**Decision Trees**

**CMSC435**

**Final Report**

**Version 1.0**

**Authors:**

**Guanhua Liu**

**Lucas Cheng**

**Ameen Ali Ateeqi**

**Hugo Wang**

**Yonatan Zvi Hersh Margulies**

**David Eitan Kaplan**

**Gregory John Segala**

**Table of Contents**

1. **Documentation 2**
   1. Overview and Integration …………….….………………………………… 2
   2. XML format ……………………………………..…….……….………… 2
   3. Imaging Algorithm .….……………..….……………………….………… 3
   4. Grading Algorithm .…………………...……………….………………… 11
   5. Online Tree Sketching ..……….......……….…………..………………… X
   6. Website ………………...….…………………....…..……………………. X
2. **Acceptance Tests X**
   1. Website ..……..………………………………………………………….. X
   2. Grading ..……..………………………………………………………….. X
   3. Imaging……….………………………………………………………….. X
3. **Project Advertisement X**
4. **Cost Estimation vs. Actual X**
5. **Appendices X**
   1. Appendix 1 ………………………………………………………………. X

**1.1 Documentation - Overview and Integration**

This product can be split into 4 key components:

1. Website → Interface where the functionality is used and its data is stored
2. Online Tree Sketcher (OTS) → Functionality where students can create decision trees
3. Imaging → Functionality where students can upload images of decision trees
4. Grading → Algorithm used for students to receive feedback on their decision trees

All parts of the project communicate with a common language, XML. We use this language as our primary way to express decision trees since XML is a standard way to express trees and has convenient parsers in multiple languages.

The first step in the process is when the professor logs into the website. He creates a class and adds students to his class. He then proceeds to create an assignment, which consists of two parts: the prompt and the solution . The prompt is just plain text that describes the question. The second is the solution tree, which is created with the OTS. The OTS is implemented with p5, a GUI specialized javascript interface. Once created, students can now log in and complete this assignment. Assuming the professor gave them permission through the website, the students use the OTS to create the tree. They can also use the imaging functionality to upload their trees. This will be a jpeg or png picture of a color coded decision tree. Once the students submit their tree (either through the OTS or the imaging), the tree is converted into XML format and their tree will be graded against the professors tree using the grading algorithm. The grading algorithm compares the solution and the submitted tree XML files and returns the grade to be displayed on the website.

**1.2 Documentation - XML Format**

The different components of the product communicate with XML. The imaging and OTS output XML to the grading algorithm, which parses the XML into its own decision tree node form. The library that is used to parse this XML file on the backend is Ruby’s Nokogiri. Nokogiri takes in an XML document and returns an easy-to-use Nokogiri document. The XML format we used for communication between the different elements is outlined as follows:

* <chance n = "name" e = "edge from parent to this" p = "chance to this not if CHANCE node parent, null otherwise" />
* <decision n = "name" e = "edge from parent to this" p = "chance to this not if CHANCE node parent, null otherwise"/>
* <final e = "edge from parent to this" p = "chance to this not if CHANCE node parent, null otherwise" v = "value at the final node"/>

Besides this node format, the conventional XML structure applies. If a node is a child of another node, then the children are listed before the ending bracket: *node\_type/>*. For example, *Figure* *1.2* would have the following output:

<xml>

<decision e="null" n="Build" p="null">

<chance e="Now" n="Soil" p="null">

<final e="Bad " p="p" v="-$1000000" />

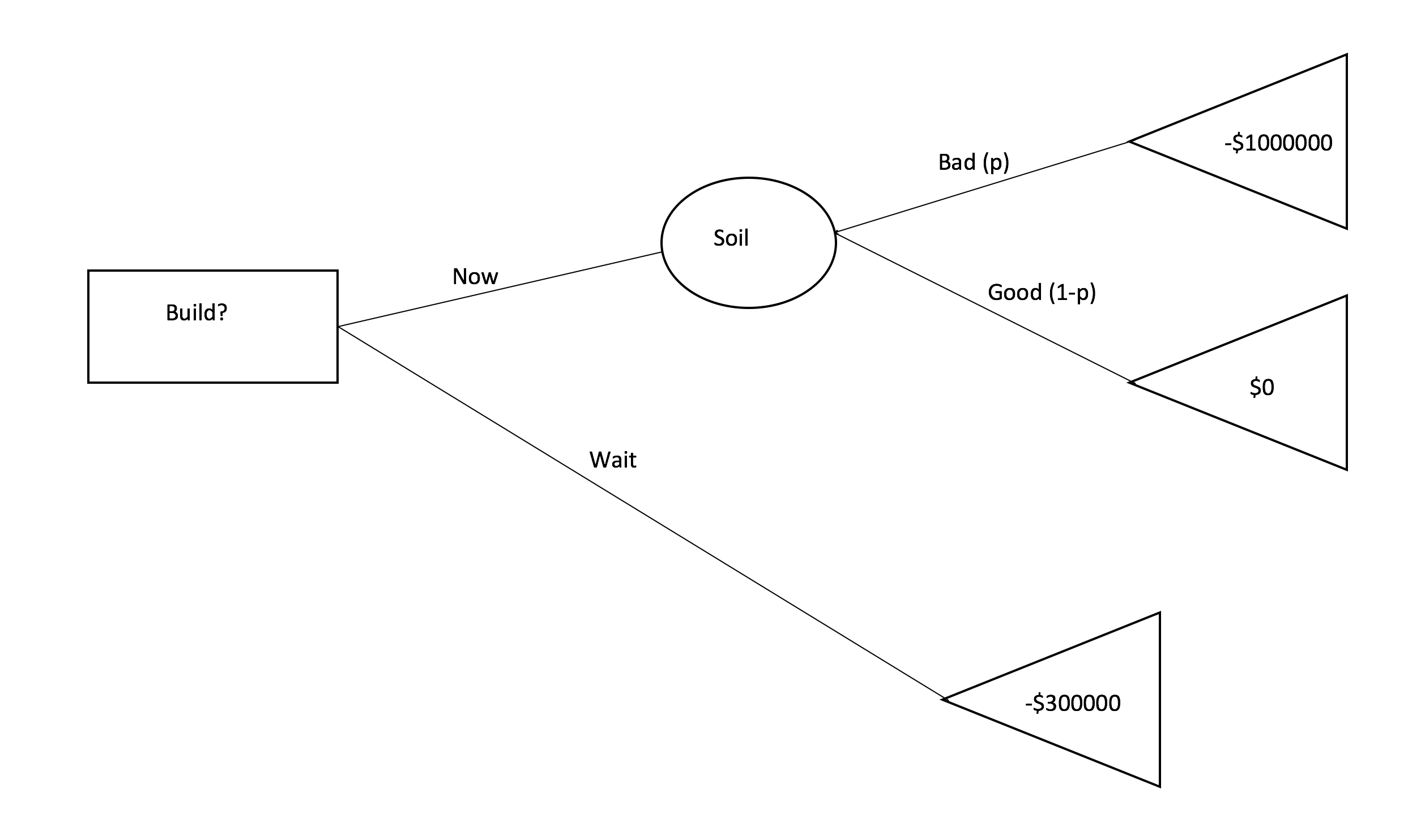
<final e="Good " p="1-p" v="$0" />

</chance>

<final e="Later" p="null" v="-$300000" />

</decision>

</xml>



*Figure 1.2: Example of a decision Tree*

**1.3 Documentation - Imaging Algorithm**

The workflow conversion of an image of a decision tree to an XML file was designed to have as little effect on the student workflow as possible and require as little work necessary for the instructor/professor. There is nothing extra that the students have to do and all of the handling of the student submissions can be done by untrained workers/volunteers. Students can draw the decision tree either in pencil or pen (blue or black ink). The tree has to be drawn on a blank, white piece of paper. More information on the workflow is outlined in 1.3.7.

The imaging algorithm converts *.jpg* and *.png* files to a XML format so that it can integrate with the rest of the website. When converting to a XML, there are intermediate steps involved with different data types. The first step is to annotate the student drawn tree with different colored markers. Once this annotated tree is scanned in, the image gets converted to a bitmap, then converted from a bitmap to a graph data structure, and finally from a graph data structure into a XML file.

The main program, function definitions, and constant definitions are defined in different files. *dt\_config.py* contains all of the constants or macros that are defined anywhere for this code, *img2xml.py* contains the main function that calls the functions defined in other files, *dt\_util\_prep.py* contains all the functions necessary to convert an image to a bitmap (including orientation), and *dt\_util.py* contains functions necessary to convert a bitmap to a XML file.

1.3.1: Color Annotation

The first step in the process of converting an image to an XML file is to annotate over the student submitted, drawn decision tree with different colored markers. Color is necessary because an algorithmic interpretation of hand drawn shapes is very unreliable. By annotating in color, the algorithm knows which color ranges to look for to determine certain types of structures in the image. For example, a red color indicates that the pixel belongs to a decision node. This is much more efficient computationally and more reliable than computationally determining the shape of each node in the decision tree.

The color markers used came from the *BIC Marking Fine Point Permanent-Marker 12-pack Assorted Colors* set. This set was chosen because of the relatively low cost compared to other markers. The Five colors chosen from this set were: *Rambunctious Red, Fandango Pink, Deep Sea Blue, Tuxedo Black,* and *Key Lime*. These colors were chosen because their Red-Green-Blue (RGB) spectrums are disjoint - meaning they each pixel can be uniquely determined to any or none of these colors. The RGB associated spectrum ranges for each color can be found in *Table 1.3.1*.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Red** | | **Green** | | **Blue** | |
| **Color** | Min | Max | Min | Max | Min | Max |
| Rambunctious Red | 100 | 255 | 60 | 120 | 70 | 119 |
| Fandango Pink | 120 | 255 | 60 | 120 | 120 | 200 |
| Deep Sea Blue | 0 | 100 | 60 | 125 | 130 | 255 |
| Tuxedo Black | 10 | 99 | 10 | 100 | 10 | 100 |
| Key Lim | 40 | 150 | 121 | 255 | 50 | 119 |

*Table 1.3.1: Spectrum ranges associated for each color. The ‘Min’ and ‘Max’ values represent the [min,max] range that the pixel value needs to fall within for that color.*

These values are hardcoded into the algorithm. If adjustment of these values are necessary, they can be found in the *dt\_config.py* file in the *Spectrums* section within the array named *spec\_robust*.

The colors associated with each node or edge can be changed by adjusting the values in *dt\_config.py* under the section *dt\_util\_prep*. For example, if you wanted to change the color of a decision node to pink instead of red, set the value of *val\_dec\_node = val\_pink*. This can be done for any variable to any defined color.

1.3.2: Orientation

The first step in the algorithm is to orient the picture so that the root node of the decision tree is the leftmost node. This is necessary because the bitmap parsing algorithm assumes that the root node is on the left side of the bitmap. This functionality is here so that the image can be scanned in any orientation and this algorithm can automatically rotate it so that it is in the right orientation. All functions are contained in *dt\_util\_prep.py*.

The way that the algorithm can tell the orientation is by looking for a black mark in one of the corners of the image. It assumes that the orientation is correct when the black mark is in the top left corner. An example is shown in *Figure 1.3.4*. Thus, the goal of the algorithm is to get the black mark in the top left corner and then getting rid of the mark so that it does not interfere with the rest of the algorithm

*Figure 1.3.2: Example orientations for a scanned in decision tree. In the left image the black marker is in the top right so the image needs to be rotated counterclockwise by 90-degrees. In the right image the black marker is in the top left corner so it does not need to be rotated.*

Pseudo code:

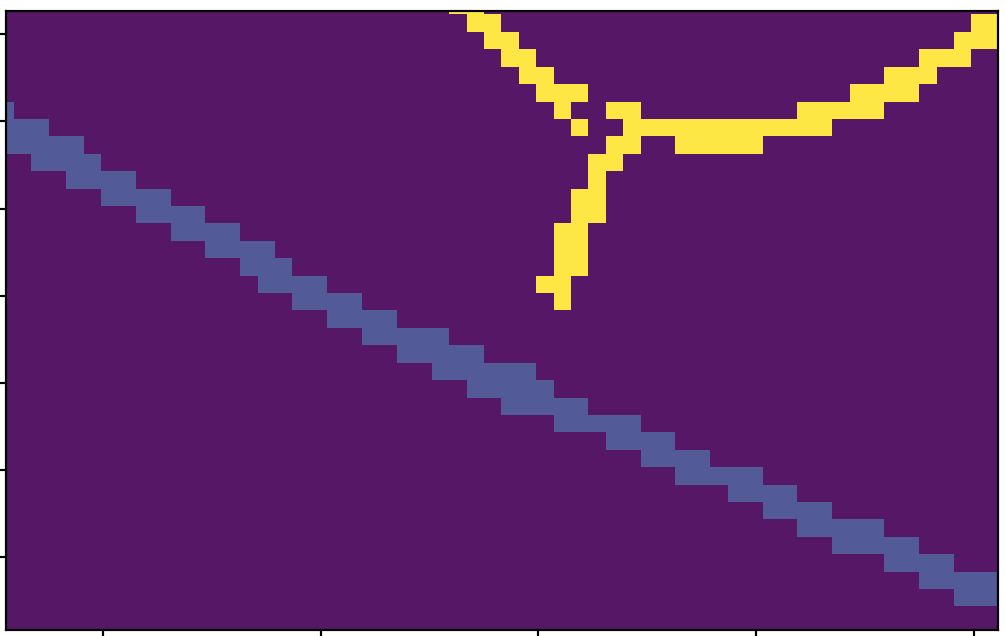
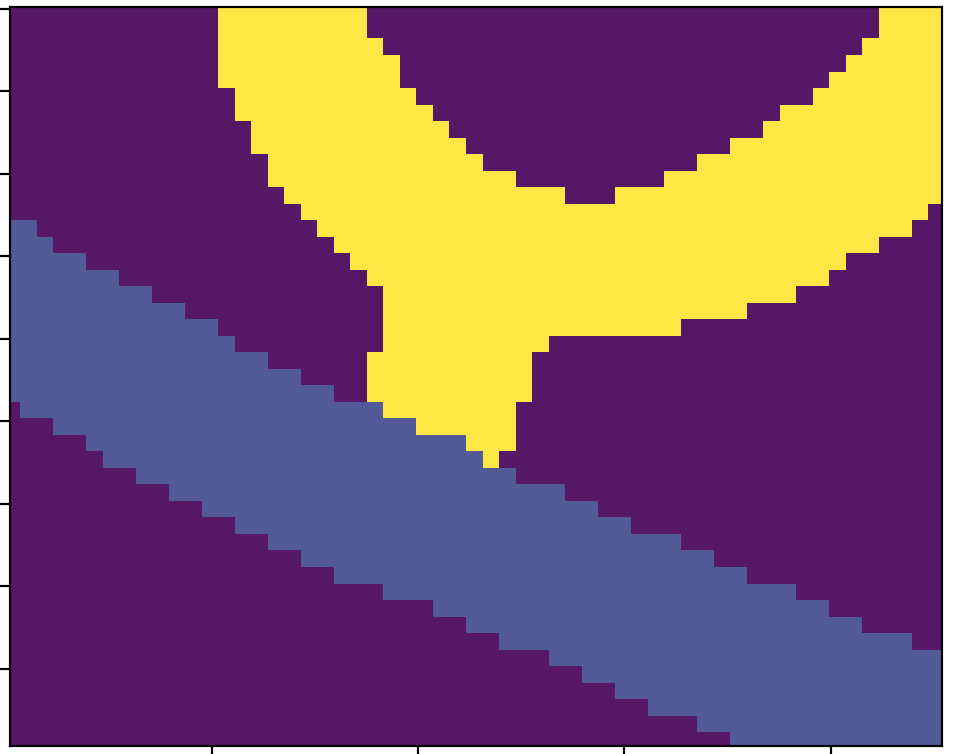
1. **change\_orientation**(img)
   1. Determines which corner the black mark is (calls border\_sweep)
   2. Rotate the image the required amount (calls rotate\_about\_center)
   3. Get rid of orientation mark (calls rid\_orientation)
2. **border\_seep**(img) - returns which corner that that has the darkest average color
   1. Get the average color for each corner
   2. Return which ever corner is the darkest
3. **rotate\_about\_center**(src,angle,scale) - rotates the image (src) an angle number of degrees to the left
   1. Derived algorithm from OpenCV - more efficient to call an OpenCV function than to manually rotate in python.
   2. It is a safe rotate - does not crop out any of the image when rotating and changes the size of the image as appropriate when rotating
4. **rid\_orientation**(img) - gets rid of the orientation mark in the top left corner
   1. Convert img to grayscale
   2. Finds all adjacent pixel to top left point (calls get\_num\_adj)
   3. Sets all pixels that were visited as white (calls or\_maps)

1.3.3: Image to Bitmap - Preprocessing

Once the image is rotated, the image is converted to a bitmap so that the determination of the different structures within the image can be parsed simpler during the image traversal. Within the code, this step is called *Preprocessing*. For each pixel in the image, if the values of the pixel fall within a certain spectrum (defined in 1.3.1), it sets the value in the corresponding location of the bitmap to the value set to the color. These values are defined in *dt\_config.py* in the *dt\_util\_prep* section with the prefix *val\_’color’*. One can change the values to any positive integer. The only **reserved** numbers are 0 and 1. 1 should always be an edge and 0 should always be nothing.

During the image traversal, the image is resized to ¼ the size. This is done because the algorithms efficiency is O(n2) and only rough relationships are needed. Thus, if we resize the image to ¼ the original size, it decreases the amount of computation by 1/16 - making it much more efficient. Once the bitmap traversal is done, the bitmap is resized to the original size so that it corresponds 1:1 with the original image. To decrease the complexity of the problem, each color is is loaded into its own bitmap and then merged at the last stage.

Additionally, the areas of each color is inflated before each spectrum bitmaps are merged together. This is done to take into account if combination of colors produces a color that is not contained in an of the spectrums. By inflating the area, the areas colors can overlap and then each color is guaranteed to touch each other. An example of this is shown in *Figure 1.3.3*.



*Figure 1.3.3: Left shows bitmap after merging without inflating the areas of each color. Right shows the bitmap after merging with inflating the areas of each color. In the right image, the different colors are touching while the left image they are not.*

Pseudo code:

1. **convert\_img2bitmap**(img1)
   1. Change the orientation so that the root is on the left (calls change\_orientation) (described in section 1.3.2).
   2. Change the color from BGR to RGB (done because OpenCV reads the image in reverse order to what this algorithm assumes). Blur to reduce noise.
   3. Resize to the inverse of the rescale factor
   4. Initialize a new bitmap for every color
   5. For each pixel in the image, if the pixel value is in the interval specified by the color, set the value to the respective color bitmap to 1.
   6. Inflate the areas and merge each bitmap (calls grow\_and\_merge)
   7. Resize back to original size
   8. Return bitmap
2. **grow\_and\_merge**(bitmaps) - inflate the areas for each bitmap (stored in a python dictionary) then merge them together
   1. Grow each bitmap (calls grow)
      1. Do this a *num\_grow* number of times. *num\_grow* is defined in
   2. Multiply the value of each color by the value associated with each color (to distinguish each color when bitmaps are merged)
   3. Merge each bitmap together (pixel by pixel). You want to do pixel by pixels because a single pixel can have multiple different values between the bitmaps. Add each color by priority:
      1. Set edge labels as the lowest priority
      2. Set edges as second lowest priority
      3. Set all other colors as highest priority (priority does not matter relative to each other).
3. **grow**(bitmap)
   1. queue all neighboring empty pixels of bitmap
   2. Set the queued up empty pixels to 1

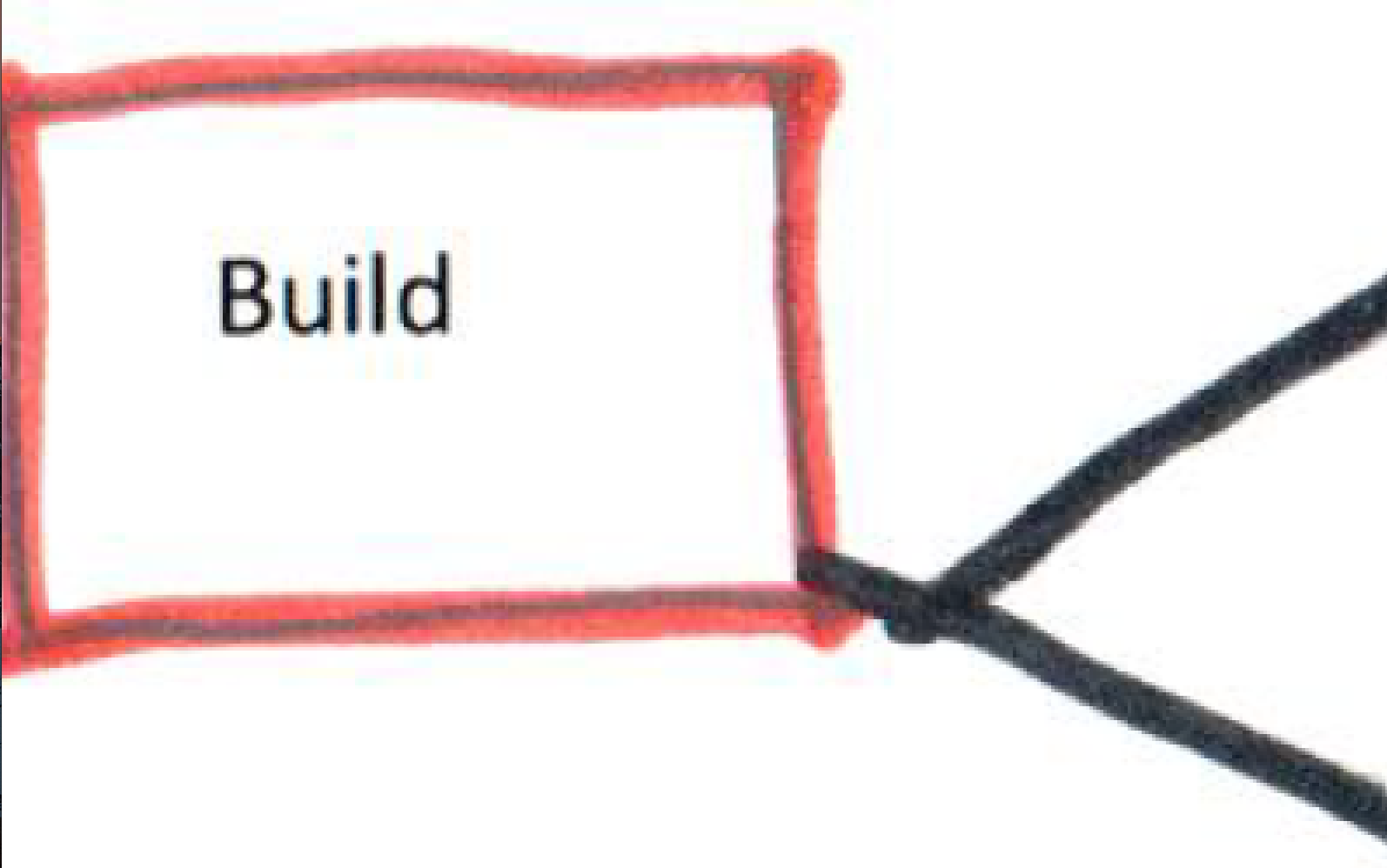
1.3.4: Bitmap to Graph Data Structure

Once the bitmap with the right format is created, the bitmap is traversed. As the algorithm traverses the bitmap from left to right, it gradually builds a graph data structure defined in *dt\_graph.py*. The bitmap is converted to an intermediate graph data structure because it is much easier to add information iteratively to a custom graph class than it is to add information iteratively to an XML file. Vast majority of the functions required for the traversal are in *dt\_util.py*.

Not all information about each node and edge is added at the same time, which is why there are a lot of different functions in the *dt\_graph* class about adding different information about each node or edge at different times in the algorithm. In general, first the pixel areas for each node, and edge are recorded and set into the graph structure. Once the locations for each are determined, the locations of the edge labels are added to the graph structure, labels are added for each edge and node, and then the node type is determined (decision, chance, or final node), in that order.

**Disclaimer**: How edge labels are associated with edges.

It is not guaranteed that each edge end point will end on a node, as supposed than another edge. This could happen because of style or by accident. To overcome the ambiguity of a neighbouring set of edges from a node being one or more actual edges, all edges are treated as one when looking at the next level of nodes from the current node. This complicates the logic of associating a label to a node (seen in the pseudocode), but it makes the imaging algorithm more robust in terms of what imaging can interpret.

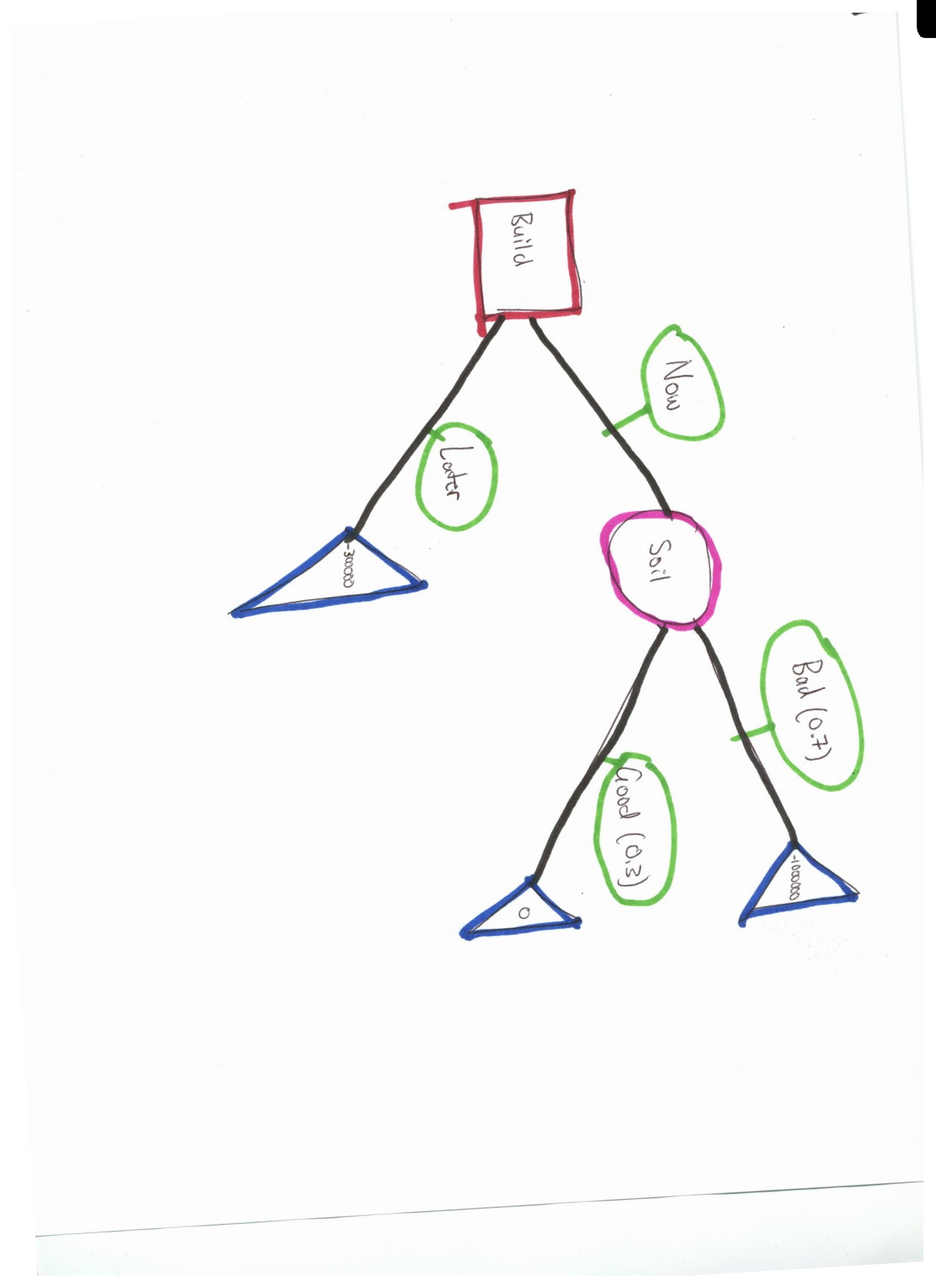
*Figure 1.3.4: Example where distinct edges connect to one edge when touching a node, instead of each edge touching the node in distinct places.*

Pseudo code:

1. **make\_graph**(img) - converts an image to a graph
   1. Convert image to a bitmap (calls convert\_img2bitmap) (defined in 1.3.3).
   2. Find the location of the root pixel (calls left\_sweep) (defined in *dt\_util\_prep.py*).
   3. Mark pixels adjacent to the initial location that are the same value as the initial location as visited. Queue up locations of all pixels of edges, get area location of node (calls mark\_node\_queue\_edges).
   4. Create a graph node with data field (*d*) set as the node location.
   5. Traverse the queued up edge pixels, marking all adjacent edge pixel nodes as visited. Queue up all adjacent node pixels. List all edge label pixels. Mark all adjacent edge label pixels as visited. Returns a list of sets of pixels from unique, unvisited, nodes (calls traverse\_edges\_to\_nodes)
   6. For each set of unique unvisited nodes, recurse to 1.c. Add the graph that is created as the child. If there are no children, continue to part 1.g.
   7. For each parent-child pair, associate each edge with an edge label. The labels are associated by finding the closest edge label node (calls add\_label\_location\_to\_graph)
   8. For each node and edge, add the label associated with it. Takes the location area provided by the graph data structure and takes the location from the original image. Within that sub-image is the label. Run the character recognition software (described in section 1.3.5) to translate the picture of the label to a string of the label. Set that string as the label (calls add\_labels).
   9. Determine the type of each node (decision, final, chance) (calls determine\_node\_type).

1.3.5: Optical Character Recognition (OCR)

The current OCR software that is used is Called Tesseract. It is an OCR engine for various operating systems. It is free software that can be downloaded. Tesseract works pretty well for computer printed words, a not very well on hand drawn words. An example is shown in *Figure 1.3.5*. Clearly from this example it does not work very well for had written words, and this misreading is very common. Later iterations of this software should use better, specially developed OCR. This implementation is designed so that the OCR is a black box (only contained in the function *add\_labels* in *dt\_util.py*) and can be easily replaced at a later date.



*Figure 1.3.5*

*n="Kn&#8220; d OCR of root node Build*

*Left picture is neat handwritten text, right picture is computer printed text.*

1.3.6: Graph Data Structure to XML file

Once the internal graph data structure is built, it is a relatively easy process to convert it to an XML file. Using the Python library ElementTree, you can build a tree with certain attributes and then it will automatically generate a XML file. The recursive algorithm is outlined in the below pseudo code:

Pseudo code:

1. **graph2xml**(graph,parent,index)
   1. Create a new XML object
   2. Set node type (set to *graph.node\_type*)
   3. Set incoming edge
   4. If parent is a chance node, set p = probability associated with this node, else set as “null”
   5. If a final node, set v = *graph.name*, else, set n = *graph.name*
   6. If graph has children, recurse on each child back to a. Set result as child of current graph. If it has no children do not recurse
   7. Convert XML object into an XML. then return.

1.3.7: Proposed Workflow

Once the students have drawn their decision trees in either pencil or blue/black pen, it is given to the instructor/TA. The trees are then annotated using the color markers specified in section 1.3.1. Using the standard convention used in this document: A person would outline all Decision nodes in red, chance nodes in pink, final nodes in blue, edges in black, and circling edge labels in green. Once the color annotation is done, the image of the annotated decision tree is scanned and uploaded into the website. Once the imaging algorithm parses the image, the read image displays what was interpreted using the Online Tree Sketching Tool. There, the person can edit the tree on the website until it matches the tree drawn on the paper in case there was any misreads by the Imaging algorithm. Once the person is happy with the tree that is on the screen, they click *Submit* and then the tree will either get graded (if it is an exam) or be saved into the database if it is for uploading a tree.

Even though there is some manual labor involved, it is an *automatic* grading of decision trees in the sense that the instructor never has to look at the files. Combined with the grading algorithm, all of the grading can be done systematically, accurately, without any intervention by the instructor.

1.3.8: How the Algorithm Fails

There are certain ways that the algorithm can fail, and certain ways that it doesn’t know it should fail. There are only four cases in which the algorithm is designed to fail:

1. No nodes were detected in the image.
2. An edge label did not get associated with an edge.
3. Two edge labels were associated to one edge.
4. If the first color detected is not a valid node color

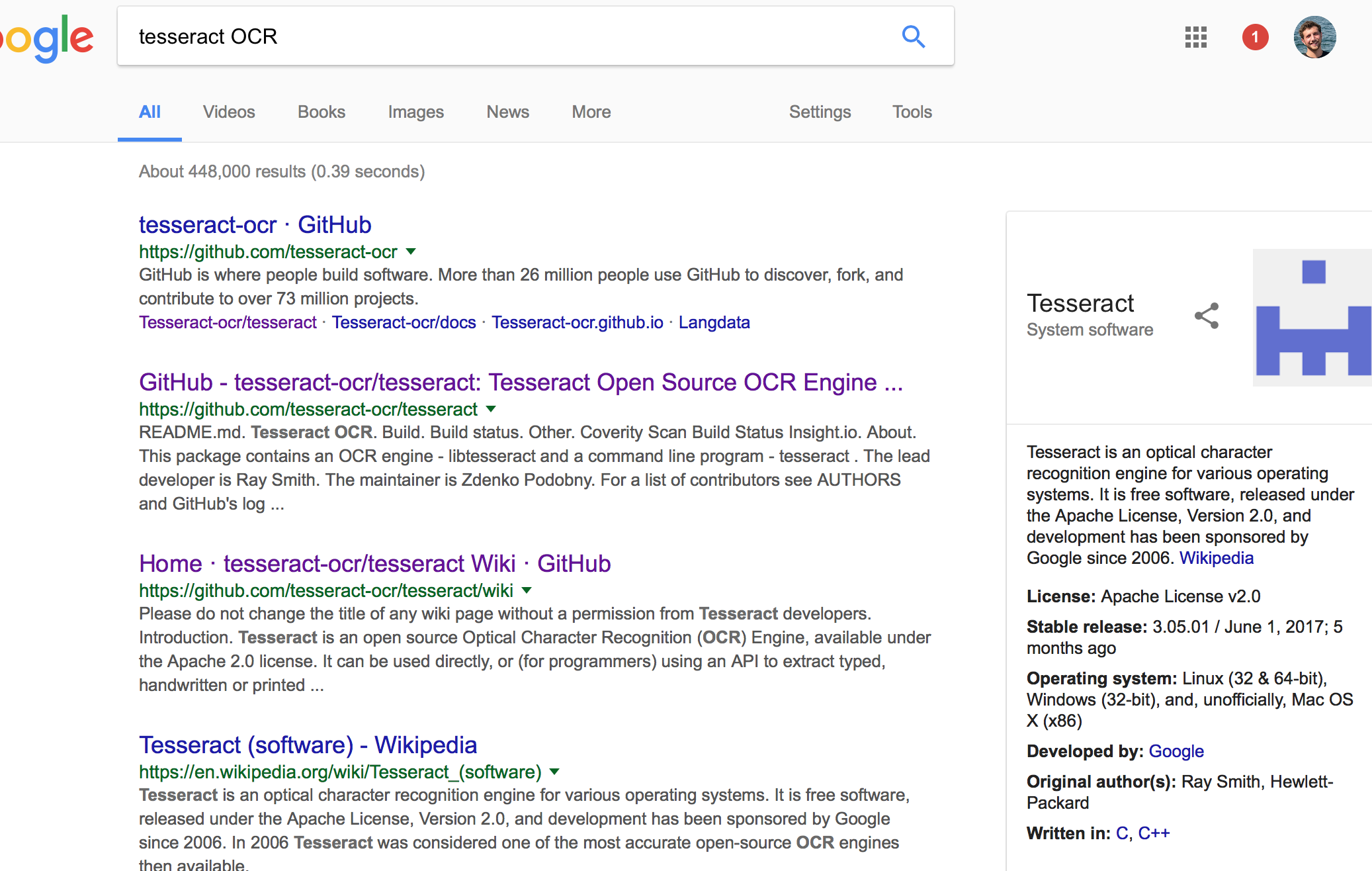
However, if a person uploads a random image. For example, *Figure 1.3.8*, there is no way implemented for the algorithm to detect it is not a tree. Thus, it can return some weird results. For example, *Figure 1.3.8* returns:

<xml>

<decision e="null" n="" p="null" />

</xml>

Clearly *Figure 1.3.8* is not a tree. More research has to be done to tell if a picture is a tree or not.

*Figure 1.3.8: A random picture that was sent through the Image Algorithm*

**1.4 Documentation - Grading Algorithm**

The tree creation and grading calculation is encompassed in the DecisionTree module, contained in the DecisionTree.rb Ruby file in the Grading directory of our SVN repository. Internal API that can be used for tree creation include the initializers for DNode and DEdge classes, the node and edge classes respectively. The DNode initializer takes in a node name, and optionally a data and type may be added. The DEdge initializer takes in a source DNode, a destination DNode, a name, and an optional data field. DNode objects have a name, data, and type fields, along with a list of children and a child-edge map that can be used for processing. The static DNode class function “grade” takes a solution tree root node and a student submitted tree root node as input and outputs the calculated final grade as a floating point value rounded to two decimal places.

The grading scheme we have chosen to adopt is based on the following three criteria: presence of all required nodes, presence of all required edges, and similarity of the tree to the solution tree. After some testing and tweaking to the numbers, the final weights settled on have 35% of the total grade coming from the node presence check, 35% of the total grade coming from the edge presence check, and 30% of the total grade coming from the similarity comparison. The presence check steps only examines the submitted tree for nodes and edges that are present in the solution tree, deducting points for missing nodes or edges, and will not deduct credit for extra incorrect nodes or edges. The similarity comparison will pick up on the extra nodes or edges.

The presence check step for nodes is done by creating a list of all the nodes in the solution tree, and a separate list of all the nodes in the submitted tree. Then, the algorithm iterates over all the nodes in the submitted tree and searches for a match in the solution list, incrementing a counter and adding its name to a set of found nodes if a match is found. The resulting grade for this portion is the percentage of nodes in the solution tree that are found in the submitted tree.

The presence check step for edges is done in a similar manner with a list of all edges in the solution tree and submitted tree created. Again, the algorithm iterates over all the edges in the submitted tree and searches for a match in the solution list, incrementing a counter and adding its name to a set of found edges if a match is found. The resulting grade for this portion is the percentage of nodes in the solution tree that are found in the submitted tree. The present set for the above two steps are included to ensure that erroneous credit is awarded for duplicate nodes.

The similarity comparison is done by checking each node in the submitted tree against the solution tree, and comparing the depth at which it’s found. A total depth difference counter is kept track of, and if there is a difference, the counter is incremented by the absolute value of the difference between the depth of the node in its own tree, and the depth at which it's found in the solution tree. An additional counter is incremented if a node is not found in the solution tree. The average depth difference value is defined as the total depth difference divided by the number of nodes in the solution tree. The final grade for the similarity comparison step begins at 100 and is subtracted from by two parts that equally represent half the grade. Half of the deduction is calculated as the percentage of nodes not found multiplied by 50, with the percentage not found defined as the not found count divided by the number of nodes in the solution tree The other half of the deduction is calculated as the average depth difference multiplied by 50. The final grade is the baseline 100 with the two deductions subtracted.

The final step of the grading algorithm takes the results of the three steps and applies the proper weighting as defined above, and outputs the final value as a float rounded to two decimal places. The benefit of defining grading in this manner is the ease of adjustment if needed as the weights for each individual section can be tweaked quickly.

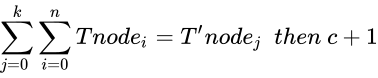
upper L e t upper T b e a. d e c i s i o n t r e e w i t h n n o d e s a. n d n minus 1 e d g e s

w h e r e for-all n o d e s there-exists StartSet d e c i s i o n comma c h a. n c e comma f i n a. l EndSet

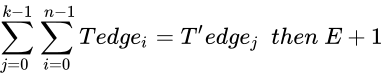
a. n d upper T prime b e a. d e c i s i o n t r e e w i t h k n o d e s a. n d k minus 1 e d g e s

w h e r e for-all n o d e s there-exists StartSet d e c i s i o n comma c h a. n c e comma f i n a. l EndSet

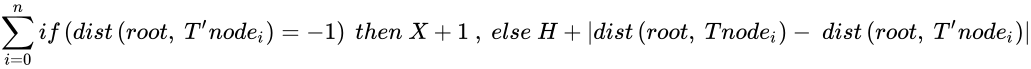
Let C (initialized at 0) be the total common nodes shared by T and T’ determined by



Let E (initialized at 0) be the total common edges share by T and T’ determined by



Let dist be a function that determines the distance of a node from another (or its own) tree’s root node and returns -1 if node not found. Let H (initialized at 0) be the total height difference of T and T’ and X (Initialized at 0) be the nodes not found in T determined by



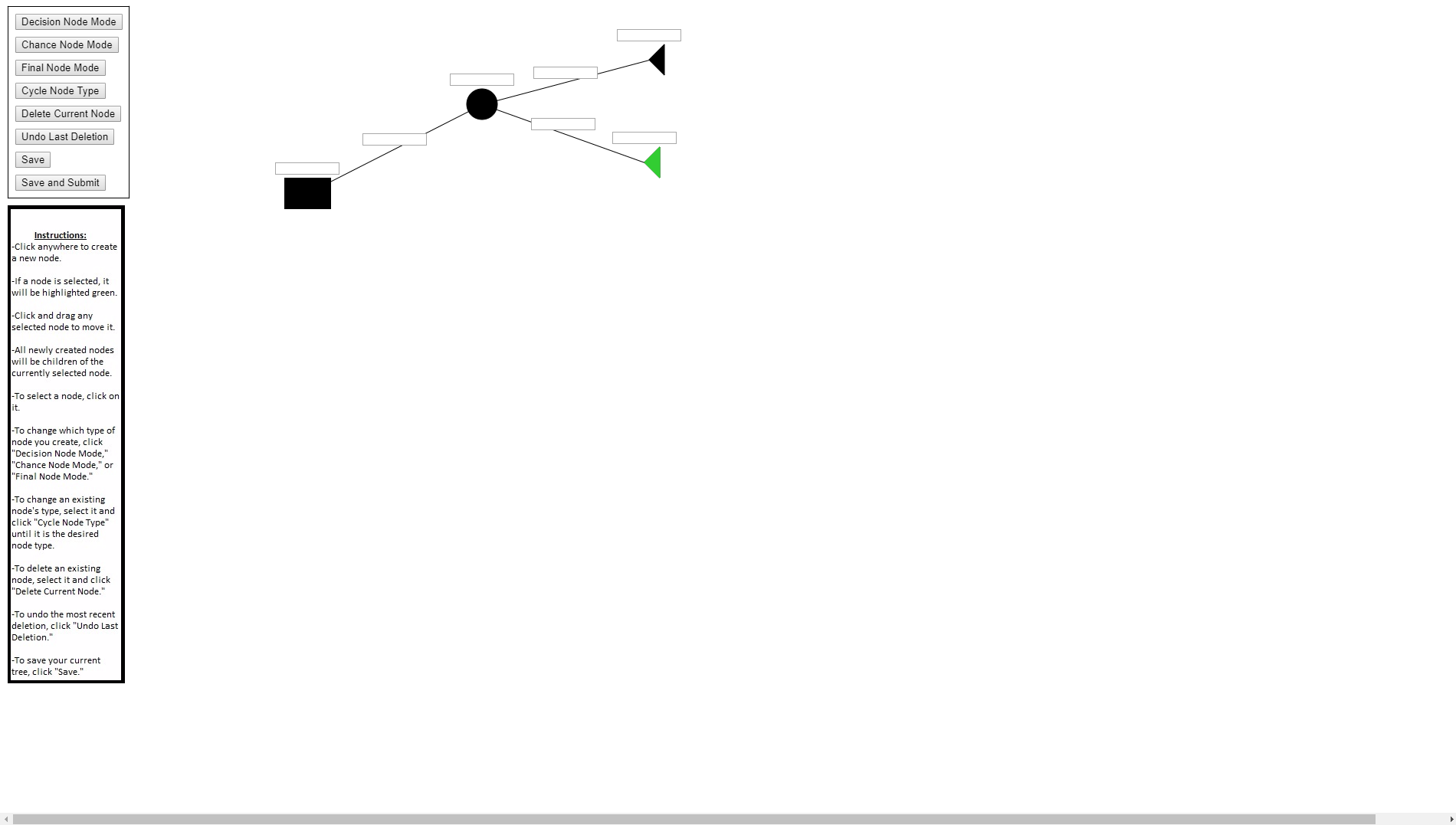
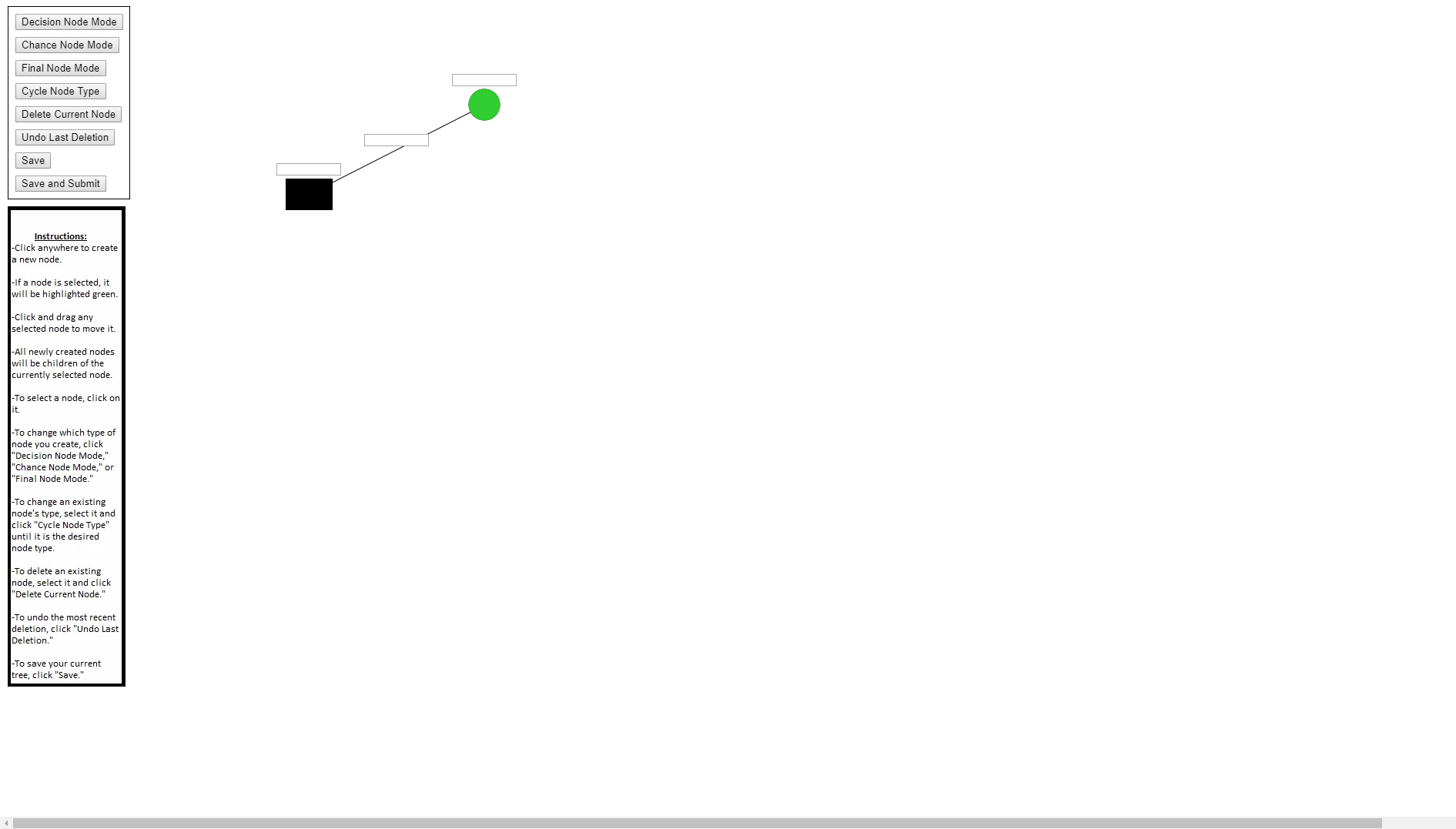
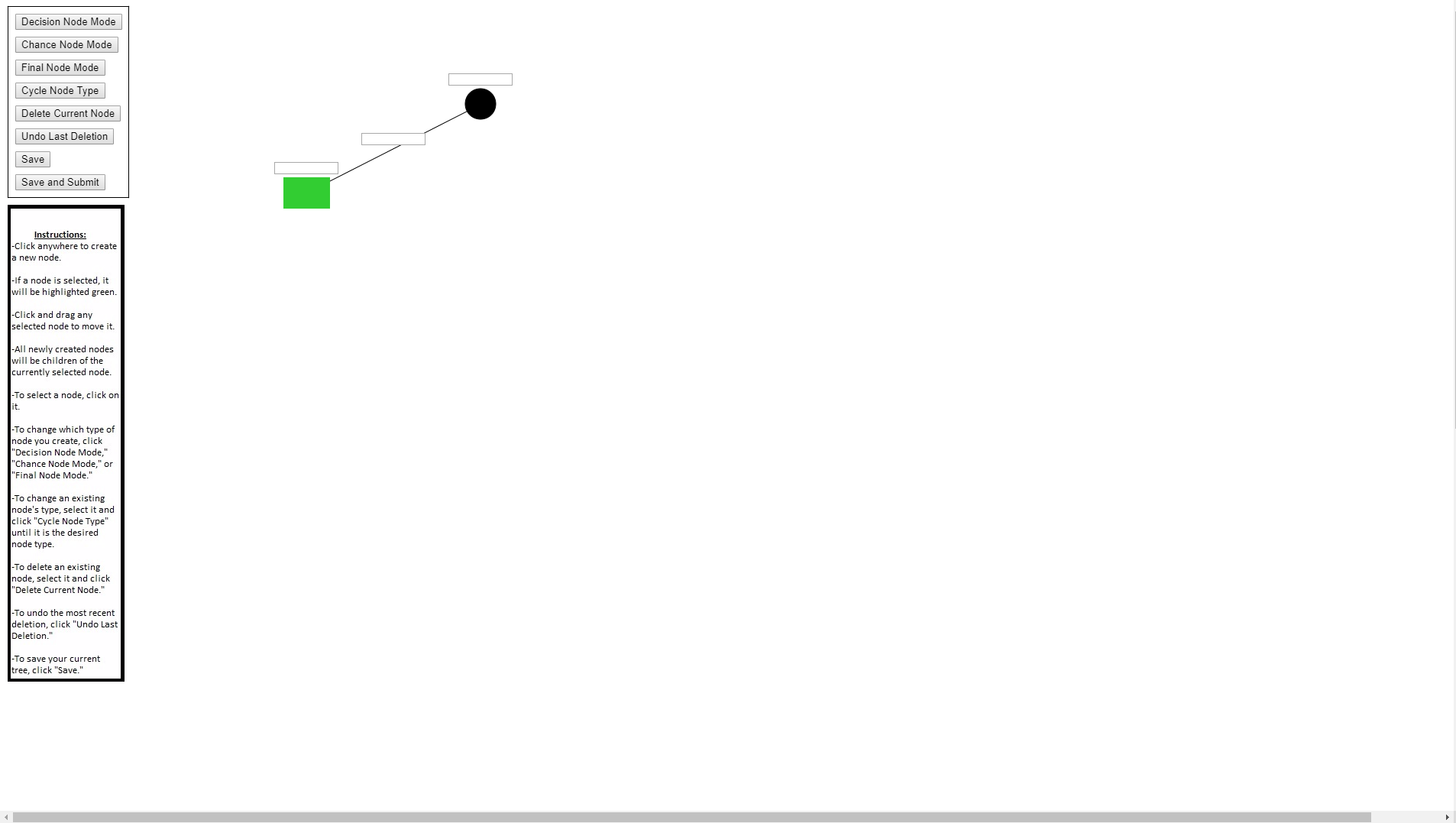
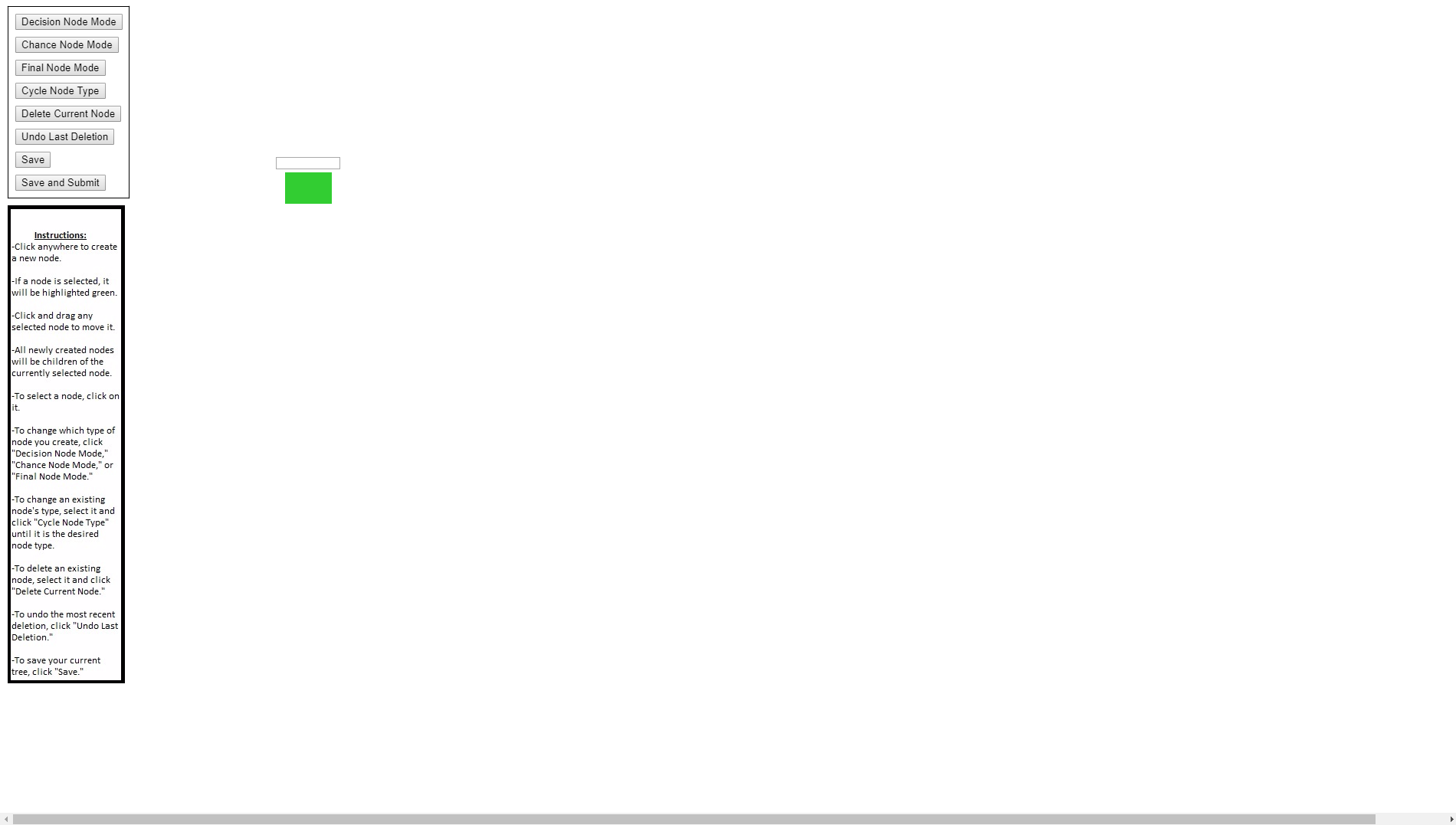
upper F i n a. l upper G r a. d e equals left-parenthesis 35 right-parenthesis  upper C Over upper N  plus left-parenthesis 35 right-parenthesis  upper T Over upper N minus 1  plus left-parenthesis 30 right-parenthesis left-parenthesis 100 minus left-parenthesis 50 upper H plus 50 upper X right-parenthesis right-parenthesis

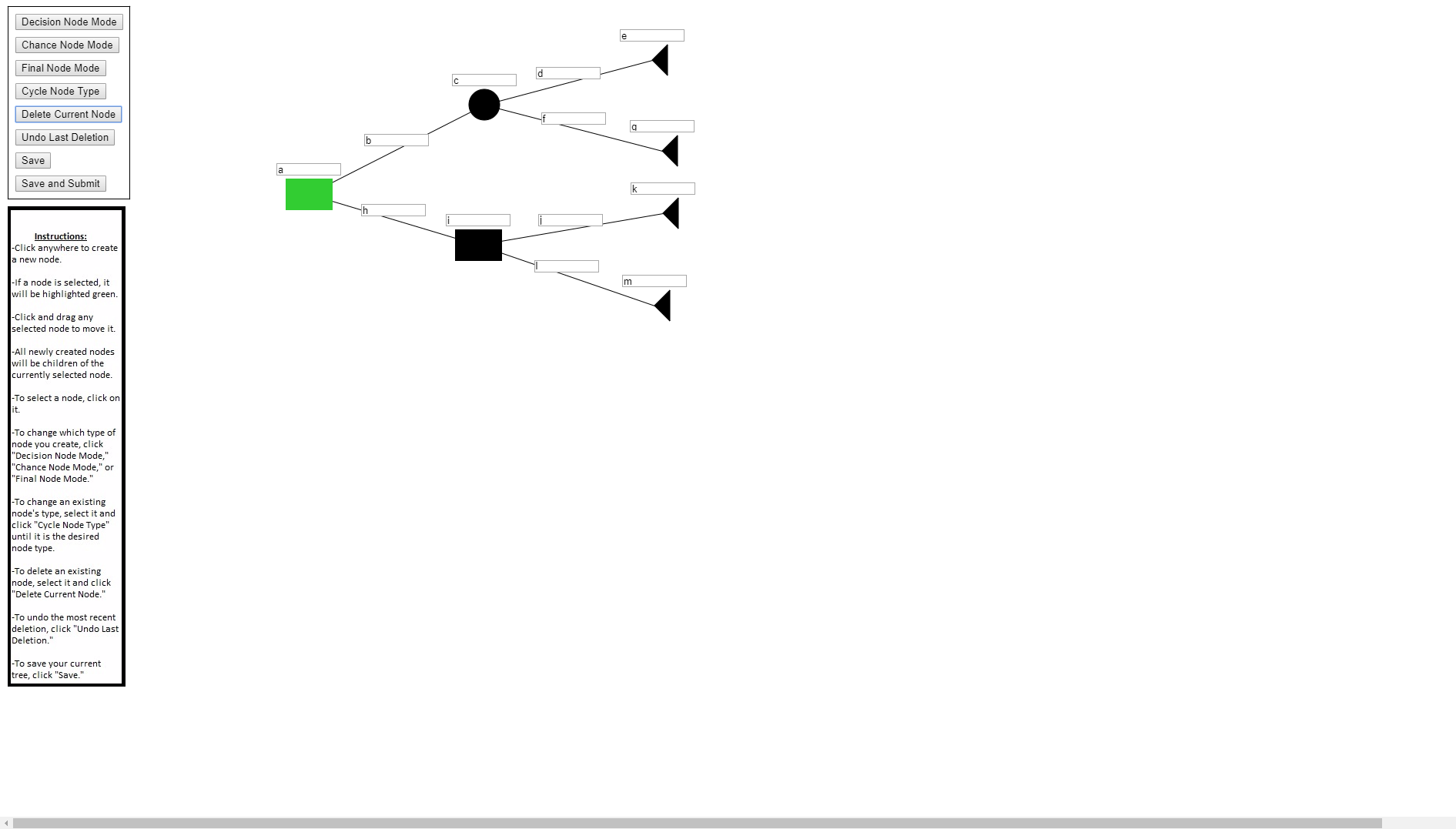
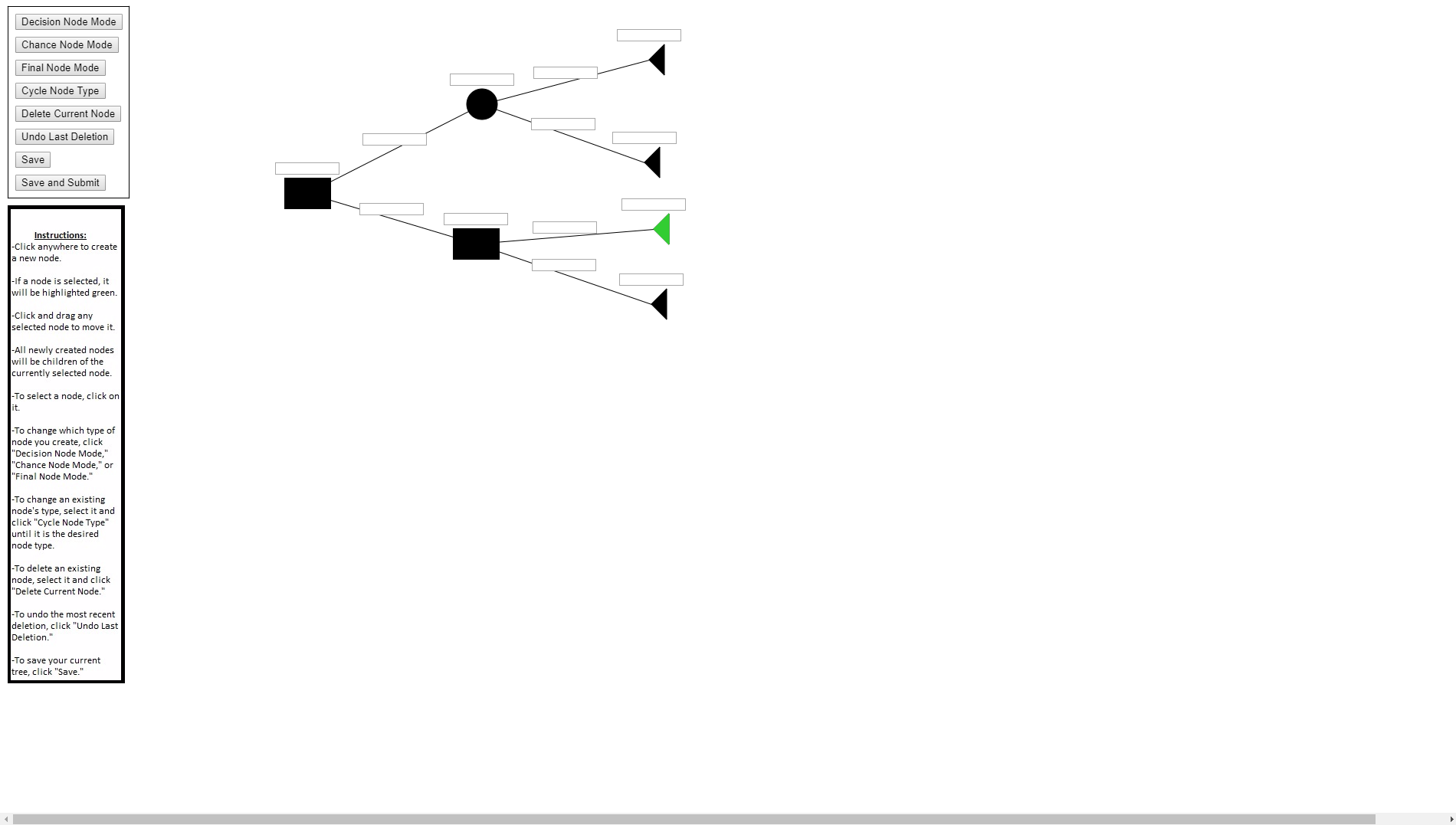
**1.5 Documentation - Online Tree Sketching**

The Tree Sketching application was constructed using p5.js. For documentation on p5.js please visit https://p5js.org/. Additionally, the Tree Sketching application is guaranteed to be compatible with Internet Explorer 11 or newer, Microsoft Edge v.23 or newer, Google Chrome v.51, and Safari v. 9 or newer. Other browsers may not be compatible.

The Tree Sketching program is simple and easy to use. The basic format of the program is as follows: click anywhere to create a new node; if a node is selected, it will be highlighted green; a selected node can be clicked and dragged to change its position; all newly created nodes will be children of the currently selected node. Furthermore, user instructions are displayed immediately upon opening the application. In order to construct a basic tree, the user can refer to these simple instructions on the left hand portion of the screen: to select a node, click on it; to change which type of node you create, click "Decision Node Mode," "Chance Node Mode," or "Final Node Mode;" to change an existing node's type, select it and click "Cycle Node Type" until it is the desired node type; to delete an existing node, select it and click "Delete Current Node;" to undo the most recent deletion, click "Undo Last Deletion;" to save your current tree, click "Save."

**Basic Tree Sketch Program Drawing Workflow:**





**Captions: (Ordered Left to Right & Up to Down)**

1. To start, the user would press click "Decision Node Mode," "Chance Node Mode," or "Final Node Mode" button to select their first node (in this case Decision Node Mode was clicked). Then the user can click anywhere on the canvas to create a node at that point.
2. The only possible node is selected. Thus, the user can click anywhere to create a new node of their choice. This new node will be a child of the first node.
3. The user can change the selected node by clicking on a new node.
4. The user can now make children for the newly selected node.
5. The user can repeat this process as necessary.
6. The user can enter text into the text boxes of all the node and edges.

The Tree Sketch program consists of four main entities: the Tree data structure, the Node data structure, the NodeCopy data structure, and the XML parser.

The Tree data structure is the part of the program which unites all the different entities. It calls all the necessary methods and functions in the correct order. It is comprised of:

1. A head node (this.head): is the head node of the decision tree node structure (is a Node object).
2. A current node (this.current): is the node that is currently selected by the user (visually highlighted as green when run) (is a Node)
3. A node type (this.nodeType): is the changeable setting which determines the type of a user’s newly created node
4. A list of all nodes (this.nodes): is an array of all the nodes that are in the tree for easy access (is a list of Node objects)
5. An id (this.id): is an integer to keep track of the nodes ids to keep them having unique ids. Increments every time a node is created
6. An undo tree (this.undoTree): is a separate head node used for an undo function to load from (is a NodeCopy object)

The Tree data structure has two unique methods both of which have similar implementation. These are select() and selectEdge(). Select is the method that determines if and which node is selected when the user clicks. SelectEdge is the method which determines if a user click on the edge text box in between two nodes.

The basic psuedo for select is as follows:

Select() {

Checks each node in the this.nodes array and finds the nodes that is the shortest distance from the click.

If the node that is closest to the click is within 50 from the click, set the current node equal to the found node and return true.

Else return false.

}

The Node data structure is the primary means of storage and calculation surrounding the decision trees. It is comprised of:

1. An id (this.id): is an unique integer id that represents this node. Used for recognizing it
2. A type (this.type): is an integer that represents the type of node (0 = decision, 1 = chance, 2 = final)
3. A list of children(this.children): is an array filled with all of this node's children
4. A x position (this.x): is a number representing x coordinate of this node
5. A y position (this.y): is a number coordinate of this node
6. An input text (this.inputText): is a p5.js Input Object which is the text box for this node
7. A list of edges (this.edges): is an array of p5.js Input Objects which are the text boxes for the edges from this node to its children

The Node data structure has several primary methods:

Add - Adds a child node with given parameters to the current node’s children list.

Delete - Recursively traverses the nodes until it finds the node in the parameter. It then removes this node from the node structure.

Draw - Draw lines from node to children and draws children recursively.

Update Edges- Recursively traverses all the edge text boxes within each node and places them properly between the parent and the child.

Update Position- Updates the position of the current node and its text box.

The basic pseudo behind the the deletion is as follows:

delete(n) {

If (current node = n) {

Delete this node and its children

Return null

} else {

Recursively check this nodes children.

}

Return this node;

}

The NodeCopy data structure is primarily used during deletion and undo of a tree. It is comprised of:

1. An id (this.id): is an unique integer id that represents this NodeCopy. Used for recognizing it
2. A type (this.type): is an integer that represents the type of NodeCopy (0 = decision, 1 = chance, 2 = final)
3. A list of children (this.children): is an array filled with all of this NodeCopy's children
4. A x position(this.x): is a number representing x coordinate of this NodeCopy
5. A y position (this.y): is a number coordinate of this NodyCopy
6. An input text string (this.inputText): is a string which is text for this NodeCopy
7. A list of edge strings (this.edges): is an array of strings which are the text forthe edges from this NodyCopy to its children

The purpose of the Node Copy stems from the fact that any input text in p5.js automatically appears and is draw when it is initialized. This causes problems when constructing a back up tree for an undo function. If you were to use a regular Node as a backup for the undo function, it would draw all the textboxes associated with a back up and they would appear on screen. Thus, it was necessary to create NodeCopy. NodeCopy is exactly like a node object except that it does not use input texts. It temporarily stores text for nodes and edges as strings so that they are not drawn on the canvas. When it is necessary to load a backed up tree, the program converts the NodeCopies to Nodes.

The basic psuedo for converting from Node Copy to Node is as follows:

Convert(){

Create new Node Object with attributes of the current Node Copy object

Initialize the Node Object’s input text with the correct value from the current Node Copy Object

Initialize all the edge input texts from this Node Object with the correct value from the current Node Copy Object

Recursively loop through the Node Copy’s children and do the same to them.

Add the newly returned children to the children list of the Node Object’s children list

}

The XML Parser is a large portion of the tree class and the node class. When loading a tree, the program loads an xml file onto a p5.j XML object. This nodes of this object are recursively called and loaded into the tree data structure. When saving, the program uses an XML writer function from https://www.codeproject.com/Articles/12504/Writing-XML

-using-JavaScript. This provided simple XML writing functions that were used when recursively constructing the xml from the tree data structure.

Basic XML reading pseudo code:

read() {

Read first xmlnode

Create new node and assign it the same values

Recursively read the children of the xmlnode and assign their results as the

children of the new node

}

BasicXMLwriting pseudo code:

write() {

Make a new XML node.

Add the corresponding attributes to the new xml node

Recurse over children

}

**1.6 Documentation - Website**

**Instructor Tasks:**

The Instructor is able to:

1. Add another instructor into database

Submit a form on instructor main page

2. Create new course into database

Submit a form on instructor main page

3. Delete an existing course into database

Select the option to delete a entry

4. Create an assignment with a standard answer for specific course

Submit a form on course page

5. Remove an assignment for specific course

Select the option to delete a entry

6. Add student into specific course

Submit a student form on course page

7. Remove student from that course

Select the option to delete a entry

8. Views all the submission of that assignment and see students’ grades

View submissions from submission page

**Professor Instructions:**

1. If you want to add other ta's or add other teacher's into the system, you can input the user's directory id into the form
2. If you want to create a course, you need to fill in the form with the course name
3. If you want to create an assignment, you need to fill the assignment name, assignment description, and assignment due date, and the correct tree answer for the assignment
4. If you want to add a student into this course, you need to fill in student directory id into that form
5. If you want to create a correct answer tree, you can do it either in uploading image or create an online sketching tree by referencing the online tree sketching instruction or images instruction

**Student Task:**

A student user is able to:

1. Update their profile through the update link on the home page
2. View all the courses the student is taking, which is listed on the home page
3. View all the assignments listed on specific course by clicking on a specific course
4. Make submissions to specific assignment by either providing an image, which is automatically uploaded to the server or create a tree through our online app

**Student Instructions:**

1. A student can update their profile through the update link on the home page
2. A student can view all they are taking, which is listed on the home page
3. Clicking on an course will take the student to a page that lists all assignments for that page
4. If there are assignments for a class, the student can submit an answer by either providing an image, which is automatically uploaded to the server or create a tree through our online app

**2.1: Website Acceptance Test**

Numbering below refers to the green light acceptance tests for the website (section 3.2).

1. Can the website be installed on a new server?

This test was failed. Even though the website can be installed on a different server, we did not have time to write detailed explanation on how to do this. Therefore a person with no prior knowledge of the operating system and framework will probably not be able to install the website on a different server.

1. Can the user log in?

We passed this test. The user, upon hitting the server gets redirected to The University of Maryland’s Central Authentication Server (CAS). Here the user provides their University ID and password. They will then be redirected back to the appropriate landing page based on their role.

1. Can an instructor create a class?

This test was passed. When instructor logs in, he or she sees a list of their courses. Here there will be a link to create a course. The instructor fills in the course name and presses submit. The course is now created.

1. Can an instructor create an assignment?

This test was passed. In the course homepage, there is a link to create a new assignment. Here the instructor fills out the assignment title, description, and due date. Then the instructor presses the “Continue To Create Tree” button. Here they have a choice of how creating the tree that will be used as the baseline in the grading process. First choice is using P5.js which launches the processing script in a new tab and when submitted, converts the tree to xml and saves to the database. The second option is Imaging. This link will take the instructor to a page with the option to attach a .png image file of a hand drawn decision tree. Upon attachment and submission, the OpenCV/Python script will be run converting the image to an xml formatted decision tree and saved to the database.

1. Can a student answer a question?

This test was passed. In the student’s course homepage will be a list of assignments. The student chooses one to see a list of submissions to this assignment. Here there will be a link to make a new submission. This will take the student to a page with the assignment title, assignment description and the same two links that the instructor had to create the assignment. The links work the same way. Only difference is once a student submits his or her assignment it will automatically get graded. The grade will show up in the list of submissions.

1. Can a student/instructor see their grade?

This test was passed. As mentioned in test 6, once the user submits their decision tree, the grade will be shown on the submission list page. The instructor can go to the course page and choose the assignment for which to see the grades. Here will be shown the list of students who have submitted their assignments along with the student UID, grade, and date of submission.

**2.2 Grading Acceptance Test**

In our conducting of acceptance tests for the grading portion of the product, there were both successes and failures. To begin testing out Decision Tree comparator, several differently sized trees had their exact and inexact copies compared, and its corresponding grade observed. The comparator was able to always assign exact solutions a perfect grade, and never assigned inexact copies a perfect grade, which was confirmation of successful basic activity. Then, we moved on to testing the accuracy of the grade assigned. To do this, many different possible solutions to a possible sample problem with a given solution were input, and the results logged. The comparator was able to quickly display a grade that we confirmed with the client, Professor Hermann, to be of an acceptable standard for an automated process. Furthermore, the algorithm proved to be much quicker than grading by hand, taking less than two seconds to output a grade when given two tree objects of around 25 nodes and 25 edges in size to compare. Another aspect of the grading process was to populate a list of nodes that were not found, which was done during the similarity comparison check step. At the end of the grading process, this list was then retrievable so that the student would be able to view nodes in their tree that were deemed incorrect. Several differently sized trees were compared for this step, and the resulting output error list was compared by hand to ensure that the nodes that were expected to be invalid were present in the list.

The above tests encompass the acceptance tests we were able to conduct that were successfully completed, and are specified by **Section 3.3** in our proposal as acceptance tests 1, 4, 5, and 6. Acceptance test #7, the accurate grading of synonyms, truncations, and misspellings proved to be a partial success. With the integration of the Ruby Gem “wordnet”, the grading algorithm was able to assess synonyms of node or edge names as a perfect copy when compared to a solution node or edge. However, truncations and misspellings were significantly harder to detect, and attempting to deliver fair grading for these cases resulted in reduced accuracy and fairness of the grading overall. Thus truncations and misspellings receiving full credit in our final product will not be supported.

Along with the successes, we experienced several failures. Acceptance tests 2 and 3, which were paired together to assure validity of decision tree collapsion, ended up being unsuccessful. During the tree collapsion step, chains of chance nodes should be combined with their resulting probabilities multiplied together. However, our grading algorithm takes into account the strict ordering of the names of nodes and edges, and finding a method to combine the instructor solution’s names in a logical manner was unsuccessful. The time constraints that came with earlier failures to plan ahead meant that incorporating accurate tree collapsing in test #2 and the following uniqueness of collapsed trees assertion in test #3 were unsuccessfully in completion.

To recap the acceptance tests for grading, the ultimate affirmation that we were on the right track was the client agreeing with our manner of approaching the automated grading process, and confirming that the output grades for the examples tested were well within the acceptable range for error. However, we were both too ambitious and ill prepared when planning for the overall conductance of acceptance tests, which can be shown in the failures encountered. One of the valuable lessons that we learned from this was how quickly small objectives balloon into larger tasks, which was a large contributing factor for the team as a whole not distributing the workload effectively across the semester. As a result we crammed too many tests into too short a period of time, and ultimately came up short. Although for this portion there were many failures, we learned of the importance of proper planning and prioritization, and how much more the minor tasks may be than they initially seem.

**2.3 Imaging Acceptance Test**

Each one of these numberings refers to the acceptance tests outlined in the Greenlight document for Imaging (section 3.4). Some structural changes were implemented since creating the green light about coloring conventions, core traversal algorithm, and constants used. Since most of the original tests do not really apply because of changes, the ***intent*** of the acceptance test was performed.

1. Can the imaging software recognize a perfect (printed solution)?
   1. Image that was used: *Figure 1.3.2* - right image.
   2. Output:

<xml>

<decision e="null" n="Build" p="null">

<chance e="Now" n="Soil" p="null">

<final e="Bad " p="0.7" v="-1000000" />

<final e="Good " p="0.3" v="" />

</chance>

<final e="" p="null" v="-300000" />

</decision>

</xml>

1. True output:

<xml>

<decision e="null" n="Build" p="null">

<chance e="Now" n="Soil" p="null">

<final e="Bad " p="0.7" v="-1000000" />

<final e="Good " p="0.3" v="0" />

</chance>

<final e="Later" p="null" v="-300000" />

</decision>

</xml>

1. Differences: two different Labels were not recognized correctly. Structure is correct. **Acceptance test failed**. Reason: OCR not optimized.
2. Can *Figure 1.1* from green light proposal be color annotated and read, per coloring conventions specified in Section 1.3.1 of the final report?
   1. Output:

<xml>

<decision e="null" n="Build" p="null">

<chance e="Now" n="Soil" p="null">

<final e="Bad " p="0.7" v="-1000000" />

<final e="Good " p="0.3" v="" />

</chance>

<final e="" p="null" v="-300000" />

</decision>

</xml>

1. True output:

<xml>

<decision e="null" n="Build" p="null">

<chance e="Now" n="Soil" p="null">

<final e="Bad " p="0.7" v="-1000000" />

<final e="Good " p="0.3" v="0" />

</chance>

<final e="Later" p="null" v="-300000" />

</decision>

</xml>

1. Differences: Same errors as in acceptance test 1. Problems with the OCR. **Acceptance Test Failed**.
2. Can a colored over hand drawn image be read?
   1. Image used: *Figure 1.3.5*, left image.
   2. Output:

<xml>

<decision e="null" n="Kn&#8220; d" p="null">

<chance e="N om" n="Sad" p="null">

<final e="" p="" v="AWDM" />

<final e="" p="" v="" />

</chance>

<final e="" p="null" v="" />

</decision>

</xml>

1. True output:

<xml>

<decision e="null" n="Build" p="null">

<chance e="Now" n="Soil" p="null">

<final e="Bad " p="0.7" v="-1000000" />

<final e="Good " p="0.3" v="0" />

</chance>

<final e="Later" p="null" v="-300000" />

</decision>

</xml>

1. Differences: all the labels. It is clear that the OCR does not work well at all on hand written words. More research and development has to be done for in the next update. **Acceptance Test Failed.**
2. Does the workflow outlined in section work?

**3 Project Advertisement**

Our software offers professors a hands-off experience for grading decision trees.Decision Tree software offers an online system that facilitates the construction of decision tree assignments. It allows the professor to post questions and correct answers to each corresponding question in decision tree assignments. Students are able to view each assignment as well as each question on each assignment. Students have a convenient way to construct, save, or submit a decision tree using an online interface. In addition, the software gives another alternative for building decision trees where professors and students are capable of scanning in images of hand drawn decision trees for exams. Assignment submissions and scanned exam trees are stored, constructively graded, and displayed along side their grade for the professor and the corresponding student to view. The navigable website allows for a smooth user experience. Within the overview page there will be navigation links to a setup walkthrough, and an information page. The setup walkthrough link provides instructions on how a user can host their own version of our decision tree formation and grading website, along with needed dependencies and their relevant documentation and download links. The information page outlines the purpose and reason for creating our product, along with repeating several of the key features such online tree submission and automated grading.

**4 Cost Estimation Vs. Actual**

In our proposal, we conducted a function point analysis in an attempt to gather what we had believed to be an accurate assessment of the time investment required for the whole group to bring the project to completion. However, as mentioned beforehand, the analysis was done with the assumption that we were to have approximately three weeks of building and testing time. As our team as a whole failed to appropriately prioritize planning and obtain our teams greenlight quickly, our build time was cut into massively. Despite the constricted schedule, the total points allotted for each sub-group ended up being not far off from accurate, surprisingly erring on the conservative side. Development time was especially eaten up by many small bugs and errors that we had not appropriately anticipated, leaving our estimations lower than the actual man-hours spent working.

Overall, there was a lack of experience in the team accurately assessing efforts for the future as it was a new topic for every single one of us. As a result, we were all pressed for time to meet deliverables, and certain aspects of the final product may have had a reduction in quality as a result. Although the planning and estimation can be classed as a failure by the team, we all learned the valuable lesson of proper prioritization and the importance of focusing on the little details, along with taking time occasionally to refocus on the overall goal. Reflecting back, if we had put more emphasis on these three seemingly minor points, much of the headache and stress regarding the crunched development time would have been reduced, and the net team efficiency greatly increased.

**5. 1 Appendix 1: Imaging Graph Data Structure Class**

1. **import** dt\_config as conf
3. **class** Graph:
4. **def** \_\_init\_\_(self,data):
5. self.d = data
6. #set to not initialized because the tesseract might return an empty string
7. self.name = 'Not initialized'
8. self.children = [] #holds coordinates of child
9. self.children\_label = [] #holds string of edge to children node
10. self.children\_label\_loc = [] #holds location of where the label to the edge to the node can be found
11. self.node\_type = -1 #set to conf.val\_XXX\_node
12. self.prob = "null" #probability associated with node
13. **def** add\_child(self,g):
14. self.children.append(g)
15. self.children\_label\_loc.append(None)
16. self.children\_label.append(None)
17. **def** add\_name(self,n):
18. self.name = n
19. **def** add\_edge\_label(self,index,s):
20. self.children\_label[index] = s
22. **def** add\_node\_type(self,node\_t):
23. self.node\_type = node\_t
25. **def** set\_prob(self,p):
26. self.prob = p
28. **def** print\_tree\_all(self):
29. #print parent data
30. **print**('\nParent ::',self.d,'label =',self.name)
31. **if** len(self.children) > 0:
32. **print**('children ::')
33. **for** i **in** range(len(self.children)):
34. **print**('child{}:'.format(i), self.children[i].d,'label = {}, edge = {}'.format(self.children[i].name,self.children\_label[i]))
35. **for** i **in** range(len(self.children)):
36. #if child has children, recurse on children
37. **if** len(self.children[i].children) > 0:
38. self.children[i].print\_tree\_all()
40. **def** print\_tree\_edge\_loc(self):
41. #print parent data
42. **print**('\nParent ::',self.d)
43. **if** len(self.children) > 0:
44. **print**('children ::')
45. **for** i **in** range(len(self.children)):
46. **print**('child{}:'.format(i), self.children[i].d,'label = {}, edge\_label\_loc = {}'.format(self.children[i].name,self.children\_label\_loc[i]))
47. **for** i **in** range(len(self.children)):
48. #if child has children, recurse on children
49. **if** len(self.children[i].children) > 0:
50. self.children[i].print\_tree\_edge\_loc()
51. #assumes all edges and nodes have labels
52. **def** print\_tree\_pretty(self):
53. **print**('\nParent ::',self.name)
54. **if** len(self.children) > 0:
55. **print**('children ::')
56. **for** i **in** range(len(self.children)):
57. **print**('child{}: {},'.format(i,self.children[i].name),'edge:',self.children\_label[i])
58. **for** i **in** range(len(self.children)):
59. #if child has children, recurse on children
60. **if** len(self.children[i].children) > 0:
61. self.children[i].print\_tree\_pretty()
63. **def** print\_node\_types(self):
64. **print**('\nParent ::',self.name,'node\_type =',conf.node\_color\_dict[self.node\_type])
65. **if** len(self.children) > 0:
66. **print**('children ::')
67. **for** i **in** range(len(self.children)):
68. **print**('child{}: {},'.format(i,self.children[i].name),'node\_type:',conf.node\_color\_dict[self.children[i].node\_type],
69. 'probability =',self.children[i].prob)
70. **for** i **in** range(len(self.children)):
71. #if child has children, recurse on children
72. **if** len(self.children[i].children) > 0:
73. self.children[i].print\_node\_types()