Graph Isomorphism and the Subgraph Isomorphism Problem

A Technical Report of Project Results

Brandon Rodriguez
04-21-18

Abstract

Graph Isomorphism is the comparison of one graph against another to see if they match. Subgraph Isomorphism is the comparison of one graph against another to check for a partial match.

This algorithm attempts to solve the subgraph problem by focusing on edge constraints early on, thus hopefully reducing the amount of work required later.

The algorithm itself was acquired from a 2013 published research paper from BMC Bioinformatics[1].

For my implementation, please see my github repository as noted in the “Code Repository” section.

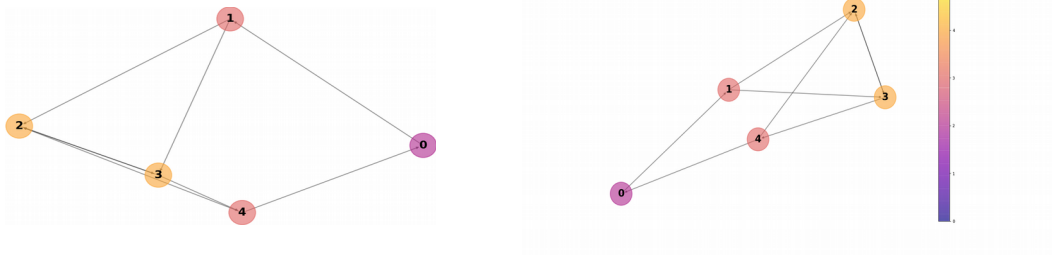
Introduction and Target Problem

The problem itself is fairly simple. If given two graphs, compare one graph against the other to see if they match. This is a Graph Isomorphism check.

An extension of this problem, and the problem examined in this project, is Subgraph Isomorphism. Essentially, if given two graphs, compare one against another to see if there is a partial match between any nodes. This can then be expanded to check for the entirety of one graph within any given subsection of the other.

For example, in the below figure, one may wish to check if the left graph is comparable or equal to the graph on the right. In this case, the two graphs have the same number of nodes with the same edge connections, so as long as the data contained is also the same, then they will be a full match (Figures acquired from *slide 3*[2]).

If looking for a subgraph, then in the below example, you would simply check for as many matching nodes as possible, instead of checking for a 100% match.



Intellectual Merit

The specific algorithm has two major parts, called “Greatest Constraints First” and “Matching”.

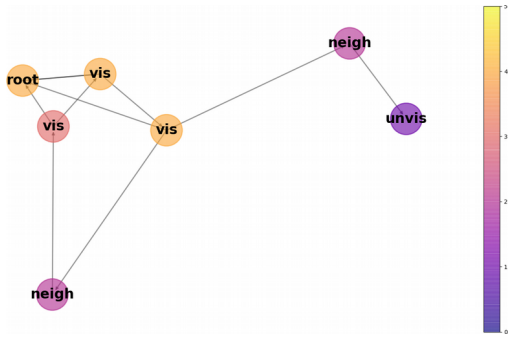
Greatest Constraints First

The “Greatest Constraints First” function only handles one graph at a time. For input, it requires a list of all nodes in the graph, ordered by number of edge connections (For an example of edge connection evaluation, see above or below figures. Both have nodes colored by edge counts). The function iterates keeps track of “previously iterated nodes” in the form of a second list. It then through all nodes in the edge-sorted list. For each node it looks at all connecting edges, giving the edge one of three possible attributes:

Visitor – Contains a direct edge connection to one or more nodes within the “previously visited” list.
Neighbor – Contains an indirect edge connection with exactly one node between.
Unvisited – Neither of the above two apply.

Once all nodes have been given an attribute, a “ranking list” is created, where all nodes are added in order of [visitor, neighbor, unvisited]. This continues for every node until all nodes have acquired a ranking list.

In the below figure, we see an example of one iteration for greatest constraints (Figures acquired from slide 12[2]).



Matching

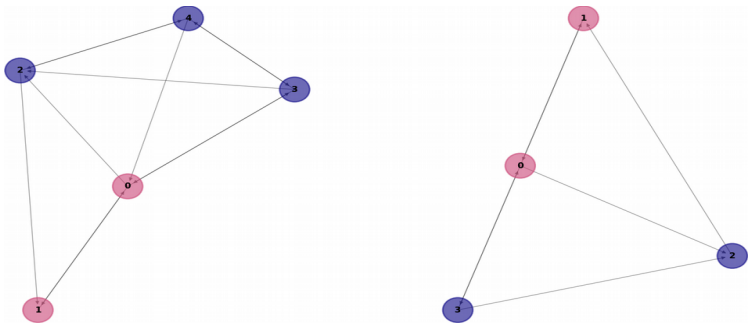
Next, the “Matching” portion of the algorithm takes place. In a list of “match pairs” is held. Nodes in the graphs are examined based on the relative ranking list values. Once a first match is found, then the lists can be used as a guide to “prune away” infeasible paths. To actually determine a match, four comparison criteria must be met between any two arbitrary comparison nodes g and h:

1) Neither g or h can be already included in any of the already located matching pairs.
2) The data held by g must match the data held by h.
3) The edge count of h must be greater than or equal to the edge count of g. Note that this is technically optional in regards to matching. However, for many “bad match” comparisons, implementing it saves significant computational costs from requirement 4.
4) The nodes at the connecting edges of g must match the nodes at the connecting edges of h.

If any above check fails, then the nodes are safely considered to not match and any following checks for those two nodes can be skipped. The result should be a list of all valid matching node pairs.

[To my knowledge, this matching section should also include a “pruning” function, but I don’t entirely understand how it works. See “**Proposed Improvements in Algorithm**”/”**Discussion and Conclusions**” for elaboration.]

In the below figure, we see an example of matching. The two graphs are identical, except that the right graph has one single node removed. In this instance, the associated edges required a 60% match or higher to be considered a node match. Red means that the nodes returned as a match. Blue means they returned as a miss. (Figures acquired from slide 13[2]).



Time Complexity

Last, the subgraph problem is considered to be a NP-Complete problem[3]. However, to be honest, I'm not sure how to fully calculate the exact run time complexity. My attempts to determine it just now resulted in a polynomial time, which surely can't be correct for a NP-complete problem.

Space Complexity

As far as space complexity, it would simplify to the cost of the size for the two graphs. To elaborate, it would amount to O(Graph_1(n * m) + Graph_2(n * m) + Extra_Graph_1_Structures + Extra_Graph_2_Structures), where n is the number of nodes and m is the number of edges in each respective graph.

For each graph, the "extra_structures" would be a list of all nodes ordered by edge length O(n), the ranking list for each graph O(n^2), and the list of matching nodes between the two graphs O(Graph_1(n) + Graph_2(n)). All of these extra list structures should refer to the node objects already present in the graphs rather than creating new node objects. Thus the lists would all be relatively small and each n would be referring to only addresses of the given nodes, not a duplicate of the nodes. This address data would clearly be smaller than the nodes in all accounts, except possibly for the single case of all nodes containing empty data and no nodes having any connections. Even in this instance, it's unlikely that a memory address would take up more space than the object itself. Thus, the list space would likely be negligible and reduce down to O(1), in comparison to the size of the node objects referred to in the graph.

Thus, overall, you'd be left with O(Graph_1(n * m) + Graph_2(n * m) + O(1)). In a complete graph, this would equate to O(Graph_1(n^2) + Graph_2(n^2) + O(1)).

Implementation

For my implementation, I created my own Graph and Node data structures. This allowed me to store values in ways that made sense to me, such as storing a graph's nodes inside a dictionary, as opposed to the research paper which (as far as I can tell) stored the graph's nodes in a list. This gave me a node access time of O(1) as opposed to a time of O(n). Furthermore, by creating my own structures, I knew how the graphs were created and could easily create additional variables specific to this algorithm, as needed.

However, when I first created these structures, I did not account for having to randomly generate my own data. To create a graph, you would pass it values for each node and the graph would build up node by node. See "**Performance Evaluation**" for why this is significant.

I also note that, while I followed the paper's algorithm to the best of my ability, I had two issues. First, the paper mentions determining a parent node for each node which is passed into the Matching section of the algorithm. However, they don't seem to mention these parent nodes again, so I'm utterly baffled as to what their purpose is.

Second, until rereading the paper just now, I was under the impression that “pruning” was part of checking for a match between two nodes, and had no impact on anything else. However, I’m now under the impression that this is something else entirely, and comes in to play when a match fails. Essentially, if at least one node match has already been found, then on subsequent node match failures, the algorithm can (somehow) use the Greatest Constraints ranking lists to “prune” away additional nodes from the search subspace. And this seems to possibly be a standard practice for Isomorphism algorithms in general? The paper doesn’t seem to elaborate very much on this or how its done so it’s not very clear to me. See “**Proposed Improvements in the Algorithm**” for elaboration.

The actual code for this project was written in Python, and a link to the github project can be found in “**Project Code Repository**”. Note that this code was written for functionality. As such, there isn’t a user interface. To change the output, the code itself needs to be changed. For the most part, the “core” functions are located in main.py under the “main” function, and simply need to be uncommented as desired.

Performance Evaluation

From what I can tell, the datasets used in the paper are no longer available (at least not at the same locations they were originally linked to). Even if they were available, I’m not sure I would have understood them, since they seemed to be of complex structures, such as proteins and other biochemical elements.

Due to this, I ended up having to just create my own datasets, which were randomly generated nodes and edges, based on varying parameters specified in code. To ensure that my two graphs would almost always have at least one match, I would create the “comparison/second” graph by copying all node values from the first graph, including the associated edges. At this point the two graphs would be identical other than graph object location within the computer’s memory.

I then proceeded to remove nodes from the second graph, based on the below described parameters, so that one graph would be a subgraph of the other. For testing/experimenting purposes, I ensured that some parameter combinations (described below) would have an extremely low match rate while others would have an extremely high match rate. IE, based on the parameters, node matches would approach either 0 or the total node count, in the extreme cases. This allowed me to gather data from all kinds of graphs, including sparse, dense, and in-between graphs with varying amounts of node matches.

Ultimately, I ended up having 5 different sections of 100, 200, 300, 400, and 500 nodes. For each section, there were 3 general groupings of edge connections. For each grouping, I had 4 different “node removal” types. And then for each type, there were 2 different “edge strictness” variations.

Edge Connection groupings were as follows:

- Sparse – Under 33% edge completeness for all nodes.
- Middle Ground – Between 33% and 66% edge completeness for all nodes.
- Dense – 66% or more edge completeness for all nodes.

Node Removal types were as follows:

- None – No removal. Second graph was identical to the first.
- Few – 33% or less nodes were randomly removed from the second graph.
- Some – Between 33% and 66% of nodes were randomly removed from the second graph.
- Many – Over 66% of nodes were randomly removed from graph.

Edge Strictness Variations:

- Loose – Edges needed a 33% or higher match rate to be considered an overall match.

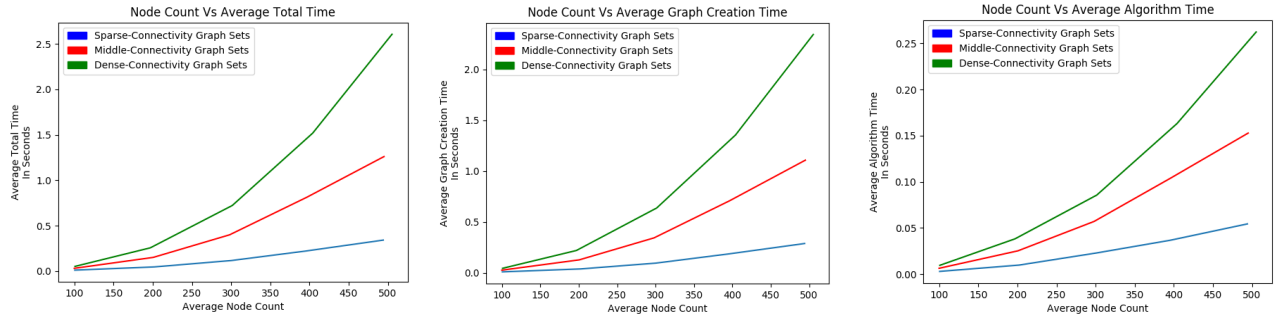
Strict – Edges needed a 66% or higher match rate to be considered an overall match.

To gather results, I iterated through many “sets”. Each set was a unique combination of the above parameter types, and ultimately, every single possible parameter type combination was examined for every given node count. For each set, 100 full iterations were ran, and full results were logged for each one. Then an average of all set results were taken, which was used as the “set’s overall results”. It was these overall results that was used to actually compare one set against another.

In regards to my actual performance, graph creation seemed to create the bound of how much data I was able to process. I had surprisingly high graph creation times when compared to algorithm run times. Either that or I somehow messed up in my saving/computing of timestamp data for my results, but I feel like that’s less likely.

I’m fairly certain this is due to how I generated my data. Currently, my program copies the original graph, node by node, and then later iterates through a second time to remove nodes. If I were to do this again, I would instead update my random graph generation to create both graphs at once. Furthermore, I would move have the node creation process determine if a node will be removed or not, instead of doing that separately after the entire graph is made. Last, I would double check the code for my Graph object, because with such abysmal times, there is likely optimization to be found there as well. I suppose I could even attempt to move the random generation element into the Graph logic, for possibly more performance gains.

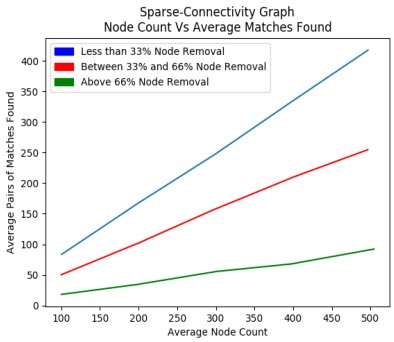
In the figures below, I plot the “total time”, “graph creation time”, and “algorithm processing time” for my datasets. In each, I plot the three graph-connectivity types separately, and I believe this instance used the “loose” category for edge strictness (Figures acquired from slide 16[2]).

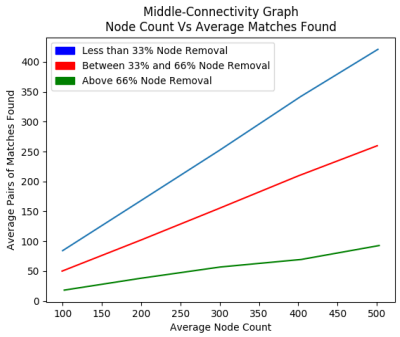


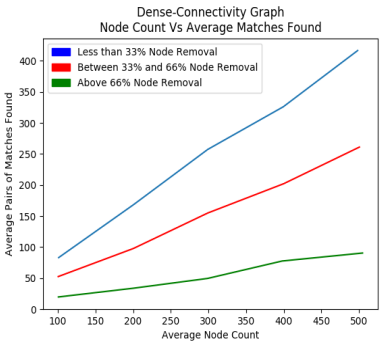
Last, I wanted to check accuracy of my implementation. First, I manually created some (small) graphs by hand and compared them against each other, to ensure that the implementation at least worked as expected on a small scale.

Next, I used my random datasets to check the average match counts between graphs of sparse, dense, and in-between connectivity. My assumption was that, if properly implemented, then the number average matches should stay the same between all graph types, regardless of the number of associated edges per node. The only thing that theoretically should change the number of matches in my datasets should be the number of nodes removed, overall.

In the figures below, I plot the accuracy of sparse, in-between, and densely connected graphs with varying amounts of removed nodes (Figures acquired from slide 17[2]). Again, I believe these plots used the “loose” paremeter for edge strictness.







Broader Impacts

There are many things Isomorphism can be used for, and not all of it is immediately obvious. For example, wikipedia claims that Graph Isomorphism is used for computer vision and pattern recognition, identifying unknown compounds against a chemical database, data mining, and verifying circuit design in electrical engineering[4]. A quick google search will lead to other links which confirm these and more[5]. But how exactly does it work?

Computer Vision and Image Recognition

For one, it's Isomorphism is used in "convolutional neural networks" as part of machine image recognition[6]. In this, images are fed into the network to "train" it to recognize a type of image. In short, it appears that each pixel ends up being a node in a graph, so the resulting graph has nodes equal to the total pixels. Then, due to how images work, edges are created to other nodes based on the relative proximity of those pixels to the node's pixel within the image. This is because it's assumed that any given two nearby pixels are far more likely to be related than, say, two pixels on the exact opposite sides of the image.

Once a network has a graph and is trained on what to look for, it's fed additional images which it compares to its training set, to see if it contains the desired object. Which at that point, it has two graphs (containing pixel data) and is comparing node by node to check for similarities/matches. Obviously there's a lot more to it which greatly increases the complexity, but at the core, that's Graph Isomorphism holding it together.

Others

Logically, given the above example, it's not that difficult to see how it could apply to chemical substance comparison or circuit verification, or really any kind of search/comparison which uses data more sophisticated than words[5]. For example, in the instance of chemical substances, one could logically assume that each atom in the substance molecule could be mapped to a node. Then atoms which are connected could be have edges mapped together. Then to compare, you simply hold a database of these, and run an Isomorphism check any time you have a new substance to identify. Or for a circuit, you could hypothetically use computer vision to teach a program what a "correct" circuit/circuit type looks like. Then compare it against newly created circuits to ensure they were created/designed. (Note, these are hypothetical, and logically deduced from how the elements in the above paragraphs logically work.)

In short, it seems that Graph Isomorphism (and thus by extension, also Subgraph Isomorphism, depending on the topic) is incredibly useful for anything type of searching/comparison where regex is not sufficient due to the type of data.

Proposed Improvement in the Algorithm

Dictionaries

From what I've observed, the authors of the paper tend to use lists to store most sets of values. While this is fine if they plan on iterating through all values anyway (such as the list of nodes sorted by edge count, for greatest constraints), it's not so ideal in instances when they want to check if a specific value exists within the set of values. In these instances, I would switch the data structure to a dictionary, which to my understanding, has a lookup time of O(1), saving a full O(n) time per look up.

Graph Matching Ordering

Furthermore, the paper seems to suggest that "constraints are deduced only from the pattern graph and not from the target graph" (page 8, first column, second to last paragraph[1]) before running the matching section of the algorithm. Perhaps I naively misunderstand the algorithm, but I feel like the overall matching process would benefit from Greatest Constraints running on even just the first node of the target graph. Or worst case scenario, at least creating a list of the target graph's nodes, based on edge connection count. Then, should there be many matches, I feel like these changes would make the matching nodes more likely to line up within the respective lists (relatively speaking) and thus find consecutive matches faster. If there are no matches, then there is no overall benefit, but the costs of these would be mostly negligible when compared to the possible benefit.

In the paper's suggestions, if every node has a match, it would result in O(n^2) overall lookup time if nodes in the second graph are coincidentally ordered to be the exact reverse of the first graph. But obviously, there would be no "second graph sorting/ordering" time so the end result lookup time would be O(n^2).

Going against the paper's suggestion and sorting by edge count on the second graph would only take an additional, single instance of O(n). Meanwhile, it would ensure that, if every node has a match, then the search between two graphs would effectively be extremely close to O(1), as the node lists will always be ordered the same way for both graphs, and on match. If you go a step further and remove the matched nodes from both lists upon discovery, then theoretically, if all nodes have a match, then the two nodes at the very front of the list should always be a match, all the way up until both lists are empty. This would result in an end result lookup time of O(n) in the case of all nodes matching, or still O(n^2) in the event of no matches.

Now obviously, the above big-O times do not account for the time necessary to compare data/edges between two nodes, as said events would be unchanged, regardless. The above proposal would only affect the computation times of actively looking for nodes to compare against each other.

Improvements Specific to my Implementation

Although, there aren't any other suggestions I feel I could make for the paper's algorithm, I do feel like there are some I could make for my own implementation. For example, I feel like I misunderstood the "pruning" part of the algorithm (page 8, first column, second to last paragraph[1]). It's only mentioned briefly, and when I first read it, I thought it was simply referring to basic node comparison in the "Matching" section. However, rereading the research paper now, I feel like "pruning" is meant to be something else entirely. It seems to be something that most Isomorphism algorithms tend to do. Basically, on any node match failure, it looks at related edges in a way such that it somehow determines other paths which aren't worth investigating and "prunes" them out of possible future search matches. Meanwhile, my implementation only uses this logic to tell if the specific node it's checking against is a match, but then it continues to iterate through all other nodes regardless.

Unfortunately, the paper doesn't go into enough detail about it either (at least for my understanding). Or if it does, it did so in a way that went way over my head. For future improvements, I would definitely look into more

sources to try and figure out exactly how this pruning logic works. While I'm at it, I would likely look into other Isomorphism algorithms in general, so that I can better understand the generic Isomorphism problem and perhaps gain insight into how to improve my implementation.

Discussion and Conclusions

Conclusions

Overall, this was an interesting project. My time results concern me a little bit, as I feel like my graph creation implementation has to be pretty bad in order to take so much more time than a supposed NP-Complete problem. However, looking at some issues I had, such as "not knowing what parent nodes are used for", and "not fully understanding pruning", I can't help but wonder if my timing results are in part due to not fully understanding the algorithm. Perhaps my implementation is a somewhat naive approach but correcting this would greatly increase computational time. Or perhaps my graph creation just really is that inefficient in its current state. Unfortunately, I simply did not have enough time to determine the cause and correct for it.

As far as my actual implementation of the algorithm, it appeared to be fairly accurate, at least in that it returned consistent results and seemed to correctly find matches. I suppose, the next step in confirming accuracy is to manually create a larger graphs (such as one with 100+ nodes) and try the algorithm against that. Since I only had time to test my implementation against smaller manually-defined graphs, I suppose it's entirely possible that it breaks at some point after so many nodes/edges are added. If this is the case, then there's no way (that I can think of) to be able to tell through the random graph generation I used for my data.

Assuming that my implementation is indeed accurate, then I'm actually fairly happy with the time it took to compute. On average, it took 1/4th of a second in the worst case, and it currently doesn't even take advantage of any branch pruning. This means that it's almost certainly can be expanded to function for much larger graphs, so long as I either improve the graph creation time or simply find a valid dataset to use. For future experiments, I would want to look up alternative algorithms/papers with data I can actually directly compare against.

Other Notes

Last, I suppose it's worth noting that I spent a considerable amount of time trying to learn the nxgraph and matplotlib Python libraries. Due to having to present our data, one of my first thoughts was "I should make sure I have a way to quickly translate my data into an easy to present, visual format". However, implementing this took a lot longer than expected, and as such, unfortunately cut into my actual algorithm implementation time more than I had hoped it would. I'm not sure it was entirely worth it, given the short timespan we were given for this project. If we had even just a few more weeks, it might have been fine, but as it is now, my time might have been better spent exclusively working on the algorithm itself, instead of also spending so long working on a way to represent my results.

Cited References

1) ***Reference Research Paper –*** Vincenzo Bonnici , Rosalba Giugno , Alfredo Pulvirenti , Dennis Shasha , Alfredo Ferro, "A Subgraph Isomorphism Algorithm and its Application to Biochemical Data", https://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-14-S7-S13, 2013

2) ***Research Presentation Powerpoint and Associated Figures*** – Brandon Rodriguez, Powerpoint located at https://docs.google.com/presentation/d/1xKKcfbQlTewS2lX00D1-4tvnRIEUNj5gP5O8uUfnRUs/edit?usp=sharing, April 2018

All figures in the powerpoint and this report were created from the code in my project, located at https://github.com/brodriguez8774/CS4310_Project, April 2018

3) ***Complexity Definition of Problem*** – https://en.wikipedia.org/wiki/Subgraph_isomorphism_problem, Last accessed 04-21-18

4) ***Applications of Isomorphism*** – https://en.wikipedia.org/wiki/Graph_isomorphism_problem, Last accessed 04-21-18

5) ***Further Applications of Isomorphism*** – https://math.stackexchange.com/questions/120408/what-are-the-applications-of-the-isomorphic-graphs, Last accessed 04-24-18

6) ***Computer Image Recognition*** **-** https://www.upwork.com/hiring/data/how-image-recognition-works/, Last accessed 04-24-18

Project Code Repository

My implementation of the algorithm (as well as all code used to generate associated figures, create data, and handle results) can be located at https://github.com/brodriguez8774/CS4310_Project.

As stated above in "**Implementation**", this code was written for functionality. As such, there isn't a user interface. To change the output, the code itself needs to be changed. For the most part, the "core" functions are located in main.py under the "main" function, and simply need to be uncommented as desired.