Determination of Graph Isomorphism by the Greatest Characteristic String Invariant

Thomas F. Hain
School of Computer & Information Sciences
University of South Alabama, Mobile, AL 36688
(251)460-6390
THain@usouthal.edu

Geof Goldbogen
School of Computer & Information Sciences
University of South Alabama, Mobile, AL 36688
(deceased)

ABSTRACT
The greatest characteristic string (GCS) of an undirected graph is a new invariant such that two such graphs are isomorphic if and only if they have the same GCS. We define the invariant, and describe an efficient algorithm for evaluating it. A formal definition of the algorithm is given, and a correctness justification is presented. Timing and operation count studies on an implementation indicate the algorithm is \( O(V^2) \) for the vast majority of random graphs, leaving only a few "pathological" cases. This complexity is consistent with theoretical arguments.

Categories and Subject Descriptors
G.2.2 [Graph Theory]

General terms
Algorithms, performance, theory.

Keywords
Algorithms, graph theory, graph isomorphism.

1. INTRODUCTION
The computation determining whether two undirected graphs are isomorphic is trivially exponential.[8] However, efficient algorithms exist \([9][7][1][3]\) that are polynomial-time procedures for significant graph classes, or are polynomial for all but a few pathological cases. We introduce an invariant among isomorphs of a graph, and an algorithm for its evaluation.[2] This algorithm performs as well as those in existence, but has a novel approach to isomorphism, and is interesting from several points of view. We compute a single unique binary number, the greatest characteristic string invariant for an undirected graph. The set of all optimizing permutations is \( \{v_1, v_2, \ldots, v_n\} \) such that

\[
a_i = \begin{cases} 
1 & \text{if } (v_i, v_j) \in E \\
0 & \text{otherwise}
\end{cases}
\]

Let \( r_i \) be the upper triangular part of the \( i \)-th row of \( A \), that is, \( r_i = [a_{1,i}, a_{2,i}, \ldots, a_{n,i}] \), \( 1 \leq i < |V| \).

We define the characteristic string of graph \( G(V,E) \), \( S_G \), as the concatenation of all the \( r_i \)’s in order, i.e., \( S_G = r_1 \circ r_2 \circ \ldots \circ r_{|V|-1} \).

The \( r_i \) is denoted as the \( i \)-th characteristic substring. Comparison operations on characteristic strings are performed by considering each characteristic string as though it were a binary number. For example, consider the graph \( G \) in Figure 1:

A permutation, (or map), \( p = [v_1, v_2, \ldots, v_n] \) is a one-to-one onto function \( p: V \to V \), corresponding to a reordering of vertices. Let the set of all possible permutations be \( \varphi \). We are interested in finding the greatest characteristic string (GCS) \( \tilde{S}_G \) of a graph \( G(V,E) \) by finding an optimizing permutation \( p(V(G)) \) of the vertices of \( G \), generating an isomorphic graph \( G'(p(V(G)), E) \) such that, for all permutations \( p \in \varphi \) on \( V(G) \), none will produce a graph \( G'(p(V(G)), E) \) with \( S_{G'} > S_G \). The set of all optimizing permutations is \( P(G) \). For
the graph \( G \) in Figure 1, \( ̂S = 110010 \) and the set of permutations
that corresponds to \( ̂S \) is \( P(G) = \{[2413][4231]\} \).

We will similarly define the least characteristic string (LCS), denoting it as \( S_0 \). In the example graph of Figure 1, \( S_0 = 001101 \) with permutations \([1 3 4 2]\) and \([3 1 4 2]\).

\[
\begin{array}{ccc}
1 & 2 & 3 \\
\begin{bmatrix}
0 & 1 & 0 & 0 \\
1 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 \\
0 & 1 & 1 & 0
\end{bmatrix} & T_1 = 0 & 0 0 0 1 \\
2 & 3 & 4
\end{array}
\]

\[
\begin{array}{ccc}
1 & 2 & 3 \\
\begin{bmatrix}
0 & 1 & 0 & 0 \\
1 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 \\
0 & 1 & 1 & 0
\end{bmatrix} & T_2 = 0 & 1 0 1 1
\end{array}
\]

\[
\begin{array}{ccc}
1 & 2 & 3 \\
\begin{bmatrix}
0 & 1 & 0 & 0 \\
1 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 \\
0 & 1 & 1 & 0
\end{bmatrix} & T_3 = 0 & 1 1 0 0
\end{array}
\]

\[
\begin{array}{ccc}
1 & 2 & 3 \\
\begin{bmatrix}
0 & 1 & 0 & 0 \\
1 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 \\
0 & 1 & 1 & 0
\end{bmatrix} & T_4 = 0 & 1 1 0 0
\end{array}
\]

Figure 1 Example Graph

Because of the diagonal symmetry of the adjacency matrix of an undirected graph, and since the upper triangular part of its adjacency matrix contains every adjacency, and since \( ̂S \) is derived from some permutation of the vertices of the graph, we have

Theorem I: Two graphs \( G_1 \) and \( G_2 \) are isomorphic if and only if
\[
\tilde{S}(G_1) = \tilde{S}(G_2)
\]

Corollary: The \( \tilde{S}(G) \) of a graph \( G \) is an invariant within the set of graphs isomorphic to \( G \).

Similarly,

Theorem II: Two graphs \( G_1 \) and \( G_2 \) are isomorphic if and only if
\[
\tilde{S}(G_1) = \tilde{S}(G_2)
\]

Corollary: The \( \tilde{S}(G) \) of a graph \( G \) is an invariant within the set of graphs isomorphic to \( G \).

Thus, computing whether two graphs \( G_1 \) and \( G_2 \) are isomorphic is equivalent to computing \( \tilde{S}(G_1) \) and \( \tilde{S}(G_2) \) and testing the GCS's for equality. \( \tilde{S}(G) \) is computed by an algorithm (given below) which finds one or more permutations in the set \( P(G) \).

The definitions and examples below are useful in understanding this process.

A partition \( T = [c_1, c_2, \ldots, c_n] \) of a graph \( G \) is a sequence disjoint, non-empty sets \( c_i \subseteq V \) such that \( \bigcup_{i=1}^{n} c_i = V \) and \( \forall i \neq j, c_i \cap c_j = \emptyset \). We refer to a \( c_i \) as a cell.

A partition \( T' = [c_1 \times c_2 \times \cdots \times c_n] \) induces a set of permutations \( P'_T = \{[c_1 \times c_2 \times \cdots \times c_n]\} \subseteq \emptyset \). For example, if \( T = \{[24]\{13]\} \), the \( T \) induces the permutations \([2413],[2431],[4213],[4231]\) whereas \([24][13][4231]\) induces the permutations \([2431],[2413]\). A complete partition is one in which \( |c_i| = 1 \) for \( i = 1 \ldots n \). That is, a complete partition induces a single permutation.

In the algorithm given below, it will be convenient to define a structured partition. An \( h \)-partition is a partition \( T_h = [\text{HEAD} \# \text{TAIL}] = [c_1, c_2, \ldots, c_\lambda, c_{\lambda+1}, \ldots, c_n] \) where

\[
\text{HEAD} = \emptyset \quad h = 0 \\
c_1, c_2, \ldots, c_\lambda \quad 0 < h \leq n \\
\text{TAIL} = c_{\lambda+1}, \ldots, c_n
\]

Furthermore, the cells of \( \text{HEAD} \) are singleton cells, i.e., \( \forall c_i \in \text{HEAD}, |c_i| = 1 \). We denote \( c_{h,1} \), the rightmost cell of \( \text{HEAD} \), as the locked cell, and its member vertex as the locked vertex. We denote \( c_{h,1} \), the leftmost cell of \( \text{TAIL} \), as the candidate cell.

a. Additional observations

The complement of a graph \( G \) is \( G = G(V,E') \) where

\[
E' = V \times V - E - [(v, v) \in E]
\]

In other words, all the 0’s and 1’s in the adjacency matrix are inverted. The 1’s and 0’s of the greatest characteristic string of \( G \) can be similarly inverted to yield the least characteristic string (LCS) of \( G' \). Thus,

Theorem III: \( \tilde{S}(G) = \tilde{S}(G') \)

It is clear that for graphs that are full or nearly full, the number of auto-isomorphs will be large, and thus the potential length of the partition list is large. While Theorem III does not help in evaluating the GCS of full or nearly full graphs, the long partition list may be avoided by the use of Theorem II. It is therefore recommended that, if two graphs having more than half the maximum number of edges are to be tested for being isomorphic, the GCS's of the complement of the graphs be calculated and compared.

b. Inforamal description of algorithm

We will use the definition of a structured partition given at the end of section 2 to aid in the description of the algorithm. The head of a partition is a static sequence of singleton cells, each containing a “locked” vertex, corresponding to the partition of the permutation matrix already determined. The tail is a dynamic

3. ALGORITHM FOR DETERMINING THE GCS

a. Overview

We now provide an overview of the algorithm to generate the GCS of a graph. The algorithm starts with a list containing a single partition having a single cell containing the set \( V \), and performs \( n \leq |V| - 2 \) iterations. Each iteration generates one or more refined partitions from every partition in the partition list in the previous iteration, and possibly removes (at the end of that iteration) some partitions. This may be thought of as respectively building and pruning a partition tree in level-order. We assert that Theorem IV, (3.d below), will hold.

b. Informal description of algorithm

We will use the definition of a structured partition given at the end of section 2 to aid in the description of the algorithm. The head of a partition is a static sequence of singleton cells, each containing a “locked” vertex, corresponding to the partition of the permutation matrix already determined. The tail is a dynamic
sequence of cells, each of which—unless it is singleton—is to be further refined. In each iteration, a new current locked vertex is extracted from the candidate cell. This adds exactly one new cell, containing the new locked vertex, to the right of the head.

We now describe the process of selecting the current (new) locked vertex from a given partition in the partition list. If the cardinality of the candidate cell is one, it becomes the current locked vertex. Otherwise, all the vertices in the candidate cell are initially denoted as "candidate" vertices. Now, for each candidate, the number of adjacencies of that vertex with vertices in the left-most cell of the tail is determined. Those candidates that have a number of adjacencies equal to the maximum adjacency survive as candidates. If a single candidate survives, that vertex is extracted from the candidate cell, and becomes the current locked vertex. If there are multiple candidates, the number of adjacencies of each of those vertices with vertices in the next cell in the tail is determined. Only those candidates that have a number of adjacencies equal to the maximum adjacency survive. This process continues until either there is a single candidate left (and used to form the current locked cell), or all partitions in the tail have been examined. In the latter case, each surviving candidate is installed in the current locked cell of first the current partition, and then replicates of the current partition which are then linked into the partition list following the original current partition.

The partition list is then traversed, and each partition (with its new current locked cell) has its tail refined. The tail is refined by splitting each cell into at most two cells, the leftmost containing all vertices adjacent to the current locked vertex, and the rightmost containing the remainder. Any null cells (having cardinality of zero) thus created are deleted. A string of 1’s (corresponding to adjacencies) and 0’s (corresponding to non-adjacencies) is maintained. After all partitions in the partition list are refined, and the maximum substring determined, all partitions whose substring is less than the maximum are deleted from the partition list. For example, partition 0 in Figure 4 is removed after the second iteration since its substring 000000 is less than the 001000 of the leftmost partition. Similarly, partitions 0 and 0 are removed after the third iteration since 01000 is less than 11000.

c. Formal specification of GCS algorithm

We will first describe three operations—h-select, h-refine, and h-substring—that will be used in the formal specification of the algorithm. This operation will also aid in the justification of the algorithm.

An h-select operation s(T, h) on a partition

\[ T_h = \{c_1, \ldots, c_a, c_{a+1}, \ldots, c_n\} \]

selects a set \( V_h \subseteq c_{a+1} \) of vertices from the candidate cell of \( T_h \) and generates a list \( L \) of \((h+1)-\)partitions, where \( |L| = |V_h| \geq 1 \). For each \( p' \in V_h \), there corresponds an element \( T_{h+1} \in L \) such that

\[ \text{HEAD}(T_{h+1}) = \text{HEAD}(T_h) \circ \{p'\} \]

\[ \text{TAIL}(T_{h+1}) = \left\{ \begin{array}{ll} \{c_{a+2}, \ldots, c_n\} & \text{if } |c_{a+1}| = 1 \\ \{c_{a+1} - \{p'\}, c_{a+2}, \ldots, c_n\} & \text{if } |c_{a+1}| > 1 \end{array} \right. \]

The set of newly locked vertices, \( V_h \), is chosen such that for each \( p' \in V_h \) the lexicographical sequence

\[ [q \mid q \in c_i, (p', q) \in E(G)] \]

where \( h' = \begin{cases} h+2 & \text{if } |c_{a+1}| = 1 \\ h+1 & \text{if } |c_{a+1}| > 1 \end{cases} \)

is at a maximum.1

An h-substring \( b(T_i, h) \) operation on a partition \( T_i = \{c_1, \ldots, c_a, c_{a+1}, \ldots, c_n\} \) generates a binary string of length \( |V| - h - 1 \) such that each successive cell \( c_i \in \text{TAIL}(T_i) \), \( h+1 \leq i \leq n \) contributes \( c_i \)'s (0's) if the vertices in \( c_i \) are (not) adjacent to the locked vertex.2

An h-refine operation \( r(T_i, h) \) on a partition \( T_i = \{c_1, \ldots, c_a, c_{a+1}, \ldots, c_n\} \) produces a single h-partition, say \( T_i' \). Here \( \text{HEAD}(T_i') = \text{HEAD}(T_i) \) and each cell \( c_i \in \text{TAIL}(T) \) is split into two adjacent disjoint cells \( c_i' \) and \( c_i'' \) such that \( c_i = \{q \mid p \subseteq c_i, q \in c_i, (p, q) \in E(G)\} \), and \( c_i' = c_i - c_i'' \). That is, those vertices of \( c_i \) adjacent to the locked vertex \( p \in c_i \) are mapped into \( c_i' \) and those not adjacent are mapped into \( c_i'' \). If either of the resulting cells \( c_i' \) or \( c_i'' \) is empty, this cell is discarded.3

Given these defined operations, we may now formally write the algorithm as shown in Figure 2.

d. Justification of validity of GCS algorithm

Theorem IV: After the last iteration of the GCS algorithm (Figure 2) the partition list, \( L \), contains partitions whose cells are singletons, and which represent a subset of permutations on the original graph whose characteristic string is maximal.

We now justify Theorem IV by an inductive argument. At the first iteration, the partition contains all vertices. In each iteration \( i \), the h-select operation generates a list of partitions such that the next vertex is fixed so as to leave the greatest number of 1’s (adjacencies) in the leftmost (candidate) cell, with ties being broken by looking at cells successively to the right, we have the material (1’s) for generating the largest number. Any remaining ties will branch the search tree. The h-refine operation shuffles vertices within the current cell so that adjacencies (1’s) are on the left and non-adjacencies (0’s) are on the right, guaranteeing that the current substring \( c_{a+1} \) is maximized. Since vertices are only shuffled within the current cell, the value of the previous

---

1 This corresponds to selecting as a locked vertex those vertices of the candidate cell that have the most adjacencies in the first cell of the residual tail (or successive cells in the case of a tie.)

2 This string will correspond to a potential (\( h+1 \)) characteristic substring of the GCS.

3 This corresponds to moving tail vertices around within their cell in such a way as to maximize \( b(T_i, h) \).
substring \( r \) will not be changed since the adjacencies of these vertices with the previous locked vertex are all the same. The operation also partitions the current cell into two cells, one for vertices adjacent to the new locked vertex, and one for vertices not adjacent to the new locked vertex. We thereby ensure that the adjacencies of all vertices within these newly formed cells with the new locked vertex are the same. Since each iteration locks exactly one vertex within all the partitions in the partition list, at the end all vertices are locked. Finally, since the fixed-size substrings have been maximized from most significant to least significant, the characteristic string is maximized (for all the partitions in the partition list). Each partition represents a permutation of the original graph to yield a graph with a greatest characteristic string.

Function \( \text{GCS}(G(V,E)) \)

Set \( T_0 = [c_1] \), where \( c_1 = V(G) \)
\[
L = (T_0)
\]
\[
\tilde{S}_G = \emptyset
\]
for \( i = 0 \) to \( |V| - 2 \)
\[
L' = L, L = \bigcup_{T_i \in L} h\text{-select}(T_i, i)
\]
\[
\forall T_i \in L, T_i = h\text{-refine}(T_i, i)
\]
\[
s_{\text{max}} = \max_{T_i \in L} (h\text{-substring}(T_i, i))
\]
\[
L = (T_i | T_i \in L, h\text{-substring}(T_i, i) = s_{\text{max}})
\]
\[
\tilde{S}_G = \tilde{S}_G \circ s_{\text{max}}
\]

Return \( \tilde{S}_G \)

Figure 2 A formal specification of the GCS algorithm

4. IMPLEMENTATION OF CCS ALGORITHM

In Figure 3 we give the pseudocode for the realization of the GCS algorithm. The implementation is quite similar to the informal description of the algorithm given in section 3.b.

Two optimizations can be performed in the implementation of this algorithm. In the first, if at the end of the current iteration the total number of 1’s encountered is equal to the number of edges in the original graph, the loop can be terminated, and the appropriate number of 0’s appended to the GCS. In the second, if the tail of the current partition is identical to the tail of the previous partition in the partition list, that partition can be deleted (since future substrings generated from this partition must necessarily be identical). For example, partition \( \emptyset \) in Figure 4 is removed since the tail is identical to that in the leftmost column. A weaker, but less costly to implement, condition is that if the sequence of vertices in the tail— independent of partition boundaries—is the same as the previous partition, this partition may be deleted.

algorithm GCS:
calcSignature(graph)
{
initialize partition_list (one partition containing one cell with all vertices)
for each level
{
   // Find candidate vertices (create new partitions)
   for each partition in partition_list
   {
      if |candidate cell| > 1
      {
         link all vertices in candidate cell to form candidate_list
         for each cell (including candidate cell)
         {
            max_degree = 0;
            for each candidate vertex
            {
               get degree of candidate in current cell
               update max_degree
            }
            delete all candidates with degree < max_degree from\ candidate_list
            swap (first eligible candidate, first vertex in candidate cell)
            for all remaining eligible candidates
            {
               create clone of current partition
               swap (eligible candidate, first vertex in candidate cell)
               insert this new partition into partition_list
            }
         }
      }
   }
   // Refine partition
   for each partition in partition_list
   {
      for each cell in tail of partition (right of locked vertex)
      {
         if necessary, split cell into two cells
         (the first containing the eligible (locked) vertex, and the second containing the remainder)
      }
   }
   // Remove partitions whose row_string is less than max string
   for all partitions after the first in partition_list
   {
      if row_string(current partition) < row_string(first partition)
      delete partition
      else if row_string(current partition) > row_string(first partition)
        delete all partitions up to current one & restart loop
      }
   }
   if all edges have been seen
   break out of loop
}

Generate GCS (max string) from first partition in partition_list
}

Figure 3 Implementation (pseudocode) of GCS algorithm
a. Performance of GCS Algorithm

Figure 5 shows the length of the partition list at the termination of the GCS algorithm for all possible graphs having 7 vertices, with all optimizations in effect (terminate when all edges have been seen, and delete partition with duplicated tails). It can be seen that the highest frequency of partition list lengths is 1. In fact, the average list length over all graphs of size 7 is 1.39. There is a relatively small number of graphs with a sufficient degree of auto-isomorphism to produce longer partition lists. For larger graphs, the average is found experimentally to be increasingly close to 1, as the probability for auto-isomorphism decreases. Thus, the average branching of the partition tree is very close to one, and the total computation is proportional to the height of the tree, \( |V| \), times the average size of the partition tail, \( O(|V|) \). The expected complexity of the algorithm is thus \( O(|V|^2) \).

Figure 6 gives experimental timing results for random graphs of sizes in the range 10–6500 vertices, with each representative size having 10 graphs with a number of edges being a random number between \( 2|V| \) and \( |V|(|V| - 1)/4 \) (i.e., less than half the maximum possible number of edges). In this figure, the line on the log-log graph has a slope of 2, representing an \( O(|V|^2) \) complexity, and it can be seen that the data points closely parallel this line. This confirms the theoretical complexity discussed above. It can also be seen that there are a few outliers, which correspond to graphs that coincidentally have a regularity or structure with large auto-isomorphism. This effect is, as would be expected, much larger at the low end of the size scale.

\[ \text{The jog in the graph at ~800 vertices may be due to cache limitations. After this jog, the graph seems to again revert to the } O(V^2) \text{ trend. Virtual memory was not involved below 6500 vertices (except possibly for pathological cases.)} \]

Figure 4 Example Graph.
5. CONCLUSION
An invariant for an undirected graph, the GCS, has been defined, and an efficient algorithm given for its evaluation. This invariant and can be used to uniquely determine whether two graphs are isomorphic, and is implementation- and well as architecture-independent (unlike the canonically labeled isomorph produced in nauty [6] used for the same purpose.) The algorithm for producing the GCS also produces a permutation matrix to map the vertices into a canonical form.

We are currently investigating other uses for this invariant, including isomorphism of directed graphs.

6. ACKNOWLEDGEMENTS
We thank Elena Galaktionova for her thoughtful perusal of drafts of this paper.

REFERENCES