

Methods for General and Special Tree Generation and their Graceful Labelings*

Soohyuck Chung
chungsrpi.edu

Szekit Ka
kasrpi.edu

Joshua Taylor
taylojrpi.edu

Benjamin Wiegner
wiegnbrpi.edu

April 28, 2005

Abstract

We discuss four algorithms and their implementation. We present an algorithm for each of the following: the generation of general unique trees, the generation of a form of ‘special’ trees, gracefully labeling the ‘special’ trees, and gracefully labeling the general unique trees. In an appendix, we provide a brief guide to interaction with our software.

1 Representation and Utilities

We first describe a basic infrastructure for construction and modification of graphs. We will not go into the lowest implementation details, as they are unnecessary at this time, but basic constructs will provide a solid understanding in the algorithms to come.

1.1 Tree Representation

At the most basic level, we represent trees as an adjacency matrix, a tree with n vertices indexing these vertices as $0, 1, 2, \dots, n-2, n-1$. A disadvantage to this approach is that trees which are isomorphic may not have the same adjacency matrix due to the ordering of vertices. For instance, a line on 3 vertices can appear as both of the following:

$$\begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Nonetheless, we use this representation internally and address this issue later.

Consistent with usual convention, we will use $V(G)$ to denote the set of vertices in G , and $E(G)$ the set of edges in G .

*PDF and HTML available at http://www.cs.rpi.edu/~tayloj/GRAPH_THEORY/

1.2 Edgeless Graphs

For the sake of convenience, we define $Edgeless(n)$ as the $n \times n$ matrix in which every element is zero. In terms of a graph, this means a graph of n vertices with no edges.

1.3 Adjacency

Perhaps the most trivial operation that can be performed on an adjacency matrix is to make two vertices adjacent to each other. Of course, performing this operation on a tree would create cycles if the vertices were not already adjacent, so this operation is defined on graphs, not only trees. To make any vertices u and v adjacent in the graph, we set the coordinates (u, v) and (v, u) in the adjacency matrix to 1. An analogous operation holds for making vertices non-adjacent. As we deal solely with simple graphs, it is an error to try to make a vertex v adjacent with itself. We denote this operation $MakeAdjacent(u, v, G)$.

$$MakeAdjacent\left(0, 1, \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}\right) = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

1.4 Line Trees

One utility that is useful in certain applications is the ability to create ‘line trees’. That is a tree consisting of n vertices $(v_0, v_1, \dots, v_{n-1})$ in which vertex v_0 is adjacent to v_1 , v_{n-1} is adjacent to v_{n-2} and every other vertex v_k in the tree is adjacent to v_{k-1} and v_{k+1} . We will denote such a tree as $Line(n)$. (Note that any tree of the form $Line(n)$ is a path of length $n - 1$.) Of course, many adjacency matrices would correspond to ‘line trees’, but this method of generation yields adjacency matrices which have adjacencies in a ‘diagonal’, e.g.:

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & \dots \\ 1 & 0 & 1 & 0 & 0 & 0 & \dots \\ 0 & 1 & 0 & 1 & 0 & 0 & \dots \\ 0 & 0 & 1 & 0 & 1 & 0 & \dots \\ 0 & 0 & 0 & 1 & 0 & 1 & \dots \\ 0 & 0 & 0 & 0 & 1 & 0 & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

1.5 Augmentation

Another utility that is necessary is *augmentation*. As we deal exclusively with trees, and not general graphs, it must be the case that any two vertices in a tree have some path between them. The only way to introduce a vertex to the tree then is also add an edge from it to some existing vertex in the tree. We denote a tree formed by adding a new vertex and edge incident to an existing vertex v in tree T as $Augment(v, T)$. Then, as an example, consider $Line(3)$ and $Augment(1, Line(3))$.

$$Line(3) = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad Augment(1, Line(3)) = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

1.6 Graph Union

In dealing with graphs we often find it useful to consider the *union* of two graphs. Mathematically the transform is simple, but the concept is even more intuitive when viewing the adjacency matrix.

$$\begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \cup \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} = \left[\begin{array}{ccc|ccc} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{array} \right]$$

Note that we specifically called this *Graph Union* and not *Tree Union*. Neither of the input graphs are required to be trees, and because no vertices in two distinct input graphs become adjacent, the resulting graph is disjoint and, thus, not a tree. We will consider *Union* as being arbitrary arity, by defining $Union(G_1, G_2, \dots, G_{n-1}, G_n) = Union(G_1, Union(G_2, Union(\dots Union(G_{n-1}, G_n))))$. We will denote the union of two graphs G_1, G_2 as either $G_1 \cup G_2$ or $Union(G_1, G_2)$.

1.7 Graph Equality

As mentioned before, the adjacency matrix representation is insufficient to determine isomorphism, at least by simple element by element comparison. While there are algorithms to determine graph isomorphism, and perhaps even more efficient algorithms to determine tree isomorphism, development of such an algorithm was not the intent of this project. Rather than implement some algorithm ourselves, we turned to the `nauty` suite.

Using the command line tools and the simplified `dreadnaut` format, we can translate graphs to `dreadnaut` and then use the utility `dretog` to translate from `dreadnaut` to `graph6` format, which is used by many of the other utilities in `nauty`. Sending the `graph6` representation to `labelg` we retrieve the canonical label of a graph. Any graphs which are isomorphic share canonical labelings. Thus, by retrieving the canonical labeling of two graphs we can determine whether they are isomorphic, which we will henceforth denote $gequal(G_1, G_2)$.

1.8 Special Trees

Later we concern ourselves with the construction and labeling of special graphs. So, while they do not influence representation at this stage, we need to be able to refer to the ‘special’ trees unambiguously later. Our special trees are those

“obtained from k copies of paths of length t , by making one end of every path adjacent to an additional vertex z . thus the input to the program is comprised of two integers: k and t .”

We will denote a tree of this form as $Special(k,t)$. Two examples are shown in Figure 1. We define a predicate $Special?(G)$ which is true if and only if G is a special tree.

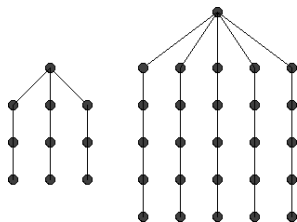


Figure 1: $Special(3,2)$ and $Special(5,4)$

2 Generation

Described are two methods of tree generation. The first will generate any special tree for a given k and t . The second will generate all trees of a given order n . Of course, the generation algorithm for general trees will generate special trees as well, however, the order of vertices within the adjacency matrix will differ for a special tree T generated by the first algorithm and the second.

2.1 Special Trees

Special trees are easy to generate, as they can be built from several smaller well defined graphs. Recall that a tree $Line(n)$ is a path of length $n - 1$, and that $Special(k,t)$ is a graph “obtained from k copies of paths of length t , by making one end of every path adjacent to an additional vertex z .” Then $Special(k,t)$ can be created from the union of a single vertex, and k copies of $Line(t + 1)$ (which we denote as L_1, L_2, \dots, L_k) with some adjacency operations.

Algorithm 1 $Special(k,t)$

Require: $k, t \geq 0$

Ensure: $S = Special(k,t)$

$S \leftarrow Union(Edgeless(1), L_1, L_2, \dots, L_k)$

for $i = 0$ below k **do**

$MakeAdjacent(0, 1 + i(t + 1), S)$

end for

2.2 General Trees

General trees are more difficult computationally, and are enumerated by systematic augmentation, and then removal of duplicates, the test of duplicity being *gequal*.

While general generation is computationally more difficult, it is conceptually simple, and easy to define in terms of several short algorithms.

Algorithm 2 *AugmentationsOfGraph(G)*

Require: G is a graph

Ensure: S is the set of unique augmentations to G

$S \leftarrow \{\}$ { S contains no duplicates, based on *gequal*}

for $v = 0$ below $|V(G)|$ **do**

 insert *Augment*(v, G) into S

end for

Algorithm 3 *AugmentSetOfGraphs(S)*

Require: S is a set of graphs (possibly empty)

Ensure: S' is the set of graphs which are all unique augmentation of graphs in S

$S' \leftarrow \{\}$ { S contains no duplicates, based on *gequal*}

for all $g \in S$ **do**

$S' \leftarrow S' \cup \text{AugmentationsOfGraph}(g)$

end for

With these two building blocks, it is now easy to generate all trees of any given order.

Algorithm 4 *Generate(n)*

Require: $n \geq 1$

Ensure: S is the set of all unique trees of order n

$S \leftarrow \{\}$

if $n = 1$ **then**

$S \leftarrow \{\text{Edgeless}(1)\}$

else

$S \leftarrow \text{AugmentSetOfGraphs}(\text{Generate}(n - 1))$

end if

3 Graceful Labeling

Describe labeling techniques for the different types of trees. We denote as *LabelOf*(v) the label applied to a vertex v , and *VertexLabels*(G) is the list of labels already used on vertices in G .

3.1 Special Labeling

Special trees generated by Algorithm 1 have the property that the vertex z , to which all paths have one endpoint adjacent, is always vertex 0 in the adjacent matrix. Furthermore, the endpoint of the first path adjacent to z is 1, the endpoint of the second is $(t+1)+1$, the third $(t+1)+(t+1)+1$, and so on. In fact, the vertices of the tree are placed in the adjacency matrix as shown in Figure 2.

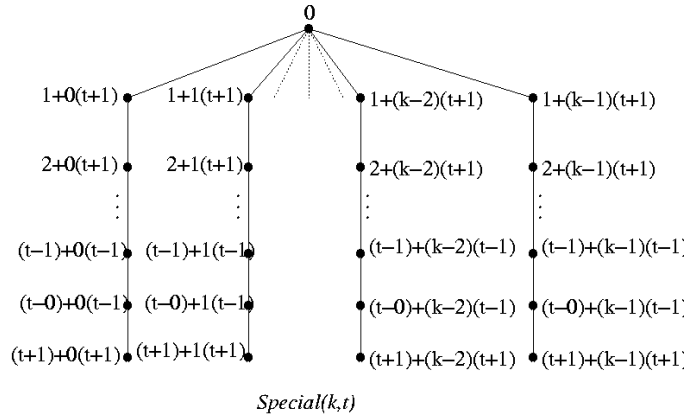


Figure 2: Structure of the vertices in a special tree

It may not immediately be clear from the labeling, but the order in which the labels are assigned begins with 0 at the 'root', and the rest proceed in the order shown in Figure 3.

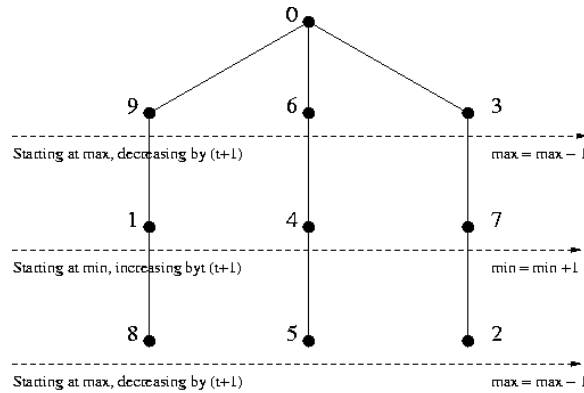


Figure 3: Applying the labels to $Special(3,2)$

Knowing that vertices in the special tree will be placed in the adjacency matrix in this way allows us to define a simple procedure to assign each vertex a label in such a way that the special tree has a graceful labeling after all assignments have been made. This labeling algorithm is efficient in that the labeling it produces is graceful, and no

‘backtracking’ or trial and error is necessary. We call this procedure *SpecialLabel(k,t)* which will assign the labels of vertices $0, 1, \dots, k(t+1)$ such that a tree with such vertices created by *Special(k,t)* is gracefully labeled.

In this algorithm, for simplification of indexing, we actually assign the labels to a matrix representing the vertices in the ‘grid’ that appears in the graphical representation of the tree. For instance, in Figure 3, the label matrix would end as

$$labels = \begin{bmatrix} 9 & 6 & 3 \\ 1 & 4 & 7 \\ 8 & 5 & 2 \end{bmatrix}$$

Algorithm 5 *SpecialLabel(k,t)*

```

counter ← 0
max ← k(t + 1)
min ← 1
Label(0) ← 0
labels ← new (t + 1) × k matrix
for r = 0 to t do
  if r is even then
    for val = max, c = 0 below k do
      labelsr,c ← val
      val ← val - (t + 1)
    end for
    max ← max - 1
  else
    for val = min, c = 0 below k do
      labelsr,c ← val
      val ← val + (t + 1)
    end for
    min ← min - 1
  end if
end for
Label(0) ← 0
for i = 0 below k do
  for j = 0 below (t + 1) do
    Label(counter) ← labelsi,j
    counter ← counter + 1
  end for
end for

```

3.2 General Labeling

The general trees are more difficult to label, but using the algorithm that we developed and refined, we are able to efficiently find graceful labelings for many trees. It is implemented as two procedures, *Label(G)* and *RecursiveLabel(G)*.

Algorithm 6 *Label*(G)

```
for all  $v \in V(G)$  do  
  label  $v$  with 0  
  if RecursiveLabel( $G$ ) returns a labeling then  
    return the labeling  
  else  
    unlabel  $v$   
  end if  
end for
```

Algorithm 7 *RecursiveLabel*(G)

```
for all  $p$  such that  $p$  is already labeled do  
  for all  $q$  such that  $q$  is adjacent to  $p$  and  $q$  is not yet labeled do  
    for all  $l$  such that  $l$  would be a label for  $q$  producing the edge with difference  
    MaxDiff do  
      if RecursiveLabel( $G$ ) returns a labeling then  
        return the labeling  
      else  
        unlabel  $q$   
      end if  
    end for  
  end for  
end for
```

4 Experiments and Results

Describe the sort of experiments we ran, and how we evaluated their results.

4.1 General Trees

How many unlabeled trees exist on n vertices is well studied, and some of this sequence is presented in Table 1, in which we also present how long¹ it took us to generate all the trees of order n . In the table, $G_w(n)$ and $G_s(n)$ denote the *wall clock* and *system* time used in generating all the trees of order n . $L_w(n)$ and $L_s(n)$ are subscripted similarly, and denote time to gracefully label all tree of order n . $T(n)$ is simply how many trees of order n exist.

n	$G_w(n)$	$G_s(n)$	$L_w(n)$	$L_s(n)$	$T(n)$
1	4.2e-5s	0.0s	5.8e-5s	0.0s	1
2	2.0e-5s	0.0s	9.4e-5s	0.0s	1
3	.028s	0.0s	1.0e-4s	0.0s	1
4	.038s	0.002s	2.99e-4s	0.0s	2
5	.123s	0.006s	6.27e-4s	0.001s	3
6	.196s	0.014s	0.002s	0.002s	6
7	.488s	0.035s	0.018s	0.013s	11
8	1.065s	0.098s	0.057s	0.057s	23
9	2.627s	0.318s	0.474s	0.443s	47
10	6.574s	1.198s	2.818s	2.449s	106
11	19.546s	6.005s	34.358	29.116s	235
12	1m6s	32.905s	2m31s	2m6s	551
13	4m54s	3m11s	44m6s	39m46s	1301
14	31m53s	21m51s	7h0m6s	6h14m35s	3159
15	3h49m11s	2h30m39s	?	?	7741
16	?	?	?	?	19320
17	?	?	?	?	48629
18	?	?	?	?	123867

Table 1: Unlabeled trees of order n , and time for their generation. $L_w(15)$ and $L_s(15)$ require at least 24 hours.

4.1.1 Generation

There are at least two interesting behaviors that should be noted. The first is that the generation algorithm $Generate(n)$ depends on the results of $Generate(n-1)$. The second is that we cache these results. That is, while $Generate(12)$ may require 1m6s to run the *first* time, if we make the query again, the answer has been cached, and is returned immediately.

¹These data were produced on a machine running CLISP 2.33.2 on Debian GNU Linux with a 3 GHz Intel chip.

In the experiment, we ran *Generate*(n) on increasing values of n , without clearing the cache. Furthermore, it means that when *Generate*(12) made a call to *Generate*(11), those results were cached, and returned immediately. Therefore, if no cache were present, then $G_w(n)$ would be (approximately) $\sum_{i=1}^n G_w(i)$ as $G_w(n)$ now appears in the table, and similarly for $G_s(n)$. This difference is significant.

There is some reason to believe that the times required for labelings are influenced more strongly by certain trees than by others. Certainly the time to label *all* the trees must increase as there are more trees to label, but by examining the labelings, we can show that certain trees require significantly more time than others.

4.1.2 Labeling

Certain trees took much longer to label than other trees. While we do not have a formal description of how to determine which trees take more time than others, the labeling itself which is produced by *Label*(G) carries some indication of how long it took to label.

Because the the vertices are labeled in order, and attempts are made to assign vertex labels in order, we can make basic inferences. For instance, if *Label*(G) returns (0 3 1 2) for some given G , we know that the assignment $Label(0) \leftarrow 0$ took place as the first assignment, and was never retracted. Vertex 1 must have been adjacent (as 0 and 3 must be adjacent in a 4 vertex graceful labeling), so the assignment $Label(1) \leftarrow 3$ was made immediately, and never retracted. These basic inferences give us some idea as to how much backtracking was performed during labeling.

As an example, there are 106 unique trees on 10 vertices. Of the labelings that our algorithm found for these trees, 82 begin (0,9,1,8,...). 96 begin with (0,9,1..). In fact, only 4 did not begin with zero. One machine used for development, labeling all 106 trees took 2m50s. These 4 graphs whose labels did not begin with 0 required 37s to label. The other 102 graphs then took, on average, only 1.3s to label. Certainly a few tree take much longer to label than others.

4.2 Special Trees

We did not stress-test the special tree algorithm insofar as running it until the resource or CPU usage became unreasonable. We were able, however, to generate and label *Special*(30,30) in a matter of seconds, and such a tree is of order 931. It is reasonable to assume that, aside from memory constraints, k and t can be rather large, quite safely.

5 Conclusion

Having invented algorithms to both generate and label general and special trees, it is clear that these algorithms have their own strengths and weaknesses.

The adjacency matrix representation of the trees may not have been the most efficient for long term storage of the trees. In retrospect, an adjacency list based approach might have been more economical.

Our generation for special trees leaves little to be desired; it is hard to see how it could be much more efficient. Generation of general trees is rather brute force, and it is conceivable that a more intelligent algorithm might be able to perform more efficiently.

The labeling of the special trees is much easier than the general case, and its algorithm is linear on the order of the tree. Determining whether or not a tree is special also seem to be linear, so inserting the special case into a general labeling algorithm would make a general labeling algorithm slightly faster. The general case seems to be fairly efficient for a large number of the trees, and an interesting further investigation would examine just how many of the trees cause significant difficulties for the algorithm presented. Perhaps it is the case that the difficult trees share some property that, like the special trees, show some simpler labeling algorithm.

APPENDIX A

A User Interaction

While user interaction is not necessary to use this system (all input and output could be coded as a program and run in batch style), for testing and evaluation purposes, real-time interaction is useful. We provide such capability, and describe it here.

A.1 Requirements `nauty`, LISP

There are two requirements for using our software: `nauty` and access to LISP (more specifically, LISPWORKS² or CLISP³). CLISP is available on `monica.cs.rpi.edu`, and on the same system, we provide access to `nauty`.

A.2 Getting into LISP

Access to CLISP should be automatic. To use the `nauty` we provide, you need to manually set `$PATH`. First, log into `monica.cs.rpi.edu` through SSH. `cd` into the directory containing `enumeration.lisp`. At the command prompt, execute the following: `"PATH=$PATH:~tayloj/nauty22 clisp"`. You should see something similar to the following:

```
tayloj@monica:~$ PATH=${PATH}:~tayloj/nauty22 clisp
  i i i i i i      ooooo  o      ooooooo  ooooo  ooooo
  I I I I I I      8 8 8      8 8  o 8 8
  I \ '+' / I      8 8      8 8 8 8 8
  \ '-+-' /      8 8      8  ooooo 8oooo
  \__|__-'      8 8      8 8 8
  |      8  o 8      8  o 8 8
  -----+-----  ooooo  8oooooo  ooo8ooo  ooooo  8
```

Copyright (c) Bruno Haible, Michael Stoll 1992, 1993
Copyright (c) Bruno Haible, Marcus Daniels 1994-1997
Copyright (c) Bruno Haible, Pierpaolo Bernardi, Sam Steingold 1998
Copyright (c) Bruno Haible, Sam Steingold 1999-2002

```
[1]>
```

“`[1]>`” is the LISP prompt, to which we will give commands. First, load the file by entering `“(load "enumeration.lisp")”`.

```
[1]> (load "enumeration.lisp")
;; Loading file enumeration.lisp ...
;; Loaded file enumeration.lisp
```

²a free personal edition is available at <http://www.lispworks.com>

³available at <http://clisp.cons.org>

T
[2]>

Now the system is loaded and ready to accept input.

A.3 Graph Format

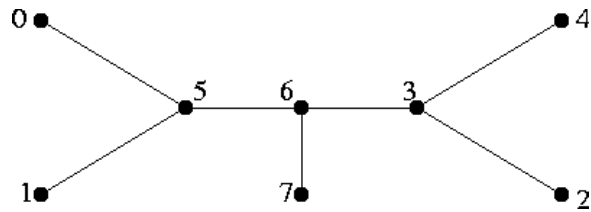


Figure 4: A graph to be entered at the LISP prompt

The format used to describe trees (more generally, graphs) is the adjacency matrix, and the order of the graph. The order is entered on a single line as an integer. The adjacency matrix is entered as a series of 1's and 0's. A 1 at position (i, j) in the adjacency matrix indicates that i and j are adjacent. To begin reading a graph manually, use the function `read-graph`. For instance, to enter the graph in Figure 4 perform the following:

```
[2]> (read-graph)
7
0 0 0 0 0 1 0 0
0 0 0 0 0 1 0 0
0 0 0 1 0 0 0 0
0 0 1 0 1 0 1 0
0 0 0 1 0 0 0 0
1 1 0 0 0 0 1 0
0 0 0 1 0 1 0 1
0 0 0 0 0 0 1 0
```

and the system will respond with

```
#S (GRAPH
  :ADJACENCY-MATRIX
  ((NIL NIL NIL NIL NIL T NIL NIL) (NIL NIL NIL NIL NIL T NIL NIL)
   (NIL NIL NIL T NIL NIL NIL NIL) (NIL NIL T NIL T NIL T NIL)
   (NIL NIL NIL T NIL NIL NIL NIL) (T T NIL NIL NIL NIL T NIL)
   (NIL NIL NIL T NIL T NIL T) (NIL NIL NIL NIL NIL NIL T NIL))
  :INTERNAL-ORDER NIL :INTERNAL-G6-STRING NIL :INTERNAL-CANONICAL-LABEL NIL
  :INTERNAL-DREADNAUT-STRING NIL :INTERNAL-GRACEFUL-LABELING NIL)
```

which is the system's internal representation of the graph structure. We needn't concern ourselves with this representation, however.

No checking is done to make sure that the input is correct. For instance, entering a graph in which a vertex is adjacent to itself will not flag an error. Nor is non-directedness flagged as an error. For instance, no error will be signalled if $(0,1)$ is marked 1 but $(1,0)$ is marked 0. Be careful with the input; otherwise later algorithms may not work correctly or as expected.

A.4 Commands

It is now known how to input trees into the system, but of more interest is generating and gracefully labelings trees. The following commands provide this interaction.

A.4.1 generate-trees-of-order

`(generate-trees-of-order n)` returns a list of all unique trees of order n . For instance:

```
CL-USER 2 > (generate-trees-of-order 4)
(#S(Graph :Adjacency-Matrix ((Nil T Nil Nil) (T Nil T T)
(Nil T Nil Nil) (Nil T Nil Nil)) :Internal-Order Nil
:Internal-G6-String "Ci" :Internal-Canonical-Label "CF"
:Internal-Dreadnaut-String Nil :Internal-Graceful-Labeling
Nil) #S(Graph :Adjacency-Matrix ((Nil T Nil Nil) (T Nil T Nil)
(Nil T Nil T) (Nil Nil T Nil)) :Internal-Order Nil
:Internal-G6-String "Ch" :Internal-Canonical-Label "CR"
:Internal-Dreadnaut-String Nil :Internal-Graceful-Labeling Nil))
```

```
CL-USER 3 > (length (generate-trees-of-order 4))
2
```

```
CL-USER 4 > (generate-trees-of-order 7)
(#S(Graph :Adjacency-Matrix ((Nil T Nil Nil Nil Nil Nil)
(T Nil T T T T T) (Nil T Nil Nil Nil Nil Nil)
(Nil T Nil Nil Nil Nil Nil) (Nil T Nil Nil Nil Nil Nil)
```

...

```
:Internal-Canonical-Label "F@IQO"
:Internal-Dreadnaut-String Nil :Internal-Graceful-Labeling Nil))
```

```
CL-USER 5 > (length (generate-trees-of-order 7))
11
```

A.4.2 print-trees-of-order

`(print-trees-of-order n)` prints the same trees that `(generate-trees-of-order n)` would generate, in the format that one would manually enter them. This can be useful in visualizing the graphs.

```
CL-USER 6 > (print-trees-of-order 4)
```

```
4  
0 1 0 0  
1 0 1 1  
0 1 0 0  
0 1 0 0
```

```
4  
0 1 0 0  
1 0 1 0  
0 1 0 1  
0 0 1 0  
Nil
```

```
CL-USER 7 > (print-trees-of-order 7)
```

```
7  
0 1 0 0 0 0 0  
1 0 1 1 1 1 1  
0 1 0 0 0 0 0  
0 1 0 0 0 0 0  
0 1 0 0 0 0 0  
0 1 0 0 0 0 0  
0 1 0 0 0 0 0  
0 1 0 0 0 0 0
```

```
...
```

```
7  
0 1 0 0 0 0 0  
1 0 1 0 0 0 0  
0 1 0 1 0 0 0  
0 0 1 0 1 0 0  
0 0 0 1 0 1 0  
0 0 0 0 1 0 1  
0 0 0 0 0 1 0  
Nil
```

A.4.3 make-special-tree

(make-special-tree num-paths path-length) will generate the tree $Special(num-paths, path-length)$. $num-paths$ corresponds to k , and $path-length$ to t in the definition of $Special(k, t)$.

```
CL-USER 8 > (make-special-tree 3 2)  
#S(Graph :Adjacency-Matrix ((Nil T Nil Nil T Nil Nil T Nil Nil)  
(T Nil T Nil Nil Nil Nil Nil Nil) (Nil T Nil T Nil Nil Nil  
Nil Nil Nil) (Nil Nil T Nil Nil Nil Nil Nil Nil Nil) (T Nil Nil  
Nil Nil T Nil Nil Nil Nil) (Nil Nil Nil Nil T Nil T Nil Nil Nil)  
(Nil Nil Nil Nil Nil T Nil Nil Nil Nil) (T Nil Nil Nil Nil Nil Nil
```

```

Nil T Nil) (Nil Nil Nil Nil Nil Nil Nil T Nil T) (Nil Nil Nil
Nil Nil Nil Nil Nil T Nil)) :Internal-Order Nil
:Internal-G6-String Nil :Internal-Canonical-Label Nil
:Internal-Dreadnaut-String Nil :Internal-Graceful-Labeling Nil)

```

A.4.4 print-special-tree

(print-special-tree num-paths path-length) will print the same tree that (make-special-tree num-paths path-length) would generate.

```

CL-USER 9 > (print-graph (make-special-tree 3 2))
10
0 1 0 0 1 0 0 1 0 0
1 0 1 0 0 0 0 0 0 0
0 1 0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0
1 0 0 0 0 1 0 0 0 0
0 0 0 0 1 0 1 0 0 0
0 0 0 0 0 1 0 0 0 0
1 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 1 0 1
0 0 0 0 0 0 0 0 1 0
Nil

```

A.4.5 Graceful Labeling Format

The graceful labeling format is a list of labels whose positions correspond to the vertex they label. For instance, a labeling might be a list (3 0 2 1) which would label a graph which has 4 vertices. $Label(0) \leftarrow 3$, $Label(1) \leftarrow 0$, $Label(1) \leftarrow 2$, etc.

A.4.6 special-tree-graceful-labeling

special-tree-graceful-labeling requires either a tree as an argument, or no argument, in which case a tree will be read from manual input. There is one caveat to special-tree-graceful-labeling and that is that if the tree provided is a special tree $Special(k,t)$, the ordering returned will be for the tree as it would be generated by (make-special-tree k t). This could lead to some confusion, so we recommend calling special-tree-graceful-labeling with a graph as an argument, and which has been generated by make-special-tree.

```

CL-USER 3 > (special-tree-graceful-labeling (make-special-tree 3 2))
(0 9 1 8 6 4 5 3 7 2)

```

This labeling is illustrated in Figure ??

A.4.7 general-tree-graceful-labeling

general-tree-graceful-labeling takes as an argument a single tree, or if an argument is not provided, will read a single tree from manual input. The labeling is returned as specified above.

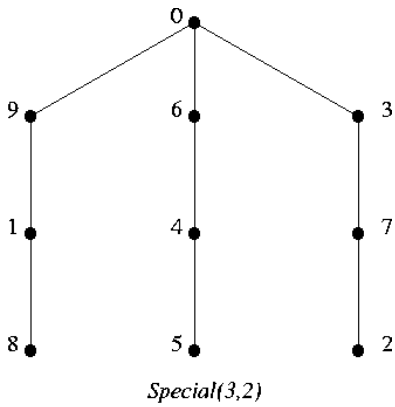


Figure 5: *Special(3,2)* labeled

```
CL-USER 5 > (general-tree-graceful-labeling)
```

```
4
0 1 0 0
1 0 1 0
0 1 0 1
0 0 1 0
```

```
(0 3 1 2)
```

```
CL-USER 6 > (setq *my-tree* (read-graph))
```

```
6
0 1 0 0 0 0
1 0 1 1 1 1
0 1 0 0 0 0
0 1 0 0 0 0
0 1 0 0 0 0
0 1 0 0 0 0
```

```
#S(Graph :Adjacency-Matrix ((Nil T Nil Nil Nil Nil)
(T Nil T T T T) (Nil T Nil Nil Nil Nil) (Nil T Nil Nil Nil Nil)
(Nil T Nil Nil Nil Nil) (Nil T Nil Nil Nil Nil)) :Internal-Order Nil
:Internal-G6-String Nil :Internal-Canonical-Label Nil
:Internal-Dreadnaut-String Nil :Internal-Graceful-Labeling Nil)
```

```
CL-USER 7 > (general-tree-graceful-labeling *my-tree*)
```

```
(0 5 1 2 3 4)
```

A.4.8 time

it may or may not be of interest to an individual, but LISP provides capability for timing the execution of arbitrary expressions. For instance, to time how long it takes to

generate all the trees of order 6, one might execute the following.

```
CL-USER 2 > (time (generate-trees-of-order 6))  
Timing the evaluation of (Generate-Trees-Of-Order 6)
```

```
user time    =      0.290  
system time  =      0.440  
Elapsed time = 0:00:03
```

```
...
```

```
CL-USER 3 >
```

APPENDIX B

B Division Of Work

B.1 Planning and Design

We started our project with several group meetings in which we discussed internal representation of trees and representations fit for interchange between programs, and we started to propose algorithms for the different tasks before us.

It was at this point that we decided on the plain text order and adjacency matrix representation for user input, and also as a convenient format by which graphs could be moved between programs, should the need arise.

B.2 Generation

The special tree generation was straightforward translation of Algorithm 1 and only an implementation detail.

The general tree algorithm was also straightforward, a simple implementation of Algorithm 4. We performed no significant optimizations or intelligent tree pruning, aside from early removal of duplicate trees.

B.3 Labeling

At the start of our work on the project, we discussed different approaches to labelings of the trees. We considered whether or not a specialized algorithm would be more appropriate for certain trees, whether or not a labeling algorithm should detect certain types of trees, and lastly whether or not we could create a special algorithm primarily for labeling the special trees.

B.3.1 Special Trees

Taylor and Wiegner worked on labeling algorithms for the special trees. Various algorithms for labeling the special trees were considered, including both algorithms which labeled along the ‘rays’ of the trees, algorithms which would ‘spiral’ out from the root, and algorithms which might work for more than just the special trees. The final algorithm which is included is an instance of the ‘spiral’ out from the root approach.

B.3.2 General Trees

Ka and Chung developed the algorithm for the general labeling based on the principle that after assigning the maximum edge length (and hence vertex labels $n - 1$ and 0), the next edge length must be adjacent to some vertex already labeled.

As a group we refined the algorithm to check for one or two minor edge cases, and the general labeling was finished.

B.4 Programming

Programming was a straightforward translation of algorithms and representations into code. Because a number of our algorithms are recursive, and we desired fast development, we chose LISP, a language with which Taylor is familiar.

B.5 Experiments

Experiments were conducted through several LISP based scripts. As a group we conducted analysis of these, including investigation of difficult labelings, and results of stress-testing.

B.6 Writeup

Typesetting was a simple task, but thanksgiving is due to D. Knuth's T_EX and Leslie Lamport's L^AT_EX, Drakos and Moore for L^AT_EX₂HTML, as well as the authors of xfig and EMACS.