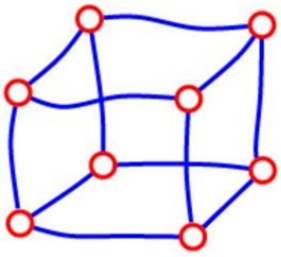
Individual 10 (Attempt 1)



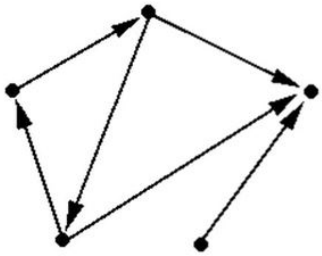
Graph 3



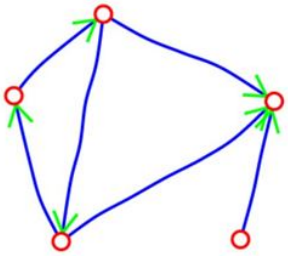
Individual 3 (Attempt 1)



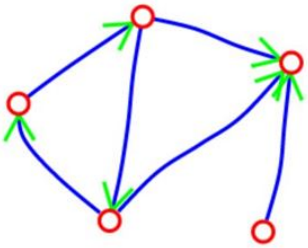
Individual 8 (Attempt 1)



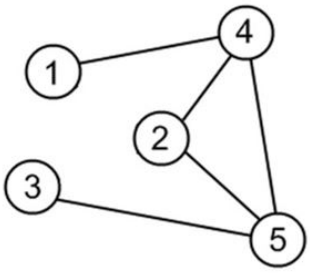
Graph 4



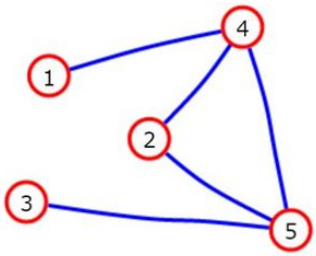
Individual 10 (Attempt 2)



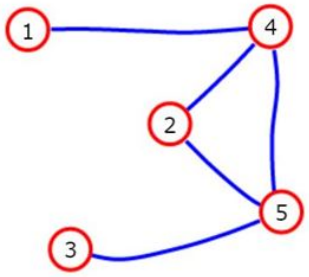
Individual 6 (Attempt 1)



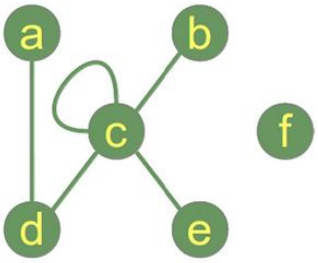
Graph 5



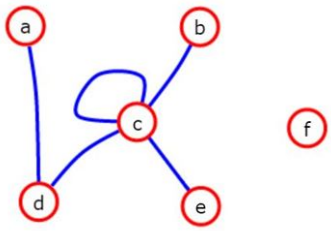
Individual 5 (Attempt 1)



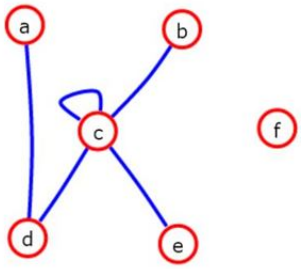
Individual 1 (Attempt 1)



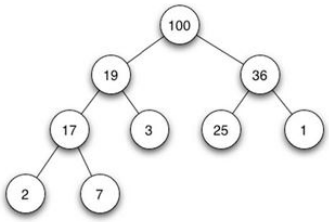
Graph 6



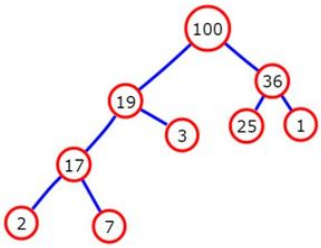
Individual 3 (Attempt 1)



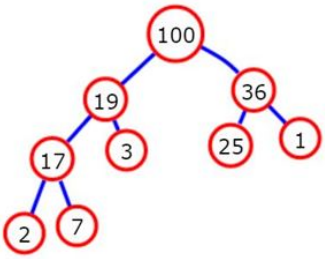
Individual 5 (Attempt 1)



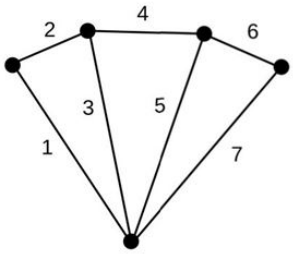
Graph 7



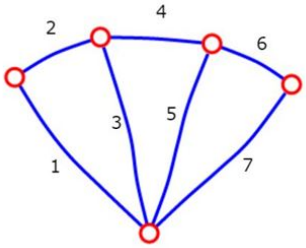
Individual 4 (Attempt 1)



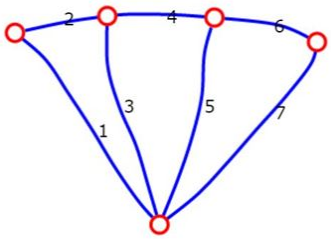
Individual 6 (Attempt 1)



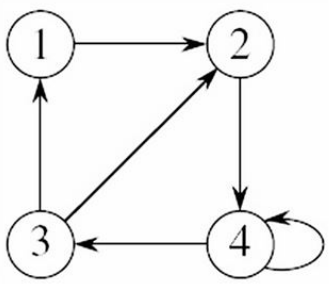
Graph 8



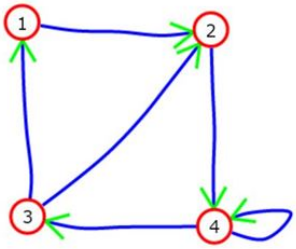
Individual 10 (Attempt 1)



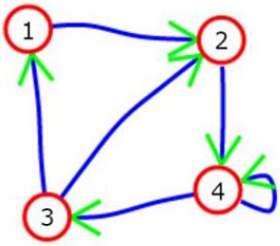
Individual 9 (Attempt 1)



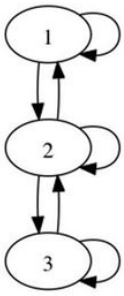
Graph 9



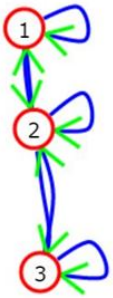
Individual 4 (Attempt 1)



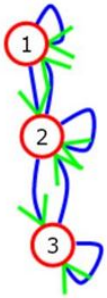
Individual 1 (Attempt 1)



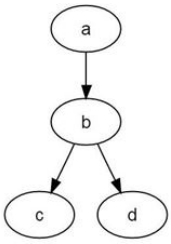
Graph 10



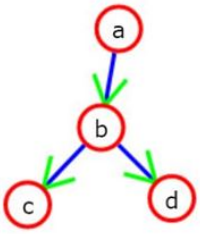
Individual 6 (Attempt 1)



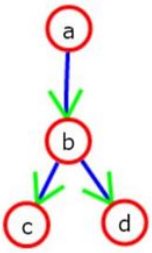
Individual 1 (Attempt 1)



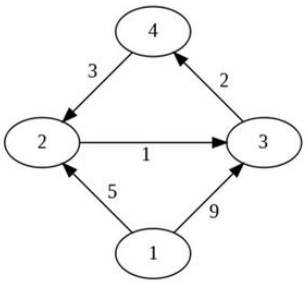
Graph 11



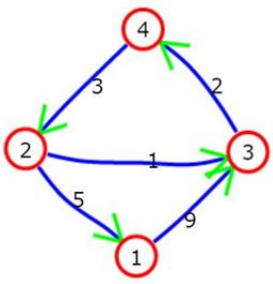
Individual 8 (Attempt 1)



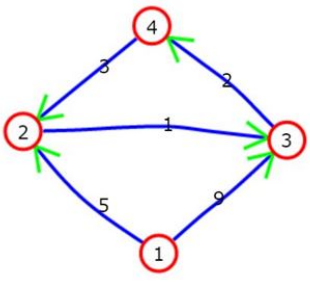
Individual 2 (Attempt 1)



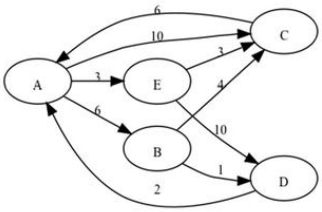
Graph 12



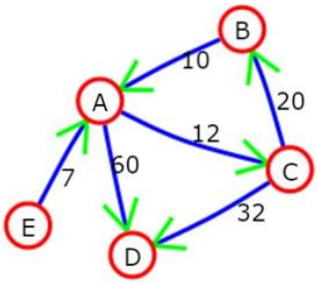
Individual 4 (Attempt 6)



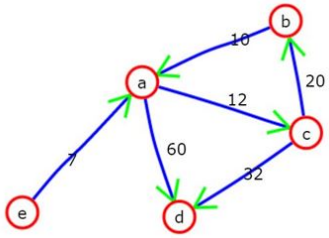
Individual 7 (Attempt 2)



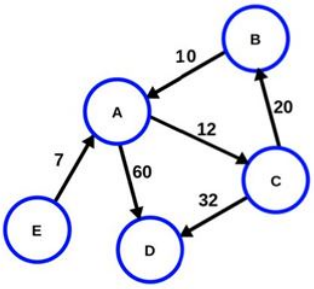
Graph 13



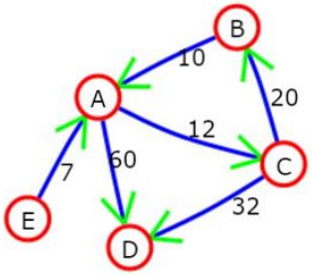
Individual 3 (Attempt 3)



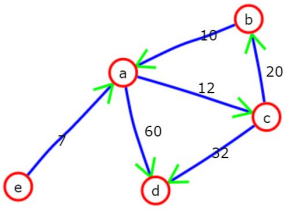
Individual 8 (Attempt 2)



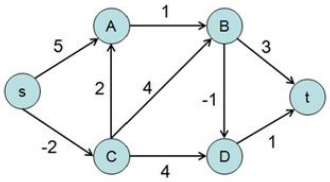
Graph 14



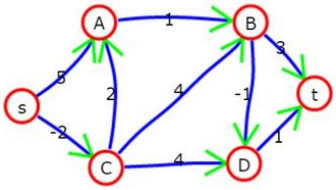
Individual 5 (Attempt 1)



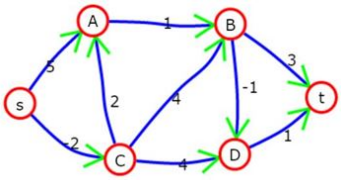
Individual 7 (Attempt 3)



Graph 15



Individual 1 (Attempt 1)



Individual 9 (Attempt 1)

Graph 1: http://upload.wikimedia.org/wikipedia/commons/thumb/5/59/Complete_graph_K4.svg/900px-Complete_graph_K4.svg.png

Graph 2: http://upload.wikimedia.org/wikipedia/commons/thumb/7/74/Shannon_multigraph_6.svg/400px-Shannon_multigraph_6.svg.png

Graph 3: https://andrewharvey4.files.wordpress.com/2008/11/lines_3-cube.png

Graph 4: http://www.colorado.edu/education/DMP/directed_graph.gif

Graph 5: http://www.algolist.net/img/graphs/DFS/DFS-example-1.png

Graph 6: http://www.python-course.eu/images/simple_graph_with_loop.png

Graph 7: http://sir.unl.edu/portal/bios/Binary-Heap-1b08b627.png

Graph 8: https://upload.wikimedia.org/wikipedia/en/thumb/5/52/Ladder_graph.svg/888px-Ladder_graph.svg.png

Graph 9: http://homepages.ius.edu/rwisman/C455/html/notes/AppendixB4/B4-1.gif

Graph 10: http://www.cs.colorado.edu/~srirams/courses/csci2824-spr14/l20f2.png

Graph 11: http://upload.wikimedia.org/wikipedia/commons/e/ec/DotLanguageDirected.svg

Graph 12: http://www.cs.ucsb.edu/~gilbert/cs140/old/cs140Win2011/cudaproject/apsp-graph.png

Graph 13: http://matteo.vaccari.name/blog/wp-content/uploads/2009/04/graph1.png

Graph 14: http://upload.wikimedia.org/wikipedia/commons/thumb/b/bc/CPT-Graphs-directed-weighted-ex1.svg/542px-CPT-Graphs-directed-weighted-ex1.svg.png

Graph 15: https://www.cpp.edu/~ftang/courses/CS241/notes/images/graph/bellman2.gif

Bibliography

[And79] Alex M Andrew. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 9(5):216–219, 1979. 38

[BE13] Bruno Bienfait and Peter Ertl. JSME: A free molecule editor in JavaScript. *Journal of Cheminformatics*, 5:24, 2013. 11

[BKL83] László Babai, William M Kantor, and Eugene M Luks. Computational complexity and the classification of finite simple groups. In *Proceedings of the 24th IEEE Symposium on Foundations of Computer Science*, pages 162–171, 1983. 73

[Bow81] Adrian Bowyer. Computing dirichlet tessellations. *The Computer Journal*, 24(2):162–166, 1981. 39

[CLRS09] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2009. 81

[Coh12] Spencer Cohen. Smooth.js, 2012. https://github.com/osuushi/Smooth.js/. 59

[DBVKOS00] Mark De Berg, Marc Van Kreveld, Mark Overmars, and Otfried Cheong Schwarzkopf. *Computational Geometry*. Springer, 2000. 38, 39

[edX15] edX. Building and running an edX course, 2015. http://edx.readthedocs.org/projects/edx-partner-course-staff/en/latest/exercises_tools/index.html. 11

[EKS83] Herbert Edelsbrunner, David Kirkpatrick, and Raimund Seidel. On the shape of a set of points in the plane. *IEEE Transactions on Information Theory*, 29(4):551–559, 1983. 38

[EM94] Herbert Edelsbrunner and Ernst P Mücke. Three-dimensional alpha shapes. *ACM Transactions on Graphics*, 13(1):43–72, 1994. 40

[Fis00] Kaspar Fischer. Introduction to alpha shapes, 2000. http://www.cs.uu.nl/docs/vakken/ddm/texts/Delaunay/alphashapes.pdf. 39

[FPJ02] Manuel J Fonseca, César Pimentel, and Joaquim A Jorge. CALI: An online scribble recognizer for calligraphic interfaces. In *Proceedings of the AAAI Spring Symposium on Sketch Understanding*, pages 51–58, 2002. 17, 36, 37, 44

[GKSS05] Leslie Gennari, Levent Burak Kara, Thomas F Stahovich, and Kenji Shimada. Combining geometry and domain knowledge to interpret hand-drawn diagrams. *Computers and Graphics*, 29(4):547–562, 2005. 13, 17, 18, 26, 53

[Gra72] Ronald L Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1:132–132, 1972. 38

[Hof13] Chris Hoffman. Not all tablet stylus are equal: Capacitve, Wacom, and Bluetooth explained, 2013. http://www.howtogeek.com/177376/not-all-tablet-styluses-are-equal-capacitive-wacom-and-bluetooth-explained. 72

[KR13] Kimberle Koile and Andee Rubin. Machine interpretation of students' hand-drawn mathematical representations. In *Proceedings of the Workshop on the Impact of Pen and Touch Technology on Education*, 2013. 11

[LKS07] WeeSan Lee, Levent Burak Kara, and Thomas F Stahovich. An efficient graph-based recognizer for hand-drawn symbols. *Computers and Graphics*, 31(4):554–567, 2007. 36

[LTC00] Edward Lank, Jeb S Thorley, and Sean Jy-Shyang Chen. An interactive system for recognizing hand drawn UML diagrams. In *Proceedings of the Centre for Advanced Studies Conference*, page 7. IBM Press, 2000. 18

[MEFC09] Akshaya Kumar Mishra, Justin A Eichel, Paul W Fieguth, and David A Clausi. VizDraw: A platform to convert online hand-drawn graphics into computer graphics. In *Proceedings of the 6th International Conference on Image Analysis and Recognition*, pages 377–386. Springer, 2009. 13, 26, 36

[Ng11] Andrew Ng. Simplified cost function and gradient descent, 2011. https://class.coursera.org/ml-005/lecture/36. 48

[Ng12] Andrew Ng. CS229 lecture notes, 2012. http://cs229.stanford.edu/notes/cs229-notes1.pdf. 37, 48, 88

[OD07] Tom Y Ouyang and Randall Davis. Recognition of hand drawn chemical diagrams. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 846–851, 2007. 13, 18, 36, 53

[PS00] Réjean Plamondon and Sargur N Srihari. On-line and off-line handwriting recognition: A comprehensive survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(1):63–84, 2000. 13

[Wat81] David F Watson. Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes. *The Computer Journal*, 24(2):167–172, 1981. 39

[ZB12] Richard Zanibbi and Dorothea Blostein. Recognition and retrieval of mathematical expressions. *International Journal on Document Analysis and Recognition*, 15(4):331–357, 2012. 13