Paths: a Powerful Graph Invariant with Interesting Limitations

Grady Ward

December 7, 2015

In searching for a polynomial time-complexity solution to the graph isomorphism problem, many researchers have focused on, *vertex invariants*, numerical properties calculated over a vertex of a graph that can be deterministically calculated, regardless of how the graph is represented or labeled. This article will aim to describe the descriptive power and information density of the $Paths(p, v)$ function, a vertex invariant that counts the number of closed paths of length p through a vertex v. The $Paths$ function, in order to be applied to the graph isomorphism problem, is used to construct an algebraic graph invariant, with the application of sorting to the results of individual vertex $Paths$ results.

Though research into the descriptive power of this function has shown that it does not fully determine the graph, the lines of inquiry have proven to be fruitful in describing patterns of graph behavior that a casual observer would not anticipate. Additionally, the invariant's descriptive power can be computationally described in a way that is meaningful, and shows that for its time complexity, it is an incredibly discriminatory algorithm. Finally, this article will hope to address the various lines of inquiry that will hopefully be addressed in further research on this topic, including a systematic analysis of the classes of graphs that the Paths function is effective for, and those for which it is insufficient to prove isomorphism.

1 Background

To begin a discussion of this work, we must first establish a mutual understanding of the functions that we are discussing, the scope of existing research on this topic, and an appropriate category of graphs to examine.

1.1 Definitions

In this analysis, we will be examining *undirected* graphs that do not contain multi-edges nor loops. Any graph under these three constraints can be represented as a symmetric adjacency matrix, with zeros along the diagonal, and ones or zeroes in the remaining locations denoting an edge or disjoint vertices respectively. We will regularly reference this matrix as A with no additional annotation.

The vertex invariant we are going to examine ($Paths(p, v)$) is the number of closed paths of a length p that contain the vertex v. A path is a sequence of vertices such that each vertex shares an edge with the next one. A Path can be notated by the sequence of the vertices that are visited (e.g. ABCDE or BACAB). A Path is *closed* if the first and last vertex in their traversal sequence are the same. Note that though $ABCDE$ and $EDCBA$ describe the same edges, they are distinct paths: even though an undirected graph doesn't have directional edges, paths maintain the order of the edges they traverse.

For this article, when we describe a path as a geometric polygon (such as a triangle, quadrilateral, pentagon, etc.), we will define to be a closed path such that the vertices of the path are non-repeating (aside from the first and last, as we assume that it is closed). Thus, $ABABA$ does not form a quadrilateral, even though it is a closed path of length four. Note that this path could equally well be described as $BABAB$, as closed paths are inherently cyclic, and do not demand a starting vertex. Additionally, we will assume that unlike a path, a polygon's directionality does not matter (thus the polygon ABCA is the same as the polygon ACBA).

1.2 Computation of the Paths Function

The $Paths(p, v)$ function counts the number of unique closed paths that pass through a given vertex, v. This information can be easily computed using A. Just as the entries of A^1 represent the existence of paths of length 1 between two vertices (edges), the entries of A^p represent the number of paths of length p between any two vertices (by examining the row and column corresponding to two vertices). Thus, to find the number of closed paths of length p that contain a given vertex v, we simply need to calculate:

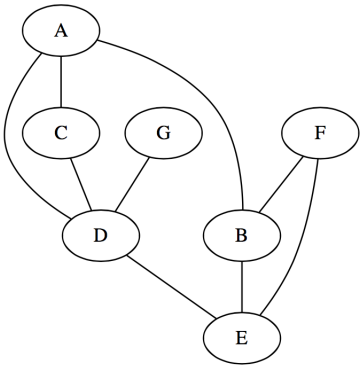
$$Paths(p, v) = A^p[v, v]$$

Where v is being used interchangeably here with its represented position within the adjacency matrix. Thus, calculating a specific value of $Paths(p, v)$ can occur in the time it takes to exponentiate A to the power p. Though it is a

well known result that matrix multiplication can be done in faster than $O(n^3)$ time, we will be using the nave assumption that matrix multiplication runs in $O(n^3)$ in order to make the computational complexity calculations more accessible. Under that assumption, it is clear that calculating $Paths(p, v)$ will occur in $O(pv^3)$ time. This is well within polynomial time, so long as we request a polynomial number of values from $Paths(p, v)$. Thus we can consider this invariant one in the traditional vein of looking for polynomial invariants with the broad aim of solving GI in polynomial time, though we have already established that it does not uniquely determine isomorphism.

1.3 Expansion to a Paths Object

Since $Paths(p, v)$ is a function that operates over a vertex, we will be examining a slightly modified version of the function which operates over a graph: $Paths(P, G)$. $Paths(P, G)$ produces a graph invariant that is related to its vertex invariant function by a simple translation. $Paths(p, v)$ is calculated for all $v \in G, p \leq P$, resulting in v vectors, each of which has P elements, representing the paths function when calculated at that node for each successive value of p. Since vectors are comparable objects, we can sort them, and return back a list of these vectors as a graph invariant, one that is computable and invariant to changes in vertex relabeling or adjacency matrix ordering. A skeptical reader should convince themselves that this holds true, even when two of the vectors to be sorted are identical, the resulting $Paths$ object is valid and deterministically constructed from the graph.



$$A = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

1.4 Co-Paths and Isomorphism

Finally, two Graphs are said to be Co-Paths if they produce the same *Paths* object. Since the Paths function is invariant to labeling and positioning, all isomorphic graphs are Co-Paths. However, a broader question is implied by this implication: does the reverse hold? Does Co-Paths determine isomorphism? The answer turns out to be no, but the line of inquiry that leads to this result is worth examining, and leads us to a broader understanding and appreciation for the discriminatory power of the *Paths* function with respect to isomorphic graphs.

2 Cardinality Analysis

An obvious choice in attempting to verify a bijection is the analysis of the cardinality of the target and domain spaces. If a function maps all inputs to a space of equal size as its own, it must be a bijection. Thus, in pursuing to quantify the relationship between the *Paths* object and the isomorphism relationship, I set about examining the cardinality of two sets: the set of all non-isomorphic graphs, and the set of all achievable *Paths* objects.

Note that this line of inquiry is what allowed us to rule out that the two relationships are the same. If we were able to prove that the relationship between the *Paths* objects and the set of all graphs was linked by a bijective function, we would have proven that if two graphs agree on their corresponding Paths's matrix, then they are the same graph. This would have been valuable because *Paths* objects can be directly compared and sorted, while graphs cannot immediately be compared in polynomial time by any currently understood algorithm.

How to we go about calculating the cardinality of both sets? Luckily, the cardinality of the first set is well established. The series A000088 in the online encyclopedia of integer sequences gives us the number of non-isomorphic graphs up to N=28. Now to calculate the number of distinguishable paths matrices of a given order, I turned to programming, with various degrees of success. Below are the various results of these computations, and a brief overview of their implementation and success rates.

2.1 Brute Force - Java

Start with what you know. As early as October, I had written a simple function in Java that computed the number of unique paths vectors by doing raw calculation of paths Matrices, sorting using a simple row comparison, and a tree set to do comparison of paths objects. Attempting to approach this

problem from the most naive standpoint first, I didn't make any smart or deductive assumptions about the number of graphs that are non-isomorphic, but instead chose to process every single adjacency matrix of a given size. Though the number of graphs of a given size is exponentially defined, the number of potential representations of those graphs has an even steeper exponential growth rate.

The final result of this line of inquiry appeared to corroborate the strength of the paths function, at an exorbitant computational cost. Running on a Macbook Air with a 1.7 gHz processor and 8 GB of RAM, this calculation took 28 hours to complete, and only described the cases with eight or fewer vertices. Far from ideal. For $V <= 8$, this analysis calculated that the number of valid paths objects of the same size exactly matched the A000088 sequence. Going forward with more advanced computations, we knew already that we would not find non-isomorphic co-paths graphs with fewer than nine vertices.

2.2 NAUTY Package

Thankfully, the generation of non-isomorphic graphs is well studied, and practical algorithms have been developed for their systematic enumeration. One of the best is the *geng* graph generator that is offered as part of the *NAUTY* package, written in C. Though computation of these graphs is still exponentially asymptotic, it is orders of magnitude faster than brute force calculations. The NAUTY package saves its graphs in a format called Graph6, which allowed us to efficiently store all possible graphs of a given cardinality and vertex count in a line delimited file, to split up the tasks of generating and processing the graphs.

All computations beyond this point were done in this way: a set of files were generated of the form [NVERTICES]-[NEDGES].txt, which stored the Graph6 format of the adjacency matrix for all non-isomorphic graphs matching those computational descriptions.

2.3 GPU Calculations

Of growing importance in scientific computing is the role of GPUs in large and distributed computation. With a twofold agenda (education and performance), I set out to utilize GPU arrays in the computation of *Paths* by using established protocols for matrix multiplication in OpenCL. The result was successful in outperforming a traditional implementation in C by a factor of four, but unfortunately was not fully parallelizable. I am hoping to work next

semester to consider broader applications of the GPU to actual *Paths* computation, collection, and comparison, in a way that takes advantage of the massive parralellizability of this problem, while trying to find creative ways to get around the highly limiting nature of the memory access procedures that differentiate GPU programming from CPU multithreading.

2.4 C Calculations

Now, most of my work on this project has been theoretical, but this piece in particular has been of great algorithmic and pragmatic interest and education. After somewhat abandoning the GPU as the method by which to speed up these calculations, I decided to try to use custom C to maximize my memory efficiency and attempt to do the minimal amount of computation necessary to be successful. What resulted is probably the best piece of code that I have ever written, and it reflects many facets of my education in computer science here at Brandeis: stream processing, advanced data structures, intentional bit manipulation, and memory allocation and reuse.

Unlike cardinality computations, this program was built with a broader purpose: to determine the maximum power necessary to differentiate all graphs within the each subset of all graphs with a set number of vertices and edges. This is not a trivial adaptation of code, and developing the structures to appropriately address this question required lots of thought and energy.

The program maintains a Trie, where the input to the Trie is the Paths object's sorted vector elements in increasing levels of power. This is far from revolutionary. What was incredibly powerful about this technique was the way in which I used stream processing to delay computation and comparison of Graphs.

When inserted into the Trie, a new graph element would only compute the first level of its exponentiated form (i.e. A^2). It would store this value (a running matrix with its current exponentiated power of A) alongside the graph within a structure which also included an elaborate (but low memory) mechanism to allow individual number output of the next number to be used in insertion into the trie (or the next level of the trie). This second part is particularly non-trivial because we have to consider past sorting decisions in making present sorting decisions, and have to store the results of those decisions in a fashion that is memory efficient and computationally easy to manipulate. My code here is particularly thoughtful, and uses a quadratic runtime (in the number of vertices) in exchange for a cubic amount of space.

Finally, when a new graph is inserted into an existing trie node, if there already exists a graph within that node, the original inhabitant is kicked out, and both are re-inserted into that node. This means that we only

exponentiate the matrix (an $O(n^3)$ operation) the minimal number of times necessary to differentiate between graphs. On average, this means that rather than exponentiating a matrix V times in the naive version, we average $0.2V$ matrix exponentiations, a five fold speed increase in multiplication alone. The number of comparisons in this methodology can also prove to be minimal, and using a trie structure over a comparable tree led to an enormous speed up in efficiency, wile still having a depth that is guaranteed to be a linear factor of the result (which is generally a linear function of the input, but not for all cases).

The result of this work, in addition to the pre-computation of non-isomorphic graphs through the NAUTY package was a program which could perform the same task as the Java function in under 2 seconds, and could expand upon the performance to $V = 10$ with all computations under 20 seconds. This aspect of the project is a great point of personal pride and growth, and demonstrates the value of a careful consideration of algorithms and data structures in scientific computation.

The result of these computations revealed a counterexample that proves that *Paths* is not a bijection. There exist two separate graphs, G_1 and G_2 with 10 vertices and 17 edges which are fundamentally distinct, but share the same paths values for all observable sizes P. A discussion of these first Co-Paths graphs will be included in the next section.

These computations also provided us with a sketch of what I will call a 'power curve', a function which describes the maximum exponentiation necessary for a graph with given properties to fully differentiate all of its constituent elements. This curve does not look like I had expected, and will certainly be a topic of further exploration and discussion.

3 Co-Paths, Non-Isomorphic Graphs

4 The Power Function

5 Future Lines of Inquiry

A thesis is apparently like a Hydra, every time that we answer one question, fifteen more appear. This independent work has given me a fantastic opportunity to pursue interesting questions that I have discovered as I explore, and given me a taste for academic research and inquiry. Below are some of the questions that are still unanswered moving into next semester, alongside plans for their address and hopeful resolution and analysis. Though I under-

stand that I will not be able to tackle every question that I have, I think that I am well on my way to crafting a thesis which explores this graph invariant robustly, and attempts to marry some graph theory with some thoughtful programming through the exploration of an interesting problem, that hopefully can be shown to be of some kind of practical signifigance.

1. Create a program (probably in java) which attempts to verify that paths functions continue to be equal for what we have begun to label as co-paths graphs. We would not want to make the deductive fallacy of assuming that simply because the paths objects line up until the long-integer limit, that that would imply that they line up to infinity. We can verify this claim (at least to a significantly higher degree of accuracy) by utilizing Java's BigInteger class, and hoping to evaluate proposed Co-Paths graphs for absurdly high values of P.

2. Create code which not only attempts to discover the maximum power at which two graphs diverge, but also creates and stores classes of graphs which share the same Paths object as output of this flawed graph invariant. Code for this purpose will have to be highly fault tolerant, and have to be based on educated assumptions about the reasonable powers that we can assume imply that the number of paths will continue to be the same beyond a large number.

3. Analyze all of the classes of 10-17 graphs, as they are easy to spot the non-isomorphic differences within. What if any features can we compute which differentiate them? Which graph invariants do they agree on? What information can this tell us about the set of graphs for which Paths is uniquely determinate? Moreover, are there broad categorizations of these graphs that might allow Paths to be calculated on the complement set of graphs with a regularity that would befit a correct algorithm for GI, even if it is over a subclass of graphs?

4. Corroborate the results that led to the abnormalities described the Power Paths function through a separate line of brute force inquiry, ideally through something like MATLAB, with high level of accuracy and a reasonable idea of correctness, without trying to duplicate the work that I have obsessed over in C.

5. Explore the Power Paths curve, and try to come up with a unifying way of describing this discrete function. If we use the granular version of it, how does that change based on our sorting mechanism? If we use the broad version of it, can we fit some sort of differentiable function to

it with a ceiling or round function being used to establish its discrete properties? How can we go about trying to fit three dimensional discrete data when we know (or suspect) that some of our resulting values are infinite?

6. Explore the power paths curve in granular detail: what percentage of all graphs fit into classes that are co-paths? What is the average depth of a leaf within this Trie, and what is the average differentiating power of two graphs? Each of these metrics sounds really interesting, and is certainly computable. There was a large surprise in the overall shape of the Power Paths function; what other secrets and patterns lie in this function which characterizes the descriptive power of the paths function, and the inherent complexity of the Graph Isomorphism problem?

7. Extend the logic of the existing C program to one which uses virtual memory and stream processing from the GENG graph generator. Currently the limitation that our program runs in to is not one of speed, but one of memory allocation and access. This should not be surprising when we are allocating at least one node, two matrices, and three arrays for each of the graphs, and the number of graphs is in the trillions. We can spare some speed on the computation side if we can use it to shift toward a disk intensive set of operations. Deciding how that will take place, and the role of multi-processing in that endeavor is a large task, but may allow us to extend existing methodologies to explore the areas of $V > 11$.

8. Explore theoretical justification for low-paths determinism: why is it that we see that most graphs are either 1) fully determined by paths at a relatively low value of P (usually linearly defined with respect to V), while others appear to exponentiate identically to infinity? Why does this divergence exist? Wouldn't we anticipate that in a system like this we would see some sort of normal distribution? What can we say about this threshold if it exists? Are there theoretical reasons (intuitive or in algebraic graph theory) that can explain this phenomena? My intuition says that this has something to do with the algebraic polynomials discussed in previous reports, and the fact that their degree is fixed as a linear function of V and E, while our paths function is allowed to wander up toward infinity...

9. When I reach a point that my data is able to draw a picture without

holes, email the Cornell professor who originally proposed this invariant, and discuss it with him.

10. Closely examine the relation that sets of Co-Paths graphs appear to have (properties of the sets, not of the graphs themselves, which will be a separate analysis). We have to realize that these sets are bound to be linked through some sorts of inverse, concatenation and union operations, and that they may be linked in ways that are more profound that we can yet imagine.

11. Set upper bound worst case approximations for the number of bits that are necessary to calculate, store and compare the Paths Object for a given value of V, E and P. Invert this function to create a calculator for the maximum size we could create for a given chunk of allocated memory.