Graph Isomorphism with Pathfinding Algorithms

Floris T. Breggeman

ABSTRACT

This paper presents exploratory research into the use of pathfinding algorithms to solve the graph isomorphism problem, where the pathfinding algorithms will be used to turn isomorphic graphs into isomorphic trees. A framework for testing such algorithms has been developed and several algorithms have been tested using this framework. The algorithms are proven to be in polynomial time, and the class of graphs on which they provide correct answers is discussed.

Keywords

Graph Isomorphism, Computational Complexity, Pathfinding, Algorithm

1. INTRODUCTION

The graph isomorphism problem has been researched by mathematicians and computer scientists alike for many years. The complexity of this problem is still open and of great interest to the graph theory and computational complexity communities, as it could have major consequences for the polynomial hierarchy and give insight into the famous P=NP question. Current state-of-the-art algorithms for general graph isomorphism rely on colouring the graph, possibly supplemented by creating a search tree [10]. On the other hand, pathfinding algorithms are well-known and almost always polynomial-time algorithms for the purpose of finding a path between two points in a graph. This paper proposes research into algorithms which omit the colouring step, and start by using a pathfinding algorithm to create a tree, where isomorphic graphs would result in isomorphic trees (and non-isomorphic graphs would result in non-isomorphic trees). Since trees can be transformed into a canonical form in linear time [1], such algorithms would be capable of generating a canonical notation of graph isomorphisms, with a complexity dependent on the pathfinding algorithm.

This research explores how well pathfinding algorithm can be used to solve graph isomorphism, as well as which algorithm is best used, what the complexity of such an algorithm is, and what graphs are best suited to the algorithm. It provides a new way to solve the isomorphism of a certain class of graphs in polynomial time, thereby continuing on previous research by the joint author [3], which did not fully explore the possibilities of the approach.

2. BACKGROUND

33rd Twente Student Conference on IT 2020-07-03, Enschede, The Netherlands

2.1 Formal Definitions and Problem Specification

A graph consists of atomic entities called nodes (often called vertices), and edges, which join nodes to other nodes. Isomorphic graphs have exactly the same structure but may differ in the labels or layout of their vertices and edges. More formally, two graphs $G_1 = (N_1, E_1)$ and $G_2 = (N_2, E_2)$ are isomorphic if there exists a one-to-one mapping $f : N_1 \to N_2$ of their nodes such that $\forall uv \in E_1[uv \in E_1 \leftrightarrow f(u)f(v) \in E_2]$. The Graph Isomorphism problem is determining whether an isomorphism exists between two graphs.

In this paper, the definition of the word tree is used as is common in computer science: a directed, acyclic, layered graph with a single root node (i.e. a node with no incoming edges), where each edge is directed from a parent node to a child node and each child has at most one parent.

This research focuses on the isomorphism of simple graphs; i.e. undirected graphs where there can be at most one edge between two nodes and an edge cannot join a node to itself. Furthermore, it discounts the possibility of labeled graphs (where nodes or edges can have labels), but adapting the presented algorithms to work on labeled graphs should not be too difficult.

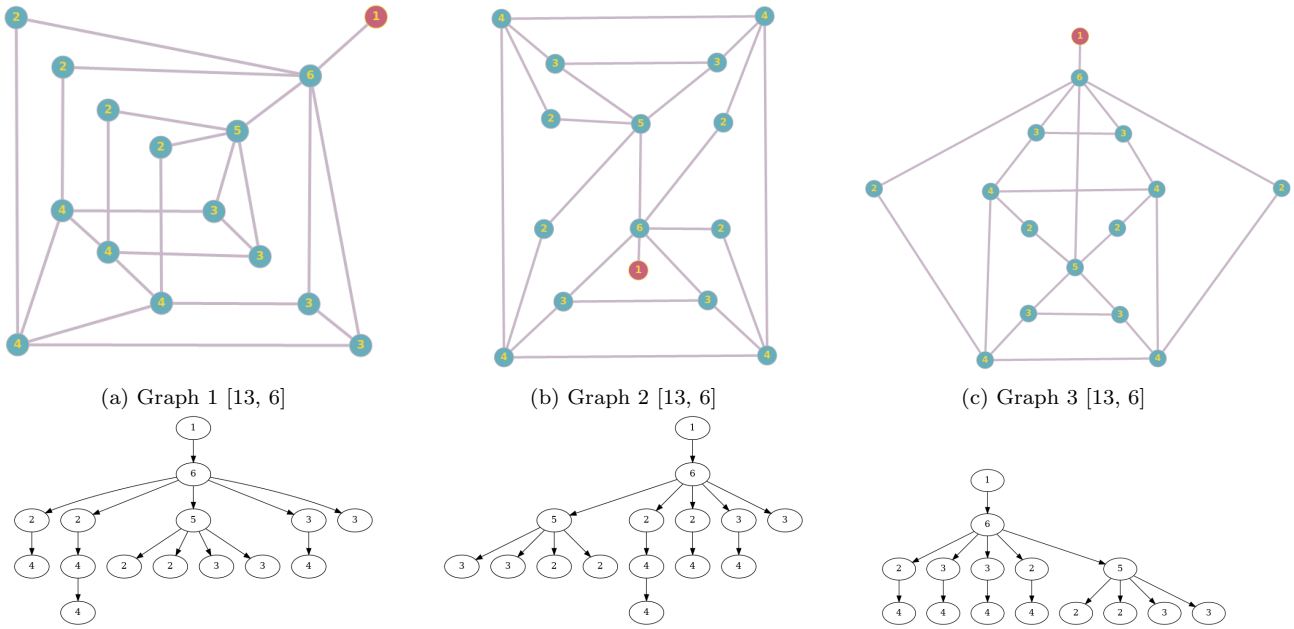
2.2 AOEU

This research continues on an algorithm jointly developed by the author during a university project on graph isomorphism, the so-called AOEU algorithm [3]. The AOEU algorithm requires a function called graph_to_tree. Given a graph and a start node, this function will create a tree that represents the graph as seen from the start node, such that isomorphic graphs will produce isomorphic trees, if an isomorphically corresponding start node is used. This algorithm is further discussed in section 2.3, and its exact implementation is the subject of a research question. The AOEU algorithm compares two graphs by using a graph_to_tree algorithm to generate a tree for every individual node in both graphs, which results in two unordered sets of trees. It is then checked that every tree in one set has an isomorphic counterpart in the other, using an existing tree isomorphism algorithm.

2.3 graph_to_tree

As mentioned previously, the exact implementation of the graph_to_tree algorithm is the subject of a research question. This section gives a small overview of a basic breadth-first implementation to illustrate how such an algorithm would work. This version is the simplest variant implemented in the previous research.

Algorithm 1 performs what is essentially a breadth-first search on the graph. The mapping *values* is used as a cost function; it contains the degrees of all parent nodes and the node itself. This specific version prioritises nodes closer to the start node, with lower degrees, making it a breadth-first search algorithm. The algorithm can be seen in action in figure 1, where the red node is the start node. Note that when a node is added to the output tree, the



(a) Graph 1 [13, 6] (b) Graph 2 [13, 6] (c) Graph 3 [13, 6]

(d) Tree for graph 1 (e) Tree for graph 2 (f) Tree for graph 3

Figure 1: an example of the graph-to-tree algorithm. The root node of a tree corresponds to the red node in a graph.

Algorithm 1: graph_to_tree

```
function graph_to_tree(Graph, start_node):
 result: a Tree with start_node as root;
 values: Mapping(node → list of integers);
 frontier ← {start_node};
 visited ← {start_node};
 values.add(start_node, [start_node.degree]);

 while there are nodes in frontier:
  best_node ← frontier_best(frontier, values);

  foreach neighbour of best_node:
   if neighbour not in visited:
    new_value ← values.get(best_node);
    new_value.append(neighbour.degree);
    values.add(neighbour, new_value);

    frontier.add(neighbour);

    add neighbour to result as child of best_node
      ↪ ;
    frontier.remove(best_node);

function frontier_best(frontier, values):
 return that node in frontier for which the
   ↪ value is:
  1. The shortest list.
  2. if nodes have list of equal length,
   ↪ return that one which is the first
   ↪ to have a lower number.
```

degree of the node is used as a label of the corresponding node in the tree. As their trees suggest, graphs 1 and 2 are isomorphic, while graph 3 is not.

2.3.1 *Ambiguous nodes*

Unfortunately, the algorithm as described above does not entirely work; in some graphs, it can occur that there are two routes of equal value to the same node. Take the graph of figure 2a. Starting at the topmost node, the generated tree would be as in figure 2b. As can be seen, the subgroup at the bottom of the graph has been bound to the left node of the square, but this could just have easily been the right node; as such, two isomorphic graphs could result in non-isomorphic trees.

Previous research identified two solutions to this problem, one of which will be expanded upon in this research. In figure 2c, the problematic node, and the nodes that are bound to it in the subgraph, have been copied to both possible positions in the tree. This approach solves the problem, but unfortunately also raises the complexity of the algorithm; in the worst case scenario (a grid-like structure) almost every node would be ambiguous, requiring an exponential amount of nodes in the tree compared to nodes in the graph. Figure 2d will be explained in section 3.2.

3. METHODOLOGY

3.1 The Framework

A new framework for testing specifically AOEU-type algorithms has been developed for this research [4]. To allow for more objective comparisons, both between variants of AOEU as well as with other GI algorithms, this framework has been written in C++. The framework contains all components of AOEU except graph_to_tree, but does provide an interface to easily test different graph_to_tree implementations. It also contains several debugging and testing tools. These components are described below.

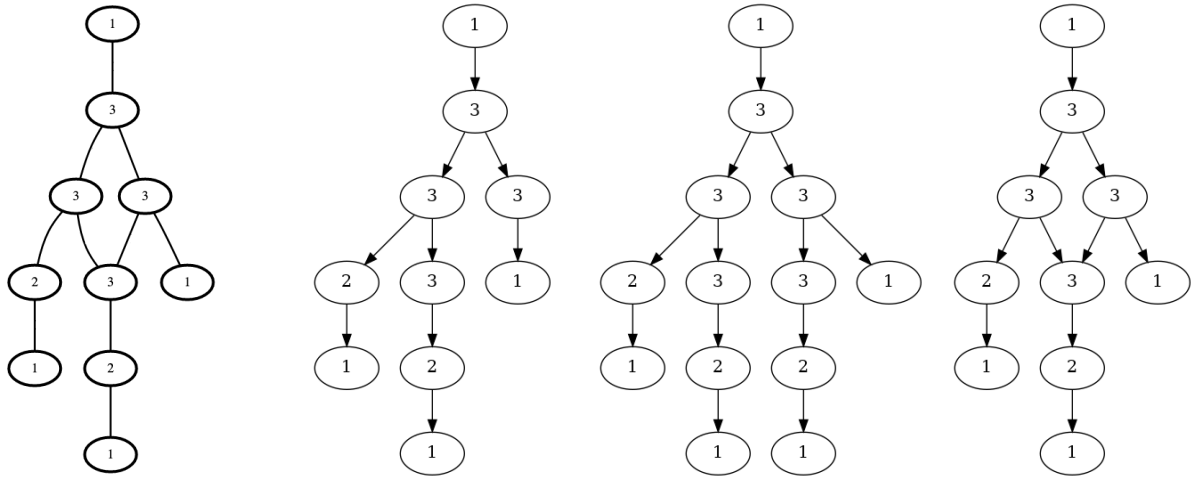
The framework also comes with a set of test graphs, taken from the Nauty-Traces website [11]. These are separate files, which the program can run upon. The framework contains 1972 readable graphs with 1000 nodes or less.

3.1.1 *Graph representation*

The graph files are in the text-based DIMACS format [9], but in order to run AOEU on them, they must be converted to a format that is easier to operate on. In the framework, this is handled by the `Graph` and `Node` classes. The `Graph` class simply has a list of all the `Node`s. The `Node` has a unique integer id, and a list of its neighbouring `Node`s.

The `Graph` class can be constructed by simply passing it a DIMACS file, which it will then read in. It can also be written to both DIMACS and dot format; the former for reusing an altered graph, the latter for visualising the graph using Graphviz [8].

Finally, the `Graph` class has a shuffle function, which ran-



(a) A graph with an ambiguous node

(b) The regular tree for this graph

(c) The tree with node copying

(d) The shadowtree

Figure 2: An ambiguous node and possible solutions

domly reorders all lists and reassigns the unique node ids. The resulting graph is guaranteed to be isomorphic. This is a useful feature, as the test graphs contain many graphs which are nonisomorphic, but not many which are isomorphic.

3.1.2 Tree representation

The framework also uses its own representation of trees, the `Tree` class. Similar to a `Graph`, the `Tree` class simply holds a collection of `TreeNode`s, as well as a separate pointer to the root node. A `TreeNode` holds pointers to its child nodes, but also to its parent nodes; as such, it is in practice a directed graph. A `TreeNode` can have multiple parents; see section 3.2.

In order to compare trees, the AHU tree isomorphism algorithm developed by *Aho, Hopcroft and Ullman [1]* is used, henceforth referred to as the AHU algorithm. Some modifications have been made to this algorithm, as described in section 3.2.1.

3.1.3 Comparison of sets of trees

As mentioned in section 2.2, the AOEU algorithm needs to isomorphically compare two unordered sets of trees. Normally, the comparison of two unordered sets would be in quadratic time, but for these sets it would be in cubic time, as the comparison of two elements (i.e. tree isomorphism) is in linear time. In order to improve this complexity, a slightly different technique is used for set comparison, which uses the fact that the sets only contain trees.

First, a new tree with a blank root node is created. Then, all the trees are added as children of this root node. The resulting tree is an isomorphic representation of the set; two equal sets of isomorphic trees result in two isomorphic trees.

As such, the comparison is linearly dependent on the amount of nodes in all the trees; as the amount of nodes in a single tree is linear in the amount of nodes in the graph n, and the amount of trees is exactly n, the final comparison is in quadratic time. It is trivial that such a comparison would always return the same result as an element-by-element set comparison on a set of trees.

3.1.4 Run modes

As the framework is used for both developing and evaluating algorithms, it has several modes in which it can be ran. This section provides a brief overview. For more details, see the code documentation [4].

- **single**: Generate a single tree from a single node in a graph, and write it in dot format. Useful for debugging and illustrative purposes.
- **check**: Check the isomorphism of two graphs.
- **shuffle**: Check the isomorphism of a graph and a shuffled version of that graph.
- **fulltest**: The complete test; see section 3.4.

Additionally, the algorithm can be selected via a runtime argument, in order to compare different algorithms via the same program without recompiling.

3.2 Shadowtrees

As described in section 2.3.1, node copying has, in the worst case, an exponential memory complexity. When an algorithm that uses node copying is used on a graph that has this worst case, it will very quickly allocate memory until the memory is full; on the Linux kernel, which was used for these tests, the program will then be unapologetically killed. This caused a problem during testing, as no full test could be completed.

To solve this issue, the algorithm has been slightly altered. Instead of producing trees, it now produces graphs which are directed, acyclic, layered[1], and have a single root node, but in which a node can have more than one incoming edge; in other words, a tree where a node can have multiple parents. This paper proposes the term "shadowtrees" for such graphs, and will adopt it for the sake of convenience.

An example of a shadowtree can be seen in figure 2d, which is the shadowtree for figure 2a. As shown, the ambiguous node is no longer duplicated; instead, it is shared between parents. While the performance improvement for this graph does not seem to be that great, consider a situation where another similarly ambiguous node would be joined to the original ambiguous node. In such a situation, the second layer of ambiguous nodes would have to be duplicated twice, being in the total graph four times; this behaviour exponentially increases memory usage. Using shadowtrees, both ambiguous nodes are only in the graph

[1] In this context, meaning that all edges go from one layer to the layer below

once; this avoids this exponentiality, thereby solving the memory issues.

3.2.1 Comparison of shadowtrees

The AHU algorithm can be easily adapted to work with shadowtrees: instead of adding the label of a node to the tuple of the parent, simply add it to the tuple of all parents.

While the isomorphism of these structures is not the focus of this paper, a short proof that the AHU algorithm [1] provides the correct result on these structures, with the modification mentioned above, will be provided in order to clear any doubts about this part of the AOEU algorithm. As this proof relies heavily on the details of the AHU algorithm, and follows from its correctness proof, it is highly recommended to get a good understanding of the AHU algorithm before attempting to verify this proof.

1. The AHU algorithm is correct for regular trees, as proven by *Aho, Hopcroft, and Ullman [1].*
2. The isomorphism of two individual substructures can be proven, as per 1, and is proven while running AHU on the tree.
3. Take two nodes, X and Y, which share (i.e. are both parents of) a child Z. If Z (and its lower subtree) were duplicated such that X and Y both have their own identical copy, X and Y receive the same label as they do when Z is shared.
4. The AHU algorithm is correct on regular trees where the nodes would be duplicated, per 2.
5. Per 3 and 4, two isomorphic shadowtrees will be labelled as isomorphic.
6. If Z were duplicated, the layer on which Z resides contains one more node, which would be spotted as an isomorphic difference to a tree where Z were shared.
7. Per 5, and 6, the modified AHU algorithm can prove that shadowtrees are isomorphic.

The complexity of this modified algorithm is trivially different than that of the original, as the combined length of the tuples is no longer linear in the amount of nodes in the tree. As it is not the focus of this paper, the algorithm's complexity is assumed to be lower than or equal to that of the graph_to_tree algorithm, which practical results seem to confirm.

3.3 Metrics of evaluation

The research has developed multiple variants on the same algorithm. These variants have been evaluated on the following metrics:

- Mathematical correctness - is there a mathematical proof that the algorithm is correct?
- Practical correctness - did the algorithm work on all graphs in the test set, and if not, on which graphs did it fail and why.
- Theoretical complexity - what is the asymptotic complexity of the algorithm?
- Practical speed - how long did the algorithm take on graphs in the test set?

3.4 The Fulltest

In order to evaluate the practical speed and correctness, the `fulltest` runmode was added. This mode evaluates all graphs in a folder, in this case the entire test set, sorted alphabetically by path; because of the way the files are named, this ensure intentionally isomorphic graphs will be directly after each other.

For each graph the set of trees is generated If the last graph had the same amount of nodes as the current one, it checks for isomorphism with this graph. Otherwise, only the time required to generate the set is recorded. It also shuffles the graph, and checks for isomorphism with the shuffled version, recording the time required to generate the set of the shuffled version as well as the time required to compare the sets.

It outputs these results into a CSV file, with the following information:

- The name of file
- The amount of nodes in the graph
- The time taken to generate the tree set
- The time taken to compare this with the tree set of the last graph. If the amount of nodes is different, this is set as 0.
- The total time taken for the comparison of the two graphs, i.e. the above two combined.
- The result of this comparison (false if none was performed).
- The time taken to generate the tree set for the shuffled graph
- The time taken to compare the tree sets of the shuffled and non-shuffled graph.
- The total time taken for the comparison of these two sets.
- The result of this comparison; if this is ever false, there is an issue with the algorithm.

In order to keep the runtime manageable the test skips any graphs with more than 1000 nodes.

4. ALGORITHMS

This section provides an overview of the different implementations of the graph-to-tree algorithms that have been developed. For the proofs of correctness and the actual runtimes of these algorithms, see section 5.

4.1 Breadth First Search

The first algorithm is a reference implementation, a direct translation of the algorithm developed in the previous research. It uses a standard breadth-first search algorithm, like algorithm 1, and it implements the node copying as described in section 2.3.1. Unfortunately, the memory complexity of this algorithm is so high that I have not been able to successfully run any tests on it, as it would repeatedly consume all memory on the system and summarily crash. As such, there are no results available for this algorithm.

4.2 BFS-shadow

A similar algorithm to BFS, but ambiguous nodes are shared between parents instead of having their entire subtree duplicated. Pseudocode for this algorithm is provided in algorithm 2. Take special notice of lines 20 to 23, which are the main difference with algorithm 1.

4.3 DFS-shadow

Similar to BFS-shadow, but the $frontier\_best$ function now prioritises nodes with lower values over shorter paths. In other words, the priorities for comparing two values are:

1. The value which has the first lower number.
2. If both are equal up to the length of the shortest path, return the node with the shortest path.
3. If two nodes have the exact same value, return both.

Algorithm 2: BFS-shadow

```
1  function BFS_shadow(Graph, start_node):
2   result: a ShadowTree with start_node as root;
3   values: Mapping(node → list of integers);
4   frontier ← {start_node};
5   visited ← {start_node};
6   values.add(start_node, [start_node.degree]);
7
8   while there are nodes in frontier:
9    best_nodes ← frontier_best(frontier, values);
10
11    foreach best_node of best_nodes:
12     foreach neighbour of best_node:
13      if neighbour not in visited:
14       new_value ← values.get(best_node);
15       new_value.append(neighbour.degree);
16       values.add(neighbour, new_value);
17
18       frontier.add(neigbour);
19
20       parents: set of nodes
21       foreach node in best_nodes:
22        if neighbour is a neighbour of node:
23         parents.add(node);
24
25       add neighbour to result as child of parents
           ↪ ;
26       frontier.remove(best_node);
27       visited.append(neighbour);
28
29 function frontier_best(frontier, values):
30  return those nodes in frontier for which the
       ↪ value is:
31  1. The shortest list.
32  2. if nodes have list of equal length,
       ↪ return that one which is the first
       ↪ to have a lower number.
33  3. If two nodes have the same value, return
       ↪ both.
```

4.4 Heuristic

Both BFS-shadow and DFS-shadow select nodes from the frontier based on their entire search path. This helps avoid ambiguous nodes, but also adds to the algorithms complexity (see section 5.1.3). The heuristic graph_to_tree algorithm avoids this complexity by using a heuristic for the search path instead: a tuple containing the length of the search path and the sum of the degrees of all nodes encountered in this search path. The *frontier_best* function simply prioritises the shortest search path, and if those are equal, the shortest sum of degrees.

5. RESULTS

This section contains the results for the different algorithms, as described in section 3.3. Table 1 provides an overview of the practical speed of the different algorithms; it compares the average time required to compare different and shuffled graphs, as well as this time divided by the size of the graph squared (as the average complexity for AOEU appears to be n^2). For figures, see the results document [2]. Raw results and the software used to produce them can be found on the git repository [4].

	avg. comp.	avg. shuffle	$\frac{comp.}{n^2}$	$\frac{shuffle}{n^2}$
BFS	6700	7732	0.0166	0.0216
DFS	6974	7892	0.0170	0.0237
heuristic	5872	6820	0.0146	0.0169

Table 1: Overview of practical speed. All times in milliseconds

5.1 BFS-shadow

5.1.1 *Mathematical correctness*

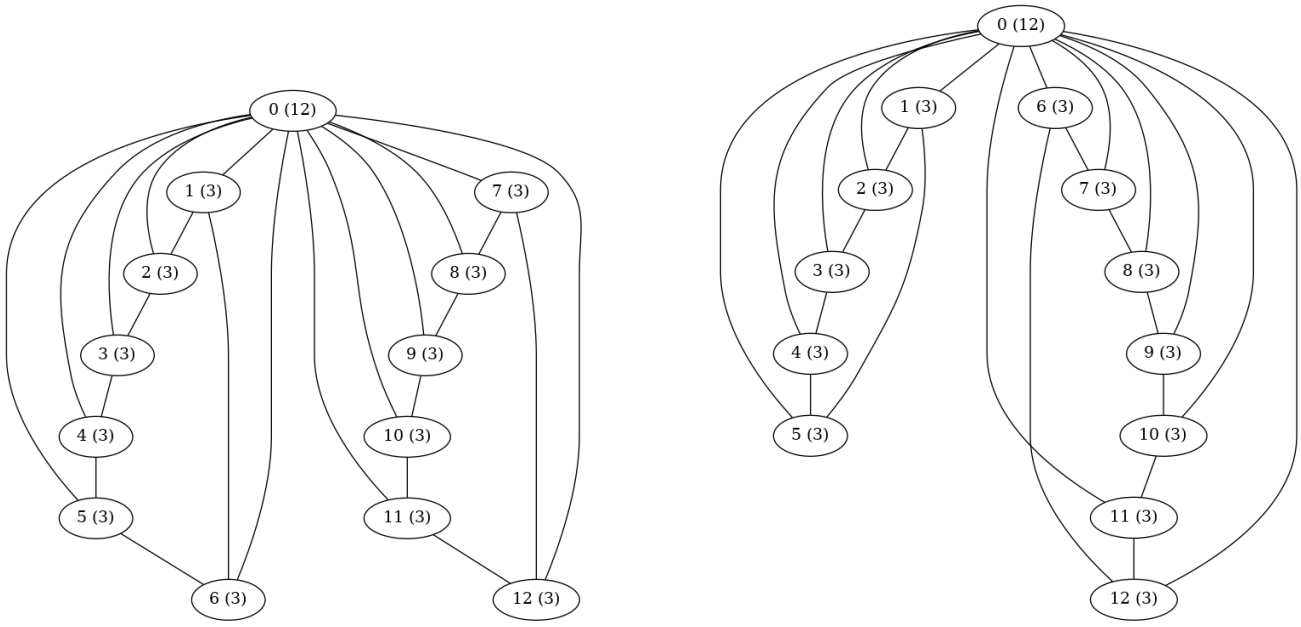
Unfortunately, the BFS-shadow algorithm can't be proven to be correct for all graphs; in fact, in can be proven to be incorrect for a very specific subset of graphs. Take the graphs of figure 3. These consist of two groups of nodes, which are fully connected to a single node. Within each group, the nodes are indistinguishable from each other; the only difference between these graphs is the size of groups (figure 3a can be described as (6,6), whereas figure 3b would be (5,7)). Starting at node 0 will obviously not result in a tree in which these differences are visible, as the edges that are not used to reach the node from the shortest path are left out. Starting at one of the other nodes, however, does not reveal the size of the groups either. Say we start at node 1 of figure 3a; node 4 is a distance of 3 removed from node 1 when traveling over nodes in the group, but only two when going via node 0; because this is breadth-first search, it will go via node 0 instead. As all nodes in the other group will also be reached via node 0, it is impossible to tell the size of the group from this tree. As this reasoning is analogous for all other nodes in the graph (except for node 0), a BFS-based algorithm can never correctly tell that these two graphs are not isomorphic.

Fortunately, this seems to be the only class of graphs with which AOEU/BFS-shadow has issues. The exact mathematical definitions of graphs for which this issue occurs is hard to define, and requires further research. Given the complexity of the algorithm as described in 5.1.3, however, it is highly probably that it is a subset of or identical to the PI-class of graphs which are closed under contraction, as described by *Ponomarenko [12]*.

There is one useful mathematical proof for AOEU/BFS-shadow: it can be proven that isomorphic graphs will always be labeled as such.

1. Given way graphs are represented in the framework (see section 3.1.1), there are two possible differences between isomorphic graphs; the order of the nodes in the list of the graph itself, and the order in which individual nodes list their neighbours.
2. The statement on line 18 could add nodes to the frontier dependent on the order in which neighbours are listed.
3. *frontier_best* will return all nodes with the best value, and is therefore not dependent on the order in which nodes are added to the frontier, up to the order of the returned set.
4. The loop on line 21 of algorithm 2 eliminates any dependence on the order in which nodes are added to the frontier, up to isomorphism of the resulting shadowtree.
5. Per 3 and 4, the resulting shadowtree is independent of the order in which nodes list their neighbours, up to isomorphism; in other words, isomorphic graphs will produce isomorphic shadowtrees, given corresponding start nodes.
6. Per section 3.2, AHU will label isomorphic shadowtrees as such.
7. Per 5 and 6, two isomorphic graphs with corresponding start nodes will be labeled isomorphic
8. In two isomorphic graphs, each node in one graph will have a corresponding equivalent in the other, per the definition of isomorphism.
9. AOEU compares the sets of trees, thereby comparing all nodes in one graph against all nodes of the other.
10. Per 7, 8, and 9, AOEU/BFS-shadow is not dependent on the two differences listed in 1; it labels isomorphic graphs as isomorphic.

In other words, while there are clearly non-isomorphic graphs which AOEU/BFS-shadow labels as isomorphic,



(a)

(b)

(c) The tree for most nodes

Figure 3: Two graphs which are not isomorphic, but are labeled as isomorphic by BFS

there are no isomorphic graphs which AOEU/BFS-shadow labels as non-isomorphic; there are false positives, but no false negatives.

5.1.2 Practical correctness

Although the issue above is definitely present on those specific graphs, the algorithm nevertheless provided the correct answer for all other graphs in the test set. This suggests the graph class for which the issue occurs is rather limited.

5.1.3 Theoretical complexity

Looking at algorithm 2, there are several loops dependent on input size.

- The loop on line 8 goes over all nodes once, and is therefore $O(n)$.
- The loop on line 11 can be discounted, as every node will removed from the frontier after going through this loop once, thus never being in the loop again; this loop simply works in accordance with the loop of line 8.
- The loop on line 12, combined with the if statement on line 13, will run through each node once, independently of the other loops; it does therefore not affect the asymptotic complexity, as $O(2 \cdot n) = O(n)$.
- The loop on line 19 can also be discounted, as the added complexity is offset by the statement on line 13
- The condition of line 20 is linear in the amount of edges in the graph, written as $O(e)$
- Running `frontier_best` is done once for every n on line 9; therefore, the complexity of this line is highly relevant. The worst-case complexity for this function would require checking the value of every node in the graph; the longest possible value is also every node in the graph. It is therefore safe to say the complexity of the function is bound by $O(n^2)$. However, the integers in a value represent nodes which have already been visited, and can therefore not be in the frontier; as such, the length of the values in inversely proportional to the amount of nodes to be checked. This inverse relationship seems to be linear, which would not change the upper bound of the complexity, but it does suggest the upper bound can never be reached.

We can therefore construct two upper bounds. The call to `frontier_best` on line 9, which is done for every n, generates a total bound of $O(n^3)$. There is also a total upper bound of $O(n \cdot e)$; however, e is maximally $\binom{n}{2} = \frac{1}{2}(n-1)n$, and can therefore never exceed n^2, making this bound irrelevant. The final upper bound can therefore be written as $O(n^3)$

The spacial complexity of BFS-shadow algorithm can be easily deduced. Because of the use of shadowtrees, each node in the graph is represented in the tree exactly once, and each edge can also be in the tree only once; the spacial complexity of a single tree is therefore $O(n \cdot e)$.

Note that both of these proofs only apply to the BFS-

shadow algorithm; as AOEU runs BFS-shadow on every node in the graph, the complexity of the full AOEU/BFS-shadow algorithm has an extra factor n.

5.1.4 Practical speed

When plotting the time required to compute the isomorphism against the size of the graph (as in done in [2]), most graphs appear very close to a polynomial line. In most cases, the line appears to be n^2; this would suggest the average-case complexity of the BFS-shadow graph_to_tree algorithm is linear.

The time required to compute a shuffled graph is slightly higher than a comparison between two graphs. This can be explained by the way the test works: while the comparison to the shuffled graph will always require both tree sets to be fully explored, the comparison with the previous graph will often be cut short because an isomorphic difference is found; in many cases, this difference is the size of the graphs, in which case the sets don't even need to be compared.

There are some instances where the time required to compare the sets of trees in significant: Cai-Fürer-Immerman graphs [5], grids, and Miyazaki graphs. For these graphs, the time required to compare the tree sets is about as large as the time required to generate them. The cause of this is immediately apparent when one would view a generated tree for such a graph: due to the layered structure of these graphs with many nodes of equal degree, the resulting trees contains almost every edge in the graph.

The graphs that took the longest to compute were Hadamard matrix graphs ([2], figure 1.9), being the only category of graphs to breach the 100 second mark. This probably has something to do with the ratio of edges to the distance between nodes, which may lead to exact worst-case behaviour. Complete graphs were significantly faster than other graphs, which seems logical given that the tree for such graphs would only have two layers. For other classes of graphs, the size of the graph was much more significant than its type.

It is hypothesised that the runtime of the graph_to_tree algorithm is proportional to the depth of its resulting tree. This would make the algorithm more suited to graphs with shorter search paths, i.e. graphs which more edges and more nodes of higher degrees. Unfortunately, such graphs also tend to fit the description of classes on which AOEU gives false positives, as in section 5.1.1. As such, it cannot be confidently said that any graph classes are particularly well suited to AOEU.

A profiling has revealed that most of the runtime of this algorithm was spend on hashtable lookups; given that the implementation of the algorithm heavily relies on sets and maps, this seems logical.

5.2 DFS-shadow

5.2.1 Theoretical Correctness

While the specific example graph used in section 5.1.1 is correctly identified as non-isomorphic by DFS-shadow, this does not mean that DFS-shadow does not share the underlying problem. Consider a graph class C of size n, where all nodes have a degree m with $m < n$. There are multiple non-isomorphic graphs that fit this description, implying $|C| > 1$. However, DFS-shadow, only relying on the length of the paths, is highly probable to run into the same problem. In these graphs, the algorithm can only rely on shortest path length, making it equivalent to BFS-shadow; it certainly does not provide a correct answer.

The second proof in section 5.1.1, however, still holds for DFS-shadow; DFS-shadow will never label isomorphic graphs as non-isomorphic.

5.2.2 Practical Correctness

DFS-shadow gave the correct answer for all graphs in the test set, including figure 3.

5.2.3 Theoretical Complexity

All proofs in section 5.1.3 hold for DFS-shadow as well; therefore, the theoretical complexity is the same: a time complexity of $O(n^4)$ and a spacial complexity of $O(n^2 \cdot e)$ for the full AOEU/DFS-shadow algorithm.

5.2.4 Practical Speed

While DFS seems to be slightly slower overall as seen in table 1, there are no significant differences in the datasets [2]. The only other conclusion that may be drawn is that the time required by DFS seems to be a bit more irregular than its breadth-first counterpart.

5.3 Heuristic

5.3.1 Theoretical Correctness

All proofs of section 5.1.1 apply to the heuristic algorithm as well, including the inability to detect the difference between figures 3a and 3b.

5.3.2 Practical Correctness

The practical correctness is identical to BFS-shadow; that is, it gave the correct answer to all graphs in the test set, except figure 3.

5.3.3 Theoretical Complexity

Because the value of a node is now constant, the time $frontier\_best$ requires is now linear in the frontier (instead of quadratic). This means that the upper bound for the algorithm generated by the call (line 9 of algorithm 2) is now $O(n^2)$ (instead of $O(n^3)$).

As a result, the $O(n \cdot e)$ call becomes relevant again, as the maximum $e = \binom{n}{2}$ exceeds n. Therefore, the upper bound for the time complexity of the heuristic graph_to_tree is $O(n^2 + n \cdot e)$, and the bound for the full AOEU/heuristic is $O(n^3 + n^2 \cdot e)$.

The spacial complexity is the same as BFS-shadow, i.e. $O(n^2 \cdot e)$ for the full AOEU/heuristic algorithm.

5.3.4 Practical Speed

Compared with AOEU/BFS-shadow, the speed on most graphs is not significantly different. Exceptions to this are random graphs with a 1/10 edge probability ([2], figure 3.24), and the Dawar Yeung graph sets [7] ([2], figures 3.26 and 3.27), on which AOEU/heuristic is about twice as fast; this descrepancy on a subset of the test set is sufficient to explain why the algorithm is so much faster on the average case, as in table 1. Curiously, Hadamard matrix graphs still pose the greatest challenge, despite the runtime of the heuristic algorithm being independent from path length. Further investigation might be required to find the root cause of why Hadamard matrix graphs pose such a challenge.

6. CONCLUSION

This paper has provided a new approach to tackle graph isomorphism, by providing three different algorithms based on pathfinding algorithms and shadowtrees. It has proven the complexity of this approach and given a handle on the graph class on which it is correct, as well as provided an overview of the practical performance.

AOEU proved 1 or 2 orders of magnitude slower than the currently most prominent solution, *traces [10]* in practice. While this may seem like a large difference, this implementation of AOEU was developed in a mere five weeks, and the actual codebase still has room for optimisation.

This version of AOEU is an enormous improvement compared to the previous version [3]. Not only is it several orders of magnitude faster[2], but it also manages to reduce the time, and more importantly memory complexity, to polynomial.

There is a now some form of proof available as to whether the algorithm works or not, and there is a handle for investigating on what graphs the algorithm works. Furthermore, there is a formal proof that the algorithm will never label two non-isomorphic graphs as isomorphic, meaning it could be used as a heuristic for isomorphism. In these cases, the runtime could be improved by only running on a subset of nodes instead of all nodes.

The difference between the different pathfinding algorithms has no significant impact on most graphs, but the heuristic algorithm seems to be significantly faster on some.

The theoretical complexity of AOEU is, most optimally, bound by $O(n^3 + n^2 \cdot e)$, and its memory complexity is $O(n^2 \cdot e)$. The practical time complexity seems to be closer to n^2.

The practical data shows that there are no graphs which can be confidently said to be particularly well-suited to AOEU, although it can be said that Hadamard-matrix graphs are particularly ill-suited.

6.1 Further Research

There are several questions that are in need of further research.

Most obviously, the exact nature of the graphs class on which AOEU is guaranteed to work is still unknown. As the algorithm runs in polynomial time, this could be a new class of of graphs the isomorphism of which can be provably solved in polynomial time: a PI-class. Furthermore, this class could be fundamentally different when AOEU is combined with different graph_to_tree algorithms, which could lead to the discovery of even more PI-classes.

As mentioned previously, the implementation still has room for improvement. For example, the *frontier_best* function has to analyse the entire frontier for every node that is analysed; this could be improved by keeping the frontier in a sorted list. Practically, the implementation relies heavily on hashtables, and changing some datastructures to be more suitable for the specific purpose could improve practical speed.

There are still many pathfinding algorithms available which are unexplored, and some may yield better results.

As mentioned previously, the algorithm could be used as a heuristic for graph isomorphism; this point is also in need of further research.

Finally, the invention of shadowtrees brings into mind an entirely different kind of algorithm. A pathfinding algorithm could also, hypothetically, be used to direct every edge in the graph, resulting in a directed acyclic graph. This would no longer be a shadowtree, as it is not layered, and therefore the AHU algorithm for tree isomorphism (or the modified variant of this paper) could no longer be used, but the directed and acyclic properties of the graph might nevertheless make it easier to check for isomorphism. As the process of making edges directed is reversible, this proposed algorithm would be trivially correct for all graphs.

[2] which could be (perhaps rightfully so) attributed to the difference between C++ and Python

Acknowledgments

The author would like to thank the following people: Rafael Dulfer, for his help with the parts of the initial algorithm other than graph-to-tree, as well as his ability to identify issues with the graph-to-tree part. Ben Willemsen, for his insight into the feasibility of the algorithm. Manuel Wackerle, for making this research possible in the first place due to his incredible computer science and mathematics skills. Mateusz Jaworski, for his incredible C++ experience and his willingness to help.

References

[1] A. V. Aho, J. E. Hopcroft and J. D. Ullman. *The design and analysis of computer algorithms*. English. Addison-Wesley series in computer science and information processing. Reading, Mass.: Addison-Wesley Pub. Co., 1974.

[2] F. T. Breggeman. *Practical Results for AOEU*. Graphs with results for this research. 2020. DOI: 10.5281/zenodo.3911836. URL: https://git.thebias.nl/floris/research-project/raw/branch/master/report/results.pdf (visited on 2020-06-18).

[3] F. T. Breggeman et al. *Excerpts form Project Documentation Group 23*. 2019. DOI: 10.5281/zenodo.3911828. URL: https://git.thebias.nl/floris/research-project/raw/branch/master/proposal/aoeu.pdf (visited on 2020-06-18).

[4] Floris Breggeman. *AOEU*. Git Repository for this project. 2020. DOI: 10.5281/zenodo.3911824. URL: https://git.thebias.nl/floris/research-project.

[5] J. Cai, M. Fürer and N. Immerman. "An optimal lower bound on the number of variables for graph identification". In: *Combinatorica* 12.4 (1992-12), 389–410. ISSN: 1439-6912. DOI: 10.1007/BF01305232. URL: https://doi.org/10.1007/BF01305232.

[6] *Create Graph online*. used to create custom graph layouts. 2015. URL: https://graphonline.ru/en/ (visited on 2020-05-07).

[7] A. Dawar and K. Khan. *Constructing Hard Examples for Graph Isomorphism*. 2018. arXiv: 1809.08154 [cs.CC]. URL: https://arxiv.org/abs/1809.08154.

[8] *Graphviz*. 2020. URL: https://graphviz.org/ (visited on 2020-05-21).

[9] B. Massey. *Coloring Problems: DIMACS Graph Format*. 2001. URL: http://prolland.free.fr/works/research/dsat/dimacs.html (visited on 2020-06-09).

[10] B. D. McKay and A. Piperno. "Practical graph isomorphism, II". In: *Journal of Symbolic Computation* 60 (2014), pp. 94–112. ISSN: 0747-7171. DOI: https://doi.org/10.1016/j.jsc.2013.09.003. URL: http://www.sciencedirect.com/science/article/pii/S0747717113001193.

[11] B.D. McKay and A. Piperno. *Nauty Traces – Graphs*. 2013. URL: http://pallini.di.uniroma1.it/Graphs.html (visited on 2020-04-26).

[12] I. N. Ponomarenko. “The isomorphism problem for classes of graphs closed under contraction”. In: *Journal of Soviet Mathematics* 55.2 (1991-06), pp. 1621–1643. ISSN: 1573-8795. DOI: 10.1007/BF01098279. URL: https://doi.org/10.1007/BF01098279.

[13] M. J. Uetz and J. De Jong. “Module 7 - Project Coaching: Graph Isomorphism and Partition Refinement”. Powerpoint for project instruction. 2019.