GENERAL IDEA FOR PAPER

When writing up actual paper on this:

* Have various implementations that work best in different cases
  + For instance, in the case of when one adds a whole graph at once, and specifically, the incidence numbers of the nodes do not change over time, one should use an implementation that does not include the Adjacency Matrix as a class variable, but rather construct it once right before calculating the Near-Isomorphism distance between this graph and another.
  + However, if our graphs change over time, and we want to calculate Near-Isomorphism distances between different subgraphs in our graph, this means that nodes and edges may be added gradually, so specifically, this means that a certain node’s incidence number may change over time, and more generally this means that our adjacency matrix will change over time. In this case, since we are calculating Near-Isomorphism Distance many times we want to avoid the duplicated work of creating the Adjacency Matrices of our subgraphs over and over. Rather we should keep the adjacency matrix of the whole graph as a class variable that has rows and columns added over time (as nodes are added) and the specific entries updated over time (as edges are added). Note: as edges are added, the incidence number of nodes may change, and since we want the nodes sorted in order of incidence number, we may need to change the ordering of certain rows and columns in our adjacency matrix as the incidence numbers of the nodes associated with each row index change. (NOTE: I just thought of this, since may want to avoid time associated with moving all entries over in our 2D array if delete node or add node, may be good to use a sort of 2D linked list-like structure, which I have actually seen somewhere…. I need to look up how this was done again, it was really cool—I remembered what the thing I saw is called! A “Skip List”!). Also, on a separate note, it would be useful to write a method that, for a subset of the nodes in the graph, returns the adjacency matrix of the induced subgraph of these nodes (with the row and column indices of the entries corresponding to the nodes sorted in decreasing order of incidence number…. If taking from class variable Adjacency Matrix, they will already be in order, we just need to extract the relevant rows and columns).

NON MACHINE LEARNING IDEAS FOR NEAR GRAPH ISOMORPHISM PROBLEM

NOTE: I will start to implement all these ideas soon and will upload code to Github (first in Python, then will translate to Java)! I just wanted to get these ideas up as I thought of them if it would help influence a potential hiring decision!

Ideas for distance metrics between graphs:

My favorite (This distance metric accounts for which vertices are connected to which in a meaningful way):

For Graph #1 and Graph #2:

* Sort the list of nodes by incidence number
* If there are chains of nodes in this sorted list could also make duplicate lists for different reorderings within these chains (within these chains that have the same incidence number could try all reorderings, or if wanted easier problem could sort within these chains by number of incident edges of a certain type, testing could determine which edge type is most useful to sort by within these chains)

Then for Graphs #1 and #2:

* Construct Adjacency Matrix (Note: this matrix would not only have an integer value at each slot, should have a tuple at each slot of the form: (signed edge weight, edge type))
* NOTE: the indices of verts for these adjacency matrices will use the ordering of nodes found in the above algorithm—in this way, we have a reasonable ordering for each node list that, if our graphs are isomorphic (resp. near-isomorphic), will be more likely to have us comparing the “same” vertices in each graph (same meaning if F is an isomorphism between G1 and G2 the row (resp. column) index of x and F(x) in Adj\_Matrix\_1 and Adj\_Matrix\_2 (respectively) would be the same (or likely the same in case of NEAR isomorphism))

THEN:

* Measure the distance between Graph 1 and Graph 2 by a distance metric on these Adjacency Matrices. VERY IMPORTANT NOTE: a distance metric defined based on these Adjacency Matrices is preferable to one like the one I will describe in the next section because it takes into account WHICH vertices are connected to WHICH vertices in a reasonable way. (Our method of choosing an ordering of the nodes for each adjacency matrix assures that if the graphs are isomorphic there will be a lower distance between them than there would be if they were not).

Possible distance metric (Note: will come up with more!): If one Adj Matrix is smaller than the other (will happen if the number of nodes in Graph1 is different than the number of nodes in Graph2), make the smaller Adj matrix the same size as the larger one by adding rows and columns.   
( NOTE: my idea for sorting by incidence number then trying all reorderings

within chains of verts w same incidence number works perfectly

(read: returns a distance of zero for isomorphic graphs) but might return too

high a distance for near-isomorphic graphs with different number of nodes

THEREFORE, would want to enlargen the smaller adj matrix in the "right way"

meaning in this case we want the empty rows in the adj matrix for the smaller graph

to coincide with where the verts in the larger graph were deleted

EG do NOT want to just add rows and cols at end

RELATED IDEA: could try defining the edge weight distance between a null node and it's

corresponding node in the other graph to be different constants then test and use

the one that gives us the right distance for our near isomorphism problem

)  
Then for each entry in Adj Matrix 1 and Adj Matrix 2 define three distance metrics between those elements (which are tuples of the form (signed edge weight, edge type)) as follows.

* + - We will have an edge weight distance metric defined as follows: dist\_edge\_weights(Graph1, Graph2) = Sum\_ (over i,j)(Adj\_matrix\_1[i][j][0]-Adj\_matrix\_2[i][j][0])\*\*2.
    - We also define an edge type metric as follows: dist\_edge\_type(Graph1,Graph2) = Sum\_(over i,j) (Adj\_matrix\_1[i][j][1] == Adj\_matrix\_2[i][j][1])  
      i.e. sum over the entries in the adjacency matrices a value of 0 if the edge types for those respective entries are not equal and 1 if they are
    - …More to come!

Completely separate idea:

Would like to make a graph visualization tool! This would be much more low priority, but once I started to get a handle on the Near Graph Isomorpshim problem would like to do this too!

MACHINE LEARNING IDEAS FOR NEAR GRAPH ISOMORPHISM PROBLEM

General Idea: If had list of all web pages taken down on dark web, and had list of new pages up which were made by the same group as those taken down. This could be a multi-class classification problem. Specifically, could classify the types of graphs generated from processing the language in these webpages up to near isomorphism using Machine Learning. It seems to me that this would be a supervised learning multi-class classification problem. Specifically, for each new webpage seen we want to figure out if its graph is near isomorphic to a graph from one of the web pages taken down, and if so which. If we have some cases of web page matches found manually be analysts (and their respective graphs), we would have a training set for our supervised learning problem. Of course the likely problem is that we would have too little training data for the number of classes in our multi-class classification problem. (I assume that the number of groups on the dark web (which would be the number of classes) is a very large number). One work around I can think of, which is not ideal, but still might work would be to generate “fake” training data. Specifically, one could randomly generate graphs that generally resemble those generated from webpages on the dark web, and Machine Learn on those. (I know this is not ideal in that the training set is supposed to resemble the testing set in a Machine Learning Problem, but in a lack of viable training examples, one must invent!) So, using this “fake” training set one could try out different features of the graphs that might be used in this multi-class classification problem. (Much testing would be needed to determine which are most useful, however…) Some might be the number of vertices in the graph, the number of edges in the graph, the number of edges of each type in the graph (different edge types could be “same page” or “same paragraph” as I understand among others), the number of cycles in the graph, the number of cycles of each size, the number of cycles using each edge type, number of cliques, number of cliques of each size, number of cliques using only a certain edge type for each edge type, entries of the adjacency matrix as generated by above algorithm (could be weird, because adjacency matrices would have different sizes) (Note: calculating some of these features e.g. number of cycles or cliques might be more computation time that it would be worth, so could test to see if certain more “expensive” features improve accuracy enough to merit including. I am just listing everything that comes to mind!)

Note: for this training set, one would need to either manually mark graphs as pairwise near isomorphic or not. OR could find satisfactory distances between graphs (using different distance metric and specific distances for each) that decide (as determined by our actual training set marked by analysts as a match) whether the two graphs are near isomorphic. In this way, we could automate the marking of the graphs as pairwise near-isomorphic or not. So then why use this machine learning approach? Might improve on accuracy and just an idea to try. It’s possible that the features I’ve chosen to “encorporate” so to speak into my distance metric aren’t the most useful in determining near-isomorphism or not, a Machine Learning approach might be a good way to quickly try out different features to see which are most useful. If find that certain features are more useful (read: give best test set accuracy), might modify the distance metric to encorporate these features. So in summary, this Machine Learning approach would most likely be a means to an end for me, the end being coming up with the most useful distance metric between graphs.