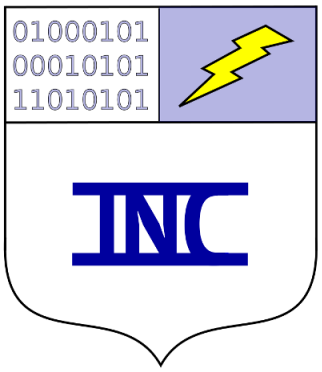
A Polynomial-Time Algorithm for Graph Isomorphism*

Brent Kirkpatrick

Intrepid Net Computing, bbkirk@intrepidnetcomputing.com

Intrepid Net Computing



www.intrepidnetcomputing.com

* Dedicated to Alan M. Turing and every minority computer scientist who has deserved better.

Document Revision History

Spring 2016 Drafted
July 21, 2016 Backed-up
October 30, 2016 Created title page

Abstract. The graph isomorphism problem is fundamental to many applications of computer science. This problem is so often presented in its technical formulation, rather than its applications, that newcomers to the field of computer science may underestimate the breadth of application for this single problem.
This paper presents a polynomial-time algorithm for counting the automorphisms of a directed acyclic graph (DAG). The same algorithm can be used to find a single automorphism in polynomial time. Due to the graph-isomorphic completeness of the DAG isomorphism problem, this result applies by reduction to solving the general problem of graph isomorphism.

1 Introduction

The question of whether the graph isomorphism problem can be solved in polynomial time, meaning that there exists an efficient algorithm for determining whether two graphs are isomorphic, has been open for over 40 years. While it has been known that graph isomorphisim is in the class NP, it has previously neither been shown to be NP-complete nor been shown to be solvable by an algorithm with a polynomial running time. A new representation of pedigree graphs first appeared specific to pedigree graphs in Kirkpatrick, 2008 [3] and GI on pedigree graphs was proven to be GI complete in 2012 [7]. This manuscript clarifies the application of this new representation by applying it to general dags and to solving GI. The method detailed here runs in $O(n^3)$ time and is a generalization of a tree isomorphism algorithm [1] that assigns labels to nodes such that any two nodes with the same label are isomorphic subgraphs rooted at those nodes.

Graph isomorphism appears in many applications ranging from software enginnering, artificial intelligence, statistics, and math, to computational biology. For example, the problem of correctly refactoring software can be expressed using directed acyclic graphs of the control flow of a program before and after refactoring. The problem in knowledge representation of comparing two logical representations of a 'fact' can also be formulated using graph isomorphism [10]. In computational biology, determining if two RNA complexes have an identical structure [9] is a graph isomorphism problem. In robotics, reconfigurable robots have isomorphic graph signatures [2].

This paper seeks to establish the existance of a polynomial-time algorithm for graph isomorphism and approaches this problem by discussing graph isomorphism for directed acyclic graphs (DAGs) and, in particular, graph isomorphism of directed subgraphs. Due to the graph-isomorphic completeness of the DAG isomorphism problem, this algorithm also applies to the general problem of graph isomorphism.

While graph isomorphism has many applications, the primary motivator for development of this algorthim is derived from the study of pedigree graphs [7, 5, 6] which are graphs of the biological relationships between individuals in a mating population.

2 Background

Let $G = (V(G), E(G))$ and $H = (V(H), E(H)$ be two directed acyclic graphs having vertices $V(.)$ and edges $E(.)$. An *isomorphism* of graphs G and H is a bijective function $f : V(G) \to V(H)$ such that $(u, v) \in E(G)$ if and only if $(f(u), f(v)) \in E(H)$. Informally, the isomorphism maps the vertices of G onto the vertices of H in such a way that the edges of G are also mapped onto the edges of H.

Recall that graph isomorphisms for DAGs is the same complexity as graph isomorphism for general graphs [1]. Recall that finding whether there is a non-trivial automorphism of a graph is polynomial-time reducible to finding a non-trivial isomorphism between a graph and itself.

Recall that counting the isomorphisms is polynomial-time reducible to counting the automorphisms of a graph which is also polynomial-time reducible to finding any isomorphism [8].

3 Algorithm

Here, we will consider the problem of finding automorphisms, as it is easier to formalize and represent a single graph than two graphs. Our graph will be a dag, as the algorithm is more intuitive with a directed graph.

Let $L(G)$ be the leaf nodes of graph G. Let $c(v)$ be the children of vertex v in graph G, i.e. $c \in c(v)$ if and only if $(v, c) \in E(G)$. Let $p(v)$ be the parents of vertex v in graph G, i.e. $p \in p(v)$ if and only if $(p, v) \in E(G)$.

For this graph G, create a new vertex s_G, add it to $V(G)$. For each source vertex, v, existing in G, create a new edge (s_G, v) to add to $V(G)$. This provides a reduction from the GI on a multi-source DAG to GI on a single-sorce DAG. A solution to the latter problem is also a solution to the former.

We will introduce some notation. We will define a descendant split (d-split) for node u as the set of nodes in the subgraph rooted at u and including u. This terminology is motivated by theory from phylogenetic tree reconstruction, where there is a similar definition and similar splits-equivalence result about the equivalence between the tree and the splits. In the next section we will prove the analagous d-splits-equivalence result for dags.

Definition 1. *Let V be the nodes of a directed acyclic graph G. The* **descendant split** *(or* **d-split***) of a node $v \in V$ is defined as a subset of the nodes*

$$D_v(V) = \{u \in V \mid u \text{ is on a directed path from } v \text{ to a leaf } \}$$

where every node is considered a descendant of itself. We refer to the set of d-splits as $\mathcal{D}_V = \{D_v(V) | v \in V\}$.

We need several algorithms in order to clearly argue that this algorithm is correct. The first algorithm is the dag reconstruction algorithm that takes the set of d-splits as input and recapitulates the topology of the dag that generated the d-splits, Algorithm 1. The remaining algorithms apply the d-split representation to counting the number of automorphisms of a graph. The automorphisms are counted by finding a single permutation that is a generating set for a group on S_n with the following property: all the permutations describing automorphisms are in one set of the equivalence class. This means that we can count the number of automorphisms by multiplying the cycle lengths.

Consider the set of d-splits for a dag. Any d-split with a single element (i.e., $|D_v(V)| = 1$) represents a leaf node. For any given node v_1 we can examine the directed path in the dag from that node to a leaf, for example the path might be $v_1 \to v_2 \to ... \to v_\ell$, where te arrow indicates a directed edge. We see that the d-splits along the path are ordered $D_{v_1} \supset D_{v_1} \supset ... \supset D_{v_\ell}$. The cardinality of the d-splits decreases as we consider nodes lower in the path. These ideas lead to the following dag reconstruction algorithm, Algorithm 1. Lemma 1 proves the equivalence of d-splits and dags by proving that the dag reconstructed by Algorithm 1 is isomorphic to the dag that generated the d-splits.

4 Example

5 Analysis

Theorem 1. *There is a non-trivial automorphism of G if and only if Auto(G) counts at least two automorphisms.*

Proof. We will develop the proof of this with several lemmas, Lemma 1 and Lemma 2. Both lemmas use inductive reasoning. The list of d-splits $\mathcal{D}_V$, encodes the topology of the graph, as we will see next.

Lemma 1. *For each $\mathcal{D}_V$ there is a distinct graph topology H that is isomorphic to the graph G. Furthermore, this H is recovered by Algorithm 1.*

Proof. A similar d-splits equivalence algorithm for pedigree graphs appears in Kirkpatrick, 2011 [4] and Kirkpatrick, 2008 [3]. Since we have a d-split for every node, the algorithm will either create parents or not create parents. Now, if we look at a single step in the algorithm, each node with a d-split that is a subset of some other d-split will be assigned the parents.

Intuitively, there is strictly increasing cardinality of d-splits as one moves up the dag (notice that this is due to a node being contained in its own d-split). The parents are represented by the descendant splits having all but one node already represented in the partial reconstruction of the dag. See Algorithm 1, line 12.

Algorithm 1 ReconstructDAG($\mathcal{D}_V$)

$Heap = (D_{v_0}, Dv_1, ..., D_{v_k})$ where $|D_{v_0}| \leq |D_{v_1}| \leq ... \leq |D_{v_k}|$ and $k = |V|$
Create graph H with nodes $\{\hat{v}_0, \hat{v}_1, ..., \hat{v}_k\}$
for every singleton set $|D_\ell| = 1$ in $\mathcal{D}_V$ **do**
    Create a node, $\hat{\ell}$, in graph H
end for
while $Heap \neq \emptyset$ **do**
    $D_{v_j} = pop(Heap)$
    for each D_u in the heap **do**
        $D_u = D_u \setminus D_{v_j} \cup \{\hat{v}_j\}$
    end for
    for each D_u in the heap in order **do**
        if there is a single node, u, without a hat in D_u **then**
            create node $\hat{u}$ in H
            for each $\hat{w} \in D_u \setminus \{u\}$ **do**
                create edge $\hat{u} \rightarrow \hat{w}$
            end for
        end if
    end for
end while
return H

Algorithm 2 Initialize(G) initialize the data structures

$I[x] = 1 \quad \forall x \in V(G) \setminus L(G)$
for each leaf $\ell \in L(G)$ **do**
    $I[\ell] = 0$
end for
for each vertex $v \in G$ in reverse topological order **do**
    $D[v] = \{v\} \cup_{c \in c(v)} D[c]$
end for
Let *glabels* be a multiset indexed array with the index for a given multiset being a unique value labeling that multiset.
for each vertex $u \in G$ in reverse topological order **do**
    {Find the multisets and labels for each graph node.}
    $m_u = \{I[x] \mid x \in D[u]\}$
    Add m_u to *glabels* if it is not already there.
    $I[u] = glabels[m_u]$
end for

Algorithm 3 Auto(G) counts all automorphisms

Let s be the source vertex in G.
Initialize(G)
let lom be a binary matrix, indicating if a pair of nodes was visited.
for each node v, let $visited[v]$ be the set of nodes v has been paired with
$map[s] = s$
Let the queue $Q = \{s, s\}$
while Q is not empty **do**
    $u = pop(Q)$
    $v = pop(Q)$
    push v onto $visited[u]$
    push u onto $visited[v]$
    for each child, $a \in c(u)$ in G **do**
        for each child, $b \in c(v)$, of G **do**
            if $I[a] == I[b]$ and $map[a]$ =undef and $map[b]$ =undef **then**
                $map[a] = b$
                $map[b] = a$
                if $visited[a, b] == 1$ **then**
                    push map on lom
                else
                    push a, b on front of queue Q
                    if map is well-defined when $u < head(Q)$ **then**
                        push map on lom
                    end if
                end if
                $map[a]$ =undef
                $map[b]$ =undef
            end if
        end for
    end for
end while
do disjoint set union to get the composition of cycles from all maps and partial maps
count the automorphisms using mutliplication of cycle sizes

Formally, we prove this by induction. The base case is that every singleton d-split has a leaf node in H.

For the inductive step, suppose that we have a partial reconstruction, H, where every connected component of H is isomorphic to some subgraph of G. By Algorithm 1, while-loop invariant is that the d-splits in the heap contain nodes in H that are the roots of every directed connected component, see line 15.

Given the base case and the inductive step, we have the inductive proof that when Algorithm 1 halts, for every source in graph G, graph H will contain a subgraph isomorphic to that sources subgraph in G. Given that G has a single source (by reduction), the graph H has a single source and is isomorphic to the graph G. Graph H was obtained from the d-splits without knowledge of G. □

Lemma 2. *The graph isomorphism labels for nodes u, v, $I[u]$, $I[v]$ are equivalent if and only if the subgraphs rooted at u and v are isomorphic.*

Proof. This proof is similar to the d-splits equivalence proof of Lemma 1. Now we use inductive reasoning.

The inductive hypothesis is that we build a (possibly disconnected) subgraph of the whole graph such that the subgraphs rooted at u and v are isomorphic if and only if the multisets of labeled d-splits are identical (i.e. $\{I[x]|x \in D[u]\} = \{I[y]|y \in D[v]\}$). This hypothesis is true for the base case which includes only the leaves of the graph and all leaves are isomorphic subgraphs. For all pairs of leaves l and m, the d-splits are $\{l\}$ and $\{m\}$ and the labels d-splits, $\{0\}$ and $\{0\}$, are equal.

For the inductive step, we add new roots to our subgraphs while maintaining the inductive invariant. Consider an arbitrary node w in the main graph whose children are all in our subgraph. Then all the descendants of w have labels assigned by function I, and $I[w] = 0$ by initialization. If w is the root of a subgraph that is isomorphic to some other z-rooted subgraph embeded in the subgraph we are building, then it must be the case that the children of w are themselves the roots of subgraphs that are isomorphic with those of the children of z. In any case, when the label of w is created

$$m_w = \{L[x] \mid \{w\} \cup_{c \in c(w)} D[c]\}. \tag{1}$$

Since the children of w are isomorphic to the children of z, then for each pair of isomorphic children $c \in c(w)$ and $d \in c(z)$, then $L[c] = L[d]$ by the inductve hypothesis. Therefore, the labeling of the set union in Equation 1 will be identical for w and z (i.e. $m_w = m_z$). Since w is an arbitrary node, we have considered both directions of the "if and only if" implication. The inductive hypothesis is proven. □

Due to Lemma 1 and Lemma 2, two subgraphs of G are isomorphic if and only if the labels L from Algorithm 2 for the roots are identical. Notice that due to Line 14 of Algorithm 3, only automorphisms are considered.

The remainder of the proof is left as an exercise to the reader.

This concludes the proof of Theorem 1. □

Corollary 1. *The problem of counting the automorphisms of G is polynomial-time reducible to finding one isomorphism.*

Since Mathon proved this [8], we will not go through a proof. We can note that the Auto(G) algorithm counts all automorphisms in polynomial time, as we see next.

Theorem 2. *Algorithm 3: Auto(G) counts all automorphisms.*

Proof. This algorithm takes the automorphisms and creates an equivalence class for them. One automorphism from each set of the equivalence class is enumerated. The other automorphisms in each set of the equivalence class are counted, not enumerated. We want to show that every map that Auto(G) counts is an automorphism. On the other hand, every automorphism is counted as a member of a set in the equivalence class for which a representative is map is printed.

First, every map printed is an automorphism. Notice that due to Line 14 of Algorithm 3, only automorphisms are considered. This means that every map that is printed is certainly an automorphism as,

$map[a] = b$ is assigned if and only if $I[a] = I[b]$, meaning that the subgraphs rooted at a and b are isomorphic by Lemma 2.

Second, every automorphism is counted as a member of a set for which a representative map is printed. Recall that Lemma 2 says two rooted subgraphs are isomorphic if and only if they have the same labels. Therefore, every matching of nodes with the same labels is legal as part of an automorphism.

The remainder of the proof is left as an exercise to the reader.

This concludes the proof. □

Theorem 3. *Algorithm 3: Auto(G) has an $O(n^3)$ running time where $n = |V|$, the node cardinality of G.*

Proof. Notice that Algorithm 2 runs in $O(n^3)$ time, since each for loop is over n nodes, and the labels m_u may be compared with up to n^2 objects in line 12. Algorithm 3 also runs in $O(n^3)$ time, since the Q will contain at most n distinct objects, and the two for loops over the children are over at most n nodes. □

6 Conclusion

References

1. Alfred V. Aho and John E. Hopcroft. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1974.
2. Masoud Asadpour, Alexander Sproewitz, Aude Billard, Pierre Dillenbourg, and Auke Jan Ijspeert. Graph signature for self-reconfiguration planning. In *International Conference on Intelligent RObots and Systems - IROS*, pages 863–869, 2008.
3. B. Kirkpatrick. Pedigree reconstruction using identity by descent. *Class project, Prof. Yun Song, 2008. Technical Report No. UCB/EECS-2010-43*, 2010.
4. B. Kirkpatrick. *Algorithms for Human Genetics.* PhD thesis, EECS Department, University of California, Berkeley, April 2011.
5. B. Kirkpatrick. Non-identifiable pedigrees and a Bayesian solution. In L Bleris, I Madoiu, R Schwarz, and Wang Jianxin, editors, *ISBRA: Int. Symp. on Bioinformatics Res. and Appl.*, volume 7292 of *Lecture Notes in Computer Science*, pages 139–152. Springer Berlin Heidelberg, 2012.
6. B. Kirkpatrick and K. Kirkpatrick. Optimal state-space reduction for pedigree hidden Markov models. *submitted manuscript (arXiv 2012)*, 2016.
7. B. Kirkpatrick, Y. Reshef, H. Finucane, H. Jiang, B. Zhu, and R. M. Karp. Comparing pedigree graphs. *J. of Comp. Biol.*, 19(9):998–1014, 2012.
8. Rudolf Mathon. A note on the graph isomorphism counting problem. *Inf. Process. Lett.*, 8(3):131–132, 1979.
9. Joseph Malcolm Schaeffer, Chris Thachuk, and Erik Winfree. Stochastic simulation of the kinetics of multiple interacting nucleic acid strands. In *Proceedings of DNA Computing and Molecular Programming (DNA21), Lecture Notes in Computer Science (LNCS)*, volume 9211, pages 194–211, 2015.
10. John F Sowa. *Principles of Semantic Networks: Explorations in the representation of knowledge.* Morgan Kaufmann, 2014.