SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY —
INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Verified Enumeration of Trees

Nils Cremer

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Verified Enumeration of Trees

Verifiziertes Aufzählen von Bäumen

Author:	Nils Cremer
Supervisor:	Prof. Tobias Nipkov
Advisor:	Emin Karayel
Submission Date:	May 15, 2023

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, May 15, 2023 Nils Cremer

Abstract

This thesis presents the verification of enumeration algorithms for trees. The first algorithm is based on the well known Prüfer-correspondence and allows the enumeration of all possible labeled trees over a fixed finite set of vertices. The second algorithm enumerates rooted, unlabeled trees of a specified size up to graph isomorphisms. It allows for the efficient enumeration without the use of an intermediate encoding of the trees with level sequences, unlike the algorithm by Beyer and Hedetniemi [1] it is based on. Both algorithms are formalized and verified in Isabelle/HOL. The formalization of trees and other graph theoretic results is also presented.

Contents

1 Introduction

Graphs are one of the most prominent combinatorial structures in computer science and mathematics. When one proves a graph theoretic statement and especially when complicated graph properties are involved, one sometimes resorts to exhaustive search. This can help find specific instances like counterexamples or count the number of instances of a specific type of graph. The enumeration of combinatorial structures is usually done by computer software. Especially in mathematical settings the correctness of the enumeration algorithm is of utmost importance. This involved using methods of formal program verification to show correctness.

In this thesis, we will focus on the generation of trees (connected acyclic graphs). An algorithm for the enumeration of trees over a given set of vertices as well as an algorithm for the enumeration of unlabeled, rooted trees are described and their correctness proved in the interactive theorem prover Isabelle/HOL.

2 Enumeration of Labeled Trees

This chapter describes an algorithm to enumerate all possible labeled trees with a given set of vertices. Then we show a correctness proof of the algorithm which then can also be used to determine the number of all possible trees.

2.1 Algorithm

The algorithm is based on Prüfer codes [5] which describe a one-to-one correspondence from trees with n vertices to sequences of vertices of length $n-2$. Thus, to enumerate all possible trees one can enumerate all $(n-2)$-sequences where each entry is one of the vertex labels and then transform each Prüfer sequence into its corresponding tree.

To understand how Prüfer sequences are constructed, we first describe a function that determines the Prüfer sequence of a tree. Given a tree T with at least 2 vertices and an arbitrary linear order on the vertices of T proceed as follows:

1. If the tree has exactly 2 vertices, then the Prüfer sequence is empty.
2. Else, find the leaf a with the smallest label.
3. Let b be the unique neighbor of a, then b is the first element of the Prüfer sequence.
4. The remaining elements are the Prüfer sequence of the tree T' where leaf a is removed.

See Figure 2.1 for an example, where we order the nodes by the natural order of natural numbers.

Instead of defining a linear order on vertex labels explicitly, the algorithm is provided with a distinct list of vertices which implicitly defines a linear order. The following function `prufer_seq_of_tree` implements this algorithm.

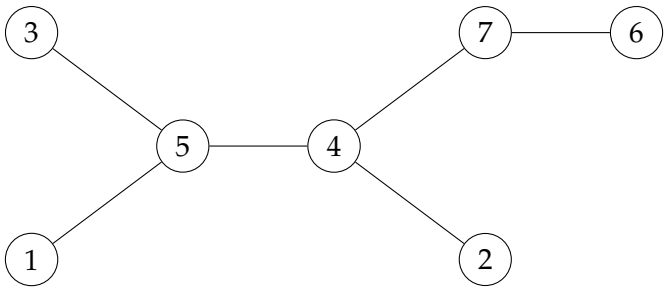

Figure 2.1: Tree with Prüfer sequence $[5, 4, 5, 4, 7]$

```
definition remove_vertex_edges :: "'a ⇒ 'a edge set ⇒ 'a edge set" where
  "remove_vertex_edges v E = {e∈E. ¬ graph_system.incident v e}"

fun prufer_seq_of_tree :: "'a list ⇒ 'a edge set ⇒ 'a list" where
  "prufer_seq_of_tree verts E =
    (if length verts ≤ 2 then []
    else (case find (tree.leaf E) verts of
      Some leaf ⇒ (THE v. ulgraph.vert_adj E leaf v) #
prufer_seq_of_tree (remove1 leaf verts) (remove_vertex_edges leaf E)))"
```

This function describes how Prüfer sequences are constructed and is relevant for the correctness proof, but for the actual enumeration algorithm the inverse operation is needed. The following function `tree_of_prufer_seq` constructs the tree given a Prüfer sequence. It takes as input the list of all n vertices, as `prufer_seq_of_tree` does, and a Prüfer sequences of length $n - 2$ and works in the following steps:

1. If the Prüfer sequence is empty, the result is a tree containing the two remaining vertices connected by an edge.
2. Otherwise, let a be the smallest vertex not contained in the Prüfer sequence, b the first element of the Prüfer sequence and s the remaining elements.
3. Remove a from the list of vertices and construct the tree of the Prüfer sequence s recursively.
4. Add the vertex a with an edge from a to b to the tree.

```
fun tree_edges_of_prufer_seq :: "'a list ⇒ 'a list ⇒ 'a edge set"
where
  "tree_edges_of_prufer_seq [u,v] [] = {{u,v}}"
| "tree_edges_of_prufer_seq verts (b#seq) =
    (case find (λx. x ∉ set (b#seq)) verts of
      Some a ⇒ insert {a,b} (tree_edges_of_prufer_seq (remove1 a verts)
seq))"

definition tree_of_prufer_seq :: "'a list ⇒ 'a list ⇒ 'a pregraph" where
  "tree_of_prufer_seq verts seq = (set verts, tree_edges_of_prufer_seq
verts seq)"
```

Then all trees can be enumerated by enumerating all possible Prüfer sequences and constructing the corresponding tree for each of them. The simple enumeration of all n-sequences is implemented in the List theory.

```
definition labeled_tree_enum :: "'a list ⇒ 'a pregraph list" where
  "labeled_tree_enum verts = map (tree_of_prufer_seq verts) (List.n_lists
(length verts - 2) verts)"
```

With this approach to enumeration, it also follows immediately that the number of labeled trees on n vertices is equal to the number of $(n-2)$-sequences of vertices, that is n^{n-2}. This is known as Cayley's Formula:

```
lemma (in valid_verts) cayleys_formula: "card (labeled_trees (set verts))
= length verts ^ (length verts - 2)"
```

With this number we can also determine the runtime of the algorithm. The n^{n-2} Prüfer sequences can be enumerated in $O(n^{n-2})$ time with the implementation in the List theory. Assuming an efficient set implementation with $O(1)$ lookup and insertion, the construction of the tree from a Prüfer sequence can be done in $O(n^2)$ time. This mean all trees are generated in $O(n^{n-2} + n^{n-2} * n^2) = O(n^n)$ time. One could potentially improve the runtime of the algorithm by using a priority queue to keep track of the number of times a vertex appears in the Prüfer sequence. This would, however, result in a considerably more complex verification and is not done in this thesis.

2.2 Proof of correctness

To begin the formal verification of the algorithm a formal definition of trees is needed in Isabelle/HOL. We build upon an existing entry in the Archive of Formal Proofs by Edmonds [3] in which undirected graphs are formalized. There are other formalizations of graphs such as the one by Noschinski [4], but the definition of graphs is far more general and would thus complicate our proofs unnecessarily. Meanwhile, Edmonds' formalization targets exactly the setting of undirected simple graphs needed in this thesis. It contains standard definitions for undirected graphs, paths, connectivity and other useful definitions and lemmas. However, there are no definitions regarding trees and thus we extend the existing theory with the necessary definitions and lemmas. Most importantly, we define a tree as a finite, connected, undirected graph with no cycles.

```
locale tree = fin_connected_ulgraph +
  assumes no_cycles: "¬ is_cycle2 c"
```

Notice that this definition does not make use of the `is_cycle` definition of the existing library as it does not enforce the edges of the cycle to be distinct. Thus, the path $a \to b \to a$ would be considered a cycle. Since this is clearly undesired in this setting we make an alternative definition that restricts the notion of cycles further:

```
definition is_cycle2 :: "'a list ⇒ bool" where
  "is_cycle2 xs ⟷ is_cycle xs ∧ distinct (walk_edges xs)"
```

In addition, we define the set of labeled trees over a fixed set of vertices, which is the set to be enumerated by the algorithm.

```
definition labeled_trees :: "'a set ⇒ 'a pregraph set" where
  "labeled_trees V = {(V,E)| E. tree V E}"
```

With this formal definition it can be shown that the Prüfer correspondence is indeed a one-to-one mapping from Prüfer sequences to trees and thus the correctness of the algorithm.

Here, and for the remainder of this chapter, we will always assume the elements of the vertex list given to the algorithm to be distinct. In addition we assume there to be at least two vertices as Prüfer sequences only exist for trees of size at least two. These assumptions are combined in the `valid_verts` locale.

```
locale valid_verts =
  fixes verts
  assumes length_verts: "length verts ≥ 2"
  and distinct_verts: "distinct verts"
```

First we show that the graph generated by `tree_of_prufer_seq` is indeed a tree. By simple induction it can be proved that the graph produced is always connected as every added vertex is immediately connected with the rest of the graph. In addition, if the vertex list has n elements the graph has n vertices and $n - 1$ edges which combined with the connectivity makes it a tree.

Next to show that the set of elements produced by the algorithm are all possible trees, it is necessary to prove that every tree has a Prüfer sequence. This is where the inverse function is useful. That is, `prufer_seq_of_tree` produces a valid Prüfer sequence for a tree.

```
lemma "tree_edges_of_prufer_seq verts (prufer_seq_of_tree verts E) = E"
```

This requires one observation, namely that the number of times an element occurs in the Prüfer sequence is one less than its degree in the graph. This means that the vertices not in the Prüfer sequence are exactly the leaves of the tree. Thus, the operations of finding the leaf with the smallest label and finding the smallest vertex not in the Prüfer sequence result in the same vertex. This can be used to prove the above statement.

This means that the algorithm generates all possible trees.

```
theorem (in valid_verts) "set (labeled_tree_enum verts) = labeled_trees
  (set verts)"
```

The last step is to show that no tree is produced more than once, that is the injectivity of `tree_of_prufer_seq`. This can be proved inductively on the length of the Prüfer sequences. Let s_1 and s_2 be two different Prüfer sequences producing trees T_1 and T_2 respectively. We want to show that T_1 and T_2 are different. Let us only consider the case that s_1 and s_2 are of the same length as otherwise they clearly produce trees of different sizes. Assume for the sake of contradiction that $T_1 = T_2$. Then due to the observation made earlier, finding the smallest vertex not in the Prüfer sequences s_1 and s_2 is the same as finding the leaves in T_1 and T_2 with the smallest label and so they are equal. Let this leaf be denoted by a. If s_1 and s_2 differ in their first element, say b_1 and b_2 respectively, then T_1 contains an edge between a and b_1 and T_2 contains an edge between a and b_2. In the algorithm a is then removed from the set of considered vertices and so no further edges containing a will be added. Therefore, T_1 does not

contain the edge from a to b_2 and T_2 does not contain the edge from a to b_1. That means the produced trees are not equal which contradicts the initial assumption.

In case s_1 and s_2 do not differ in their first element, they differ in some later position and we can use the inductive hypothesis to show the statement. Thus, all trees produced by the algorithm are distinct.

```
theorem (in valid_verts) "distinct (labeled_tree_enum verts)"
```

3 Enumeration of Unlabeled Rooted Trees

This Chapter describes the verification of an algorithm enumerating all unlabeled, rooted trees with a specific number of nodes. Let $G = (V, E, r)$ denote the rooted graph with vertex set V, edge set E and a distinguished root $r \in V$.

```
locale rgraph = graph_system +
  fixes r
  assumes root_wf: "r ∈ V"
```

Similarly, a rooted tree is a tree with a fixed root.

```
locale rtree = tree + rgraph
```

As can be seen in the definition the nodes of rooted graphs are still labeled. To get a notion of unlabeled trees we define isomorphism on these graphs. A graph isomorphism for (unrooted) graphs is a bijection between the vertex sets of two graphs which preserves edge connectivity.

```
locale graph_isomorphism =
  G: graph_system V_G E_G for V_G E_G +
  fixes V_H E_H f
  assumes bij_f: "bij_betw f V_G V_H"
  and edge_preserving: "((‘) f) ‘ E_G = E_H"
```

Similarly, graph isomorphism is defined on rooted graphs with the additional requirement that the isomorphism also preserves the root node.

```
locale rgraph_isomorphism =
  G: rgraph V_G E_G r_G + graph_isomorphism V_G E_G V_H E_H f for V_G E_G r_G
V_H E_H r_H f +
  assumes root_preserving: "f r_G = r_H"
```

Two rooted trees T_1 and T_2 are isomorphic (denoted with $T_1 \simeq_r T_2$) if there exists an isomorphism between them.

The algorithm described here produces a list of all rooted trees with n vertices modulo isomorphism. That is, every rooted tree is isomorphic to exactly one rooted tree in the resulting list.

3.1 Algorithm

The idea of the algorithm is based on one described by Beyer and Hedetniemi [1]. It works on ordered, rooted trees. An ordered, rooted tree is a rooted tree with a linear order with which subtrees can be ordered. We will represent such a tree graphically as in Figure 3.1 by drawing the root node at the top and the subtrees in the corresponding order beneath it. Since we are interested in unlabeled trees, node labels will generally be omitted. The two trees shown differ in the ordering of their subtrees but represent the same underlying rooted tree. Of the possibly many ordered trees corresponding to a rooted tree, one canonical tree will be defined. For this, a linear order will be defined on ordered, rooted trees recursively. First we define the type of ordered, rooted trees as a root node with a list of ordered subtrees.

```
datatype tree = Node "tree list"
```

The order defined on these trees will be the reversed lexicographical ordering. No tree is smaller than the singleton tree (a tree only containing a root node). The singleton tree is smaller than every other tree besides itself. Otherwise, the subtrees are compared elementwise from the back. To simplify the definition in a functional setting, we first define the regular lexicographical order on trees and then define the desired ordering by mirroring its arguments.

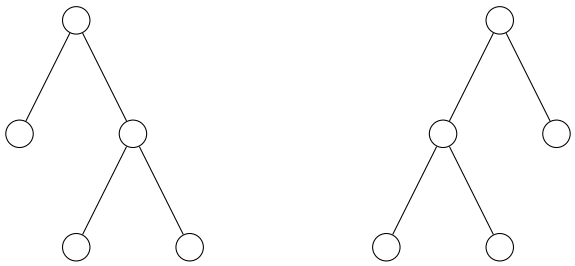

Figure 3.1: Example rooted trees

```
fun lexord_tree where
  "lexord_tree t (Node []) ⟷ False"
| "lexord_tree (Node []) r ⟷ True"
| "lexord_tree (Node (t#ts)) (Node (r#rs)) ⟷
    lexord_tree t r ∨ (t = r ∧ lexord_tree (Node ts) (Node rs))"

fun mirror :: "tree ⇒ tree" where
  "mirror (Node ts) = Node (map mirror (rev ts))"

definition
  tree_less_def: "(t::tree) < r ⟷ lexord_tree (mirror t) (mirror r)"
```

For a rooted tree T, we now define the canonical ordered tree as the greatest ordered tree corresponding to T. In Figure 3.1 the left tree is the canonical ordered tree.

The algorithm will produce all canonical ordered trees. Instead of working with rooted trees directly, the algorithm of Beyer and Hedetniemi uses an alternative representation of rooted trees, namely level sequences. The algorithm described here enumerates rooted trees in the same order but is defined recursively on a recursive tree datatype instead of on the level sequences.

To traverse all rooted trees a successor function is defined giving the next smallest canonical tree. To accomplish this, some auxiliary functions are needed. We define a function that trims a tree, *trim_tree n t* trims t to n nodes by removing nodes in postorder until the tree has at most n nodes. See Figure 3.2 for an example of how *trim_tree* works.

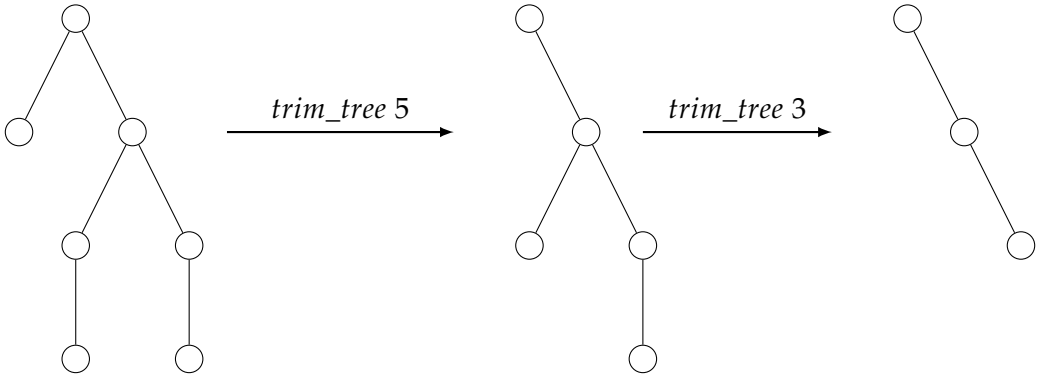

Figure 3.2: Example of *trim_tree*

```
fun trim_tree :: "nat ⇒ tree ⇒ nat × tree" where
  "trim_tree 0 t = (0, t)"
| "trim_tree (Suc 0) t = (0, Node [])"
| "trim_tree (Suc n) (Node []) = (n, Node [])"
| "trim_tree n (Node (t#ts)) =
  (case trim_tree n (Node ts) of
    (0, t') ⇒ (0, t') |
    (n1, Node ts') ⇒
      let (n2, t') = trim_tree n1 t
      in (n2, Node (t'#ts')))"
```

With the help of this a function *fill_tree* is defined, *fill_tree n t* produces a list of trees with a total number of n nodes. Each tree in the list will be t except the first one which might be trimmed to a suitable size in order for the total number of nodes to be n.

```
fun fill_tree :: "nat ⇒ tree ⇒ tree list" where
  "fill_tree 0 _ = []"
| "fill_tree n t =
    (let (n', t') = trim_tree n t
    in fill_tree n' t' @ [t'])"
```

Now we define the successor function which produces, given a tree, the next smallest canonical tree. Generating the next smallest tree follows these 4 steps:

1. Remove all direct subtrees of the root node which are singleton trees.
2. Find the first node v (in preorder) which has a singleton tree as the first subtree. Let the subtree of v be T_v.
3. Remove the first singleton subtree of T_v which results in a tree T'_v.
4. Let p be the parent of v. Fill the subtree list of p with T'_v until the entire tree consists of n nodes again.

See Figure 3.3 for examples of how the function operates.
This function can be directly defined recursively on the tree datatype.

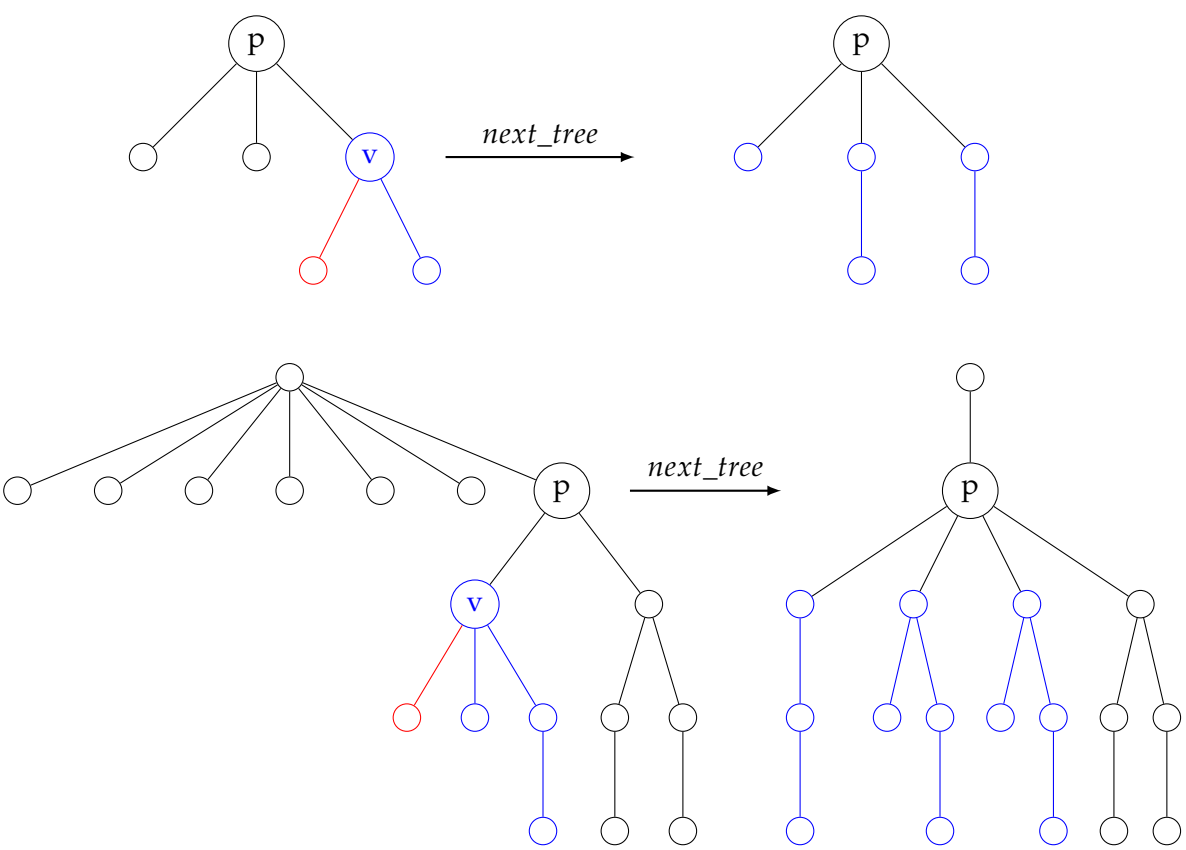

Figure 3.3: Example of *next_tree*

```
fun next_tree_aux :: "nat ⇒ tree ⇒ tree option" where
  "next_tree_aux n (Node []) = None"
| "next_tree_aux n (Node (Node [] # ts)) =
      next_tree_aux (Suc n) (Node ts)"
| "next_tree_aux n (Node (Node (Node [] # rs) # ts)) =
      Some (Node (fill_tree (Suc n) (Node rs) @ (Node rs) # ts))"
| "next_tree_aux n (Node (t # ts)) =
      Some (Node (the (next_tree_aux n t) # ts))"

fun next_tree :: "tree ⇒ tree option" where
  "next_tree t = next_tree_aux 0 t"
```

None will be returned if the argument is already the smallest tree and thus there is no next smaller tree.

To now enumerate all rooted trees with n nodes, one starts with the greatest tree on n nodes and repeatedly calls *next_tree* until the smallest tree is reached.

```
fun greatest_tree :: "nat ⇒ tree" where
  "greatest_tree (Suc 0) = Node []"
| "greatest_tree (Suc n) = Node [greatest_tree n]"

function n_tree_enum_aux :: "tree ⇒ tree list" where
  "n_tree_enum_aux t =
  (case next_tree t of None ⇒ [t] | Some t' ⇒ t # n_tree_enum_aux t')"
by pat_completeness auto

fun n_tree_enum :: "nat ⇒ tree list" where
  "n_tree_enum 0 = []"
| "n_tree_enum n = n_tree_enum_aux (greatest_tree n)"
```

To prove termination of the enumeration we show that the output of *next_tree* (if it is not *None*) is strictly smaller than its input and that there are only finitely many rooted trees with n nodes.

To determine the runtime of the algorithm let N be the number of rooted trees of size n. Then `next_tree` is called N times during the enumeration of all trees. `next_tree` traverses each node at most once and since `fill_tree` runs in linear time with respect to the number of nodes produced, `next_tree` runs in $O(n)$. Thus the runtime of the algorithm is $O(n * N)$. Beyer and Hedetniemi [1] prove that their imperative implementation of `next_tree` runs on average in constant time over all N calls. This means their algorithm runs in $O(N)$ time. Even if the implementation presented here performs the same operations on the tree, it traverses some extra nodes while doing so.

This means the result cannot be transferred directly. However, it may well be possible to show a better bound or perhaps even $O(N)$ here as well with a more careful analysis, as on average still only a small portion of nodes is actually traversed.

3.2 Proof of correctness

3.2.1 Totality

Instead of working with canonical rooted trees, we work with an alternative (equivalent) definition of regularity. An ordered, rooted tree is regular iff the subtrees are ordered with respect to the linear order previously defined and all subtrees are regular themselves.

```
fun regular :: "tree ⇒ bool" where
  "regular (Node ts) ⟷ sorted ts ∧ (∀t∈set ts. regular t)"
```

To prove correctness we first show that the algorithm enumerates all regular ordered, rooted trees and then that the set of regular ordered rooted trees is equivalent to the set of rooted trees modulo isomorphism.

By simple induction all defined functions can be shown to preserve regularity. In addition to that, `next_tree` preserves the size of the tree, again by induction and simple auxiliary lemmas on the size of the output of the defined auxiliary functions. Since `greatest_tree n` is regular and of size n, the algorithm produces only regular trees of size n. Due to the strict monotonicity of `next_tree` proven earlier it immediately follows that no tree is generated twice. In addition, we prove that there is a unique minimal tree of size n, namely the tree consisting of a root node and $n-1$ singleton subtrees. The enumeration stops only if this minimal tree is reached. What is left to prove is that the algorithm does not skip any regular rooted tree. For this some auxiliary lemmas are needed. First, observe that due to the order in which `trim_tree` removes nodes, `trim_tree n t` is the greatest tree with at most n nodes that is smaller or equal to t. This also means that a tree which has as subtrees the list of trees generated by `fill_tree n t` is the greatest tree of size $n+1$ (observe the additional root node) where every subtree is smaller or equal to t. Now to get an intuition of why `next_tree` generates the next smallest tree consider counting down numbers as an analogy. When counting down from 14200 one tries to decrease the rightmost digit that can be decreased, i.e. the 2. Then set the digits to the right of the decreased number to their maximum resulting in 14199.

Since subtrees are compared in reverse order, the leftmost subtree that can be decreased (i.e. is not a singleton) will be decreased by removing its leftmost leaf. Then

using `fill_tree` the subtree left to the decreased tree is maximized. This argument can be made formal by induction on `next_tree` and with some technicalities we omit here. Thus, we have proven that all regular trees of size n are enumerated by the algorithm.

```
fun tree_size :: "tree ⇒ nat" where
  "tree_size (Node ts) = Suc (∑t←ts. tree_size t)"

definition regular_n_trees :: "nat ⇒ tree set" where
  "regular_n_trees n = {t. tree_size t = n ∧ regular t}"

theorem "set (n_tree_enum n) = regular_n_trees n"
```

3.2.2 Distinctness modulo Isomorphism

In order for the algorithm to meet its specification we need to show that the regular, ordered, rooted trees are equivalent to all rooted trees. We define a function to map an ordered, unlabeled, rooted tree to a rooted, labeled tree in graph form. This is done via two additional intermediate steps which simplify the proofs. First we define ordered, rooted, labeled trees which are like ordered, rooted trees except that each node has a label associated with it. A function is defined to label the nodes of an unlabeled tree. This can be done arbitrarily as long as the nodes have different labels since any labeling will produce an isomorphic tree. Next we define unordered, labeled trees where the subtrees of a node are represented as a finite set instead of an ordered list. As the last step one can transform an unordered, labeled tree t into its graph representation simply by the graph $G = (V, E)$ where V is the set of nodes in t and E contains an edge for every node to the roots of its subtrees. Let t be an ordered, unlabeled, rooted tree, then G_t denotes the graph representation of t. It can be shown that G_t is indeed a tree, i.e. connected and acyclic.

We now want to show for every tree graph T there exists exactly one t such that T and G_t are isomorphic. Let r be the root of T. Moreover, let C be the set of connected components of $T - \{r\}$ (the tree without its root). Then one can generate unordered unlabeled trees for each connected component recursively by taking the unique vertex in it that is connected to r as its root. The sorted list of these trees will be the subtrees of t. This construction results in a regular tree t such that G_t is isomorphic to T. Denote this tree by $t(T)$.

The remaining step to show uniqueness is to show that G_{t_1} and G_{t_2} are not isomorphic for two distinct regular trees t_1 and t_2, or equivalently that $t_1 = t_2$ is implied by G_{t_1} and G_{t_2} being isomorphic.

lemma *"regular t1 ⟹ regular t2 ⟹ G$_{t1}$ ≃$_r$ G$_{t2}$ ⟹ t1 = t2"*

One can prove that $t(G_t) = t$ for all regular trees t. Also, since the construction of $t(T)$ does not depend on the labels of the nodes, but only on the structure of the tree, we have $t(G_{t_1}) = t(G_{t_2})$ since G_{t_1} and G_{t_2} are isomorphic. This means

$$t1 = t(G_{t_1}) = t(G_{t_2}) = t2$$

which finishes the proof.

This concludes the entire proof and shows that the described algorithm generates each tree exactly once modulo isomorphism, that is it generates all unlabeled, rooted trees.

theorem *"G ∈ n_rtree_graphs n*
⟹ ∃!G' ∈ set (n_rtree_graph_enum n). G' ≃$_r$ G"

4 Formalization of Graph Theoretic Results

In the previous chapters, the main focus was on the algorithms at hand and the approach to formalize them. While we mentioned some graph theoretic definitions we took well known definitions and lemmas on trees and graphs for granted. Some results were already formalized by the undirected graph library but further ones were needed for this project. This chapter tries to summarize some graph theoretic results that were formalized in the process. These are well suited to be reused in other projects.

We already discussed the definition of trees as acyclic, connected graphs. We also provide the standard definition of leaves as vertices of degree 1. When proving properties of trees it is often useful to prove them by induction on the number of vertices. To do this we show that removing a leaf from a tree results in another tree. To simplify proving properties on trees inductively we provide a custom induction rule. One then only needs to prove the property for the singleton tree and that the property is preserved when adding a new leaf.

```
lemma tree_induct [case_names singolton insert, induct set: tree]:
  assumes tree: "tree V E"
    and trivial: "⋀v. tree {v} {} ⟹ P {v} {}"
    and insert: "⋀l v V E. tree V E ⟹ P V E ⟹ l ∉ V ⟹ v ∈ V ⟹
      {l,v} ∉ E ⟹ tree.leaf (insert {l,v} E) l ⟹
      P (insert l V) (insert {l,v} E)"
  shows "P V E"
```

With the additional assumptions in the inductive step, proofs of properties on trees, such as that a tree with n vertices has $n - 1$ edges, can be simplified a great deal.

When proving that a graph is a tree, arguing about acyclicity is in many cases difficult as it can require one to argue about all possible paths in the graphs. Instead, like in Section 2.2, we use the well-known fact that a connected graph with n vertices and $n - 1$ edges is a tree. This lemma turns out to require a fair bit of setup. First, we define a spanning tree of a graph G to be a subgraph of G that is a tree and that contains all vertices of G.

```
locale spanning_tree = ulgraph V E + T: tree V T for V E T +
  assumes subgraph: "T ⊆ E"
```

Next, we prove that every finite, connected graph has a spanning tree. This is done inductively by removing edges of cycles, which keeps the graph connected, until the graph is acyclic. Now to prove the lemma we were interested in initially, that every connected graph with n vertices and $n-1$ edges is a tree, let us assume for contradiction that there exists such a graph that is not a tree. As the graph is connected, it has a spanning tree and since it is not a tree itself, the spanning tree has to be a strict subgraph. Since a spanning tree is a tree it has $n-1$ edges by the statement proven earlier. That means the original graph has more than $n-1$ edges, which is a contradiction. That concludes the proof.

Another concept that had not yet been formalized is connected components. This was required for rooted trees, where removing the root node leaves the principal subtrees as connected components. For a formal definition we first define a relation on nodes that indicates connectivity.

```
abbreviation "vert_connected_rel ≡ {(u,v). vert_connected u v}"
```

We prove this to be an equivalence relation and define connected components to be the equivalence classes of this relation.

```
definition connected_components :: "'a set set" where
  "connected_components = V // vert_connected_rel"
```

A couple of interesting properties of connected components that were needed for the project were proven as well.

Additionally, there are dozens more lemmas for graphs, paths, connectivity, subgraphs, vertex degrees etc. formalized. One well known fact which might be of interest for other projects is the handshaking lemma which relates the number of edges with the sum of vertex degrees.

```
lemma (in fin_ulgraph) handshaking: "(∑v∈V. degree v) = 2 * card E"
```

The interested reader is referred to the Archive of Formal Proofs entry for the full development.

5 Conclusions

The aim of this thesis was to formalize algorithms for the enumeration of trees and to prove their correctness. Algorithms for the enumeration of labeled trees and unlabeled, rooted trees were described and successfully verified in Isabelle/HOL. To accomplish this, trees and other graph theoretic results were formalized as well. These results can be used by a wide variety of other projects involving trees or undirected graphs in general.

Furthermore, there are other algorithms for the enumeration of other tree structures that could be formalized in future projects. This includes algorithms for the generation of free trees [6] or spanning trees of graphs [2].

Bibliography

[1] T. Beyer and S. M. Hedetniemi. "Constant Time Generation of Rooted Trees." In: *SIAM Journal on Computing* 9.4 (1980), pp. 706–712. DOI: 10.1137/0209055. eprint: https://doi.org/10.1137/0209055.

[2] M. Chakraborty, S. Chowdhury, J. Chakraborty, R. Mehera, and R. Pal. "Algorithms for generating all possible spanning trees of a simple undirected connected graph: an extensive review." In: *Complex & Intelligent Systems* 5 (Aug. 2018). DOI: 10.1007/s40747-018-0079-7.

[3] C. Edmonds. "Undirected Graph Theory." In: *Archive of Formal Proofs* (Sept. 2022). https://isa-afp.org/entries/Undirected_Graph_Theory.html, Formal proof development. ISSN: 2150-914x.

[4] L. Noschinski. "Graph Theory." In: *Archive of Formal Proofs* (Apr. 2013). https://isa-afp.org/entries/Graph_Theory.html, Formal proof development. ISSN: 2150-914x.

[5] H. Prüfer. "Neuer Beweis eines Satzes über Permutationen." In: *Archiv der Mathematischen Physik* 27 (1918), pp. 742–744.

[6] R. A. Wright, L. B. Richmond, A. M. Odlyzko, and B. D. McKay. "Constant Time Generation of Free Trees." In: *SIAM J. Comput.* 15 (1986), pp. 540–548.