# The Labelled Cycle-Decomposition Trees of a Connected Graph

Ortho Flint and Stuart Rankin

September 22, 2009

**Abstract**

The notion of a labelled cycle-decomposition tree for an arbitrary graph is introduced in this paper. The idea behind the labelled cycle-decomposition tree, one constructed for each vertex in the graph, is to attach to each vertex a data structure that gives more than local information about the vertex, where the data structures can be compared in polynomial time. The fact that trees can be compared in linear time, with particularly efficient algorithms for solving the isomorphism problem for rooted and labelled trees, led us to consider how we could represent the vertex's view of the graph by means of a labelled rooted tree. Of course, cycles in the graph can't be directly recorded by means of a tree strucure, but in this article, we present one method for recording certain of the cycles that are encountered during a breadth-first search from the vertex in question. The collection of labelled, so-called cycle-decomposition trees for the graph, one for each vertex, provides an invariant of the graph.

## 1 Introduction

The existence of a polynomial time algorithm for determining graph isomorphism is still an open problem. A striking result, due to Laszlo Babai (see [2]) established a simple vertex classification algorithm that would in linear time produce a canonical labeling of an $n$-vertex random graph with probability $1 - e^{(-O(n))}$. This result is considered to explain why many graph isomorphism algorithms behave well in practice.

Most graph isomorphism algorithms employ the technique of graph canonization (see [1], [2], [5], [6] for example), whereby one attempts to construct for each graph a canonical representation of it such that two graphs are isomorphic if and only if their canonical representations are identical. At the present time, B. McKay's algorithm `nauty` ([5], [6]) is widely considered to be the preminent graph isomorphism algorithm, and it is based on graph canonization. As described in McKay's work, it is possible to provide some initial data for the graph that can greatly reduce the amount of work required to construct a canonical representation. One might still hope that it is will be possible to find

some such data that would lead to polynomial time construction of a canonical representation, and this was one of the motivations for our development of the labelled cycle-decomposition trees for a graph. However, it is quite likely that the labelled cycle-decomposition trees will be able to play an important role is the related study of graph similarity. One such application would be in data mining, for example in chemical database applications (see [3]).

The idea behind the labelled cycle-decomposition tree, one constructed for each vertex in the graph, was to try to attach to each vertex a data structure that gave more than local information about the vertex, where the data structures could be compared in polynomial time. The fact that trees can be compared in linear time (see [4]), with particularly efficient algorithms for solving the isomorphism problem for rooted and labelled trees led us to consider how we could represent the vertex's view of the graph by means of a labelled rooted tree. Of course, cycles in the graph can't be directly recorded by means of a tree strucure, and the purpose of this article is to present one method for recording certain of the cycles that are encountered during a breadth-first search from the vertex in question. The collection of labelled, so-called cycle-decomposition trees for the graph, one for each vertex, provides an invariant of the graph. While it is unlikely that this is a complete invariant, at this time we do not have any examples of non-isomorphic graphs with identical sets of labelled cycle-decomposition trees. However, we do finish up with a study of an interesting example, due to R. Mathon [7], of a graph with trivial automorphism group on 50 vertices for which the 50 labelled cycle-decomposition trees are of exactly two kinds.

## 2 Constructing the labelled trees

In this section, we describe an algorithm for decomposing an arbitrary connected graph into a labelled rooted tree. The root may be selected to be any vertex, and the resulting labelled rooted tree provides a view of the graph from the perspective of the selected vertex.

The assembly of the labels will make use of the following result, the proof of which is straightforward.

**Proposition 2.1** *The relation $R$ on $\mathbb{Z} \times \mathbb{Z}$ defined by $(m,n)\,R\,(r,s)$ if one of the following holds:*

*(i)* $m + \lfloor \frac{n}{2} \rfloor < r + \lfloor \frac{s}{2} \rfloor$;

*(ii)* $m + \lfloor \frac{n}{2} \rfloor = r + \lfloor \frac{s}{2} \rfloor$ *and* $m + n < r + s$;

*(iii)* $m + \lfloor \frac{n}{2} \rfloor = r + \lfloor \frac{s}{2} \rfloor$, $m + n = r + s$, *and* $m \geq r$

*is a total order relation.*

Let $V$ denote the set of vertices of the graph $G$, and let $n = |V|$. Name the vertices 1 to $n$ in an arbitrary way. Create an array $L$ of size $n$, whose entries are vertex label data structures, as described next.

| | |
|---|---|
| `parent`: | a sequence of integers from 1 to $n$ (at initialization, $L[i]$.`parent` := `null`). |
| `depth`: | an integer from 0 to $n-1$ (used to store the depth of the vertex as set by the breadth-first search, initially null). |
| `adj_list`: | a sequence of positive integers, initialized using the data from $G$. |
| `labels`: | a sequence of label records, initally `null`. |
| `tag`: | an integer, initially `null`. |

During the construction of the labelled tree, new vertices may be added to the graph, and each new vertex is named with the next available positive integer. However, each newly created vertex, say $j$, is linked by the construction to exactly one of the original $n$ vertices, say $i$, and the `parent` field in $L[j]$ is set to $i$, the `depth` field is set to the current value of `depth`, and the `label` sequence will be set. This causes $j$ to become a terminal node in the current tree.

In general, the construction procedure will gradually convert more and more of the original graph's edges into directed edges pointing back to the root, breaking cycles as they are encountered. For each vertex, there will be an appropriate time at which the `parent` field will be set to point to the immediate predecessor in the unique path from the vertex in question back to the root. It remains a null pointer until it gets set (with the possible exception of a situation that may be encountered in Step 2, but in such a case, this will be rectified in Step 3).

To begin with, select a vertex to serve as the root of the labelled tree that we are to build. We shall let $R$ denote the root vertex. If $R$ has name $i$, then set $L[i]$.`parent` := $i$ (to indicate that this vertex is the root), and set $L[i]$.`depth` := 0. Initialize a counter `depth` to 1 (this will hold the depth of the vertices that are going to be considered next; namely those attached to the vertices of the current frontier $F$). Set $F := \{ R \}$ (this will be the set of vertices from which we will extend a breadth-first search). Initialize a counter `tag_counter` to 1. This counter will hold the next available tag value to be used to set the `tag` field of a vertex. The `tag` field is not part of the label, but is kept to allow reconstruction of the graph from the full tree data structure. Several vertices may be assigned the same `tag` value, and one might incorporate the information as to which vertices had the same `tag` value, without actually having the value itself be part of the label. This would still be a graph invariant, but at the present time, we have not extended the labels to contain this information.

We remark that at each step, the current tree consists of those $L[i]$ with non-null `depth` field, initally just the root vertex.

While $F \neq \varnothing$, perform the following procedure.

**Begin**

Let $T$ denote the set of vertices in $G$ that have not yet been assigned a breadth-first search depth from $R$ but are adjacent to at least one vertex in $F$ (the elements of $T$ are found by examining the adjacency list of each vertex in $F$, selecting those vertices on the adjacency list that have not yet had their `depth` field set).

**While** $T \neq \varnothing$, perform Step 1, then Step 2.

**Step 1.** Initialize $E$ to be the set of edges in the induced subgraph $G[T]$. While $E \neq \varnothing$, perform the following actions. Choose $e \in E$ and replace $E$ by $E - \{ e \}$. Let $i$, $j$ denote $e$'s endpoints ($e$ could be a loop, in which case $i = j$). Remove $i$ from $L[j].\texttt{adj\_list}$ and remove $j$ from $L[i].\texttt{adj\_list}$. Create two new vertices, so if we currently have $s$ vertices, name the new vertices $s+1$ and $s+2$. Set

$L[s+1].\texttt{parent} := i$, $L[s+2].\texttt{parent} := j$,
$L[s+1].\texttt{depth} := \texttt{depth}$, $L[s+2].\texttt{depth} := \texttt{depth}$,
$L[s+1].\texttt{adj\_list} := i$, $L[s+2].\texttt{adj\_list} := j$
$L[s+1].\texttt{tag} := \texttt{tag\_counter}$, $L[s+2].\texttt{tag} := \texttt{tag\_counter}$,

and increment `tag_counter`. Then append $s+1$ to $L[i].\texttt{adj\_list}$ and $s+2$ to $L[j].\texttt{adj\_list}$. If $e$ is a loop, then set $L[s+1].\texttt{labels}$ and $L[s+2].\texttt{labels}$ equal to the list whose only entry is $(\texttt{depth}, 1)$. Otherwise, create a temporary linear array $M$ of label records, initially null, and append entries determined as follows. Construct an array $P_i$, respectively $P_j$ containing the distinct paths from $i$, respectively $j$, which pass first through a vertex of $F$ and thereafter visit only vertices whose breadth-first depth has been set (the tree as constructed so far). Note that all of these paths have the same (weighted) length `depth`. Let the number of paths in $P_i$ be $m$ and the number of paths in $P_j$ be $n$. Then perform the $mn$ path comparisons, wherein each path in $P_i$ is compared to each path in $P_j$ in order to identify the first vertex in common to the two paths. Since all paths terminate at $R$, such a vertex exists. Suppose that we are comparing path $P_i[r]$ to path $P_j[s]$. Let $k$ denote the first vertex in common to $P_i[r]$ and $P_j[s]$ on the traversal from $i$ to $R$, respectively from $j$ to $R$. Then append the ordered pair $(L[k].\texttt{depth}, 2(\texttt{depth} - L[k].\texttt{depth}) + 1)$ to $M$. The first entry in the pair is the depth of attachment of the cycle we are breaking, and the second entry is the size of the cycle.

Once all $mn$ pairs have been processed, sort $M$ in ascending order according to the total order as described in Proposition 2.1, deleting duplicates. Then set $L[s+1].\texttt{labels} := M$ and $L[s+2].\texttt{labels} := M$.

Interpretation: the new vertex $s+1$, respectively $s+2$, has unique path to $R$ passing through $i$, respectively $j$, and then through vertices in the current tree. The fact that the `depth` field has been set means that $s+1$ and $s+2$ have been added to the current tree, and in recognition of the fact that each is a terminal vertex, neither has bee added to the frontier for the next iteration. Each cycle through the edge $e$ that has $u + \lfloor \frac{v}{2} \rfloor = \texttt{depth}$, where $u$ is the vertex on the cycle that is closest to $R$, and $v$ is the cycle size, has been accounted for with an entry in the `labels` list.

Step 1 is completed when $E = \varnothing$. At this point, we are ready to begin Step 2. $G$ has been modified during Step 1 so that any edge that joined two vertices in $T$ has been removed and replaced by two new edges, each with one endpoint a vertex in $T$ but the other a new vertex (which is already in the current tree). No path from a vertex in $T$ to $R$ for which the first edge leads to a vertex in $F$ has been modified. Furthermore, neither $F$ nor $T$ has not been changed during Step 1.

**Step 2.** Let $H$ denote the subgraph of $G$ whose vertex set is $F \cup T$ and whose

edge set consists of those edges of $G$ with one endpoint in $F$ and the other endpoint in $T$. For each vertex $i$ of $T$ with $\deg_H(i) = 1$ (there may be no such vertices, but this is where all vertices that were created in Step 1 of the previous iteration will be processed), set $L[i].\texttt{depth} := \texttt{depth}$ and $L[i].\texttt{parent}$ equal to the unique vertex in $F$ that is adjacent to $i$ in $H$. Next, process each vertex of $T$ for which $\deg_H(i) > 1$ (there may be no such vertices).

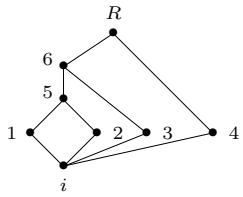
For each $i$ in $T$ with $\deg_H(i) > 1$, perform the actions described in this paragraph. Let $m = \deg_H(i)$ and suppose that $i_1, i_2, \ldots, i_m$ are the vertices of $F$ that are adjacent to $i$. For each $j = 1, \ldots, m$, there is a unique path in $G$ from $i_j$ to $R$ that passes through vertices that have had their `depth` field set, and this path necessarily has (weighted) length $\texttt{depth} - 1$. Create a temporary array $P$ of size $m$, such that for each $j$, $P[j]$ contains the path in $G$ from $i_j$ to $R$ that passes through only vertices that have had their `depth` field set. We shall treat $P[j]$ itself as an array of size at most $\texttt{depth} - 1$ whose entries are the names of the vertices on the path (due to our construction, it is possible that paths might contain edges that are weighted greater than 1). Note that $P[j][1] = i_j$. Create a temporary array $S$ of size $m$, with each entry initially set to `null`. For an index $j$, $S[j]$ will be a list of pairs of non-negative integers of the form $(L[z].\texttt{depth}, z)$, and the $k^{th}$ entry (indexing from 1) in the list $S[j]$ will be accessed as $S[j][k].\texttt{depth}$ and $S[j][k].\texttt{vertex}$, respectively. Each list $S[j]$ is sorted first in nonincreasing order with respect to the first entry (the depth value), then in increasing order with respect to their second entry (the vertex label $z$). For each of the $\binom{m}{2}$ subsets $\{ r, s \}$ of $\{ 1, 2, \ldots, m \}$, determine the first vertex $z$ to be encountered in common to the two paths $P[r]$ and $P[s]$ on their way back to $R$. Insert the pair $(L[z].\texttt{depth}, z)$ so as to maintain the order described above into each of the two lists $S[r]$ and $S[s]$. When all pairs of paths have been examined, sort $S$ and $P$ by the requirement that $S[r]$ ($P[r]$) precedes $S[s]$ ($P[s]$) if $S[r][1].\texttt{depth} > S[s][1].\texttt{depth}$ or if $S[r][1].\texttt{depth} = S[s][1].\texttt{depth}$ and $S[r][1].\texttt{vertex} < S[s][1].\texttt{vertex}$. This will arrange it so that all paths from $i$ back to $R$ that meet at a deepest vertex will come first, and of those, the ones for which this deepest vertex is least of all such will come first (most importantly, all paths with the same deepest vertex in common will be contiguous in the sort order). Now process $P$ in order first to last. Let $j = S[1][1].\texttt{vertex}$, and suppose that the first $t$ entries (and no more) in the array $P$ have first common vertex on the way back to $R$ equal to $j$. If $t = 1$, remove $P[1]$ from the $P$ array and save in a new temporary array $P'$, and remove $S[1]$ from the $S$ array, and save in a new temporary array $S'$. On the other hand, if $t > 1$, then process the $t$ paths as described next. Let $i_{j_1}, i_{j_2}, \ldots, i_{j_t}$ denote the initial vertices of these $t$ paths. Remove $i$ from the adjacency list of each of these $t$ vertices, and at the same time, remove each of these vertices from the adjacency list of $i$ (which may cause $i$ to become isolated, or it may cause the number of components in the graph to increase). Insert the label $(L[j].\texttt{depth}, 2(\texttt{depth} - L[j].\texttt{depth}))$ at the appropriate place in the sequence $L[i].\texttt{labels}$ so as to maintain the order according to Proposition 2.1. The next step requires a bit of explanation. It is intended to retain knowledge of the deepest points of attachment of the even cycles that are being

broken during the processing of vertex $i$ at this depth. If $L[i].\texttt{parent} = \texttt{null}$, then set $L[i].\texttt{parent} := j$. If $L[i].\texttt{parent} \neq \texttt{null}$ (so that it has been processed earlier in this step, or at an earlier time), then we consult the last entry $(r, s)$ in the sequence $L[i].\texttt{labels}$. If $(L[j].\texttt{depth}, 2(\texttt{depth} - L[j].\texttt{depth})) = (r, s)$, then append $j$ to $L[i].\texttt{parent}$ (this is how the `parent` field can come to contain a sequence of integers rather than a single integer–see Step 3 for a discussion of how this sequence will eventually be turned back into a single integer entry). If $L[i].\texttt{tag} = \texttt{null}$, then set $L[i].\texttt{tag} := \texttt{tag\_counter}$ and increment `tag_counter` (we only assign a tag once, and the same tag value is then given to each vertex that is created from this one, whether at this stage or a later stage). Create $t$ new vertices, and if we currently have $s$ vertices, then name the new vertices $s + 1, \ldots, s + t$. For each integer $k$ with $1 \leq k \leq t$, initialize $L[s + k].\texttt{parent} := i_{j_k}$, $L[s + k].\texttt{depth} := \texttt{depth}$, $L[s + k].\texttt{adj\_list} := i_{j_k}$, $L[s + k].\texttt{labels} := L[i].\texttt{labels}$, $L[s + k].\texttt{tag} := L[i].\texttt{tag}$, and adjoin $s + k$ to $L[i_{j_k}].\texttt{adj\_list}$. Finally, remove $P[1], P[2], \ldots, P[t]$ from $P$ and remove the corresponding entries from $S$. It is intended in this description that removing the entries from $P$ causes every entry index to be reduced by $t$, so the next path to process is now $P[1]$. Similarly, removing the corresponding entries from $S$ causes the indicies of the following entries to be reduced by $t$. Repeat this process until either $P$ is completely used up or else has one path left. If the temporary array $P'$ is null, then either $P$ finished up with one path, in which case we set $L[i].\texttt{depth} := \texttt{depth}$ and $L[i].\texttt{parent} := j$, where $j$ is the vertex adjacent to $i$ on the unique path left in $P$, otherwise we remove $i$ from $T$. On the other hand, if the temporary array $P'$ is not null, then there is more processing to do. Either $P$ was not completely used up, so it has one path left, and we then adjoin this last path of $P$ to $P'$ and adjoin the last entry of $S$ to $S'$ and then set $P := P'$ and $S := S'$, otherwise $P$ was completely used up and we set $P := P'$ and $S := S'$. The $S$ array and the $P$ array are in sorted order.

We now iterate the following procedure until $P$ is either empty or consists of a single path Remove the first entry from $S[1]$, and then move the modified $S[1]$ to the first possible correct position in $S$ according to the sort order described above, and suppose that its new index is $k$. Move $P[1]$ to the corresponding position in $P$. If $j = S[k][1].\texttt{parent} = S[k + 1][1].\texttt{parent}$, then process as described above ($t = 2$ in all of this processing, and when the processing of these two is finished, their entries will be removed from $P$ and $S$). Eventually, $P$ will either be empty or consist of a single path. If $P$ finished up with one path, then set $L[i].\texttt{depth} := \texttt{depth}$ and $L[i].\texttt{parent} := j$, where $j$ is the vertex adjacent to $i$ on the unique path left in $P$, otherwise remove $i$ from $T$. See Figure 1 for an illustration of such a situation.

Finally, set $F := T$ and increment `depth`. This completes Step 2.

At the completion of Step 2, the subgraph of $G$ that is induced by the vertices whose `depth` field has been set is a tree. Furthermore, every vertex that was in $T$ at the beginning of Step 2 has been processed. It is possible that $T$ has had some vertices removed during Step 2, and in fact, $T$ will eventuall be empty. When $T = \varnothing$, the algorithm will exit from the **While** loop, and will continue




| j | P[j] | S[j] |
|---|---|---|
| 1 | 1,5,6,7 | (2,5),(1,6),(0,7) |
| 2 | 2,5,6,7 | (2,5),(1,6),(0,7) |
| 3 | 3,6,7 | (1,6),(0,7) $\rightarrow P'$ |
| 4 | 4,7 | (0,7) |

| j | P[j] | S[j] |
|---|---|---|
| 1 | 3,6,7 | (0,7) |
| 2 | 4,7 | (0,7) |

Figure 1:

with Step 3. At this point, the component of $G$ that contains $R$ is a tree. We also point out that every vertex not in the original graph was created at some stage as a terminal node of the tree at depth no greater than $|V| - 1$, so after at most $|V| - 1$ iterations of Step 1 and Step 2, we will reach a stage where $T = \varnothing$. However, even though we had started with a connected graph, the various cycle decompositions that have been carried out may have resulted in a graph of several components, and we will only have a spanning tree for the component that contains the root vertex. Step 3 will allow us to deal with the other components.

**Step 3**. Set $F := \varnothing$, and form the set $X$ of all vertices of the original graph for which the `depth` field is still `null`.

If $X = \varnothing$, then we are done, so we return to the top, at which point the condition $F = \varnothing$ is true and the process outputs $L$ and terminates. The graph that has been constructed provides us with the labelled cycle decomposition tree for $G$ with root $R$. Otherwise, $X \neq \varnothing$. Of the vertices in $X$, at least one must have a non-null label sequence with entries having even second coordinate. Remove all vertices from $X$ that have either a null label sequence or else a non-null label sequence in which the first entry has odd second coordinate (in which case, all entries in the label sequence have odd second coordinate). Next, form the set

$$Y = \{\, (m,n) \mid \text{there exists } i \in X \text{ for which } (m,n) \text{ is the first entry in the sequence } L[i].\texttt{labels} \,\}$$

and, using the total order introduced in Proposition 2.1, determine the smallest element in $Y$, say $(r,s)$. Finally, let

$$S = \{\, i \in X \mid \text{ the first entry in } L[i].\texttt{labels} \text{ is equal to } (r,s) \,\},$$

and set $\texttt{depth} := r + \left\lfloor \frac{s}{2} \right\rfloor$ (this sets the `depth` field to the correct depth in preparation for a return to Step 1).

For each $i \in S$, carry out the following actions for each $j$ in the sequence $L[i].\texttt{parent}$. Set $F := F \cup \{\, j \,\}$, append $j$ to $L[i].\texttt{adj\_list}$ (we are in effect attaching $i$ by a new edge, of weight greater than 1, to the point of attachment of the cycle which was broken when $i$ was first processed, which was recorded in the `parent` field of $i$ at that time), and append $i$ to $L[j].\texttt{adj\_list}$. Then set $L[i].\texttt{parent} := \texttt{null}$.

**End**.

Eventually, Step 3 will result in $X = \varnothing$ which will cause control to return to the top with $F = \varnothing$, thereby causing the termination of the algorithm. At that point, $L$ is the labelled cycle decomposition tree for the original graph.

We remark that during the course of execution of the algorithm for a particular vertex $v$, each new vertex that is created will be incorporated into the current tree immediately upon creation, while any vertex that is an endpoint of an edge being processed by Step 1 but that is not the midpoint of an even cycle (with reference to the point of attachment to $v$); that is, will be added to the current tree at the beginning of Step 2 of the current iteration. Vertices of the original graph that do not end up being added to the current tree during their processing in Step 2 will be processed again during Step 2 of a later iteration due to Step 3. However, if such a vertex is reattached during Step 3, its degree upon reattachment will be less than its degree at the time of its previous processing. Thus the number of times that a given vertex can get repeatedly reattached by Step 3 is at most the degree of the vertex, and so eventually it will be incorporated into the current tree. Thus any given vertex will be incorporated into the current tree in a number of iterations equal at most to its degree in the original graph, and so it follows that the algorithm will terminate after at most $|V|^2$ iterations.

# 3 The labelled cycle decomposition trees do not determine the automorphism group orbits

While the labelled cycle decomposition trees do in general serve to distinguish vertices that are in different orbits of the graph's automorphism group, they are not infallible in this regard. However, they can be used to provide an initial vertex colouring for the application of canonical labelling algorithms such as `nauty` [McKay]. We also expect that the contruction of full or partial labelled cycle-decomposition trees will provide a useful tool for similarity testing [chemists].

The graph $A_{50}$, shown in Figure 2 and derived by Mathon (see [Ma]) from a Steiner triple system BIBD(15,31), has trivial automorphism group. It is a bipartite graph with 35 vertices of degree 3 and 15 vertices of degree 7.

All 15 vertices of degree 7 have the same labelled cycle-decomposition tree. The tree structure is shown below, where each vertex has null label except for the terminal vertices (84 of them), each of which has label $(0, 6)$. In the diagram, $R$ denotes the root vertex. Each of the seven children of $R$, labelled as $v_1$ through $v_7$ in the diagram, has the subtree shown below at the right. As well, there were 28 vertices of degree 3 that were decomposed in the creation of the $(0, 6)$ labels, and so there will be 28 new vertices attached to $R$ by an edge of weight 3, and these have not been shown.

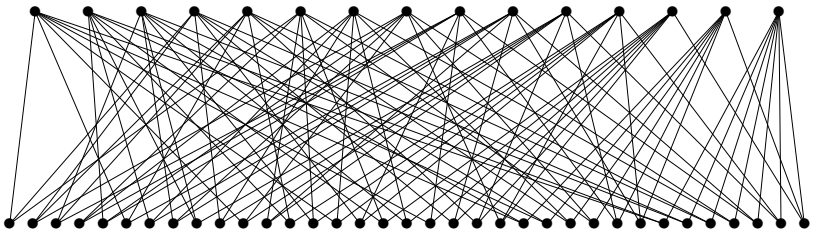
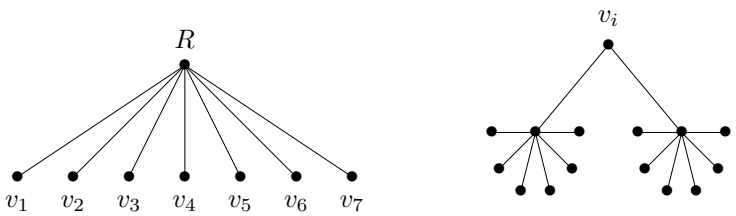
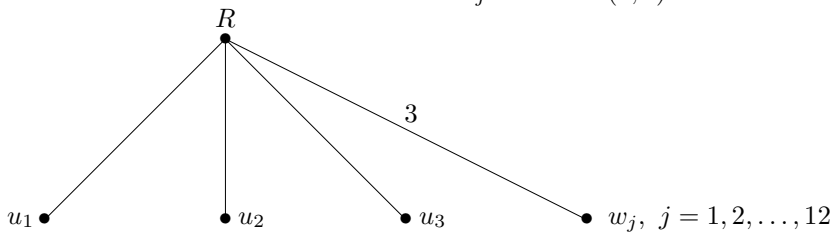
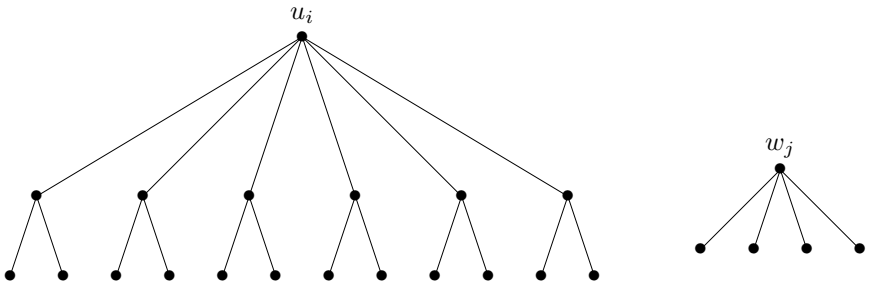


Figure 2: $A_{50}$




As for the 35 vertices of degree 3, all have identical labelled cycle-decomposition trees, with structure as described below. Let $R$ denote any vertex of degree 3. Then in the labelled cycle-decomposition tree for $R$, $R$ will have three children labelled $u_1$, $u_2$ and $u_3$, shown below, twelve children labelled $w_1$ through $w_{12}$, represented by $w_j$ on the diagram below and connected to $R$ by an edge of weight 3, and 16 terminal vertices (not shown on the diagram below), each connected to $R$ by an edge of weight 4. The subtrees connected to $u_1$, $u_2$, and $u_3$ are identical, each equal to the tree shown as rooted at $u_i$ in the diagram below. As well, there are twelve children of $R$ joined to $R$ by an edge of weight 3, and each of these has subtree as shown below at right, marked as rooted at $w_j$. Each of the terminal vertices on the three subtrees with root $u_i$, $i = 1, 2, 3$ has label $(0, 6)$, and for $j = 1, 2, \ldots, 12$, $w_j$ has label $(0, 6)$, while each of the terminal vertices on the subtree with root $w_j$ has label $(0, 8)$.






# References

[1] L. Babai and E. M. Luks, *Canonical labeling of graphs*, Proc. 15th ACM Symposium on Theory of Computing, 1983, 1719–183.

[2] L. Babai and L. Kucera, *Canonical labeling of graphs in average linear time*, Proc. 20th Annual IEEE Symposium on Foundations of Computer Science, 1979, 39–46.

[3] D. J. Cook and L. B. Holder, *Mining Graph Data, ISBN: 978-0-471-73190-0, Section 6.2.1, Canonical Labeling*, Wiley-Interscience, 2007, 119–121.

[4] J. E. Hopcroft and J. K. Wong, *Linear time algorithm for isomorphism of planar graphs (Preliminary Report)*, Proceedings of the sixth annual ACM symposium on Theory of Computing, Seattle, 1974, 172–184.

[5] B. D. McKay, *Practical graph isomorphism*, **30**, Congressus Numerantium 10th. Manitoba Conference on Numerical Mathematics and Computing (Winnipeg, 1980), 1981, 45–87.

[6] B. D. McKay,*Computing automorphisms and canonical labellings of graphs*, Combinatorial Mathematics, Lecture Notes in Mathematics, 686 (Springer-Verlag, Berlin, 1978), 223-232.

[7] R. Mathon, *Sample Graphs for Isomorphism Testing*, **21**, Congressus Numerantium, 1978, 499-517.