Graph Isomorphism Testing Without Full Automorphism Group Computation*

José Luis López-Presa
DIATEL, Universidad Politécnica de Madrid, Spain, jllopez@diatel.upm.es

Antonio Fernández
GSyC, Universidad Rey Juan Carlos, Spain, afernandez@acm.org

May 3, 2004

Abstract

In this paper we present an algorithm for testing the isomorphism of two graphs. The algorithm works in three steps. First, it builds a sequence of partitions on the vertices of one of the graphs. Then, it looks for some automorphisms in that graph. Finally, it uses backtracking to try to find another sequence of partitions for the second graph that is compatible with that for the first graph.

We compare the performance of an implementation of this algorithm with other isomorphism testing programs. For this purpose we have chosen *nauty*, that is the fastest program we know of (it works computing a canonical form of the graphs), and *vf2* that uses a completely different approach which looks useful for certain types of graphs. Several types of graphs have been used for the tests in their directed and undirected versions. Our program is faster that the other two in some cases, and behaves more uniformly in all of them.

Keywords: graph isomorphism, automorphism discovery, backtracking algorithm.

1 Introduction

The general graph isomorphism problem has been traditionally faced using mainly two different approaches. A first approach uses backtracking to find a mapping between the vertices of the graphs by exploring the full search tree, using heuristics to prune that tree. This approach can be very fast for irregular graphs and graphs with small automorphism groups. This approach has been used, for example, by Ullmann [13], and Cordella et al. [1]. However, this kind of algorithms can spend a long time exploring automorphic solutions (paths in the search tree) that will not lead to a valid match. This is especially serious when the graphs being tested are not isomorphic (since all possibilities are checked).

The second approach obtains a canonical form for each of the graphs to be checked, and then compares them for equality. The algorithms in the literature that use this approach, also use backtracking, but they learn about automorphisms and use this knowledge to prune the search tree. However, a canonical form is not always easy to construct (efficiently). This technique is used by the fastest general isomorphism programs, like McKay's *nauty* [8] which applies the ideas presented in [6]. Previously, Weisfeiler and Lehman had discussed in [14] the problem of finding a canonical form of a graph and using a refinement technique to find stable partitions of the vertices in a graph that has been used by many others. Similar algorithms are described in [5] and [12].

In this paper we present an algorithm that uses a combination of both techniques to check whether two given graphs are isomorphic. Like in the first approach, it relies on a backtracking algorithm that tries to find a mapping between the graphs, but instead of using heuristics to prune the search tree, prior to the

*Partially supported by the Spanish MCyT under grant TIC2001-1586-C03-01 and the Comunidad de Madrid under grant 07T/0022/2003.

search it tries to find some automorphisms in one of the graphs. Then, this automorphisms are used to prune the search tree. This helps when the graphs have many automorphisms. Unlike the algorithms that use the second approach above, our algorithm does not try to compute the full automorphism group.

Our algorithm works in three phases. First, it builds a sequence of vertex partitions for one of the graphs, using modifications of well known techniques (like those presented in [14]). Then, it tries to discover automorphisms in that graph, without backtracking. Finally, using backtracking, it tries to generate, for the second graph, a sequence of partitions compatible with the one previously generated for the first graph. If this is possible, both graphs are isomorphic. If it is not possible, they are not isomorphic.

When the graphs tested have a large automorphism group, like Strongly Regular Graphs or Cubic Symmetric Graphs, or graphs based on Fürer gadgets like those described in [4] (and used by Miyazaki in [9]) it would be extremely hard to find the compatible sequence of partitions for the second graph without knowing the automorphisms of the graphs. Since determining the whole automorphism group can be costly, our algorithm only tries to find those automorphisms whose knowledge will be useful in the third step and which do not need backtracking to be found. Then the information obtained about automorphisms in the first graph can be used during the generation of the second sequence of partitions to prune the search tree induced by the backtracking algorithm.

We have evaluated the space complexity of the algorithm and we show here that the amount of storage required is $O(n^2)$. Unfortunately we have not been able to bound the time complexity beyond the trivial bounds. Hence, in order to evaluate the practical performance of our algorithm, we have implemented it. Then we have compared the performance of this implementation with other isomorphism testing programs. For this purpose, we have chosen *nauty* [8, 7], which uses the second approach above and is considered the fastest isomorphism testing program, and *vf2* [11], that uses the first approach above and has been shown to work faster than nauty for certain types of graphs. Several types of graphs have been used for the tests in their directed and undirected versions. We have found that our program is faster than the other two in some cases. Furthermore, for all cases it behaves uniformly, while the other two have specific classes of graphs that make them to take a large amount of time even for small sizes.

The rest of the paper is organized as follows. In Section 2 we define some basic concepts that will be used widely throughout the paper. In Section 3 we describe the algorithm in detail. We present an example of the operation of the algorithm in Section 4, which shows most of its functionality and power. In Section 5 we prove the correctness of the algorithm along with all the assertions made, but not proved, in Section 3. Section 6 tries to analyze the complexity of the algorithm in time and space. In section 7 we compare the performance of our implementation with that of the other two using different families of graphs of up to 1000 vertices.

2 Basic Definitions

A *directed graph* $G = (V, R)$ consists of a finite set V of vertices and a binary relation R, i. e. a subset $R \subseteq V \times V$[1]. The elements of R are called *arcs*. An arc $(u, v) \in R$ is considered to be oriented from u to v. An *undirected graph* is a graph whose arc set R is symmetrical, i.e., $(u, v) \in R$ iff $(v, u) \in R$. From now on, we will use the term *graph* to refer to a *directed graph*. Undirected graphs are just a particular case of directed graphs.

Given a graph $G = (V, R)$, R can be represented by an *adjacency matrix* [2] $Adj(G) = A$ with size $|V| \times |V|$ in the following way:

$$A_{ij} = \begin{cases} 0 & \text{if } (i,j) \notin R \wedge (j,i) \notin R \\ 1 & \text{if } (i,j) \notin R \wedge (j,i) \in R \\ 2 & \text{if } (i,j) \in R \wedge (j,i) \notin R \\ 3 & \text{if } (i,j) \in R \wedge (j,i) \in R \end{cases}$$

Given a graph $G = (V, R)$ and its adjacency matrix $Adj(G) = A$, the *degree* [3] of a vertex $v \in V$ under

[1] Multigraphs are not considered in this paper.

[2] Note the difference with the traditional definition of the adjacency matrix where $A_{ij} = 1$ if $(i, j) \in R$ and $A_{ij} = 0$ if $(i, j) \notin R$.

[3] Note the difference with the traditional degrees. This is a combination of the in-degree, the out-degree and the number of neighbors of a vertex.

graph G, denoted by $Deg(v, G)$, is the 3-tuple (D_3, D_2, D_1) where $D_i = |\{u \in V : A_{vu} = i\}|$ for $i \in \{1, 2, 3\}$. Let $V_1 \subseteq V$. The *available degree* of v in V_1 under G, denoted by $ADeg(v, V_1, G)$, is the 3-tuple (D_3, D_2, D_1) where $D_i = |\{u \in V_1 : A_{vu} = i\}|$ for $i \in \{1, 2, 3\}$.

Extending the notation for some $V_1 \subseteq V$ such that $\forall u, v \in V_1, Deg(u, G) = Deg(v, G) = d$, we define $Deg(V_1, G) = d$. The same extension can be applied to the available degree. Let $V_2 \subseteq V$ and $V_1 \subseteq V$ such that $\forall u, v \in V_1, ADeg(u, V_2, G) = ADeg(v, V_2, G) = d$. Then, $ADeg(V_1, V_2, G) = d$.

We will say a 3-tuple $(D_3, D_2, D_1) \prec (E_3, E_2, E_1)$ when the first one precedes the second one in lexicographic order and $(D_3, D_2, D_1) \succ (E_3, E_2, E_1)$ when the second one precedes the first one in lexicographic order. This notation will be used to order the degree and the available degree of both vertices and sets.

A *partition* of a set S is a sequence $\mathcal{S} = (S_1, ..., S_r)$ of disjoint nonempty subsets of S such that $S = \bigcup_{i=1}^{r} S_i$. The sets S_i are called the *cells* of $\mathcal{S}$.

The symbol $\emptyset$ denotes the empty partition and the empty set, and the symbol $\circ$ denotes the partition concatenation operator. Let $\mathcal{S} = (S_1, ..., S_r)$ and $\mathcal{S}' = (S'_1, ..., S'_s)$ be partitions of two disjoint sets S and S', respectively. Then, $\mathcal{S} \circ \mathcal{S}' = (S_1, ..., S_r, S'_1, ..., S'_s)$. Clearly, $\emptyset \circ \mathcal{S} = \mathcal{S} = \mathcal{S} \circ \emptyset = \mathcal{S}$.

Let $G = (V, R)$ be a graph. The *degree partition* of V in G, denoted by $DegreePartition(G)$, is the empty partition if $V = \emptyset$, and, otherwise, it is a partition $\mathcal{V} = (V_1, ..., V_r)$ of V such that:

1 $\forall i \in \{1, ..., r\}, \forall v, u \in V_i, Deg(v, G) = Deg(u, G)$
2 $\forall i, j \in \{1, ..., r\}, i < j$ implies $Deg(V_i, G) \succ Deg(V_j, G)$

Let $G = (V, R)$ be a graph, $v \in V$, $V_1 \subseteq V \setminus \{v\}$, and $Adj(G) = A$. The *vertex partition* of V_1 by v, denoted $PartitionByVertex(V_1, v, G)$, is the empty partition $\emptyset$ if $V_1 = \emptyset$, and, otherwise, a partition $(S_1, ..., S_r)$ of V_1 such that:

1 $\forall i \in \{1, ..., r\}, \forall w, u \in S_i, A_{vw} = A_{vu}$
2 $\forall i, j \in \{1, ..., r\}, i < j$ implies $\forall u \in S_i, \forall w \in S_j, A_{vu} > A_{vw}$

Let $G = (V, R)$ be a graph, and $V_1, V_2 \subseteq V$. The *set partition* of V_1 by V_2, denoted $PartitionBySet(V_1, V_2, G)$, is a partition $(S_1, ..., S_r)$ of V_1 such that:

1 $\forall i \in \{1, ..., r\}, \forall v, u \in S_i, ADeg(v, V_2, G) = ADeg(u, V_2, G)$
2 $\forall i, j \in \{1, ..., r\}, i < j$ implies $ADeg(S_i, V_2, G) \succ ADeg(S_j, V_2, G)$

Let $G = (V, R)$ be a graph with $Adj(G) = A$, and $V_1, V_2 \subseteq V$ such that $\forall u, v \in V_1, ADeg(u, V_2, G) = ADeg(v, V_2, G)$. Let $ADeg(V_1, V_2, G) = (D_3, D_2, D_1)$. Then, $NumLinks(V_1, V_2, G) = D_3 + D_2 + D_1$, and $HasLinks(V_1, V_2, G) = (NumLinks(V_1, V_2, G) > 0)$.

A *permutation* $\pi : V \longrightarrow V$ acting on the finite set V is a one-to-one mapping from V onto itself. The image of an element $x \in V$ with respect to the permutation π is denoted by $\pi(x)$.

Let $G = (V, R_G)$ and $H = (V, R_H)$ be two graphs with the same vertex set. A permutation $\pi : V \longrightarrow V$ of V is called an *isomorphism* of G and H if $\forall u, v \in V, (v, u) \in R_G \iff (\pi(v), \pi(u)) \in R_H$. G and H are called *isomorphic*, written $G \simeq H$, if there is at least one isomorphism π of them.

Let $G = (V, R)$ be a graph. An *automorphism* of G is an isomorphism of G and itself.

3 The Algorithm

This section describes the algorithm that tests if two graphs are isomorphic. Algorithm *AreIsomorphic*, shown in Figure 1, receives two graphs G and H as parameters and returns TRUE if both graphs are isomorphic, and FALSE if they are not.

This algorithm tests first if both graphs have the same number of vertices and arcs. It is easy to see that this is a necessary condition for isomorphism. Then, it generates initial partitions of the vertices of both graphs based on their degrees; $\mathcal{S}^0$ is the degree partition of G and $\mathcal{T}^0$ the degree partition of H. If these partitions are not compatible (G and H differ in the number of vertices of some degree), the graphs cannot be isomorphic. Generating the degree partitions and checking for their compatibility is fast and can simplify the search for an isomorphism between G and H, since vertices in one cell of $\mathcal{S}^0$ can only be mapped to vertices in the corresponding cell of $\mathcal{T}^0$ (they can only be mapped to vertices with their same degree).

```
AreIsomorphic(G, H) : boolean
1  - - let G = (V_G, R_G) and H = (V_H, R_H)
2  if (|V_G| ≠ |V_H|) ∨ (|R_G| ≠ |R_H|) then
3      return FALSE
4  else
5      S^0 ← DegreePartition(G)
6      T^0 ← DegreePartition(H)
7      if ¬DegreePartitionsAreCompatible(G, H) then
8          return FALSE
9      else
10         GenerateFirstSequenceOfPartitions(G)
11         SearchAutomorphisms(G)
12         return Match(0, G, H)
13     end if
14 end if
```

Figure 1: Algorithm that tests whether G and H are isomorphic.

Unfortunately, for regular graphs, this *DegreePartition* has only one cell, which means that each vertex in one partition (or graph) can be mapped to any one in the other partition (or graph).

Algorithm *DegreePartitionsAreCompatible*, shown in Figure 2, tests if $\mathcal{S}^0$ under graph G and $\mathcal{T}^0$ under graph H are compatible. It checks whether they have the same number of cells, and whether cells in the same position in their respective partitions have the same degree and size.

```
DegreePartitionsAreCompatible(G, H) : boolean
1  if |S^0| ≠ |T^0| then
2      return FALSE
3  else
4      - - let r = |S^0| = |T^0| , S^0 = (S_1^0, ..., S_r^0) , T^0 = (T_1^0, ..., T_r^0)
5      return ∀i ∈ {1, ..., r}, (|S_i^0| = |T_i^0|) ∧ (Deg(S_i^0, G) = Deg(T_i^0, H))
6  end if
```

Figure 2: Algorithm that tests whether the degree partitions of G and H are compatible.

If the degree partitions $\mathcal{S}^0$ and $\mathcal{T}^0$ are compatible, algorithm *GenerateFirstSequenceOfPartitions* is used to generate a sequence of partitions $\mathcal{S}^0, ..., \mathcal{S}^{ll}$ for graph G. Then, considering this sequence, graph G is searched for automorphisms. This search does not attempt to discover all the automorphisms in G. Instead, it only tries to find those that will later reduce the search for a sequence of partitions $\mathcal{T}^1, ..., \mathcal{T}^{ll}$ (equivalent to the sequence for G) for graph H. Finally, algorithm *Match* attempts to generate this sequence of partitions $\mathcal{T}^1, ..., \mathcal{T}^{ll}$ for graph H. If algorithm *Match* succeeds, G and H are isomorphic.

3.1 Generation of the Sequence of Partitions for Graph G

Algorithm *GenerateFirstSequenceOfPartitions*, shown in Figure 3, starts from the degree partition $\mathcal{S}^0$ of graph G and generates successive partitions until it finds a partition $\mathcal{S}^{ll}$ such that the vertices in cells with more than one vertex have no adjacencies with the remaining vertices in that partition. The generation of each new partition $\mathcal{S}^{l+1}$ from its previous one $\mathcal{S}^l$ is done in the following way:

First, all the cells without links are discarded. This reduces the complexity of the problem reducing the number of vertices to be handled, without loss of information about adjacencies, since a discarded vertex has no adjacencies with the remaining vertices in the partition.

Then, among the others:

1. If there are cells of size one, one of them is chosen as the *pivot set* for that partition and its only vertex is used to split the other cells (with algorithm *RefineByVertex*).

2. Otherwise, the algorithm looks for a cell that is able to split some cell (maybe itself) in that partition. If such a cell is found, it is chosen as the pivot set, and it is used to generate $\mathcal{S}^{l+1}$ (with algorithm *RefineBySet*).

3. If no cell meeting the condition of Case 2 has been found, then some cell is chosen as the pivot set, and a vertex in that cell is used to generate $\mathcal{S}^{l+1}$ (with algorithm *RefineByVertex*).

P^l stores which of the previous three cases has been met to partition $\mathcal{S}^l$, t^l identifies the pivot set used (S_{t^l} is the pivot set for $\mathcal{S}^l$), and p^l is the pivot vertex used in cases 1 and 3. The values P^l and t^l are necessary to generate the sequence of partitions $\mathcal{T}^1, ..., \mathcal{T}^{ll}$ for graph H using these same refinements and the corresponding pivot sets.

We use an attribute *Valid* of the cells to improve the performance in case 2. This attribute stores whether a cell is marked as valid or invalid. We mark a cell S_i^l as invalid if we know that it cannot split any of the cells in $\mathcal{S}^l$, and it is valid otherwise (if it has not been proved to be invalid). Before a cell is used as the pivot set for a refinement by set, it is marked invalid in advance, because, if it is not able to split any cell, it will be invalid, and if it is able to split some cell, once it has been used, it has split all the cells to its best, and it will never be able to split any of the subcells it has generated (otherwise, it would have split them at this point). This way, if it dos not split itself with this refinement, it will remain invalid, whilst if it does, its subcells will be valid, since they are new, and we still do not know if they will be able to split some cell in future refinements.

```
GenerateFirstSequenceOfPartitions(G)
1  for each S_i^0 ∈ S^0 do
2      Valid(S_i^0) ← (|S^0| > 1) ∧ HasLinks(S_i^0, V, G)
3  end for
4  l ← 0
5  - - let r = |S^l| , S^l = (S_1^l, ..., S_r^l) and V^l = ⋃_{i=1}^r S_i^l
6  while ∃S_j^l ∈ S^l : (|S_j^l| > 1) ∧ HasLinks(S_j^l, V^l, G) do
7      t^l ← IndexBestPivot(l, G)
8      if |S_{t^l}^l| = 1 then
9          P^l ← VERTEX
10         p^l ← any vertex in S_{t^l}^l
11         RefineByVertex(l, G)
12     else
13         success ← FALSE
14         while Valid(S_{t^l}^l) ∧ ¬success do
15             Valid(S_{t^l}^l) ← FALSE
16             P^l ← GROUP
17             RefineBySet(l, success, G)
18             if ¬success then
19                 t^l ← IndexBestPivot(l, G)
20             end if
21         end while
22         if ¬success then
23             P^l ← UNKNOWN
24             p^l ← any vertex in S_{t^l}^l
25             RefineByVertex(l, G)
26         end if
27     end if
28     l ← l + 1
29 end while
30 ll ← l
```

Figure 3: Algorithm that generates the sequence of partitions for graph G.

The task of choosing the pivot set among a set of cells is done by algorithm *IndexBestPivot*, shown in Figure 4. This algorithm behaves as follows:

- It chooses a cell with only one vertex and which has links, if such a cell exists. This corresponds to Case 1 above.
- If there is not such a cell, it chooses some valid cell. Algorithm *GenerateFirstSequenceOfPartitions* will check whether this cell breaks some cell and, if so, will stick to it as the pivot set (Case 2 above). Otherwise, it marks the cell as invalid and asks *IndexBestPivot* for another candidate.
- If there is not such a cell, it chooses some cell in the partition. This happens if we are in Case 3 above.

```
IndexBestPivot(l, G) : integer
1  - - let r = |S^l| , S^l = (S_1^l, ..., S_r^l) and V^l = ⋃_{i=1}^r S_i^l
2  b ← 1
3  for i ← 2 to r do
4      if Valid(S_i^l) then
5          if ¬Valid(S_b^l) ∨ (|S_b^l| > |S_i^l|) ∨ ((|S_i^l| = 1) ∧ (NumLinks(S_i^l, V^l, G) > NumLinks(S_b^l, V^l, G))) then
6              b ← i
7      else
8          if ¬Valid(S_b^l) ∧ HasLinks(S_i^l, V^l, G) ∧ (¬HasLinks(S_b^l, V^l, G) ∨ (|S_b^l| > |S_i^l|)) then
9              b ← i
10     end if
11 end for
12 return b
```

Figure 4: Algorithm that finds the best set $S_i^l \in \mathcal{S}^l$ to be used as a pivot.

IndexBestPivot uses some heuristics to break ties that, in our experiments, have shown to be efficient. It first chooses the cell of smallest size, and among cells of the same size, it chooses the cell with the smallest index. The order in which the cells are chosen as pivots has a deep impact on the ability of the algorithm to find automorphisms, and therefore, on its performance.

```
RefineByVertex(l, G)
1  - - let r = |S^l| , S^l = (S_1^l, ..., S_r^l) and V^l = ⋃_{i=1}^r S_i^l
2  S^{l+1} ← ∅
3  for i ← 1 to r do
4      if HasLinks(S_i^l, V^l, G) then
5          if i = t^l then
6              X ← PartitionByVertex(S_i^l \ {p^l}, p^l, G)
7          else
8              X ← PartitionByVertex(S_i^l, p^l, G)
9          end if
10         S^{l+1} ← S^{l+1} ∘ X
11         for each S_j^{l+1} ∈ S^{l+1} : S_j^{l+1} ∈ X do
12             Valid(S_j^{l+1}) ← Valid(S_i^l) ∨ (|X| > 1) ∨ (i = t^l)
13         end for
14     end if
15 end for
16 - - let r = |S^{l+1}| , S^{l+1} = (S_1^{l+1}, ..., S_r^{l+1}) and V^{l+1} = ⋃_{i=1}^r S_i^{l+1}
17 for each S_j^{l+1} ∈ S^{l+1} : ¬HasLinks(S_j^{l+1}, V^{l+1}, G) do
18     Valid(S_j^{l+1}) ← FALSE
19 end for
```

Figure 5: Algorithm that generates $\mathcal{S}^{l+1}$ applying vertex partition on $\mathcal{S}^l$.

Algorithm *RefineByVertex* is shown in Figure 5. It generates a new partition $\mathcal{S}^{l+1}$ from $\mathcal{S}^l$ by applying *PartitionByVertex* to every cell with links in the partition. The cells without links are discarded since all

their adjacencies have been previously dealt with and all their adjacent vertices have been already discarded (if there were any). Moreover, all the vertices in a cell without links are equivalent (they are all adjacent to the same vertices in the same way). The vertex used to compute the partition by vertex is also discarded, because all its adjacencies are used up at this point. If the vertex refinement is applied with a pivot set with more than one vertex, any vertex in that set can be chosen as the pivot vertex. This vertex is stored in p^l so it can be used during the search for automorphisms.

The cells in the new partition are valid if they come from a cell in the previous partition which was already valid, or they are new cells (proper subsets of a cell in the previous partition); being new cells, we still do not know if they will be able to split any cell or not. Finally, when a cell has lost all its links, it becomes invalid, since it will not be able to split any cell.

Algorithm *RefineBySet* is shown in Figure 6. It computes a new partition $\mathcal{S}^{l+1}$ from $\mathcal{S}^l$ by applying *PartitionBySet* to all the sets in the partition. As in the case of refinement by vertex, cells without links are discarded since they are useless. Parameter *success* tells the caller if the pivot set has been able to split at least one cell in the partition. Otherwise, it will be necessary to try another pivot set, or another kind of refinement. The cells in the new partition are marked valid if they come from a valid cell (note that the pivot cell has already be marked as invalid in algorithm *GenerateFirstSequenceOfPartitions* and therefore, if it does not split itself, it will remain invalid in the new partition) or they are new (they come from a cell that has been split by this refinement).

```
RefineBySet(l, success, G)
1  - - let r = |S^l| , S^l = (S^l_1, ..., S^l_r) and V^l = ⋃_{i=1}^{r} S^l_i
2  success ← FALSE
3  S^{l+1} ← ∅
4  for i ← 1 to r do
5      if HasLinks(S^l_i, V^l, G) then
6          X ← PartitionBySet(S^l_i, S^l_{t^l}, G)
7          S^{l+1} ← S^{l+1} ∘ X
8          for each S^{l+1}_j ∈ S^{l+1} : S^{l+1}_j ∈ X do
9              Valid(S^{l+1}_j) ← Valid(S^l_i) ∨ (|X| > 1)
10         end for
11         success ← success ∨ (|X| > 1)
12     end if
13 end for
```

Figure 6: Algorithm that generates $\mathcal{S}^{l+1}$ applying set partition on $\mathcal{S}^l$.

The sequence of partitions generated by algorithm *GenerateFirstSequenceOfPartitions* induces an order on the vertices of the graph. The order of the vertices is defined by the order in which they are discarded by the algorithm. This happens when a vertex is used as the pivot for a vertex refinement, and when a cell is discarded for not having links (in this case, the order among the vertices in the cell is irrelevant since they are equivalent, but the order among cells removed in the same refinement is indeed relevant). Finally, the vertices in the last partition follow all the previously removed vertices, and their relative order is defined by their position in this last partition (again, the order among the vertices in a cell is irrelevant). See Section 5 for more details.

3.2 Automorphisms Discovery

After generating the sequence of partitions for graph G, the algorithm searches for automorphisms in G. The aim of this search is to remove potential backtracking points in the search for the sequence of partitions for graph H. Backtracking will only be necessary when refinement by vertex is applied with a pivot set that has more than one vertex. In this case, P^l takes value UNKNOWN.

Algorithm *SearchAutomorphisms*, shown in Figure 7, starts with partition $\mathcal{S}^{ll-1}$ and runs backwards through the sequence of partitions $\mathcal{S}^0, ..., \mathcal{S}^{ll}$. When it finds a partition $\mathcal{S}^l$ such that P^l is UNKNOWN, it tests if all the vertices in the pivot set are equivalent (for more details, see Section 5). In such a case, P^l

```
SearchAutomorphisms(G)
1  for l ← ll − 1 downto 0 do
2      if P^l = UNKNOWN then
3          if TheVerticesInThePivotSetAreEquivalent(l, G) then
4              P^l ← VERTEX
5          else
6              P^l ← BACKTR
7          end if
8      end if
9  end for
10 end if
```

Figure 7: Algorithm that finds some automorphisms in the first graph.

will be changed to VERTEX. This way, during the generation of the sequence of partitions for graph H, it will be sufficient to try one of the vertices in the pivot set, since, if G and H are isomorphic, then the vertices in the pivot set must also be equivalent in partition $\mathcal{T}^l$ for graph H. All this is thoroughly discussed in Section 5. If not all the vertices in the pivot set are equivalent, backtracking may be necessary when generating partition $\mathcal{T}^{l+1}$ from partition $\mathcal{T}^l$ for graph H. In this case, P^l is set to BACKTR to indicate this fact.

Algorithm *TheVerticesInThePivotSetAreEquivalent*, shown in Figure 8, tests all the vertices in the pivot set for equivalence with the pivot vertex. If it finds a vertex in the pivot set which is not equivalent to the pivot vertex, it concludes that not all the vertices in the pivot set are equivalent and returns. Otherwise, it returns indicating that all of them are equivalent. This is why choosing the right cells as pivot sets is so crucial for the performance of the algorithm (it is directly related with the amount of possible backtracking needed by algorithm *Match* to find the sequence of partitions for graph H). Ideally, only cells with equivalent vertices should be taken as pivot sets. Algorithm *IsEquivalentToThePivotVertex*, shown in Figure 9, is used to test individual vertices for equivalence with the pivot vertex.

```
TheVerticesInThePivotSetAreEquivalent(l, G) : boolean
1  - - let r = |S^l| and S^l = (S^l_1, ..., S^l_r)
2  X ← S^l_{t^l} \ {p^l}
3  repeat
4      x ← any vertex in X
5      X ← X \ {x}
6      success ← IsEquivalentToThePivotVertex(l, x, G)
7  until (X = ∅) ∨ ¬success
8  return success
```

Figure 8: Algorithm that tests equivalence among the vertices in the pivot group at Partition $\mathcal{S}^l$.

Algorithm *IsEquivalentToThePivotVertex* tests if vertex x is equivalent to the pivot vertex for partition $\mathcal{W}^l$. To do so, it generates a sequence of partitions $\mathcal{W}^{l+1}, ..., \mathcal{W}^{ll}$. This partitions are stored in $\mathcal{X}$ (the old partition) and $\mathcal{Y}$ (the new partition) to minimize memory usage. Partition $\mathcal{W}^{l+1}$ is generated by applying vertex partitioning to $\mathcal{S}^l$, with x as the pivot vertex. If $\mathcal{W}^{l+1}$ and $\mathcal{S}^{l+1}$ are compatible, partition $\mathcal{W}^{l+2}$ is generated using the partitioning technique P^{l+1} used to generate partition $\mathcal{S}^{l+2}$ from $\mathcal{S}^{l+1}$. If they are compatible, new partitions are generated in this way and tested for compatibility, until the final partition $\mathcal{W}^{ll}$ is reached or incompatibility is found. If the algorithm finds an incompatibility, it returns FALSE. Otherwise, the final partitions are tested for equivalence with algorithm *FinalPartitionsAreEquivalent*. If they are equivalent (i.e. both partitions are the same), then x and p^l are equivalent. Note that in Line 24 of the algorithm of Figure 9 we are assuming that the evaluation of the boolean expression is done from left to right and only if needed (short circuit evaluation). Hence, *FinalPartitionsAreEquivalent* will be invoked only if *compatible* is TRUE and *equivalent* is FALSE (and hence $m = ll$).

If there is an i such that partition $\mathcal{W}^i$ and partition $\mathcal{S}^i$ are the same, then it is not necessary to go on

```
IsEquivalentToThePivotVertex(l, x, G) : boolean
1  𝒳 ← 𝒮^l
2  ComputePartitionByGivenVertex(l, x, 𝒳, 𝒴, G)
3  compatible ← VertexPartitionIsCompatible(l + 1, G, 𝒴, G)
4  equivalent ← compatible ∧ (𝒴 = 𝒮^{l+1})
5  m ← l + 1
6  while compatible ∧ ¬equivalent ∧ (m < ll) do
7      if P^m = BACKTR then
8          compatible ← FALSE
9      else
10         𝒳 ← 𝒴
11         - - let r = |𝒳| and 𝒳 = (X_1, ..., X_r)
12         if P^m = VERTEX then
13             z ← any vertex in X_{t^l}
14             ComputePartitionByGivenVertex(m, z, 𝒳, 𝒴, G)
15             compatible ← VertexPartitionIsCompatible(m + 1, G, 𝒴, G)
16         else
17             ComputePartitionByPivotSet(m, 𝒳, 𝒴, G)
18             compatible ← SetPartitionIsCompatible(m + 1, G, X_{t^l}, 𝒴, G)
19         end if
20         equivalent ← compatible ∧ (𝒴 = 𝒮^{m+1})
21     end if
22     m ← m + 1
23 end while
24 return compatible ∧ (equivalent ∨ FinalPartitionsAreEquivalent(G, 𝒴, G))
```

Figure 9: Algorithm that tests equivalence of vertex x with p^l.

generating the remaining partitions $\mathcal{W}^{i+1}, ..., \mathcal{W}^{ll}$ since they would be the same as $\mathcal{S}^{i+1}, ..., \mathcal{S}^{ll}$. In such a case, they are equivalent, and the algorithm returns immediately.

When checking the equivalence between vertex x and vertex p^l, the algorithm may reach a partition $\mathcal{W}^i$ such that P^i = VERTEX but $|W^i_{t^i}| = |S^i_{t^i}| > 1$. This is the case when vertex equivalence has been established previously for the pivot set $S^i_{t^i}$ of partition $\mathcal{S}^i$. Then, any vertex in cell $W^i_{t^i}$ may be used for the refinement (to obtain $\mathcal{W}^{i+1}$) since all of them must be equivalent. See Section 5 for the details.

```
ComputePartitionByGivenVertex(l, x, 𝒳, 𝒴, I)
1  - - let r = |𝒳| , 𝒳 = (X_1, ..., X_r) and W = ⋃_{i=1}^{r} X_i
2  𝒴 ← ∅
3  for i ← 1 to r do
4      if HasLinks(X_i, W, I) do
5          if i = t^l then
6              𝒵 ← PartitionByVertex(X_i \ {x}, x, I)
7          else
8              𝒵 ← PartitionByVertex(X_i, x, I)
9          end if
10         𝒴 ← 𝒴 ∘ 𝒵
11     end if
12 end for
```

Figure 10: Algorithm that computes a new partition applying vertex partitioning.

Algorithms *ComputePartitionByGivenVertex*, shown in Figure 10, and *ComputePartitionByPivotSet*, shown in Figure 11, are very similar to *RefineByVertex*, previously shown in Figure 5, and *RefineBySet*, previously shown in Figure 6, respectively, but simpler, since they do not need to store information for future use like p^l, P^l, etc. These algorithms will also be used during the generation of the sequence of

```
ComputePartitionByPivotSet(l, 𝒳, 𝒴, I)
1  - - let r = |𝒳| , 𝒳 = (X_1, ..., X_r) and W = ⋃_{i=1}^{r} X_i
2  𝒴 ← ∅
3  for i ← 1 to r do
4      if HasLinks(X_i, W, I) do
5          𝒵 ← PartitionBySet(X_i, X_{t^l}, I)
6          𝒴 ← 𝒴 ∘ 𝒵
7      end if
8  end for
```

Figure 11: Algorithm that computes a new partition applying group partitioning.

partitions for graph H. That is why they have a generic graph parameter I.

When a partition $\mathcal{X}$ is generated applying vertex partitioning to partition $\mathcal{Y}$, the cells in $\mathcal{Y}$ may split according to the adjacencies of its vertices with the pivot vertex used. Since the pivot vertex is discarded in $\mathcal{X}$, the available degree of a cell in $\mathcal{X}$ is enough to know if its vertices were adjacent to the pivot vertex, and their type of adjacency. Two partitions are compatible if they have the same number of cells, and their respective cells have the same available degree. Therefore, if two vertices are equivalent, the partitions they generate from two compatible partitions must have their respective cells with the same available degree.

```
VertexPartitionIsCompatible(l, G, 𝒳, I) : boolean
1  if |𝒮^l| ≠ |𝒳| then
2      return FALSE
3  else
4      - - let r = |𝒮^l| = |𝒳| , 𝒮^l = (S_1^l, ..., S_r^l) , 𝒳 = (X_1, ..., X_r) , V^l = ⋃_{i=1}^{r} S_i^l and W = ⋃_{i=1}^{r} X_i
5      return ∀i ∈ {1, ..., r}, (|S_i^l| = |X_i|) ∧ (ADeg(S_i^l, V^l, G) = ADeg(X_i, W, I))
6  end if
```

Figure 12: Algorithm that tests compatibility between $\mathcal{X}$ under graph I and $\mathcal{S}^l$ under graph G after a vertex refinement.

Algorithm *VertexPartitionIsCompatible*, shown in Figure 12, tests the compatibility of partition $\mathcal{S}^l$ under graph G with partition $\mathcal{X}$ under graph I. After vertex refinement, two partitions are compatible if they have the same number of cells and the available degrees of the corresponding cells in their respective partitions are the same.

```
SetPartitionIsCompatible(l, G, Q, 𝒳, I) : boolean
1  if |𝒮^l| ≠ |𝒳| then
2      return FALSE
3  else
4      - - let r = |𝒮^l| = |𝒳| , 𝒮^l = (S_1^l, ..., S_r^l) , 𝒳 = (X_1, ..., X_r) and s = |𝒮^{l-1}| , 𝒮^{l-1} = (S_1^{l-1}, ..., S_s^{l-1})
5      return ∀i ∈ {1, ..., r}, (|S_i^l| = |X_i|) ∧ (ADeg(S_i^l, S_{t^{l-1}}^{l-1}, G) = ADeg(X_i, Q, I))
6  end if
```

Figure 13: Algorithm that tests compatibility between $\mathcal{X}$ under graph I and $\mathcal{S}^l$ under graph G after a set refinement.

Algorithm *SetPartitionIsCompatible*, shown in Figure 13, tests if after applying refinement by set to two compatible partitions, the new partitions are also compatible. Specifically, it tests if partition $\mathcal{S}^l$ under graph G is compatible with partition $\mathcal{X}$ under graph parameter I. After a refinement by set, $\mathcal{S}^l$ and $\mathcal{X}$ are compatible if they have the same number of cells and the available degrees of the corresponding cells with respect to their pivot sets $S_{t^{l-1}}^{l-1}$ (the pivot set used to generate partition $\mathcal{S}^l$ from $\mathcal{S}^{l-1}$) and Q, respectively, are the same.

Algorithm *FinalPartitionsAreEquivalent*, shown in Figure 14, tests if partition $\mathcal{S}^{ll}$ under graph G is

FinalPartitionsAreEquivalent$(G, \mathcal{Y}, I)$: boolean
1 - - let $r = |\mathcal{S}^{ll}| = |\mathcal{Y}|$, $\mathcal{S}^{ll} = (S_1^{ll}, ..., S_r^{ll})$, $V^l = \bigcup_{i=1}^{r} S_i^{ll}$, $\mathcal{Y} = (Y_1, ..., Y_r)$ and $W = \bigcup_{i=1}^{r} Y_i$
2 - - since $\forall i \in \{1, ..., r\} : HasLinks(S_i^{ll}, V^l, G), |S_i^{ll}| = |Y_i| = 1$
3 - - let s_i be the vertex in S_i^{ll} and y_i the vertex in Y_i for all $i \in \{1, ..., r\} : HasLinks(S_i^{ll}, V^l, G)$
4 - - let $A = Adj(G)$ and $B = Adj(I)$
5 **return** $\forall i, j \in \{1, ..., r\} : HasLinks(S_i^{ll}, V^l, G) \wedge HasLinks(S_j^{ll}, V^l, G), A_{s_i s_j} = B_{y_i y_j}$

Figure 14: Algorithm that tests if final partitions $\mathcal{X}^{ll}$ under graph G and $\mathcal{Y}$ under graph H are compatible.

equivalent to partition $\mathcal{Y}$ under graph parameter I. Since the final partitions define orders on their vertices, they will be equivalent if the adjacency between every pair of vertices in partition $\mathcal{S}^{ll}$ under graph G and the adjacency between their corresponding vertices in partition $\mathcal{Y}$ under graph parameter I are the same.

3.3 Generation of the Sequence of Partitions for Graph H

Algorithm *Match*, shown in Figure 15, looks for a sequence of partitions for graph H that is equivalent to the sequence of partitions previously generated for graph G. *Match* is a recursive backtracking algorithm which generates a new partition each time it is run. It processes partition $\mathcal{T}^l$, which is compatible with partition $\mathcal{S}^l$, to generate a new partition $\mathcal{T}^{l+1}$ which is tested for compatibility with partition $\mathcal{S}^{l+1}$. If they are compatible, *Match* calls itself to process the new partition $\mathcal{T}^{l+1}$. If they are not compatible, the sequence of partitions for graph H we have generated is not valid, and backtracking will be necessary (if possible). Success is achieved when partition $\mathcal{T}^{ll}$ is reached and found to be equivalent to $\mathcal{S}^{ll}$.

To decide which cell should be used as the pivot set, *Match* uses the information previously recorded when the sequence of partitions for graph G was generated. Since cell $S_{t^l}^l$ was used with G, cell $T_{t^l}^l$ will be used here for graph H. P^l will be used, as well, to choose the partitioning technique to use to generate the next partition.

The algorithm behaves as follows, depending on the partitioning technique to be applied:

- If the partitioning technique to be applied is VERTEX, a new partition $\mathcal{T}^{l+1}$ is generated applying algorithm *ComputePartitionByGivenVertex* taking, as pivot vertex, any vertex in the pivot set. Then, it is tested for compatibility with partition $\mathcal{S}^{l+1}$.
  - If they are not compatible, then *Match* returns FALSE.
  - If they are compatible and the final partition has been reached, equivalence between the final partitions needs to be tested. If the final partitions are equivalent, *Match* returns TRUE, since an isomorphism has been found. Otherwise, the sequence of partitions is not valid and it returns FALSE.
  - If they are compatible but the final partition has not been reached yet, a recursive call to *Match* is necessary to generate and test partitions $\mathcal{T}^{l+2}, ..., \mathcal{T}^{ll}$.
- If the partitioning technique to be applied is GROUP a new partition $\mathcal{T}^{l+1}$ is generated using algorithm *ComputePartitionByPivotSet* and tested for compatibility with partition $\mathcal{S}^{l+1}$.
  - If they are not compatible, like in the previous case, *Match* returns FALSE.
  - If they are compatible and the final partition has been reached, the final partitions are tested for equivalence. If they are equivalent, *Match* returns TRUE, like in the previous case, and otherwise, it returns FALSE.
  - If they are compatible but the final partition has not been reached yet, a recursive call to *Match* is necessary, like in the previous case, to generate and test partitions $\mathcal{T}^{l+2}, ..., \mathcal{T}^{ll}$.
- If P^l is BACKTR, this is a possible point of backtracking. In this case, all the vertices in the pivot set will be tried as pivot vertex, since any of them could be the one that matches the pivot vertex chosen for graph G. If no one leads to a valid sequence of partitions, then *Match* returns FALSE.

```
Match(l, G, H) : boolean
1  - - let r = |T^l| and T^l = (T_1^l, ..., T_r^l)
2  Q ← T_{t^l}^l
3  if P^l = BACKTR then
4      repeat
5          q ← any vertex in Q
6          Q ← Q \ {q}
7          ComputePartitionByGivenVertex(l, q, T^l, T^{l+1}, H)
8          if VertexPartitionIsCompatible(l + 1, G, T^{l+1}, H) then
9              if (l + 1) = ll then
10                 success ← FinalPartitionsAreEquivalent(G, T^{ll}, H)
11             else
12                 success ← Match(l + 1, G, H)
13             end if
14         else
15             success ← FALSE
16         end if
17     until (Q = ∅) ∨ success
18 else
19     if P^l = GROUP then
20         ComputePartitionByPivotSet(l, T^l, T^{l+1}, H)
21         compatible ← SetPartitionIsCompatible(l + 1, G, Q, T^{l+1}, H)
22     else
23         q ← any vertex in Q
24         ComputePartitionByGivenVertex(l, q, T^l, T^{l+1}, H)
25         compatible ← VertexPartitionIsCompatible(l + 1, G, T^{l+1}, H)
26     end if
27     if compatible then
28         if (l + 1) = ll then
29             success ← FinalPartitionsAreEquivalent(G, T^{ll}, H)
30         else
31             success ← Match(l + 1, G, H)
32         end if
33     else
34         success ← FALSE
35     end if
36 end if
37 return success
```

Figure 15: Algorithm that finds a match from graph H onto G.

For each vertex in the pivot set, it is necessary to compute a new partition applying algorithm *ComputePartitionByGivenVertex* with that vertex as the pivot vertex, and test it for compatibility with $\mathcal{S}^{l+1}$:

- If they are not compatible, the new partition is discarded.
- If they are compatible and the final partition has been reached, then equivalence between the final partitions is tested. *Match* returns TRUE if they are equivalent, and FALSE otherwise.
- If they are equivalent but the final partition has not been reached, *Match* is called recursively to generate and test the remaining partitions in this sequence.

4 Example

To show how the algorithm works, we will present an example of its behavior. We will use the graphs G and H shown in Figure 16. These graphs are a directed version of the Fürer gadgets used by Miyazaki [9] to

prove an exponential lower bound for McKay's program nauty [7]. As it will be seen, our algorithm works fine with this family of graphs and needs little backtracking. In fact, for these sample graphs it needs no backtracking whatsoever, as it is shown throughout this section.

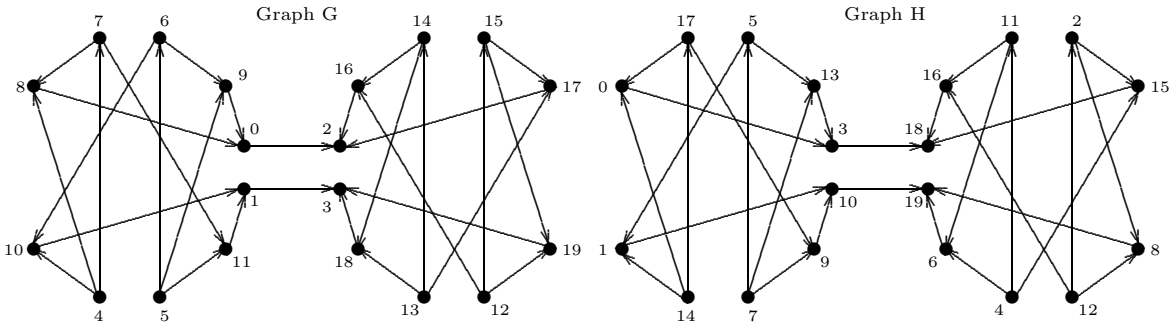

Figure 16: Sample graphs G and H to test isomorphism.

The algorithm needs to generate the data structures described in the previous section. First, it tests if graphs $G = (V_G, R_G)$ and $H = (V_H, R_H)$ have the same number of vertices and arcs. Since $|V_G| = 20$, $|V_H| = 20$, $|R_G| = 30$ and $|R_H| = 30$, they pass the first test for compatibility and partitions $\mathcal{S}^0$ for graph G and $\mathcal{T}^0$ for graph H are generated.

For graph G, we have:

$Deg(4,G) = Deg(5,G) = Deg(12,G) = Deg(13,G) = (0,3,0)$
$Deg(6,G) = Deg(7,G) = Deg(14,G) = Deg(15,G) = (0,2,1)$
$Deg(0,G) = Deg(1,G) = Deg(8,G) = Deg(9,G) = Deg(10,G) =$
$= Deg(11,G) = Deg(16,G) = Deg(17,G) = Deg(18,G) = Deg(19,G) = (0,1,2)$
$Deg(2,G) = Deg(3,G) = (0,0,3)$

Then, $\mathcal{S}^0 = (S_1^0, S_2^0, S_3^0, S_4^0)$ where:

$S_1^0 = \{4,5,12,13\}$ with $Deg(S_1^0, G) = (0,3,0)$ and $|S_1^0| = 4$
$S_2^0 = \{6,7,14,15\}$ with $Deg(S_2^0, G) = (0,2,1)$ and $|S_2^0| = 4$
$S_3^0 = \{0,1,8,9,10,11,16,17,18,19\}$ with $Deg(S_3^0, G) = (0,1,2)$ and $|S_3^0| = 10$
$S_4^0 = \{2,3\}$ with $Deg(S_4^0, G) = (0,0,3)$ and $|S_4^0| = 2$

For graph H, we have:

$Deg(4,H) = Deg(7,H) = Deg(12,H) = Deg(14,H) = (0,3,0)$
$Deg(2,H) = Deg(5,H) = Deg(11,H) = Deg(17,H) = (0,2,1)$
$Deg(0,H) = Deg(1,H) = Deg(3,H) = Deg(6,H) = Deg(8,H) =$
$= Deg(9,H) = Deg(10,H) = Deg(13,H) = Deg(15,H) = Deg(16,H) = (0,1,2)$
$Deg(18,H) = Deg(19,H) = (0,0,3)$

Then, $\mathcal{T}^0 = (T_1^0, T_2^0, T_3^0, T_4^0)$ where:

$T_1^0 = \{4,7,12,14\}$ with $Deg(T_1^0, H) = (0,3,0)$ and $|T_1^0| = 4$
$T_2^0 = \{2,5,11,17\}$ with $Deg(T_2^0, H) = (0,2,1)$ and $|T_2^0| = 4$
$T_3^0 = \{0,1,3,6,8,9,10,13,15,16\}$ with $Deg(T_3^0, H) = (0,1,2)$ and $|T_3^0| = 10$
$T_4^0 = \{18,19\}$ with $Deg(T_4^0, H) = (0,0,3)$ and $|T_4^0| = 2$

It is easy to see that partitions $\mathcal{S}^0$ and $\mathcal{T}^0$ are compatible, and the vertices in cell T_i^0 can only be mapped to the vertices in cell S_i^0 for $i \in \{1,2,3,4\}$. Hence, the sequence of partitions for graph G will be generated starting with $\mathcal{S}^0$.

4.1 Generating the sequence of partitions for graph G

Initially, all the cells in partition $\mathcal{S}^0$ are marked *Valid*. We choose a pivot set with algorithm *IndexBestPivot*. Since the smallest valid cell is S_4^0, *IndexBestPivot* returns 4, which is stored in t^0. Then, using S_4^0 as the pivot set, we try to generate a new partition $\mathcal{S}^1$ from $\mathcal{S}^0$. We start by marking S_4^0 as not valid, and setting $P^0 = \text{GROUP}$, and then, we try *RefineBySet*. This requires to compute the partitions by set of all the cells in $\mathcal{S}^0$. Since this is an easy task, we assume the reader can easily come to this result:

$\mathcal{S}^1 = (S_1^1, S_2^1, S_3^1, S_4^1, S_5^1)$ where:

$S_1^1 = \{4, 5, 12, 13\}$ with $ADeg(S_1^1, S_4^0, G) = (0, 0, 0)$, $|S_1^1| = 4$ and $Valid(S_1^1) = \text{TRUE}$
$S_2^1 = \{6, 7, 14, 15\}$ with $ADeg(S_2^1, S_4^0, G) = (0, 0, 0)$, $|S_2^1| = 4$ and $Valid(S_2^1) = \text{TRUE}$
$S_3^1 = \{0, 1, 16, 17, 18, 19\}$ with $ADeg(S_3^1, S_4^0, G) = (0, 1, 0)$, $|S_3^1| = 6$ and $Valid(S_3^1) = \text{TRUE}$
$S_4^1 = \{8, 9, 10, 11\}$ with $ADeg(S_4^1, S_4^0, G) = (0, 0, 0)$, $|S_5^1| = 4$ and $Valid(S_4^1) = \text{TRUE}$
$S_5^1 = \{2, 3\}$ with $ADeg(S_5^1, S_4^0, G) = (0, 0, 0)$, $|S_5^1| = 2$ and $Valid(S_5^1) = \text{FALSE}$

Choosing the pivot set to generate $\mathcal{S}^2$, yields $t^1 = 1$, since S_1^1 is the smallest valid cell found in left-to-right order by algorithm *IndexBestPivot*. We also set $P^1 = \text{GROUP}$ and apply *RefineBySet* to $\mathcal{S}^1$ to get:

$\mathcal{S}^2 = (S_1^2, S_2^2, S_3^2, S_4^2, S_5^2, S_6^2)$ where:

$S_1^2 = \{4, 5, 12, 13\}$ with $ADeg(S_1^2, S_1^1, G) = (0, 0, 0)$, $|S_1^2| = 4$ and $Valid(S_1^2) = \text{FALSE}$
$S_2^2 = \{6, 7, 14, 15\}$ with $ADeg(S_2^2, S_1^1, G) = (0, 0, 1)$, $|S_2^2| = 4$ and $Valid(S_2^2) = \text{TRUE}$
$S_3^2 = \{16, 17, 18, 19\}$ with $ADeg(S_3^2, S_1^1, G) = (0, 0, 1)$, $|S_3^2| = 4$ and $Valid(S_3^2) = \text{TRUE}$
$S_4^2 = \{0, 1\}$ with $ADeg(S_4^2, S_1^1, G) = (0, 0, 0)$, $|S_4^2| = 2$ and $Valid(S_4^2) = \text{TRUE}$
$S_5^2 = \{8, 9, 10, 11\}$ with $ADeg(S_5^2, S_1^1, G) = (0, 0, 1)$, $|S_5^2| = 4$ and $Valid(S_5^2) = \text{TRUE}$
$S_6^2 = \{2, 3\}$ with $ADeg(S_6^2, S_1^1, G) = (0, 0, 0)$, $|S_6^2| = 2$ and $Valid(S_6^2) = \text{FALSE}$

Cell S_4^2 is tried next as pivot set and discarded since it is not capable of splitting any cell in $\mathcal{S}^2$. Next, S_2^2 is tried and also discarded, since it is not able to split any cell in $\mathcal{S}^2$ either. Finally, we set $t^2 = 3$ and $P^2 = \text{GROUP}$. After the refinement, we obtain:

$\mathcal{S}^3 = (S_1^3, S_2^3, S_3^3, S_4^3, S_5^3, S_6^3, S_7^3, S_8^3)$ where:

$S_1^3 = \{12, 13\}$ with $ADeg(S_1^3, S_3^2, G) = (0, 2, 0)$, $|S_1^3| = 2$ and $Valid(S_1^3) = \text{TRUE}$
$S_2^3 = \{4, 5\}$ with $ADeg(S_2^3, S_3^2, G) = (0, 0, 0)$, $|S_2^3| = 2$ and $Valid(S_2^3) = \text{TRUE}$
$S_3^3 = \{14, 15\}$ with $ADeg(S_3^3, S_3^2, G) = (0, 2, 0)$, $|S_3^3| = 2$ and $Valid(S_3^3) = \text{TRUE}$
$S_4^3 = \{6, 7\}$ with $ADeg(S_4^3, S_3^2, G) = (0, 0, 0)$, $|S_4^3| = 2$ and $Valid(S_4^3) = \text{TRUE}$
$S_5^3 = \{16, 17, 18, 19\}$ with $ADeg(S_5^3, S_3^2, G) = (0, 0, 0)$, $|S_5^3| = 4$ and $Valid(S_5^3) = \text{FALSE}$
$S_6^3 = \{0, 1\}$ with $ADeg(S_6^3, S_3^2, G) = (0, 0, 0)$, $|S_6^3| = 2$ and $Valid(S_6^3) = \text{FALSE}$
$S_7^3 = \{8, 9, 10, 11\}$ with $ADeg(S_7^3, S_3^2, G) = (0, 0, 0)$, $|S_7^3| = 4$ and $Valid(S_7^3) = \text{TRUE}$
$S_8^3 = \{2, 3\}$ with $ADeg(S_8^3, S_3^2, G) = (0, 0, 2)$, $|S_8^3| = 2$ and $Valid(S_8^3) = \text{FALSE}$

Note that, since cell S_1^2 has been split into cells S_1^3 and S_2^3, these new cells are marked valid, even though cell S_1^2 was not valid. Now, all the valid cells are tried as pivots, discarded, and marked as non-valid, since none of them is able to split any cell in the partition. In this case, the algorithm chooses the leftmost cell of the smallest size, S_1^3, as the pivot set to use for vertex refinement, so $t^3 = 1$ and $P^3 = \text{UNKNOWN}$ to indicate that backtracking might be necessary at this point. After a vertex refinement with pivot vertex $p^3 = 12$, we have:

$\mathcal{S}^4 = (S_1^4, S_2^4, S_3^4, S_4^4, S_5^4, S_6^4, S_7^4, S_8^4, S_9^4, S_{10}^4)$ where $V^4 = \bigcup_{i=1}^{10} S_i^4$ and:

$S_1^4 = \{13\}$ with $ADeg(S_1^4, V^4, G) = (0, 3, 0)$, $|S_1^4| = 1$ and $Valid(S_1^4) = \text{TRUE}$
$S_2^4 = \{4, 5\}$ with $ADeg(S_2^4, V^4, G) = (0, 3, 0)$, $|S_2^4| = 2$ and $Valid(S_2^4) = \text{FALSE}$
$S_3^4 = \{15\}$ with $ADeg(S_3^4, V^4, G) = (0, 2, 0)$, $|S_3^4| = 1$ and $Valid(S_3^4) = \text{TRUE}$
$S_4^4 = \{14\}$ with $ADeg(S_4^4, V^4, G) = (0, 2, 1)$, $|S_4^4| = 1$ and $Valid(S_4^4) = \text{TRUE}$
$S_5^4 = \{6, 7\}$ with $ADeg(S_5^4, V^4, G) = (0, 2, 1)$, $|S_5^4| = 2$ and $Valid(S_5^4) = \text{FALSE}$
$S_6^4 = \{16, 19\}$ with $ADeg(S_6^4, V^4, G) = (0, 1, 1)$, $|S_6^4| = 2$ and $Valid(S_6^4) = \text{TRUE}$
$S_7^4 = \{17, 18\}$ with $ADeg(S_7^4, V^4, G) = (0, 1, 2)$, $|S_7^4| = 2$ and $Valid(S_7^4) = \text{TRUE}$
$S_8^4 = \{0, 1\}$ with $ADeg(S_8^4, V^4, G) = (0, 1, 2)$, $|S_8^4| = 2$ and $Valid(S_8^4) = \text{FALSE}$
$S_9^4 = \{8, 9, 10, 11\}$ with $ADeg(S_9^4, V^4, G) = (0, 1, 2)$, $|S_9^4| = 4$ and $Valid(S_9^4) = \text{FALSE}$
$S_{10}^4 = \{2, 3\}$ with $ADeg(S_{10}^4, V^4, G) = (0, 0, 3)$, $|S_{10}^4| = 2$ and $Valid(S_{10}^4) = \text{FALSE}$

Note that vertex 12 is not considered anymore, since it has been used as a pivot vertex, and therefore, all its adjacencies have been processed at this point. Now, since there are cells with only one vertex, *IndexBestPivot* chooses cell S_1^4 as the leftmost cell with the largest number of links (computed by $NumLinks(S_1^4, V^4, G)$), among the cells with a single vertex. Thus, $t^4 = 1$ and $P^4 = \text{VERTEX}$. After such refinement, we have:

$\mathcal{S}^5 = (S_1^5, S_2^5, S_3^5, S_4^5, S_5^5, S_6^5, S_7^5, S_8^5, S_9^5)$ where $V^5 = \bigcup_{i=1}^{9} S_i^5$ and:

$S_1^5 = \{4,5\}$ with $ADeg(S_1^5, V^5, G) = (0,3,0)$, $|S_1^5| = 2$ and $Valid(S_1^5)$ = FALSE
$S_2^5 = \{15\}$ with $ADeg(S_2^5, V^5, G) = (0,2,0)$, $|S_2^5| = 1$ and $Valid(S_2^5)$ = TRUE
$S_3^5 = \{14\}$ with $ADeg(S_3^5, V^5, G) = (0,2,0)$, $|S_3^5| = 1$ and $Valid(S_3^5)$ = TRUE
$S_4^5 = \{6,7\}$ with $ADeg(S_4^5, V^5, G) = (0,2,1)$, $|S_4^5| = 2$ and $Valid(S_4^5)$ = FALSE
$S_5^5 = \{16,19\}$ with $ADeg(S_5^5, V^5, G) = (0,1,1)$, $|S_5^5| = 2$ and $Valid(S_5^5)$ = TRUE
$S_6^5 = \{17,18\}$ with $ADeg(S_6^5, V^5, G) = (0,1,1)$, $|S_6^5| = 2$ and $Valid(S_6^5)$ = TRUE
$S_7^5 = \{0,1\}$ with $ADeg(S_7^5, V^5, G) = (0,1,2)$, $|S_7^5| = 2$ and $Valid(S_7^5)$ = FALSE
$S_8^5 = \{8,9,10,11\}$ with $ADeg(S_8^5, V^5, G) = (0,1,2)$, $|S_8^5| = 4$ and $Valid(S_8^5)$ = FALSE
$S_9^5 = \{2,3\}$ with $ADeg(S_9^5, V^5, G) = (0,0,3)$, $|S_9^5| = 2$ and $Valid(S_9^5)$ = FALSE

Note that vertex 13 has not been able to split any cell. However, the pivot set has been discarded and the number of links of cells S_3^5 (previously S_4^4) and S_6^5 (previously S_7^4) has been decremented. S_2^5 will be used as the next pivot set, $t^5 = 2$, and vertex refinement, P^5 = VERTEX, will be used to generate $\mathcal{S}^6$ which will be:

$\mathcal{S}^6 = (S_1^6, S_2^6, S_3^6, S_4^6, S_5^6, S_6^6, S_7^6, S_8^6, S_9^6, S_{10}^6)$ where $V^6 = \bigcup_{i=1}^{10} S_i^6$ and:
$S_1^6 = \{4,5\}$ with $ADeg(S_1^6, V^6, G) = (0,3,0)$, $|S_1^6| = 2$ and $Valid(S_1^6)$ = FALSE
$S_2^6 = \{14\}$ with $ADeg(S_2^6, V^6, G) = (0,2,0)$, $|S_2^6| = 1$ and $Valid(S_2^6)$ = TRUE
$S_3^6 = \{6,7\}$ with $ADeg(S_3^6, V^6, G) = (0,2,1)$, $|S_3^6| = 2$ and $Valid(S_3^6)$ = FALSE
$S_4^6 = \{19\}$ with $ADeg(S_4^6, V^6, G) = (0,1,0)$, $|S_4^6| = 1$ and $Valid(S_4^6)$ = TRUE
$S_5^6 = \{16\}$ with $ADeg(S_5^6, V^6, G) = (0,1,1)$, $|S_5^6| = 1$ and $Valid(S_5^6)$ = TRUE
$S_6^6 = \{17\}$ with $ADeg(S_6^6, V^6, G) = (0,1,0)$, $|S_6^6| = 1$ and $Valid(S_6^6)$ = TRUE
$S_7^6 = \{18\}$ with $ADeg(S_7^6, V^6, G) = (0,1,1)$, $|S_7^6| = 1$ and $Valid(S_7^6)$ = TRUE
$S_8^6 = \{0,1\}$ with $ADeg(S_8^6, V^6, G) = (0,1,2)$, $|S_8^6| = 2$ and $Valid(S_8^6)$ = FALSE
$S_9^6 = \{8,9,10,11\}$ with $ADeg(S_9^6, V^6, G) = (0,1,2)$, $|S_9^6| = 4$ and $Valid(S_9^6)$ = FALSE
$S_{10}^6 = \{2,3\}$ with $ADeg(S_{10}^6, V^6, G) = (0,0,3)$, $|S_{10}^6| = 2$ and $Valid(S_{10}^6)$ = FALSE

Continuing with the cells with a single vertex, the algorithm chooses S_2^6 for the next refinement. It sets $t^6 = 2$, and P^6 = VERTEX, and generates the new partition $\mathcal{S}^7$:

$\mathcal{S}^7 = (S_1^7, S_2^7, S_3^7, S_4^7, S_5^7, S_6^7, S_7^7, S_8^7, S_9^7)$ where $V^7 = \bigcup_{i=1}^{9} S_i^7$ and:
$S_1^7 = \{4,5\}$ with $ADeg(S_1^7, V^7, G) = (0,3,0)$, $|S_1^7| = 2$ and $Valid(S_1^7)$ = FALSE
$S_2^7 = \{6,7\}$ with $ADeg(S_2^7, V^7, G) = (0,2,1)$, $|S_2^7| = 2$ and $Valid(S_2^7)$ = FALSE
$S_3^7 = \{19\}$ with $ADeg(S_3^7, V^7, G) = (0,1,0)$, $|S_3^7| = 1$ and $Valid(S_3^7)$ = TRUE
$S_4^7 = \{16\}$ with $ADeg(S_4^7, V^7, G) = (0,1,0)$, $|S_4^7| = 1$ and $Valid(S_4^7)$ = TRUE
$S_5^7 = \{17\}$ with $ADeg(S_5^7, V^7, G) = (0,1,0)$, $|S_5^7| = 1$ and $Valid(S_5^7)$ = TRUE
$S_6^7 = \{18\}$ with $ADeg(S_6^7, V^7, G) = (0,1,0)$, $|S_6^7| = 1$ and $Valid(S_6^7)$ = TRUE
$S_7^7 = \{0,1\}$ with $ADeg(S_7^7, V^7, G) = (0,1,2)$, $|S_7^7| = 2$ and $Valid(S_7^7)$ = FALSE
$S_8^7 = \{8,9,10,11\}$ with $ADeg(S_8^7, V^7, G) = (0,1,2)$, $|S_8^7| = 4$ and $Valid(S_8^7)$ = FALSE
$S_9^7 = \{2,3\}$ with $ADeg(S_9^7, V^7, G) = (0,0,3)$, $|S_9^7| = 2$ and $Valid(S_9^7)$ = FALSE

The next cell to be used is S_3^7, so we set $t^7 = 3$ and P^7 = VERTEX. Then, the new partition $\mathcal{S}^8$ is generated. Note that, since cell S_9^7 is split, the new cells S_8^8 and S_9^8 are marked as valid, even though S_9^7 was not valid.

$\mathcal{S}^8 = (S_1^8, S_2^8, S_3^8, S_4^8, S_5^8, S_6^8, S_7^8, S_8^8, S_9^8)$ where $V^8 = \bigcup_{i=1}^{9} S_i^8$ and:
$S_1^8 = \{4,5\}$ with $ADeg(S_1^8, V^8, G) = (0,3,0)$, $|S_1^8| = 2$ and $Valid(S_1^8)$ = FALSE
$S_2^8 = \{6,7\}$ with $ADeg(S_2^8, V^8, G) = (0,2,1)$, $|S_2^8| = 2$ and $Valid(S_2^8)$ = FALSE
$S_3^8 = \{16\}$ with $ADeg(S_3^8, V^8, G) = (0,1,0)$, $|S_3^8| = 1$ and $Valid(S_3^8)$ = TRUE
$S_4^8 = \{17\}$ with $ADeg(S_4^8, V^8, G) = (0,1,0)$, $|S_4^8| = 1$ and $Valid(S_4^8)$ = TRUE
$S_5^8 = \{18\}$ with $ADeg(S_5^8, V^8, G) = (0,1,0)$, $|S_5^8| = 1$ and $Valid(S_5^8)$ = TRUE
$S_6^8 = \{0,1\}$ with $ADeg(S_6^8, V^8, G) = (0,1,2)$, $|S_6^8| = 2$ and $Valid(S_6^8)$ = FALSE
$S_7^8 = \{8,9,10,11\}$ with $ADeg(S_7^8, V^8, G) = (0,1,2)$, $|S_7^8| = 4$ and $Valid(S_7^8)$ = FALSE
$S_8^8 = \{3\}$ with $ADeg(S_8^8, V^8, G) = (0,0,2)$, $|S_8^8| = 1$ and $Valid(S_8^8)$ = TRUE
$S_9^8 = \{2\}$ with $ADeg(S_9^8, V^8, G) = (0,0,3)$, $|S_9^8| = 1$ and $Valid(S_9^8)$ = TRUE

Now, the algorithm selects cell S_9^8 as the pivot set, since it is a cell with a single vertex, and, among the cells with a single vertex, it is the one with the most available links (computed by $NumLinks(S_9^8, V^8, G)$). Therefore, we set $t^8 = 9$ and P^8 = VERTEX. Note that in the following partition $\mathcal{S}^9$, there are a couple of cells, S_3^9 and S_4^9, which do not have links (their available degree has become $(0,0,0)$), so they are marked as

non-valid, and will be discarded in the next refinement. Besides, the new cells split from S_6^8 are marked as valid.

$\mathcal{S}^9 = (S_1^9, S_2^9, S_3^9, S_4^9, S_5^9, S_6^9, S_7^9, S_8^9, S_9^9)$ where $V^9 = \bigcup_{i=1}^{9} S_i^9$ and:

$S_1^9 = \{4,5\}$ with $ADeg(S_1^9, V^9, G) = (0,3,0)$, $|S_1^9| = 2$ and $Valid(S_1^9) = \text{FALSE}$
$S_2^9 = \{6,7\}$ with $ADeg(S_2^9, V^9, G) = (0,2,1)$, $|S_2^9| = 2$ and $Valid(S_2^9) = \text{FALSE}$
$S_3^9 = \{16\}$ with $ADeg(S_3^9, V^9, G) = (0,0,0)$, $|S_3^9| = 1$ and $Valid(S_3^9) = \text{FALSE}$
$S_4^9 = \{17\}$ with $ADeg(S_4^9, V^9, G) = (0,0,0)$, $|S_4^9| = 1$ and $Valid(S_4^9) = \text{FALSE}$
$S_5^9 = \{18\}$ with $ADeg(S_5^9, V^9, G) = (0,1,0)$, $|S_5^9| = 1$ and $Valid(S_5^9) = \text{TRUE}$
$S_6^9 = \{0\}$ with $ADeg(S_6^9, V^9, G) = (0,0,2)$, $|S_6^9| = 1$ and $Valid(S_6^9) = \text{TRUE}$
$S_7^9 = \{1\}$ with $ADeg(S_7^9, V^9, G) = (0,1,2)$, $|S_7^9| = 1$ and $Valid(S_7^9) = \text{TRUE}$
$S_8^9 = \{8,9,10,11\}$ with $ADeg(S_8^9, V^9, G) = (0,1,2)$, $|S_7^9| = 4$ and $Valid(S_7^9) = \text{FALSE}$
$S_9^9 = \{3\}$ with $ADeg(S_8^9, V^9, G) = (0,0,2)$, $|S_9^9| = 1$ and $Valid(S_9^9) = \text{TRUE}$

From left-to-right, the smallest cell with most links is S_7^9, since it has three links and size one. Therefore, we set $t^9 = 7$ and $P^9 = \text{VERTEX}$. Refining by vertex yields:

$\mathcal{S}^{10} = (S_1^{10}, S_2^{10}, S_3^{10}, S_4^{10}, S_5^{10}, S_6^{10}, S_7^{10})$ where $V^{10} = \bigcup_{i=1}^{7} S_i^{10}$ and:

$S_1^{10} = \{4,5\}$ with $ADeg(S_1^{10}, V^{10}, G) = (0,3,0)$, $|S_1^{10}| = 2$ and $Valid(S_1^{10}) = \text{FALSE}$
$S_2^{10} = \{6,7\}$ with $ADeg(S_2^{10}, V^{10}, G) = (0,2,1)$, $|S_2^{10}| = 2$ and $Valid(S_2^{10}) = \text{FALSE}$
$S_3^{10} = \{18\}$ with $ADeg(S_3^{10}, V^{10}, G) = (0,1,0)$, $|S_3^{10}| = 1$ and $Valid(S_3^{10}) = \text{TRUE}$
$S_4^{10} = \{0\}$ with $ADeg(S_4^{10}, V^{10}, G) = (0,0,2)$, $|S_4^{10}| = 1$ and $Valid(S_4^{10}) = \text{TRUE}$
$S_5^{10} = \{10,11\}$ with $ADeg(S_5^{10}, V^{10}, G) = (0,0,2)$, $|S_5^{10}| = 2$ and $Valid(S_5^{10}) = \text{TRUE}$
$S_6^{10} = \{8,9\}$ with $ADeg(S_6^{10}, V^{10}, G) = (0,1,2)$, $|S_6^{10}| = 2$ and $Valid(S_6^{10}) = \text{TRUE}$
$S_7^{10} = \{3\}$ with $ADeg(S_7^{10}, V^{10}, G) = (0,0,1)$, $|S_7^{10}| = 1$ and $Valid(S_7^{10}) = \text{TRUE}$

Note that cells S_5^{10} and S_6^{10} are new, and, therefore, marked valid. The best pivot set in partition $\mathcal{S}^{10}$ is S_4^{10} since it is the one with most links, among the two with only one vertex. We set $t^{10} = 4$ and $P^{10} = \text{VERTEX}$ and refine by vertex to obtain the next partition:

$\mathcal{S}^{11} = (S_1^{11}, S_2^{11}, S_3^{11}, S_4^{11}, S_5^{11}, S_6^{11})$ where $V^{11} = \bigcup_{i=1}^{6} S_i^{11}$ and:

$S_1^{11} = \{4,5\}$ with $ADeg(S_1^{11}, V^{11}, G) = (0,3,0)$, $|S_1^{11}| = 2$ and $Valid(S_1^{11}) = \text{FALSE}$
$S_2^{11} = \{6,7\}$ with $ADeg(S_2^{11}, V^{11}, G) = (0,2,1)$, $|S_2^{11}| = 2$ and $Valid(S_2^{11}) = \text{FALSE}$
$S_3^{11} = \{18\}$ with $ADeg(S_3^{11}, V^{11}, G) = (0,1,0)$, $|S_3^{11}| = 1$ and $Valid(S_3^{11}) = \text{TRUE}$
$S_4^{11} = \{10,11\}$ with $ADeg(S_4^{11}, V^{11}, G) = (0,0,2)$, $|S_4^{11}| = 2$ and $Valid(S_4^{11}) = \text{TRUE}$
$S_5^{11} = \{8,9\}$ with $ADeg(S_5^{11}, V^{11}, G) = (0,0,2)$, $|S_5^{11}| = 2$ and $Valid(S_5^{11}) = \text{TRUE}$
$S_6^{11} = \{3\}$ with $ADeg(S_6^{11}, V^{11}, G) = (0,0,1)$, $|S_6^{11}| = 1$ and $Valid(S_6^{11}) = \text{TRUE}$

This refinement has not been able to split any cell, but the pivot set has been discarded and the available degree of one cell has been decremented. That, though slight, is an improvement. Now, S_3^{11} is chosen and $t^{11} = 3$ and $P^{11} = \text{VERTEX}$ are set. After the refinement, we have:

$\mathcal{S}^{12} = (S_1^{12}, S_2^{12}, S_3^{12}, S_4^{12}, S_5^{12})$ where $V^{12} = \bigcup_{i=1}^{5} S_i^{12}$ and:

$S_1^{12} = \{4,5\}$ with $ADeg(S_1^{12}, V^{12}, G) = (0,3,0)$, $|S_1^{12}| = 2$ and $Valid(S_1^{12}) = \text{FALSE}$
$S_2^{12} = \{6,7\}$ with $ADeg(S_2^{12}, V^{12}, G) = (0,2,1)$, $|S_2^{12}| = 2$ and $Valid(S_2^{12}) = \text{FALSE}$
$S_3^{12} = \{10,11\}$ with $ADeg(S_3^{12}, V^{12}, G) = (0,0,2)$, $|S_3^{12}| = 2$ and $Valid(S_3^{12}) = \text{TRUE}$
$S_4^{12} = \{8,9\}$ with $ADeg(S_4^{12}, V^{12}, G) = (0,0,2)$, $|S_4^{12}| = 2$ and $Valid(S_4^{12}) = \text{TRUE}$
$S_5^{12} = \{3\}$ with $ADeg(S_5^{12}, V^{12}, G) = (0,0,0)$, $|S_5^{12}| = 1$ and $Valid(S_5^{12}) = \text{FALSE}$

Since S_5^{12} has no links, it is marked as non valid. Therefore, the pivot set will be chosen among the valid cells. These are S_3^{12} and S_4^{12}. They are tried for refinement by set, discarded, and marked as non-valid, since they are not able to split any cell in the partition. Then, S_1^{12} is used for vertex partitioning with possible backtracking. Therefore, we set $t^{12} = 1$, $p^{12} = 4$ and $P^{12} = \text{UNKNOWN}$. After the refinement, we get:

$\mathcal{S}^{13} = (S_1^{13}, S_2^{13}, S_3^{13}, S_4^{13}, S_5^{13}, S_6^{13}, S_7^{13})$ where $V^{13} = \bigcup_{i=1}^{7} S_i^{13}$ and:

$S_1^{13} = \{5\}$ with $ADeg(S_1^{13}, V^{13}, G) = (0,3,0)$, $|S_1^{13}| = 1$ and $Valid(S_1^{13}) = \text{TRUE}$
$S_2^{13} = \{7\}$ with $ADeg(S_2^{13}, V^{13}, G) = (0,2,0)$, $|S_2^{13}| = 1$ and $Valid(S_2^{13}) = \text{TRUE}$
$S_3^{13} = \{6\}$ with $ADeg(S_3^{13}, V^{13}, G) = (0,2,1)$, $|S_3^{13}| = 1$ and $Valid(S_3^{13}) = \text{TRUE}$
$S_4^{13} = \{10\}$ with $ADeg(S_4^{13}, V^{13}, G) = (0,0,1)$, $|S_4^{13}| = 1$ and $Valid(S_4^{13}) = \text{TRUE}$
$S_5^{13} = \{11\}$ with $ADeg(S_5^{13}, V^{13}, G) = (0,0,2)$, $|S_5^{13}| = 1$ and $Valid(S_5^{13}) = \text{TRUE}$
$S_6^{13} = \{8\}$ with $ADeg(S_6^{13}, V^{13}, G) = (0,0,1)$, $|S_6^{13}| = 1$ and $Valid(S_6^{13}) = \text{TRUE}$
$S_7^{13} = \{9\}$ with $ADeg(S_7^{13}, V^{13}, G) = (0,0,2)$, $|S_7^{13}| = 1$ and $Valid(S_7^{13}) = \text{TRUE}$

We have come to a partition where all the cells have only one vertex. Therefore, the generation of the sequence of partitions has finished and we set $ll = 13$. Implicitly, we have an order on the vertices in the graph that, later, may be used to establish a correspondence between the vertices in graph G and the vertices in graph H.

4.2 Searching for Automorphisms

After the generation of the sequence of partitions for graph G, the algorithm looks for automorphisms. To do so, it tries to change $P^l =$ UNKNOWN for $P^l =$ VERTEX proving the equivalence of all the vertices in $S^l_{t^l}$; if that is not possible, it sets $P^l =$ BACKTR. In particular, the two partitions marked UNKNOWN are $\mathcal{S}^{13}$, which is tested first, and $\mathcal{S}^3$, which will be tested next.

Choosing vertex 5 instead of vertex 4 in cell S^{12}_1 and applying a vertex refinement yields a new partition $(\{4\},\{6\},\{7\},\{11\},\{10\},\{9\},\{8\})$ such that:

$ADeg(\{4\},\{4,6,7,11,10,9,8\},G) = (0,3,0)$, $|\{4\}| = 1$
$ADeg(\{6\},\{4,6,7,11,10,9,8\},G) = (0,2,0)$, $|\{6\}| = 1$
$ADeg(\{7\},\{4,6,7,11,10,9,8\},G) = (0,2,1)$, $|\{7\}| = 1$
$ADeg(\{11\},\{4,6,7,11,10,9,8\},G) = (0,0,1)$, $|\{11\}| = 1$
$ADeg(\{10\},\{4,6,7,11,10,9,8\},G) = (0,0,2)$, $|\{10\}| = 1$
$ADeg(\{9\},\{4,6,7,11,10,9,8\},G) = (0,0,1)$, $|\{9\}| = 1$
$ADeg(\{8\},\{4,6,7,11,10,9,8\},G) = (0,0,2)$, $|\{8\}| = 1$

It is easy to see that it is compatible with partition $\mathcal{S}^{13} = (\{5\},\{7\},\{6\},\{10\},\{11\},\{8\},\{9\})$. This is tested by algorithm *VertexPartitionIsCompatible*. Finally, their equivalence is established with algorithm *FinalPartitionsAreEquivalent* based on their adjacencies, which are graphically shown in Figure 17. This equivalence of all the vertices in the pivot set lets us set $P^{12} =$ VERTEX.

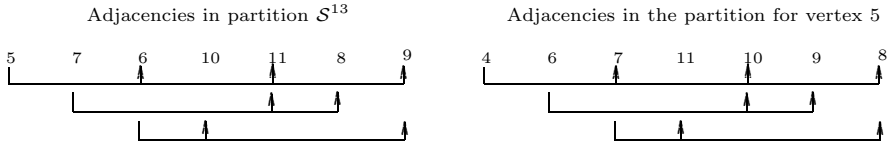

Figure 17: Equivalence of the final partitions for vertices 5 and 4.

Now, we test the equivalence of the vertices in cell S^3_1. Vertex 12 was the one previously used, so now, vertex 13 will be tested for equivalence. To do so, first, a new partition is generated via vertex refinement, using 13 as the pivot vertex. This yields:

$\mathcal{W}^4 = (S^4_1, S^4_2, S^4_3, S^4_4, S^4_5, S^4_6, S^4_7, S^4_8, S^4_9, S^4_{10})$ where $W^4 = \bigcup_{i=1}^{10} W^4_i$ and:

$W^4_1 = \{12\}$ with $ADeg(W^4_1, W^4, G) = (0,3,0)$, $|W^4_1| = 1$
$W^4_2 = \{4,5\}$ with $ADeg(W^4_2, W^4, G) = (0,3,0)$, $|W^4_2| = 2$
$W^4_3 = \{14\}$ with $ADeg(W^4_3, W^4, G) = (0,2,0)$, $|W^4_3| = 1$
$W^4_4 = \{15\}$ with $ADeg(W^4_4, W^4, G) = (0,2,1)$, $|W^4_4| = 1$
$W^4_5 = \{6,7\}$ with $ADeg(W^4_5, W^4, G) = (0,2,1)$, $|W^4_5| = 2$
$W^4_6 = \{17,18\}$ with $ADeg(W^4_6, W^4, G) = (0,1,1)$, $|W^4_6| = 2$
$W^4_7 = \{16,19\}$ with $ADeg(W^4_7, W^4, G) = (0,1,2)$, $|W^4_7| = 2$
$W^4_8 = \{0,1\}$ with $ADeg(W^4_8, W^4, G) = (0,1,2)$, $|W^4_8| = 2$
$W^4_9 = \{8,9,10,11\}$ with $ADeg(W^4_9, W^4, G) = (0,1,2)$, $|W^4_9| = 4$
$W^4_{10} = \{2,3\}$ with $ADeg(W^4_{10}, W^4, G) = (0,0,3)$, $|W^4_{10}| = 2$

Since they are compatible but not equivalent (they are not the same), a new partition $\mathcal{W}^5$ is generated applying the same refinement that was used to generate $\mathcal{S}^5$, with the corresponding pivot set $W^4_{t^4}$, that is, W^4_1. Thus, we obtain:

$\mathcal{W}^5 = (W^5_1, W^5_2, W^5_3, W^5_4, W^5_5, W^5_6, W^5_7, W^5_8, W^5_9)$ where $W^5 = \bigcup_{i=1}^{9} W^5_i$ and:

$W_1^5 = \{4,5\}$ with $ADeg(W_1^5, W^5, G) = (0,3,0)$, $|W_1^5| = 2$
$W_2^5 = \{14\}$ with $ADeg(W_2^5, W^5, G) = (0,2,0)$, $|W_2^5| = 1$
$W_3^5 = \{15\}$ with $ADeg(W_3^5, W^5, G) = (0,2,0)$, $|W_3^5| = 1$
$W_4^5 = \{6,7\}$ with $ADeg(W_4^5, W^5, G) = (0,2,1)$, $|W_4^5| = 2$
$W_5^5 = \{17,18\}$ with $ADeg(W_5^5, W^5, G) = (0,1,1)$, $|W_5^5| = 2$
$W_6^5 = \{16,19\}$ with $ADeg(W_6^5, W^5, G) = (0,1,1)$, $|W_6^5| = 2$
$W_7^5 = \{0,1\}$ with $ADeg(W_7^5, W^5, G) = (0,1,2)$, $|W_7^5| = 2$
$W_8^5 = \{8,9,10,11\}$ with $ADeg(W_8^5, W^5, G) = (0,1,2)$, $|W_8^5| = 4$
$W_9^5 = \{2,3\}$ with $ADeg(W_9^5, W^5, G) = (0,0,3)$, $|W_9^5| = 2$

Again, partition $\mathcal{W}^5$ is compatible with partition $\mathcal{S}^5$, but they are not equivalent. Therefore, we must go on generating new partitions. According to P^5 and t^5, vertex refinement is applied using cell W_2^5 as the pivot vertex. This yields:

$\mathcal{W}^6 = (W_1^6, W_2^6, W_3^6, W_4^6, W_5^6, W_6^6, W_7^6, W_8^6, W_9^6, W_{10}^6)$ where $W^6 = \bigcup_{i=1}^{10} W_i^6$ and:

$W_1^6 = \{4,5\}$ with $ADeg(W_1^6, W^6, G) = (0,3,0)$, $|W_1^6| = 2$
$W_2^6 = \{15\}$ with $ADeg(W_2^6, W^6, G) = (0,2,0)$, $|W_2^6| = 1$
$W_3^6 = \{6,7\}$ with $ADeg(W_3^6, W^6, G) = (0,2,1)$, $|W_3^6| = 2$
$W_4^6 = \{18\}$ with $ADeg(W_4^6, W^6, G) = (0,1,0)$, $|W_4^6| = 1$
$W_5^6 = \{17\}$ with $ADeg(W_5^6, W^6, G) = (0,1,1)$, $|W_5^6| = 1$
$W_6^6 = \{16\}$ with $ADeg(W_6^6, W^6, G) = (0,1,0)$, $|W_6^6| = 1$
$W_7^6 = \{19\}$ with $ADeg(W_7^6, W^6, G) = (0,1,1)$, $|W_7^6| = 1$
$W_8^6 = \{0,1\}$ with $ADeg(W_8^6, W^6, G) = (0,1,2)$, $|W_8^6| = 2$
$W_9^6 = \{8,9,10,11\}$ with $ADeg(W_9^6, W^6, G) = (0,1,2)$, $|W_9^6| = 4$
$W_{10}^6 = \{2,3\}$ with $ADeg(W_{10}^6, W^6, G) = (0,0,3)$, $|W_{10}^6| = 2$

It is easy to see that partitions $\mathcal{W}^6$ and $\mathcal{S}^6$ are compatible but not equivalent. Since $P^6 = \text{VERTEX}$ and $t^6 = 2$, we apply vertex refinement to $\mathcal{W}^6$, with W_2^6 as the pivot vertex. As a result, we get:

$\mathcal{W}^7 = (W_1^7, W_2^7, W_3^7, W_4^7, W_5^7, W_6^7, W_7^7, W_8^7, W_9^7)$ where $W^7 = \bigcup_{i=1}^{9} W_i^7$ and:

$W_1^7 = \{4,5\}$ with $ADeg(W_1^7, W^7, G) = (0,3,0)$, $|W_1^7| = 2$
$W_2^7 = \{6,7\}$ with $ADeg(W_2^7, W^7, G) = (0,2,1)$, $|W_2^7| = 2$
$W_3^7 = \{18\}$ with $ADeg(W_3^7, W^7, G) = (0,1,0)$, $|W_3^7| = 1$
$W_4^7 = \{17\}$ with $ADeg(W_4^7, W^7, G) = (0,1,0)$, $|W_4^7| = 1$
$W_5^7 = \{16\}$ with $ADeg(W_5^7, W^7, G) = (0,1,0)$, $|W_5^7| = 1$
$W_6^7 = \{19\}$ with $ADeg(W_6^7, W^7, G) = (0,1,0)$, $|W_6^7| = 1$
$W_7^7 = \{0,1\}$ with $ADeg(W_7^7, W^7, G) = (0,1,2)$, $|W_7^7| = 2$
$W_8^7 = \{8,9,10,11\}$ with $ADeg(W_8^7, W^7, G) = (0,1,2)$, $|W_8^7| = 4$
$W_9^7 = \{2,3\}$ with $ADeg(W_9^7, W^7, G) = (0,0,3)$, $|W_9^7| = 2$

Partition $\mathcal{W}^7$ is compatible with partition $\mathcal{S}^7$, so next refinement will be done with $W_{t^7}^7$, that is, W_3^7. After the corresponding vertex refinement, we have:

$\mathcal{W}^8 = (W_1^8, W_2^8, W_3^8, W_4^8, W_5^8, W_6^8, W_7^8, W_8^8, W_9^8)$ where $W^8 = \bigcup_{i=1}^{9} W_i^8$ and:

$W_1^8 = \{4,5\}$ with $ADeg(W_1^8, W^8, G) = (0,3,0)$, $|W_1^8| = 2$
$W_2^8 = \{6,7\}$ with $ADeg(W_2^8, W^8, G) = (0,2,1)$, $|W_2^8| = 2$
$W_3^8 = \{17\}$ with $ADeg(W_3^8, W^8, G) = (0,1,0)$, $|W_3^8| = 1$
$W_4^8 = \{16\}$ with $ADeg(W_4^8, W^8, G) = (0,1,0)$, $|W_4^8| = 1$
$W_5^8 = \{19\}$ with $ADeg(W_5^8, W^8, G) = (0,1,0)$, $|W_5^8| = 1$
$W_6^8 = \{0,1\}$ with $ADeg(W_6^8, W^8, G) = (0,1,2)$, $|W_6^8| = 2$
$W_7^8 = \{8,9,10,11\}$ with $ADeg(W_7^8, W^8, G) = (0,1,2)$, $|W_7^8| = 4$
$W_8^8 = \{3\}$ with $ADeg(W_8^8, W^8, G) = (0,0,2)$, $|W_8^8| = 1$
$W_9^8 = \{2\}$ with $ADeg(W_9^8, W^8, G) = (0,0,3)$, $|W_9^8| = 1$

Partition $\mathcal{W}^8$ and partition $\mathcal{S}^8$ are also compatible, so another refinement step is performed, with W_9^8 as the pivot set. This refinement yields:

$\mathcal{W}^9 = (W_1^9, W_2^9, W_3^9, W_4^9, W_5^9, W_6^9, W_7^9, W_8^9, W_9^9)$ where $W^9 = \bigcup_{i=1}^{9} W_i^9$ and:

$W_1^9 = \{4,5\}$ with $ADeg(W_1^9, W^9, G) = (0,3,0)$, $|W_1^9| = 2$
$W_2^9 = \{6,7\}$ with $ADeg(W_2^9, W^9, G) = (0,2,1)$, $|W_2^9| = 2$
$W_3^9 = \{17\}$ with $ADeg(W_3^9, W^9, G) = (0,0,0)$, $|W_3^9| = 1$
$W_4^9 = \{16\}$ with $ADeg(W_4^9, W^9, G) = (0,0,0)$, $|W_4^9| = 1$
$W_5^9 = \{19\}$ with $ADeg(W_5^9, W^9, G) = (0,1,0)$, $|W_5^9| = 1$
$W_6^9 = \{0\}$ with $ADeg(W_6^9, W^9, G) = (0,0,2)$, $|W_6^9| = 1$
$W_7^9 = \{1\}$ with $ADeg(W_7^9, W^9, G) = (0,1,2)$, $|W_7^9| = 1$
$W_8^9 = \{8,9,10,11\}$ with $ADeg(W_8^9, W^9, G) = (0,1,2)$, $|W_7^9| = 4$
$W_9^9 = \{3\}$ with $ADeg(W_8^9, W^9, G) = (0,0,2)$, $|W_9^9| = 1$

Again, $\mathcal{W}^9$ is compatible with $\mathcal{S}^9$. Therefore, a refinement by vertex is performed using W_7^9 as the pivot vertex to get the next partition:

$\mathcal{W}^{10} = (W_1^{10}, W_2^{10}, W_3^{10}, W_4^{10}, W_5^{10}, W_6^{10}, W_7^{10})$ where $W^{10} = \bigcup_{i=1}^{7} W_i^{10}$ and:
$W_1^{10} = \{4,5\}$ with $ADeg(W_1^{10}, W^{10}, G) = (0,3,0)$, $|W_1^{10}| = 2$
$W_2^{10} = \{6,7\}$ with $ADeg(W_2^{10}, W^{10}, G) = (0,2,1)$, $|W_2^{10}| = 2$
$W_3^{10} = \{19\}$ with $ADeg(W_3^{10}, W^{10}, G) = (0,1,0)$, $|W_3^{10}| = 1$
$W_4^{10} = \{0\}$ with $ADeg(W_4^{10}, W^{10}, G) = (0,0,2)$, $|W_4^{10}| = 1$
$W_5^{10} = \{10,11\}$ with $ADeg(W_5^{10}, W^{10}, G) = (0,0,2)$, $|W_5^{10}| = 2$
$W_6^{10} = \{8,9\}$ with $ADeg(W_6^{10}, W^{10}, G) = (0,1,2)$, $|W_6^{10}| = 2$
$W_7^{10} = \{3\}$ with $ADeg(W_7^{10}, W^{10}, G) = (0,0,1)$, $|W_7^{10}| = 1$

Note that cells W_3^9 and W_4^9 have been discarded prior to refinement since they had no links. We have got a partition $\mathcal{W}^{10}$ which is compatible with $\mathcal{S}^{10}$, so we apply a new refinement. Using cell W_4^{10} as the pivot set, we apply a vertex refinement, and get:

$\mathcal{W}^{11} = (W_1^{11}, W_2^{11}, W_3^{11}, W_4^{11}, W_5^{11}, W_6^{11})$ where $W^{11} = \bigcup_{i=1}^{6} W_i^{11}$ and:
$W_1^{11} = \{4,5\}$ with $ADeg(W_1^{11}, W^{11}, G) = (0,3,0)$, $|W_1^{11}| = 2$
$W_2^{11} = \{6,7\}$ with $ADeg(W_2^{11}, W^{11}, G) = (0,2,1)$, $|W_2^{11}| = 2$
$W_3^{11} = \{19\}$ with $ADeg(W_3^{11}, W^{11}, G) = (0,1,0)$, $|W_3^{11}| = 1$
$W_4^{11} = \{10,11\}$ with $ADeg(W_4^{11}, W^{11}, G) = (0,0,2)$, $|W_4^{11}| = 2$
$W_5^{11} = \{8,9\}$ with $ADeg(W_5^{11}, W^{11}, G) = (0,0,2)$, $|W_5^{11}| = 2$
$W_6^{11} = \{3\}$ with $ADeg(W_6^{11}, W^{11}, G) = (0,0,1)$, $|W_6^{11}| = 1$

Since $\mathcal{W}^{11}$ is compatible with $\mathcal{S}^{11}$, we perform another refinement using W_3^{11} as the pivot set (remember that $t^{11} = 3$). After a vertex refinement, we obtain a new partition:

$\mathcal{W}^{12} = (W_1^{12}, W_2^{12}, W_3^{12}, W_4^{12}, W_5^{12})$ where $W^{12} = \bigcup_{i=1}^{5} W_i^{12}$ and:
$W_1^{12} = \{4,5\}$ with $ADeg(W_1^{12}, W^{12}, G) = (0,3,0)$, $|W_1^{12}| = 2$
$W_2^{12} = \{6,7\}$ with $ADeg(W_2^{12}, W^{12}, G) = (0,2,1)$, $|W_2^{12}| = 2$
$W_3^{12} = \{10,11\}$ with $ADeg(W_3^{12}, W^{12}, G) = (0,0,2)$, $|W_3^{12}| = 2$
$W_4^{12} = \{8,9\}$ with $ADeg(W_4^{12}, W^{12}, G) = (0,0,2)$, $|W_4^{12}| = 2$
$W_5^{12} = \{3\}$ with $ADeg(W_5^{12}, W^{12}, G) = (0,0,0)$, $|W_5^{12}| = 1$

This partition $\mathcal{W}^{12}$ is compatible with partition $\mathcal{S}^{12}$. Even more, it is the same partition. Therefore, they are equivalent, and we can determine the equivalence of vertices 12 and 13 in cell S_1^3 without the need of more refinements. Hence, we set $P^3 =$ VERTEX. Since there is no refinement marked BACKTR, there will be no backtracking in the next phase of the algorithm (the search for an equivalent sequence of partitions for graph H).

4.3 Generating the sequence of partitions for graph H

We begin from the partition $\mathcal{T}^0$ built previously. Then, we start generating new partitions applying the refinement technique used for graph G at each point (according to P^l where l ranges from 0 to $ll - 1$), with the corresponding pivot set (according to t^l for the same values of l). In this way, we obtain the following sequence of partitions (we do not present the details since it is easy to follow from Figure 16 and the generation of the sequence of partitions for graph G):

$\mathcal{T}^1 = (\{4,7,12,14\}, \{2,5,11,17\}, \{3,6,8,10,15,16\}, \{0,1,9,13\}, \{18,19\})$
$\mathcal{T}^2 = (\{4,7,12,14\}, \{2,5,11,17\}, \{6,8,15,16\}, \{3,10\}, \{0,1,9,13\}, \{18,19\})$
$\mathcal{T}^3 = (\{4,12\}, \{7,14\}, \{2,11\}, \{5,17\}, \{6,8,15,16\}, \{3,10\}, \{0,1,9,13\}, \{18,19\})$

Now, cell $\{4, 12\}$ is the next pivot set to use for vertex refinement. Since we have discarded this as a point of possible backtracking, any vertex in the cell may be chosen as the pivot vertex (if graph H is isomorphic to graph G, vertex equivalence in graph G implies the same equivalence in graph H). For this example, we will choose vertex 4. Having taken this choice, we can go on generating new partitions:

$\mathcal{T}^4 = (\{12\}, \{7, 14\}, \{11\}, \{2\}, \{5, 17\}, \{6, 15\}, \{8, 16\}, \{3, 10\}, \{0, 1, 9, 13\}, \{18, 19\})$
$\mathcal{T}^5 = (\{7, 14\}, \{11\}, \{2\}, \{5, 17\}, \{6, 15\}, \{8, 16\}, \{3, 10\}, \{0, 1, 9, 13\}, \{18, 19\})$
$\mathcal{T}^6 = (\{7, 14\}, \{2\}, \{5, 17\}, \{6\}, \{15\}, \{16\}, \{8\}, \{3, 10\}, \{0, 1, 9, 13\}, \{18, 19\})$
$\mathcal{T}^7 = (\{7, 14\}, \{5, 17\}, \{6\}, \{15\}, \{16\}, \{8\}, \{3, 10\}, \{0, 1, 9, 13\}, \{18, 19\})$
$\mathcal{T}^8 = (\{7, 14\}, \{5, 17\}, \{15\}, \{16\}, \{8\}, \{3, 10\}, \{0, 1, 9, 13\}, \{19\}, \{18\})$
$\mathcal{T}^9 = (\{7, 14\}, \{5, 17\}, \{15\}, \{16\}, \{8\}, \{3\}, \{10\}, \{0, 1, 9, 13\}, \{19\})$
$\mathcal{T}^{10} = (\{7, 14\}, \{5, 17\}, \{8\}, \{3\}, \{1, 9\}, \{0, 13\}, \{19\})$
$\mathcal{T}^{11} = (\{7, 14\}, \{5, 17\}, \{8\}, \{1, 9\}, \{0, 13\}, \{19\})$
$\mathcal{T}^{12} = (\{7, 14\}, \{5, 17\}, \{1, 9\}, \{0, 13\}, \{19\})$

Again, we have a pivot cell with two vertices, but a vertex refinement without the need of backtracking is performed, with cell $\{7, 14\}$ as the pivot set. Since we can choose any vertex in the pivot set, vertex 7 will do. Using it as the pivot vertex, after a vertex refinement, we obtain:

$\mathcal{T}^{13} = (\{14\}, \{5\}, \{17\}, \{9\}, \{1\}, \{13\}, \{0\})$

We have reached the final partition, and now we have to test if this partition is compatible with partition $\mathcal{S}^{13}$. This equivalence is validated by algorithm *FinalPartitionsAreEquivalent*. Figure 18 shows the adjacencies between the vertices in partition $\mathcal{T}^{13}$ from graph H and partition $\mathcal{S}^{13}$ from graph G, where this equivalence is easily observed. Hence, we can conclude that G and H are isomorphic.

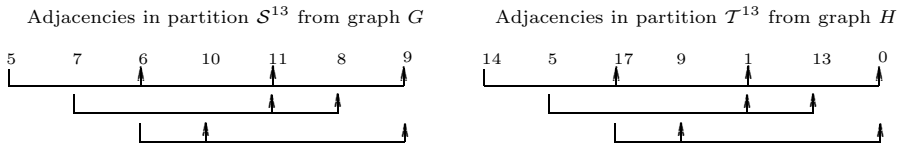

Figure 18: Equivalence of the final partitions $\mathcal{T}^{13}$ and $\mathcal{S}^{13}$.

The sequences of partitions obtained induce orders on the vertices of G and H that let us establish a correspondence between the vertices of both graphs. Remember that this orders are not canonical (isomorphic graphs can lead to different non-automorphic orders). The orders induced by the sequences of partitions generated are:

G	12	13	15	14	19	2	16	17	1	0	18	3	4	5	7	6	10	11	8	9
H	4	12	11	2	6	18	15	16	10	3	8	19	7	14	5	17	9	1	13	0

This correspondence is not obtained with the algorithm presented, though it could be found with little effort. However, it is useful to explicitly present it here to verify the isomorphism found by the algorithm.

5 Correctness of the Algorithm

In this section we show that the algorithm in fact correctly determines whether two graphs are isomorphic. The algorithm presented generates a sequence of partitions for both graphs tested. We first define the concepts of final partition, sequence of partitions, and compatibility between two sequences of partitions.

Definition 1 *Let $G = (V_G, R_G)$ be a graph. Let $\mathcal{S} = (S_1, ..., S_r)$ be a partition of $V \subseteq V_G$. Then, $\mathcal{S}$ is a* final partition *if there is no cell $S_i \in \mathcal{S}$ such that $|S_i| > 1$ and $HasLinks(S_i, V, G)$.*

Definition 2 *Let $G = (V_G, R_G)$ be a graph. $\mathcal{S}^0, ..., \mathcal{S}^{ll}$ is a* sequence of partitions *for graph G if:*

1. *$\mathcal{S}^0 = DegreePartition(G)$.*
2. *For each $i \in \{1, ..., ll\}$, $\mathcal{S}^i$ is generated from $\mathcal{S}^{i-1}$ using vertex refinement or set refinement as described in algorithms RefineByVertex (see Figure 5) and RefineBySet (see Figure 6).*

3. $\mathcal{S}^{ll}$ *is a final partition.*

Definition 3 *Two sequences of partitions,* $\mathcal{S}^0, ..., \mathcal{S}^{ll}$ *for graph* G*, and* $\mathcal{T}^0, ..., \mathcal{T}^{ll}$ *for graph* H*, are* compatible *if:*

1. $\mathcal{S}^0$ *and* $\mathcal{T}^0$ *are compatible according to the criteria used by algorithm DegreePartitionsAreCompatible (see Figure 2).*
2. *Partitions* $\mathcal{S}^{l+1}$ *and* $\mathcal{T}^{l+1}$ *are generated from* $\mathcal{S}^l$ *and* $\mathcal{T}^l$ *using the same kind of refinement* P^l *and the same pivot set index* t^l *for all* $l \in \{0, ..., ll-1\}$.
3. *Partitions* $\mathcal{S}^l$ *and* $\mathcal{T}^l$ *are compatible according to the corresponding kind of refinement* P^{l-1} *and pivot set used to generate them (VertexPartitionIsCompatible or SetPartitionIsCompatible, Figures 12 and 13, respectively) for all* $l \in \{1, ..., ll\}$.
4. $\mathcal{S}^{ll}$ *and* $\mathcal{T}^{ll}$ *are equivalent according to algorithm FinalPartitionsAreEquivalent (see Figure 14).*

Lemma 1 *Let* G *and* H *be two isomorphic graphs. Then, there are two compatible sequences of partitions* $\mathcal{S}^0, ..., \mathcal{S}^{ll}$ *and* $\mathcal{T}^0, ..., \mathcal{T}^{ll}$ *for* G *and* H*, respectively.*

Proof: Since G and H are isomorphic, there must be (at least) one mapping m from the vertices of G to the vertices of H that preserves adjacencies. Also, the degree partitions $\mathcal{S}^0$ and $\mathcal{T}^0$, for G and H respectively, must be compatible. Moreover, let $r = |\mathcal{S}^0| = |\mathcal{T}^0|$, $\mathcal{S}^0 = (S_1^0, ..., S_r^0)$, and $\mathcal{T}^0 = (T_1^0, ..., T_r^0)$, then, m maps the vertices in S_i^0 to the vertices in T_i^0, for each $i \in \{1, ..., r\}$.

Now, by induction, we assume that compatibility exists up to partitions $\mathcal{S}^l$ and $\mathcal{T}^l$, and that m maps the vertices in the ith cell of $\mathcal{S}^l$ to the vertices in the ith cell of $\mathcal{T}^l$. Then we prove that this also holds for partitions $\mathcal{S}^{l+1}$ and $\mathcal{T}^{l+1}$.

Note first that if a cell has been discarded when deriving $\mathcal{S}^{l+1}$ from $\mathcal{S}^l$, that was because it had no remaining links. Since the vertices of that cell are mapped by m to the vertices of its corresponding cell in $\mathcal{T}^l$, this last cell can not have links and will also be discarded in $\mathcal{T}^{l+1}$.

Now, partition $\mathcal{S}^{l+1}$ is generated from $\mathcal{S}^l$ under one of the following three different circumstances:

1. The pivot set $S_{t^l}^l$ in $\mathcal{S}^l$ has only one vertex, so vertex refinement is applied.
2. The pivot set $S_{t^l}^l$ in $\mathcal{S}^l$ has more than one vertex and set refinement is applied.
3. The pivot set $S_{t^l}^l$ in $\mathcal{S}^l$ has more than one vertex and vertex refinement is applied.

In Case 1, for graph H, we can generate a new partition $\mathcal{T}^{l+1}$ from $\mathcal{T}^l$ using vertex refinement with the pivot set $T_{t^l}^l$ (which,from the induction hypothesis, contains a single vertex, image under m of the only vertex in $S_{t^l}^l$). Also from the induction hypothesis, the vertices in cell $S_i^l \in \mathcal{S}^l$ are mapped under m to the vertices in cell $T_i^l \in \mathcal{T}^l$ for $i \in \{1, ..., |\mathcal{S}^l|\}$. Hence, if the pivot vertex in $S_{t^l}^l$ has a certain kind of link with k vertices in some cell S_i^l, then the vertex in $T_{t^l}^l$ must also have a link of that kind with k vertices in cell T_i^l. Otherwise, there would be vertices in S_i^l which could not be mapped by m to vertices in T_i^l for having different adjacencies. Therefore, the new cells generated will have the same number of vertices and their vertices will have the same kind of adjacency with the respective pivot vertex. Hence, the new partition generated $\mathcal{T}^{l+1}$ must be compatible with partition $\mathcal{S}^{l+1}$, and the vertices in cell $S_i^{l+1} \in \mathcal{S}^{l+1}$ can only be mapped, under mapping m, to the vertices in $T_i^{l+1} \in \mathcal{T}^{l+1}$.

In Case 2, we generate partition $\mathcal{T}^{l+1}$ using set refinement with the corresponding pivot set $T_{t^l}^l$. By the induction hypothesis, cells $S_{t^l}^l$ and $T_{t^l}^l$ must have the same adjacencies with the corresponding cells in both partitions. Therefore, the new cells generated will have the same adjacencies with the pivot set in both graphs. Hence, the new cells in $\mathcal{S}^{l+1}$ must be mapped under m to the corresponding new cells in $\mathcal{T}^{l+1}$, and $\mathcal{S}^{l+1}$ and $\mathcal{T}^{l+1}$ must be compatible.

In Case 3, the pivot vertex p^l chosen from cell $S_{t^l}^l$ could be mapped to any vertex in $T_{t^l}^l$. However, one of them must be $m(p^l)$ since $S_{t^l}^l$ and $T_{t^l}^l$ are compatible and the vertices in $S_{t^l}^l$ can only be mapped to vertices in $T_{t^l}^l$. Using $m(p^l)$ as the pivot vertex, we generate a new partition $\mathcal{T}^{l+1}$ compatible with $\mathcal{S}^{l+1}$ since p^l and $m(p^l)$ have the same adjacencies with the same cells. Hence, the new partition generated $\mathcal{T}^{l+1}$ must be

compatible with partition $\mathcal{S}^{l+1}$, and the vertices in cell $S_i^{l+1} \in \mathcal{S}^{l+1}$ can only be mapped, under mapping m, to the vertices in $T_i^{l+1} \in \mathcal{T}^{l+1}$ for $i \in \{1, ..., |\mathcal{S}^{l+1}|\}$.

This way, it is possible to generate a sequence of partitions $\mathcal{T}^0, ..., \mathcal{T}^{ll}$ compatible with $\mathcal{S}^0, ..., \mathcal{S}^{ll}$ up to the final partitions (those that have no cells with remaining links and more than one vertex). The equivalence of these partitions is easy to see. Remember that the vertices in one cell in $\mathcal{S}^{ll}$ can only be mapped, under mapping m, to vertices of its corresponding cell in $\mathcal{T}^{ll}$. ■

A partition of a set of vertices of a graph induces a (partial) order on these vertices. A partition where every cell is of size one induces a total order on these vertices. Besides, a sequence of partitions induces also a partial order as follows.

Definition 4 *Let $\mathcal{S}^0, ..., \mathcal{S}^{ll}$ be a sequence of partitions for some graph $G = (V_G, R_G)$. Let V^l be the vertices in partition $\mathcal{S}^l$ for all $l \in \{0, ..., ll\}$. Note that $V_G = V^0 \supseteq ... \supseteq V^{ll}$. The* order induced *on the vertices of V_G by the sequence of partitions $\mathcal{S}^0, ..., \mathcal{S}^{ll}$ is that which satisfies the following conditions:*

- *For all $l \in \{0, ..., ll-1\}$, vertex $v \in V^l \setminus V^{l+1}$ precedes vertex $w \in V^{l+1}$ (i.e. if v was discarded in an earlier refinement than w).*
- *For all $l \in \{0, ..., ll-1\}$, if $P^l =$* VERTEX*, then p^l precedes any other vertex $v \in V^l \setminus V^{l+1}$.*
- *For all $l \in \{0, ..., ll-1\}$, let $\mathcal{S}^l = (S_1^l, ..., S_r^l)$. For all $i, j \in \{1, ..., r\}$ such that S_i^l and S_j^l have no links, let $v \in S_i^l$ and $w \in S_j^l$. Then, v precedes w if $i < j$.*
- *Let $\mathcal{S}^{ll} = (S_1^{ll}, ..., S_r^{ll})$. Let $v \in S_i^{ll}, w \in S_j^{ll}$, $i, j \in \{1, ..., r\}$. Then, v precedes w if $i < j$.*

Note that there are pairs of vertices that are not ordered with Definition 4. These vertices are interchangeable. When referring to the order induced by a partition or a sequence of partitions, we mean any total order that respects the (partial) order defined.

Let us start analyzing the final partitions $\mathcal{S}^{ll}$ and $\mathcal{T}^{ll}$ of two sequences of partitions: $\mathcal{S}^0, ..., \mathcal{S}^{ll}$ and $\mathcal{T}^0, ..., \mathcal{T}^{ll}$ obtained from graphs G and H respectively. Partitions $\mathcal{S}^{ll}$ and $\mathcal{T}^{ll}$ induce orders on their vertices, say $v_1, ..., v_k$ and $w_1, ..., w_k$ respectively. If these partitions are equivalent according to algorithm *FinalPartitionsAreEquivalent*, there is a correspondence between v_i and w_i for all $i \in \{1, ..., k\}$ that preserves adjacencies.

Observation 1 *Let $\mathcal{S}^{ll}$ under graph G and $\mathcal{T}^{ll}$ under graph H be two final partitions equivalent according to algorithm FinalPartitionsAreEquivalent. Let $v_1, ..., v_k$ and $w_1, ..., w_k$ be the respective induced orders of their vertices. Mapping $m : \{v_1, ..., v_k\} \longrightarrow \{w_1, ..., w_k\}$ such that $m(v_i) = w_i$ for all $i \in \{1, ..., k\}$ is an isomorphism between the subgraphs induced by $\{v_1, ..., v_k\}$ on G and $\{w_1, ..., w_k\}$ on H. If G and H are the same graph and $\{v_1, ..., v_k\} = \{w_1, ..., w_k\}$, then m is an automorphism.*

Let $\mathcal{S}^0, ..., \mathcal{S}^{ll}$ and $\mathcal{T}^0, ..., \mathcal{T}^{ll}$ be two compatible sequences of partitions for graphs G and H respectively. Let $v_1, ..., v_n$ be the order induced by the sequence of partitions $\mathcal{S}^0, ..., \mathcal{S}^{ll}$ on the vertices of graph G. Let $w_1, ..., w_n$ be the order induced by the sequence of partitions $\mathcal{T}^0, ..., \mathcal{T}^{ll}$ on the vertices of graph H. Let m be the mapping $m : \{v_1, ..., v_n\} \longrightarrow \{w_1, ..., w_n\}$ such that $m(v_i) = w_i$, for all $i \in \{1, ..., n\}$.

Lemma 2 *For all $k \in \{1, ..., n\}$, m is an isomorphism between the subgraphs induced by $\{v_k, ..., v_n\}$ on G and $\{w_k, ..., w_n\}$ on H.*

Proof: Let $v_k, ..., v_n$ be the order induced by $\mathcal{S}^{ll}$ on its vertices. Let $w_k, ..., w_n$ be the order induced by $\mathcal{T}^{ll}$ on its vertices. From Observation 1, mapping m is an isomorphism between the subgraphs induced by $\mathcal{S}^{ll}$ on G and $\mathcal{T}^{ll}$ on H.

Now, by induction, let $v_j, ..., v_n$ be the last $n - j + 1$ vertices in the order induced by the sequence of partitions $\mathcal{S}^0, ..., \mathcal{S}^{ll}$ on the vertices of graph G, and let $w_j, ..., w_n$ be the last $n - j + 1$ vertices in the order induced by the sequence of partitions $\mathcal{T}^0, ..., \mathcal{T}^{ll}$ on the vertices of graph H. We assume that m is an isomorphism between the subgraphs induced on G and H by $\{v_j, ..., v_n\}$ and $\{w_j, ..., w_n\}$ respectively and prove that m is also an isomorphism between the subgraphs induced by $\{v_{j-1}, v_j, ..., v_n\}$ and $\{w_{j-1}, w_j, ..., w_n\}$ on G and H respectively. Vertices v_{j-1} and w_{j-1} are vertices which where discarded for one of the following reasons:

1. They belonged to cells with no links.

2. They were used as the pivot vertices for a vertex refinement.

In Case 1, the subgraphs induced by $\{v_{j-1}, v_j, ..., v_n\}$ and $\{w_{j-1}, w_j, ..., w_n\}$ on G and H respectively are the previous ones with a new disconnected vertex. These new vertices are v_{j-1} and $m(v_{j-1}) = w_{j-1}$. Clearly, m is an isomorphism between the new subgraphs.

In Case 2, since the sequences of partitions are compatible, if there is an adjacency between v_{j-1} and any vertex $v \in \{v_{j-1}, v_j, ..., v_n\}$, then the same adjacency must exist between w_{j-1} and $m(v)$. Therefore, m is an isomorphism between the new subgraphs.

Since we can do this up to vertices v_1 and w_1, m is an isomorphism for all $k \in \{1, ..., n\}$. ■

Corollary 1 *Mapping m maps vertices in corresponding cells for all partitions $\mathcal{S}^l$ and $\mathcal{T}^l$, $l \in \{0, ..., ll\}$. I.e., let $\mathcal{S}^l = (S_1^l, ..., S_r^l)$ and $\mathcal{T}^l = (T_1^l, ..., T_r^l)$, then $v \in S_i^l$ implies $m(v) \in T_i^l$, for all $i \in \{1, ..., r\}$.*

Corollary 2 *Mapping m is an isomorphism between G and H. If G and H are the same graph, m is an automorphism.*

Proof: This is just the case for $k = 1$ in Lemma 2. ■

Corollary 3 *Let V^l be the vertices in $\mathcal{S}^l$ and W^l be the vertices in $\mathcal{T}^l$ for all $l \in \{0, ..., ll\}$. Then, mapping m is an isomorphism between the subgraph induced by V^l on G and the subgraph induced by W^l on H.*

Proof: It follows directly from Lemma 2. ■

Theorem 1 *Two graphs G and H are isomorphic if and only if there are two equivalent sequences of partitions $\mathcal{S}^0, ..., \mathcal{S}^{ll}$ for graph G and $\mathcal{T}^0, ..., \mathcal{T}^{ll}$ for graph H.*

Proof: Immediate from Lemma 1 and Corollary 2. ■

We say that two vertices of a pivot set used for vertex refinement are *equivalent* if they generate compatible sequences of partitions. Algorithm *SearchAutomorphisms* (see Figure 7) tests for equivalence among the vertices in the pivot sets of partitions $\mathcal{S}^l$ in decreasing order of the index l. If it finds that all the vertices in the pivot set are equivalent, it changes P^l from UNKNOWN to VERTEX, and to BACKTR otherwise.

Let $\mathcal{S}^0, ..., \mathcal{S}^{ll}$ be a sequence of partitions for graph G. Let $\mathcal{S}^l = (S_1^l, ..., S_r^l)$ be a partition such that, for all $k \in \{l+1, ..., ll-1\}$, if $\mathcal{S}^k$ has been refined by vertex with a pivot set $S_{t^k}^k$ with more than one vertex, then all the vertices in $S_{t^k}^k$ have been proved to be equivalent (and, hence, P^k has been changed from UNKNOWN to VERTEX). Note that algorithm *IsEquivalentToThePivotVertex* tries only one vertex in the pivot set $S_{t^k}^k$ when $P^k =$ VERTEX, no matter the size of the pivot set. The following lemma and its corollary prove that is enough.

Lemma 3 *If using a vertex $x \in S_{t^l}^l$, $x \neq p^l$, when refining partition $\mathcal{S}^l$, it is possible to generate a sequence of partitions $\mathcal{S}^0, ..., \mathcal{S}^l, \mathcal{W}^{l+1}, ..., \mathcal{W}^{ll}$ compatible with $\mathcal{S}^0, ..., \mathcal{S}^{ll}$, then, for all $k \in \{l+1, ..., ll-1\}$ such that $P^k =$ VERTEX and $|S_{t^k}^k| > 1$, choosing any vertex in the pivot set $W_{t^k}^k$ when refining partition $\mathcal{W}^k$ by vertex, will yield compatible sequences of partitions.*

Proof: Let $\mathcal{S}^0, ..., \mathcal{S}^l, \mathcal{S}^{l+1}, ..., \mathcal{S}^k, \mathcal{S}^{k+1}, ..., \mathcal{S}^{ll}$ be the sequence of partitions generated using vertex p^l to refine partition $\mathcal{S}^l$. Let $\mathcal{S}^0, ..., \mathcal{S}^l, \mathcal{W}^{l+1}, ..., \mathcal{W}^k, \mathcal{W}^{k+1}, ..., \mathcal{W}^{ll}$ be the sequence of partitions generated using vertex x instead of p^l, which is compatible with the previous one. Let $P^k =$ VERTEX and $|W_{t^k}^k| > 1$, such that $\mathcal{W}^{k+1}$ has been generated using vertex $y \in W_{t^k}^k$. By the way of contradiction, let us assume that there is a vertex $z \in W_{t^k}^k$ with which it is not possible to generate a sequence of partitions compatible with $\mathcal{S}^0, ..., \mathcal{S}^l, \mathcal{S}^{l+1}, ..., \mathcal{S}^k, \mathcal{S}^{k+1}, ..., \mathcal{S}^{ll}$.

Let V be the vertices of $\mathcal{S}^k$ and W the vertices of $\mathcal{W}^k$. From Corollary 3, m is an isomorphism between the subgraphs induced by V and W on G. From Corollary 1, the vertices in each cell of $\mathcal{S}^k$ are mapped by m to the vertices in the corresponding cell of $\mathcal{W}^k$. Then, $m^{-1}(z) \in S^k_{t^k}$ and, since all the vertices in $S^k_{t^k}$ are equivalent, using $m^{-1}(z)$ to refine partition $\mathcal{S}^k$ yields a sequence of partitions compatible with $\mathcal{S}^0, ..., \mathcal{S}^{ll}$. Using the same argument used in the proof of Lemma 1, we can generate a sequence of partitions, compatible with this one, using vertex z to refine partition $\mathcal{W}^k$. Hence, we have come to a contradiction. ■

Corollary 4 *If using a vertex $x \in S^l_{t^l}$, $x \neq p^l$, when refining partition $\mathcal{S}^l$, we reach a partition $\mathcal{W}^k$ such that $P^k =$ VERTEX and $|S^k_{t^k}| > 1$, and using a vertex $y \in W^k_{t^k}$ leads to an incompatibility in the sequence of partitions being generated, trying another vertex $z \in W^k_{t^k}$ will never yield a compatible sequence of partitions.*

Proof: By the way of contradiction, let $z \in W^k_{t^k}$ be a vertex that, used as the pivot vertex to refine partition $\mathcal{W}^k$, yields a sequence of partitions compatible with $\mathcal{S}^0, ..., \mathcal{S}^{ll}$. Then, from Lemma 3, any other vertex $y \in W^k_{t^k}$ will also yield a compatible sequence of partitions. Hence, we have reached a contradiction. ■

These results show that, when looking for automorphisms in graph G, the algorithm learns about partial automorphisms. When it has used some vertex to refine partition $\mathcal{S}^l$ by vertex, if $P^l =$ VERTEX (no matter the size of the pivot set), and it gets to a partition which is not equivalent to its corresponding one in $\mathcal{S}^0, ..., \mathcal{S}^{ll}$, it knows (from Corollary 4) that trying another vertex will not work either.

Observation 2 *If two graphs are isomorphic, then they have the same automorphisms.*

Lemma 4 *Let $\mathcal{S}^0, ..., \mathcal{S}^{ll}$ and $\mathcal{T}^0, ..., \mathcal{T}^{ll}$ be two compatible sequences of partitions for two isomorphic graphs G and H. Then, if there is some l such that $P^l =$ VERTEX and $|S^l_{t^l}| > 1$ (i.e. all the vertices in that pivot set are equivalent), then, all the vertices in $T^l_{t^l}$ will also be equivalent.*

Proof: Let $v_1, ..., v_n$ be the order induced by the sequence of partitions $\mathcal{S}^0, ..., \mathcal{S}^{ll}$ on the vertices of graph G. Let $w_1, ..., w_n$ be the order induced by the sequence of partitions $\mathcal{T}^0, ..., \mathcal{T}^{ll}$ on the vertices of graph H. Let m be the mapping $m : \{v_1, ..., v_n\} \longrightarrow \{w_1, ..., w_n\}$ such that $m(v_i) = w_i$, for all $i \in \{1, ..., n\}$. From Corollary 1, m maps the vertices in $S^l_{t^l}$ to the vertices in $T^l_{t^l}$. From Observation 2, since G and H are isomorphic, they have the same automorphisms. Therefore, if the vertices of cell $S^l_{t^l}$ are equivalent, their images under m (the vertices of $T^l_{t^l}$) must also be equivalent. ■

Corollary 5 *Let $\mathcal{S}^0, ..., \mathcal{S}^{ll}$ be a sequence of partitions for graph G, such that for some $l \in \{0, ..., ll\}$, $P^l =$ VERTEX and $|S^l_{t^l}| > 1$. If trying to generate a sequence of partitions for graph H compatible with this one, $\mathcal{T}^0, ..., \mathcal{T}^{ll}$, using some vertex $x \in T^l_{t^l}$ we can not satisfy the compatibility, trying another vertex $y \in T^l_{t^l}, y \neq x$ will not work either.*

Proof: By the way of contradiction, let us assume, there exists such a vertex y which yields a sequence of partitions compatible with $\mathcal{S}^0, ..., \mathcal{S}^{ll}$. In that case, from Lemma 4, vertex x would also yield and equivalent sequence of partitions. Therefore, we reach a contradiction. ■

These results are used by algorithm *Match* to reduce the amount of backtracking needed to find a sequence of partitions, for graph H, compatible with that previously generated for graph G. Only one branch in the search tree is explored if all the vertices of a pivot set are known to be equivalent. This greatly improves the performance of the algorithm.

Theorem 2 *Graphs G and H are isomorphic if and only if AreIsomorphic(G, H) returns* TRUE.

Proof: If graphs G and H do not have the same number of vertices and arcs, or their degree partitions are not compatible, they can not be isomorphic and *AreIsomorphic* returns FALSE. If their degree partitions are compatible, a sequence of partitions for graph G is generated and searched for automorphisms. Then, algorithm *Match* tries to find a sequence of partitions for graph H compatible with the one generated by *GenerateFirstSequenceOfPartitions* for graph G. In this process, four cases arise when refining partition $\mathcal{S}^l$ for $l \in \{0, ..., ll - 1\}$:

1. P^l = VERTEX and $|S^l_{t^l}| = 1$.
2. P^l = GROUP.
3. P^l = VERTEX and $|S^l_{t^l}| > 1$.
4. P^l = BACKTR.

In Case 1, *Match* uses vertex refinement with the corresponding pivot set, and tests the new partition $\mathcal{T}^{l+1}$ for compatibility with $\mathcal{S}^{l+1}$. If they are compatible, it follows that branch in the search tree. This corresponds to Case 1 in the proof of Lemma 1. If they are not compatible, it backtracks looking for an unexplored branch in the search tree.

In Case 2, *Match* applies a set refinement with the corresponding pivot set, testing the new partition for compatibility, and taking the same actions as in the previous case. This corresponds to Case 2 in the proof of Lemma 1.

In Case 3, only one vertex in the pivot set needs to be tried, since, in case the choice made does not lead to a compatible sequence of partitions, from Corollary 5, no other vertex in the pivot set would work. The actions taken in this case are the same as in the previous cases.

In Case 4, any of the vertices may match the pivot vertex p^l. Therefore, every vertex in $T^l_{t^l}$ is tried. If none of them matches, then the algorithm backtracks.

This way, algorithm *Match*, explores every plausible branch. Hence, if it is possible to generate a compatible sequence of partitions for graph H, it will find it. If it is not possible to find such a sequence of partitions, it does not exist and graphs G and H are not isomorphic. ■

6 Complexity of the algorithm

In this section we will briefly study the space and time complexities of the proposed algorithm.

6.1 Space Complexity

Let us first deal with the space complexity of the algorithm. Note that a space complexity of n^3, with n the number of vertices of the graphs, would make the algorithm unable in practice to deal with graphs of thousands of vertices, since that would require Gigabytes of storing space. Hence, we have make an effort to limit the memory space required to kn^2, for a small constant k. The most costly memory requirement comes from the fact that the algorithm needs to store the sequences of partitions for both graphs, and all their associated information (pivot set index, pivot vertex, refinement technique used, etc.). Each partition is represented by a data structure that requires $O(n)$ space (in our implementation we use no more than $23n$ bytes). We show now that the total number of partitions that have to be stored is at most $2n$.

Observe that Algorithm *GenerateFirstSequenceOfPartitions* stops when it reaches a partition whose cells with more than one vertex do not have links. Each partition is generated from the previous one using two kinds of refinement: vertex refinement and set refinement. Let us first consider only vertex refinements. In this case, the new partition has, at least, one vertex less than the previous one. That means that, in the worst case, after $n-1$ vertex refinements, we will get a partition with only one cell and with only one vertex (the other $n-1$ vertices have been discarded). On the other hand, considering only set refinements, in each refinement at least one cell is split. Hence, after $n-1$ set refinements, there will be n cells, all of them with only one vertex. At this point, the final partition has been reached. Combining both techniques, at most $2(n-1)$ refinements are necessary to reach the final partition. Hence, we have that a sequence of partitions can be stored in $O(n^2)$ space.

Note that at most two sequences of partitions have to be maintained simultaneously in memory at any point in the algorithm. When the sequence of partitions for graph G is generated, that sequence of partitions has to be stored. When the search for automorphisms is performed, it is necessary to store a second sequence of partitions: the one being generated and tested for equivalence with the first one. Once when the search for automorphisms ends, this sequence is discarded. Then, during the search for an equivalent sequence of

partitions for graph H, again it is necessary to keep another sequence in memory. Hence, at any time, only two sequences of partitions are needed and the total memory is still $O(n^2)$. Note that this is also the order of the space required to store each adjacency matrix.

6.2 Time complexity

Let us now consider the time complexity of the different parts of the algorithm. The generation of the degree partitions is done by ordering the vertices by their degree, with cost $O(n \log n)$, and then by generating the cells in time $O(n)$. The other partitions are generated by refining their previous partition with either *RefineByVertex* or *RefineBySet*. *RefineByVertex* can be implemented with a cost in time of $O(n)$, being n the number of vertices in the partition being refined. However, *RefineBySet* has a much higher cost since, for each cell in the partition being refined, it is necessary to compute the available degree of each of its vertices with respect to the pivot set. If a cell has k vertices and the pivot set has p vertices, this process requires time $O(kp)$ for this cell. Then, it is necessary to order the vertices of each cell (which takes $O(k \log k)$ for a k-vertex cell). Finally, the cells in the new partition have to be generated, which can be done in time $O(n)$. In the worst case, to generate a new partition it is necessary to try all the cell in the partition as pivot sets for unsuccessful set refinements, and finally use a vertex refinement. It is easy to see that this whole process requires at most $O(n^2)$ time. Since at most $O(n)$ refinements are necessary, generating the first sequence of partitions for graph G takes time $O(n^3)$.

The search for automorphisms is done generating subsequences of partitions, starting from the one whose pivot set is being checked for vertex equivalence. The time required for this search depends greatly on the original sequence of partitions. The worst case would be that all the pivot sets used have to be tested. Hence at most $O(n^2)$ sequences of partitions have to be generated and, as we saw previously, each can take at most $O(n^3)$ time, which yields a (loose) bound of $O(n^5)$.

The bounds we have established for the first two steps in the algorithm are clearly polynomial. However, as far as we know, algorithm *Match* may take an exponential time since it is a backtracking algorithm and we have not been able to bound the number of potential backtracking points below $\Omega(n)$. Its practical performance will rely on the ability of the previous steps to reduce the amount of backtracking needed here.

7 Performance comparison

As we said before, we have compared the performance of an implementation of our algorithm, which we will call *conauto*, with two other programs: *nauty-2.0* [8] and *vf2* [11]. In our implementation, we have slightly modified our algorithm, so that in the automorphisms discovery phase our program does not check the first potential backtracking point. Testing at this point for equivalence among the vertices in the pivot set may be very expensive if the pivot set has $O(n)$ vertices, which is the case for vertex-transitive regular graphs. Moreover, the benefit from knowing that the vertices in this pivot set are equivalent is that algorithm *Match* will not backtrack at this point. However, that has the cost of testing every vertex in this pivot set for equivalence with the pivot vertex previously chosen. Not doing so here leaves this work for *Match*, that, probably, will not need so much work to find the equivalent sequence of partitions (if it exists). If these vertices are equivalent and the graphs are isomorphic, *Match* will not need to backtrack at this point (any vertex chosen will work). If the vertices are not equivalent, we do not lose time checking that. In case the vertices are equivalent but graphs G and H are not isomorphic, no more work will be done by *Match* than it would be done by *SearchAutomorphisms*. Therefore, testing for automorphisms is not performed at this point. Note that, if instead of comparing only two graphs G and H, we would be comparing G with many graphs $H_1, ..., H_k$, it would possibly be more convenient to search for equivalence among the vertices in every pivot set used for vertex refinement with more than one vertex.

The tests have been carried out on a Pentium III at 1.0 GHz with 256 MB of main memory under Linux RedHat 9.0. All the programs have been compiled with the same compiler and using the same optimizing options. The execution time considered was the real time (not CPU time) used by the programs once the graphs had been loaded in memory, thus skipping the overload of accessing a disk file. The CPU time limit for each program run was set to 1800 seconds. If a test could not finish in that period of time, it was considered to last 1800 seconds. We know that this may slightly change the results, but since our algorithm

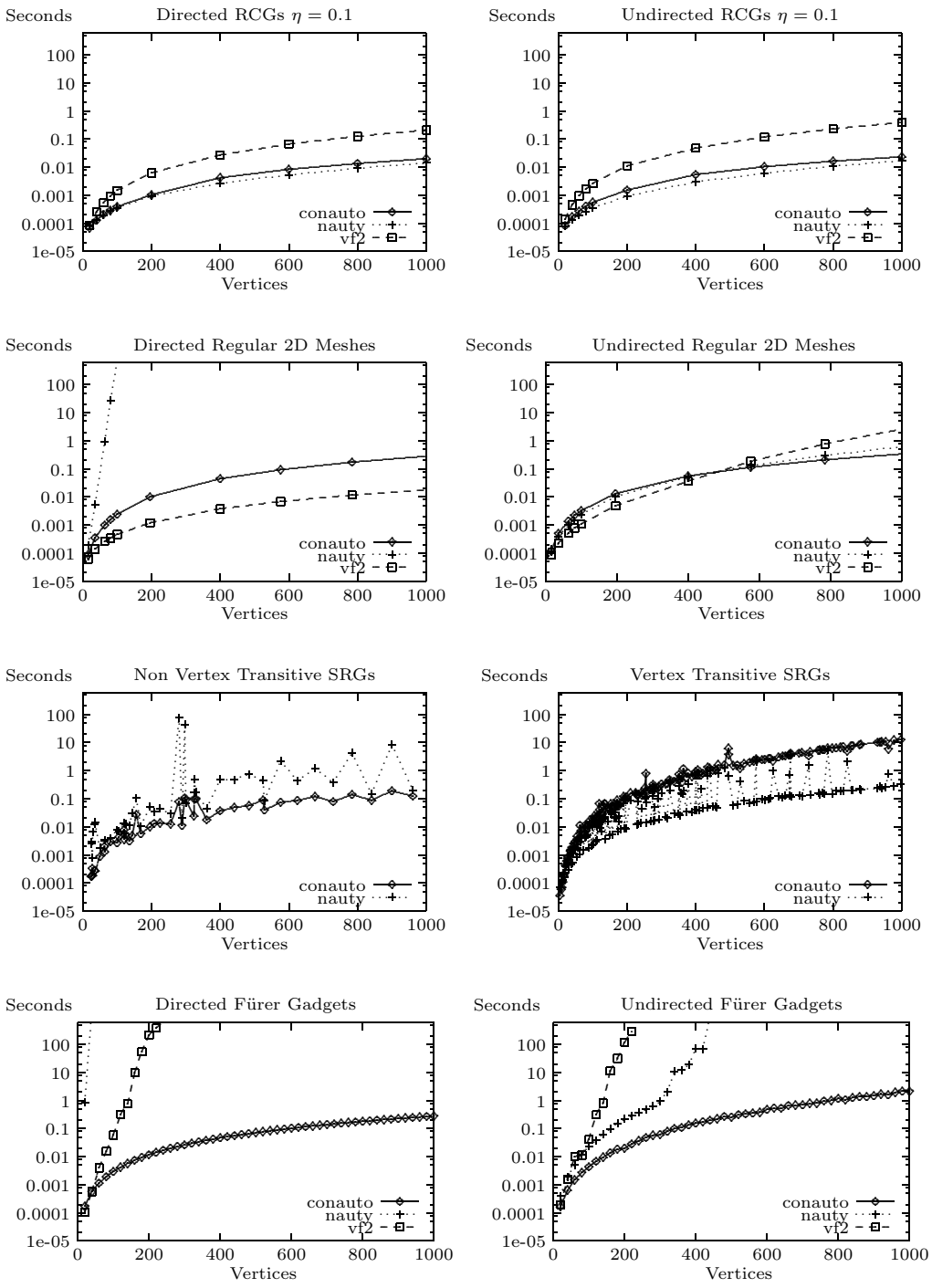

Figure 19: The performance of the three algorithms.

never reached 1800 seconds, it only overestimated the performance of the other two algorithms. For each type and size of the graphs considered, several different graphs or permutations of the same graph were tried and their average execution time computed.

We have restricted the tests, whose results are shown in Figure 19, to positive cases. Negative cases are usually more difficult for algorithms that rely on backtracking for finding the correspondence between the vertices of the graphs, while it is not important for algorithms that build a canonical form of the graphs. The study of negative cases has been left for future work due to the difficulty of generating hard negative cases. For our tests, we have chosen four types of graphs:

- Randomly connected graphs with $\eta = 0.1$ (RCGs) in directed and undirected versions. The directed version of these graphs has been developed by Foggia et al.[2] and has been obtained from [11]. The undirected version has been derived from this graphs, converting arcs into edges.
- 2D Meshes, in directed and undirected versions. The directed version is due also to Foggia et al. [2] and the undirected version has been derived directly from the directed version as in the case of randomly connected graphs.
- Strongly Regular Graphs (SRGs), classified in two categories: vertex transitive and not vertex transitive. These graphs were supplied by Sven Reichard and many of them are available at [10].
- Fürer gadgets, in directed and undirected versions. These graphs have been generated with a program by Takunari Miyazaki, slightly modified to generate both the directed and the undirected versions in the same format used in [2].

First, we must note that, for random graphs, the three programs have a good behavior. These are simple graphs and they are supposed to have probably no automorphisms. However, *nauty* and *conauto* are rather faster than *vf2*, which suggests that vertex classification and partition refinement are very helpful with this kind of graphs.

With regular 2D meshes, both *conauto* and *vf2* have similar behaviors, though *conauto* performs more uniformly for the directed and the undirected versions. *nauty*, though, has a very bad behavior with the directed version of the graphs, which was already known [3]. With the undirected versions its performance si similar to that of the other two algorithms. *nauty* seems to have problems to gather automorphisms of directed graphs or to take advantage of directed edges (arcs) in the process of partition refinement.

Strongly Regular Graphs (SRGs) have been split into two groups: vertex-transitive and non vertex-transitive. The automorphism group of a vertex-transitive graph is easier to compute and that seems to be the reason why *nauty* is faster for vertex-transitive SRGs than for non vertex-transitive ones. However, for these positive tests, *conauto* performs better for non vertex-transitive graphs. This is due to the fact that it does not compute the full automorphism group of the graphs. The curves also show that SRGs are an heterogeneous family of graphs. *vf2* has not been included in the SRGs curves for its very erratic behavior with these graphs. Note that *conauto* has a quite regular behavior in both cases.

Regarding Fürer gadgets, both in the directed and undirected versions, *conauto* behaves quite regularly, and noticeably better than the other two programs. In fact, it looks like the *conauto* behavior is polynomial, while that of the other two is exponential. Again, *nauty* behaves especially bad with the directed version of the graphs (it could not finish in 1800 seconds for the graphs of 40 vertices). This seems to be due to the difficulty it experiments to find the automorphisms of this family of graphs. However, the order in which *conauto* choses its pivot sets and the method it uses to look for automorphisms makes these graphs easily manageable. For each size, 20 permutations of the same graph were considered, and, while *conauto* performed quite uniformly with all the combinations, the other two found some permutations much harder than the others.

8 Final considerations

Our algorithm seems to work reasonably well with the graphs we have tried. However, it does not exploit yet all the power of automorphism discovery to prune the search tree. We believe that with a more sophisticated technique, it could detect more automorphisms, which would allow it to manage graphs with a regular

structure more efficiently. Some of the results presented in [6] could also be applied to our algorithm with little effort. The hardest graphs to deal with seem to be those regular but not vertex-transitive. For these graphs, a more sophisticated partitioning technique could help to distinguishing non equivalent vertices, thus helping to determine the orbits of the graphs.

References

[1] L.P. Cordella, P. Foggia, C. Sansone, and M. Vento. An improved algorithm for matching large graphs. In *Proceedings of the 3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition*, Ischia (Italy), May 2001.

[2] P. Foggia, C. Sansone, and M.Vento. A database of graphs for isomorphism and sub-graph isomorphism benchmarkin. In *Proceedings of the 3rd IAPR-TC15 Workshop on Graph-based Representationsin Pattern Recognition*, Ischia (Italy), May 2001.

[3] P. Foggia, C. Sansone, and M. Vento. A performance comparison of five algorithms for graph isomorphism. In *Proceedings of the 3rd IAPR-TC15 Workshop on Graph-based Representationsin Pattern Recognition*, Ischia (Italy), May 2001.

[4] Martin Fürer. A counterexample in graph isomorphism testing. Tech. Rep. CS-87-36, Department of Computer Science, The Pennsylvania State University, University Park, Penna., 1987.

[5] Donald L. Kreher and Douglas R. Stinson. *Combinatorial Algorithms: Generation, enumeration and search.* CRC Press, 1999.

[6] Brendan D. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981.

[7] Brendan D. McKay. nauty user's guide (version 1.5). Technical report, Computer Science Department, Australian National University, 1990.

[8] Brendan D. McKay. The nauty page, March 2004. `http://cs.anu.edu.au/~bdm/nauty/`.

[9] Takunari Miyazaki. The complexity of McKay's canonical labelling algorithm. *Groups and Computation II*, 28, 1996.

[10] Sven Reichard. Strongly regular graphs, May 2000. `http://www.math.udel.edu/~reichard/srg_new/index.html`.

[11] SIVALab. The graph database, May 2003. `http://amalfi.dis.unina.it/graph`.

[12] G. Tinhofer and M. Klin. Algebraic combinatorics in mathematical chemistry iii. graph invariants and stabilization methods, 1999.

[13] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM*, 23(1):31–42, January 1976.

[14] B. Weisfeiler, editor. *On Construction and Identification of Graphs*, volume 558 of *Lecture Notes in Math.* Springer, Berlin, 1976.