Fast Generation of Unlabelled Free Trees using Weight Sequences

Paul Brown and Trevor Fenner

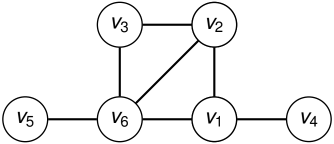
Department of Computer Science and Information Systems
Birkbeck, University of London

December 2020

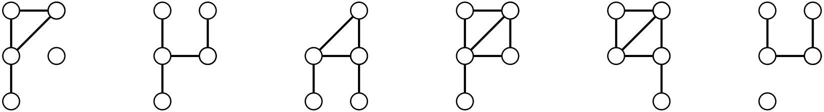
Motivation - Reconstruction I

- The *vertex-deleted subgraph* or *card* $G - v$ is graph obtained by deleting the vertex v and all edges incident to v
- The *deck* of a graph G of order n is the *multiset* of the n *unlabelled* cards of G
- The number of common cards of non-isomorphic simple graphs G and H, denoted by $b(G, H)$, is the cardinality of the multiset intersection of the decks of G and H
- Reconstruction conjecture (Kelly, Ulam 1941):
$$b(G, H) \leq n - 1 \text{ for } n \geq 3$$
- Conjecture shown to hold for many classes of graphs, e.g., Trees, Disconnected Graphs, Planar graphs

Example deck



A Graph G



$G - v_1$ $G - v_2$ $G - v_3$ $G - v_4$ $G - v_5$ $G - v_6$

Figure 1: Cards of G

Motivation - Reconstruction II

- **Strong Reconstruction conjecture** (BBF 2010):
  $b(G, H) \leq 2\left\lfloor \frac{1}{3}(n-1) \right\rfloor$, for all $n \geq 13$
  - Can construct pairs of graphs from many families with $b(G, H) \approx \frac{2n}{3}$
  - In particular, there exists an infinite family of pairs of trees with $b(G, H) = 2\left\lfloor \frac{1}{3}(n-5) \right\rfloor$

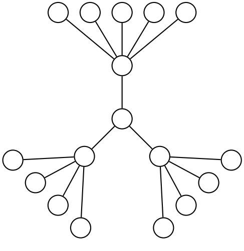


Figure 2: Pair of trees order 17 with 8 common cards

Motivation - Reconstruction III

- Wanted empirical evidence that **Stong** RC true for trees
  - Rob Cook (BBK MSc CS - 2010) tabulated $b(G, H)$ up to $n = 24$
  - We want to verify then extend his results
- Need a representation for trees and efficient method to construct their decks and check whether two cards are isomorphic
- *Weight sequences* are a suitable representation for this task
- Require fast method to generate the weight sequences of all trees of order n
- Generating the trees using NetWorkX and converting to weight sequences was proving cumbersome and too slow

Tree basics

- A *(free) tree* is a connected undirected graph with no cycles
  - A *rooted tree* is a free tree with a distinguished vertex - its *root*
  - An *ordered tree* is a rooted tree in which there is an ordering defined on the *children* of each vertex
- In a *labelled* tree, each vertex is assigned a unique label
- We assume that the vertices of an ordered tree are labelled in preorder, e.g., $v_1, v_2, \ldots, v_n$
- Two labelled *free* trees are *f-isomorphic* if there is a bijection between their vertex sets that preserves adjacency and non-adjacency

Isomorphic free trees

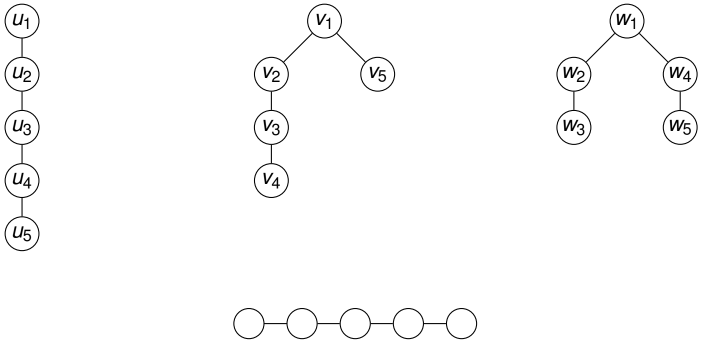


Figure 3: Isomorphic labelled free trees

Figure 3 shows three labelled free trees isomorphic to P_5 and the unlabelled representative of the isomorphism class.

Isomorphic rooted /ordered trees

- Two labelled *rooted* trees are *r-isomorphic* if there exists an isomorphism between their underlying free trees that *maps the root of one onto the root of the other*
- If they are ordered and there exists an r-isomorphism that preserves the orderings of the children then they are *o-isomorphic*

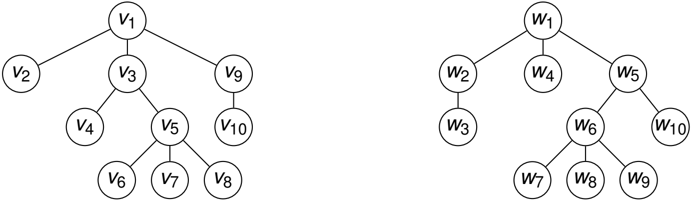


Figure 4: Two r-isomorphic trees that are *not* o-isomorphic

Some history

- "Cayley's formula": n^{n-2} free trees on *n labelled* vertices, Borchard (1860)
- Asymptotic estimates for numbers of both *unlabelled* free and rooted trees, Otter (1948): $O(\alpha^n)$ where $\alpha \approx 2.996$
- Generating functions for the numbers of both unlabelled free and rooted trees, Knuth (1997)

- Algorithm for *generating* unlabelled *rooted* trees, Beyer and Hedetniemi (1980)
- Algorithm extended to unlabelled *free* trees, Wright, Richmond, Odlyzko and McKay (1986) - WROM algorithm
- Alternative algorithm for generating unlabelled free trees, Li and Ruskey (1996)

Tree representations

- *Representation sequence* of an ordered tree: an *integer sequence* obtained by traversing the tree in a specified order (usually preorder) and recording some property of each vertex
- *Valid* representation for ordered trees: a well-defined representation sequence such that any two ordered trees with the same sequence are o-isomorphic
  - Analogous definitions for rooted and free trees
- *Level* sequence: level of root is 1, its children 2 etc.
  - Used in WROM algorithm
  - E.g., for Figure 4: 1 2 2 3 3 4 4 4 2 3 and 1 2 3 2 2 3 4 4 4 3
- *Parent* sequence: index of label of parent of each vertex
  - Used in Li & Ruskey algorithm
  - E.g., for Figure 4: 1 1 3 3 5 5 5 1 9 and 1 2 1 1 5 6 6 6 5

Birkbeck UNIVERSITY OF LONDON

Weight sequences

- The *weight* of a vertex v in an ordered tree R is the order of the ordered subtree $R(v)$, rooted at v, that consists of v and its descendants
- The *weight sequence* $\mathbf{ws}(R)$ of an ordered tree R is obtained by recording the *weight* of each vertex in a preorder traversal of R
  - E.g., for Figure 4: 10 1 6 1 4 1 1 1 2 1 and 10 2 1 1 6 4 1 1 1 1
- $\mathbf{ws}(R)$ preserves *referential transparency*:
  - $\mathbf{ws}(R(v_k))$ is the subsequence of $\mathbf{ws}(R)$ corresponding to $R(v_k)$
  - We can construct the weight sequence of any ordered tree directly from the weight sequences of its subtrees:
    if $u_1, u_2, \ldots, u_p$ are the children of the root of R then
    $$\mathbf{ws}(R) = n \oplus \mathbf{ws}(R(u_1)) \oplus \mathbf{ws}(R(u_2)) \oplus \ldots \oplus \mathbf{ws}(R(u_p))$$
- Easily shown to be a valid representation for ordered trees

Canonical weight sequences I

- Weight sequences for *rooted* trees: we need a unique representative from each r-isomorphism class of ordered trees
- An ordered tree R is *canonically-ordered* if, for each vertex u of R having a "next" sibling v, $\mathbf{ws}(R(u)) \geq \mathbf{ws}(R(v))$

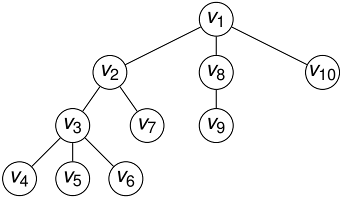


Figure 5: Canonically ordered tree r-isomorphic to trees in Fig 4

Canonical weight sequences II

- The *canonical weight sequence* **cws**(R) of R is the weight sequence of any canonically ordered tree r-isomorphic to R
- It is straightforward to show that
  - If R and R' are r-isomorphic canonically-ordered trees then **ws**$(R) =$ **ws**(R'), and R and R' are o-isomorphic
  - By permuting the subtrees of each vertex of any ordered tree R we can obtain a canonically-ordered tree r-isomorphic to R
- It follows that **cws**$(\cdot)$ is a valid representation for rooted trees
  - We may represent any rooted tree by a "unique" canonically-ordered tree (up to vertex labelling)
  - It is easy to see that **cws**$(\cdot)$ is lexicographically maximal
- The weight sequence of the tree in Figure 5 is 10 6 4 1 1 1 1 2 1 1

Centroids

- Weight sequences for *free* trees: we need a unique representative from each f-isomorphism class of ordered trees
- A *centroidal* vertex u of T is a vertex such that each component of the forest $T - u$ is of order at most $\frac{n}{2}$
- A tree is either unicentroidal or bicentroidal, having two adjacent centroidal vertices
  - unicentroidal: largest component of $T - u$ is of order at most $\frac{n-1}{2}$
  - bicentroidal: the largest component of $T - u$ is of order $\frac{n}{2}$
- The centroids of two f-isomorphic free trees must map to each other under any f-isomorphism
- Therefore consider the two types of free tree separately

Unicentroidal trees

- The *free weight sequence* $\mathbf{fws}(T)$ of a unicentroidal free tree T is the weight sequence of any canonically-ordered tree that is f-isomorphic to T and *rooted at its centroid*
- Easy to show that this is a valid representation for unicentroidal free trees
  - We may represent any unicentroidal free by a unique canonically-ordered tree rooted at its centroid
  - Figure 6 shows this representation for the free tree T that is f-isomorphic to the trees in Figures 4 and 5

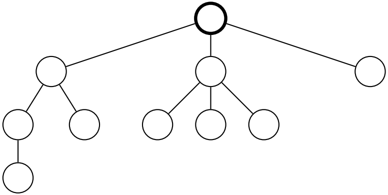


Figure 6: $\mathbf{fws}(T) = 10\,4\,2\,1\,1\,4\,1\,1\,1\,1$

Bicentroidal trees

- Let T_u and T_v be the rooted trees obtained by deleting the edge between the centroidal vertices u and v, where we assume that $\mathbf{cws}(T_u) \geq \mathbf{cws}(T_v)$
- The *free weight sequence* of T is the concatenation of $\mathbf{cws}(T_u)$ and $\mathbf{cws}(T_v)$, i.e., $\mathbf{fws}(T) = \mathbf{cws}(T_u) \oplus \mathbf{cws}(T_v)$
- Easy to show this is a valid representation for bicentroidal trees
- Therefore $\mathbf{fws}(T)$ is a valid representation for free trees

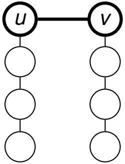


Figure 7: $\mathbf{fws}(P_8) = 4\,3\,2\,1\,4\,3\,2\,1$

Algorithm notation

- $\mathcal{B}(n)$ denotes the relex (*reverse lexicographically*) ordered set of the canonical weight sequences of all rooted trees of order n
  - Let $\mathcal{B}_q(n) \subseteq \mathcal{B}(n)$ denote the canonical weight sequences for which the first subtree of the root is of order q
  - $\mathcal{B}_q(n)$ contains the sequences for which the second element is q
- If $\mathbf{s} = s_1\, s_2\, \ldots s_n$ is an integer sequence, then $\mathbf{s}^{\#} = s_2\, s_3\, \ldots s_n$
  - So if $\mathbf{s}$ is the weight sequence of an ordered tree R, then $\mathbf{s}^{\#}$ is the weight sequence of the ordered *forest* obtained by removing the root of R
- We write $\mathbf{s} \succcurlyeq \mathbf{t}$ if $\mathbf{t}$ is some other sequence such that either $\mathbf{s} \geq \mathbf{t}$ or $\mathbf{s}$ is a prefix of $\mathbf{t}$, i.e., $\mathbf{t} = \mathbf{s} \oplus \mathbf{x}$ for some sequence $\mathbf{x}$

Main algorithm idea

- An alternative decomposition of the weight sequence is given by

$$\mathbf{ws}(R) = n \oplus \mathbf{ws}(R(u_1)) \oplus \mathbf{ws}(R - R(u_1))^{\#}$$

- Let $\mathcal{A}_q(n) = \{<\mathbf{a}, \mathbf{b}> \in \mathcal{B}(q) \times \mathcal{B}(n-q) \mid \mathbf{a} \succcurlyeq \mathbf{b}^{\#}\}$ and define β by $\beta(<\mathbf{a}, \mathbf{b}>) = n \oplus \mathbf{a} \oplus \mathbf{b}^{\#}$
  - We can prove that β is a bijection from $\mathcal{A}_q(n)$ to $\mathcal{B}_q(n)$ as the weight sequences are canonical
  - So $\mathcal{B}(n)$ can be constructed from the $\mathcal{B}(q)$ for $1 \leq q \leq n-1$
- To generate all rooted trees, for each q and for each $\mathbf{a}$ in $\mathcal{B}(q)$, we therefore need to find those $\mathbf{b}$ in $\mathcal{B}(n-q)$ for which $\mathbf{a} \succcurlyeq \mathbf{b}^{\#}$
  - Then form the integer sequence $n \oplus \mathbf{a} \oplus \mathbf{b}^{\#}$ to obtain the appropriate element of $\mathcal{B}(n)$
  - Easy to see we only need to consider those elements that are in $\mathcal{B}_r(n-q)$, where $1 \leq r \leq \min(n-q-1, q)$

Birkbeck University of London

Function RootedTrees(n)

    if $n = 1$ **then return** [1]

    Bn $\leftarrow$ [ ]

    for q **from** $n - 1$ **downto** 1 **do** Bn $\leftarrow$ Bn $\oplus$ RTHelper(n, q)

    return Bn

Function RTHelper(n, q)

    Bqn $\leftarrow$ [ ]

    $newq \leftarrow \mathtt{min}(n - q - 1, q)$

    if $newq = 0$ **then**

        for a **in** RootedTrees(q) **do** Bqn $\leftarrow$ Bqn $\oplus$ [$n \oplus$ a]

    else

        for a **in** RootedTrees(q) **do**

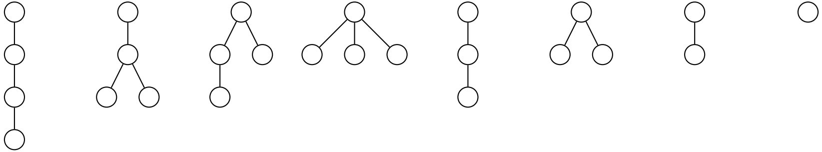
            for r **from** $newq$ **downto** 1 **do**

                for b **in** RTHelper($n - q, r$) **do**

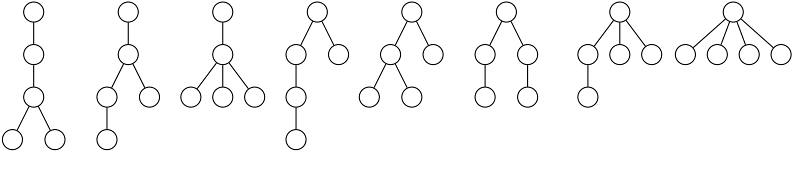
                    if a $\succcurlyeq$ b$^{\#}$ **then** Bqn $\leftarrow$ Bqn $\oplus$ [$n \oplus$ a $\oplus$ b$^{\#}$]

    return Bqn

All rooted trees of order at most 5



a *b* *c* *d* *e* *f* *g* *h*



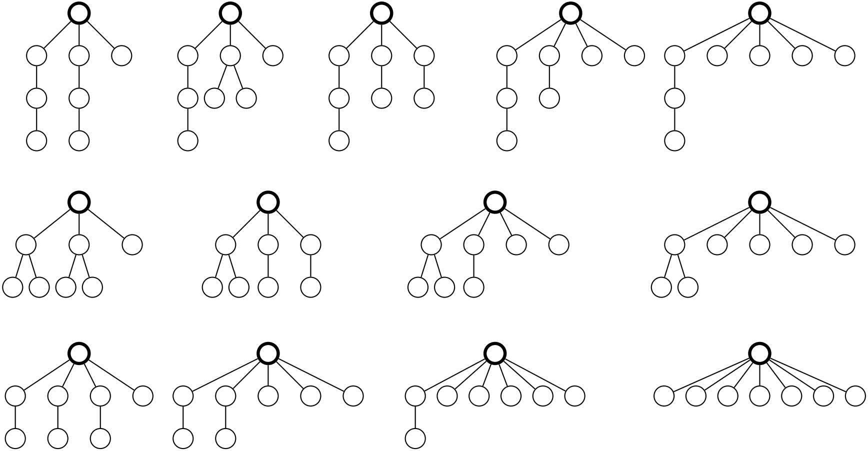
ah *bh* *ch* *dh* *eg* *fg* *ge* *gf* *hd*

Unicentroidal free tree generation

- $\mathcal{F}_U(n)$ denotes the relex ordered set of the free weight sequences of all unicentroidal free trees of order n
- In the canonically ordered tree that represents a unicentroidal free tree, the sub-trees of the root are of order at most $\frac{n-1}{2}$
- So the mapping β above is a bijection from $\bigcup_{i=1}^{maxq} \mathcal{A}_q(n)$ to $\mathcal{F}_U(n)$, where $maxq = \left\lfloor \frac{n-1}{2} \right\rfloor$

Function UFT(n)
  if $n = 1$ **then return** [1]
  UFn $\leftarrow$ [ ]
  $maxq \leftarrow \left\lfloor \frac{1}{2}(n-1) \right\rfloor$
  for q **from** $maxq$ **downto** 1 **do** UFn $\leftarrow$ UFn $\oplus$ RTHelper(n, q)
  return UFn

Unicentroidal free trees of order 8



Bicentroidal free tree generation

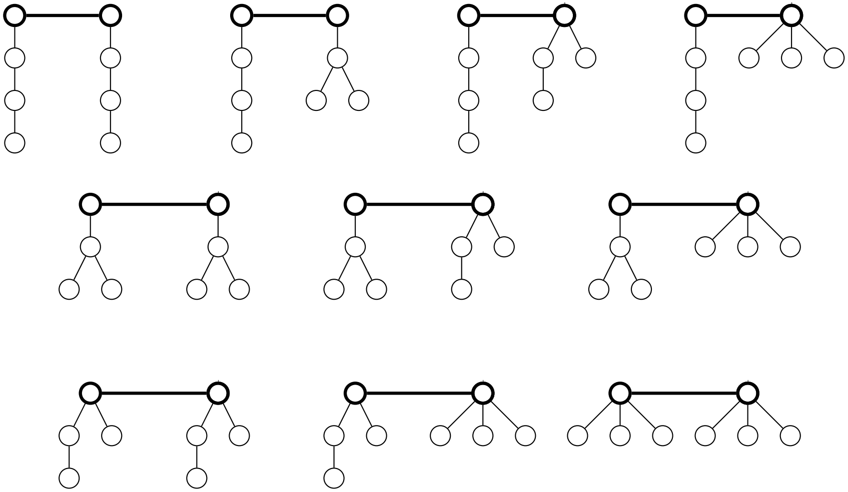
- $\mathcal{F}_B(n)$ denotes the relex ordered set of the free weight sequences of all bicentroidal free trees of order n
- For any n, $\mathcal{F}_B(n)$ can be constructed from the set $\mathcal{B}(\frac{n}{2})$

Function BFT(n)
  BFn $\leftarrow$ [ ]
  for a1 **in** RootedTrees($\frac{n}{2}$) **do**
    for a2 **in** RootedTrees($\frac{n}{2}$) **do**
      if a1 $\geq$ a2 **then** BFn $\leftarrow$ BFn $\oplus$ [a1 $\oplus$ a2]
  return BFn

Function FreeTrees(n)
  return UFT(n) $\oplus$ BFT(n)

Birkbeck UNIVERSITY OF LONDON

Bicentroidal free trees of order 8



Improvements to the algorithms

- We store the weight sequences as strings instead of lists, using $1, 2, \ldots, 9, A, B, C, \ldots$ for weights $1, 2, \ldots 9, 10, 11, 12, \ldots$
- We return generators instead of lists in the algorithms
- We make the following additional improvements to RTHelper
  - There is only one rooted tree of order 1 and one of order 2, so we may compute the result explicitly when q is 1 or 2
  - When $q = n - 2$, the second subtree of the root contains just a single vertex, so $\mathbf{b}^{\#}$ is just 1 in this case
  - Checking whether $\mathbf{a} \succcurlyeq \mathbf{b}^{\#}$ is only necessary when r, the order of the second child of the root, equals q, the order of the first child
  - After $\mathbf{a} \succcurlyeq \mathbf{b}^{\#}$ for the first time, this also holds for all subsequent $\mathbf{b}$, removing the need to check whether $\mathbf{a} \succcurlyeq \mathbf{b}^{\#}$ from then on

Function RTHelper2(n, q)

 if $q = 1$ **then return** $n \oplus 1^{n-1}$
 Bqn $\leftarrow$ [ ]
 if $q = 2$ **then**
  for t **from** $\lfloor \frac{1}{2}(n-1) \rfloor$ **downto** 1 **do**
   Bqn $\leftarrow$ Bqn $\oplus$ [$n \oplus (2\,1)^t \oplus 1^{n-1-2t}$]
 else if $q = n - 1$ **then**
  for a **in** RootedTrees(q) **do** Bqn $\leftarrow$ Bqn $\oplus$ [$n \oplus$ a]
 else if $q = n - 2$ **then**
  for a **in** RootedTrees(q) **do** Bqn $\leftarrow$ Bqn $\oplus$ [$n \oplus$ a $\oplus$ 1]
 else
  $newq \leftarrow \min(n - q - 1, q)$
  for a **in** RootedTrees(q) **do**
   for r **from** $newq$ **downto** 1 **do**
    RTHelperList $\leftarrow$ RTHelper2($n - q, r$)
    $start \leftarrow 1$
    if $r = q$ **then**
     for b **in** RTHelperList **do**
      if a $\succcurlyeq$ b$^{\#}$ **then break**
      $start \leftarrow start + 1$
    for b **in** RTHelperList[$start \ldots$] **do**
     Bqn $\leftarrow$ Bqn $\oplus$ [$n \oplus$ a $\oplus$ b$^{\#}$]
 return Bqn

Caching I

- We can obtain a significant increase in the efficiency of RTHelper2(n, q) if we cache in memory $\mathcal{B}(k)$, for $1 \leq k \leq q$
- RootedTrees(n) calls RTHelper2(n, q), for $1 \leq q \leq n - 1$
  - There are nearly 10^{12} rooted trees of order 31, so generating all rooted trees of order 32 would require at least 10 terabytes
- For free trees, we only need to cache $\mathcal{B}(k)$ for $1 \leq k \leq \frac{n}{2}$ to avoid all calls of RootedTrees for both UFT(n) and BFT(n)
  - UFT(n) calls RTHelper2(n, q), for $1 \leq q \leq \left\lfloor \frac{n-1}{2} \right\rfloor$
  - BFT(n) only calls RootedTrees($\frac{n}{2}$)
  - Although there are around 11×10^{10} free trees of order 32, there are fewer than $380,000$ rooted trees of order 16 or less

Caching II

- To avoid calling RootedTrees, we cache $\mathcal{B}(k)$ for $1 \leq k \leq L$, and store the sets $\mathcal{B}(1), \mathcal{B}(2), \ldots, \mathcal{B}(L)$ in the list RTList
- We choose $L \geq \left\lfloor \frac{n}{2} \right\rfloor + 1$ to improve efficiency when $q = \left\lfloor \frac{n-1}{2} \right\rfloor$
- We also use RTList instead of RTHelper2 when $q \geq n - L$
- When $n - L \leq q < \left\lfloor \frac{n-1}{2} \right\rfloor$, we can skip some of the initial elements of RTList[$n - q$] by pre-computing another small array
- Furthermore, we can dispense with checking whether $\mathbf{a} \succcurlyeq \mathbf{b}^{\#}$:
  - when $q \geq \left\lfloor \frac{n+1}{2} \right\rfloor$, since $n - q - 1 < q$
  - when $q = \left\lfloor \frac{n-1}{2} \right\rfloor$, by skipping the initial elements of RTList[$n - q$]
- We avoid removing the first element of **b** by also caching the set of sequences obtained by replacing each sequence **b** by $\mathbf{b}^{\#}$
- We also improve the efficiency of BFT by using the same idea as that used for the case $q = \left\lfloor \frac{n-1}{2} \right\rfloor$

Birkbeck
UNIVERSITY OF LONDON

Function RTHelper3(n, q)

. . . . # as RTHelper2 #

else if $q = n - 2$ **then**

  for a **in** RTList[q] **do** Bqn $\leftarrow$ Bqn $\oplus$ [$n \oplus$ a $\oplus$ 1]

else if $q \geq \left\lfloor \frac{n+1}{2} \right\rfloor$ **then**

  for a **in** RTList[q] **do**

    for b **in** RTList[$n - q$] **do** Bqn $\leftarrow$ Bqn $\oplus$ [$n \oplus$ a $\oplus$ b$^{\#}$]

else if $q = \left\lfloor \frac{n-1}{2} \right\rfloor$ **then**

  start $\leftarrow$ 1

  if $n \equiv 0 \pmod 2$ **then** *start* $\leftarrow$ length(RTList[$\frac{n}{2}$]) + 1

  for a **in** RTList[q] **do**

    for b **in** RTList[$n - q$][*start* . . .] **do**

      Bqn $\leftarrow$ Bqn $\oplus$ [$n \oplus$ a $\oplus$ b$^{\#}$]

    start $\leftarrow$ *start* $+ 1$

```
else if q ≥ n − L then
    start ← RTqstart[n − q, q]
    for a in RTList[q] do
        for b in RTList[n − q][start . . .] do
            if a ⋡ b# then break
            start ← start + 1
        for b in RTList[n − q][start . . .] do
            Bqn ← Bqn ⊕ [n ⊕ a ⊕ b#]
else
    for a in RTList[q] do
        for r from q downto 1 do
            RTHelperList ← RTHelper3(n − q, r)
            . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . # as RTHelper2 #
return Bqn
```

Adjacency lists and matrices

- For most purposes a more conventional representation, such as adjacency lists or adjacency matrices, is required
- We have created simple algorithms that return the adjacency list or matrix of a free tree given its weight sequence
- We can speed up the generation of the adjacency list / matrix representations by caching these for small values of n
- We can then construct the adjacency list / matrix representations while we construct their weight sequences
  - We retrieve the adjacency list / matrix for the subtree, and then change the vertex label / column number appropriately

Results

- We implemented our algorithms in Python and compared these with the Python implementation of the WROM algorithm taken from NetworkX
  - All computations were performed using Python 3.7 and the JIT compiler PyPy3.6-v7.3.1, using a Pentium i7 with 16GB RAM
- The runtimes for generating the weight sequences using BRFE are less than a quarter of those for generating the level sequences using WROM
  - Compilation time overhead signifcantly affects the times for smaller values of n
  - Runtimes for adjacency list and matrix representations are less than a third of those for generating these using WROM
  - The timings in the table are for $L = \frac{n}{2} + 1$
  - Increasing the value of L can significantly reduce runtimes, e.g., the runtime is 556 seconds when $n = 29$ and $L = 19$

Runtimes in seconds of the implementations of the BRFE and WROM algorithms

n	No. of trees	BRFE wt. seq	WROM level seq	BRFE list	WROM list	BRFE matrix	WROM matrix
18	123867	0.05	0.22	0.21	0.33	0.18	0.38
19	317955	0.08	0.29	0.29	0.53	0.26	0.69
20	823065	0.12	0.53	0.47	1.14	0.44	1.58
21	2144505	0.28	1.10	0.93	2.62	0.84	3.82
22	5623756	0.53	2.64	2.02	6.79	1.86	10.12
23	14828074	1.61	6.50	5.33	17.95	4.66	27.34
24	39299897	3.41	17.72	13.56	48.41	12.34	75.49
25	104636890	11.09	49.73	40.47	129.57	37.29	207.03
26	279793450	24.24	127.31	104.66	364.85	102.16	594.39
27	751065460	80.26	336.57	319.91	–	288.38	–
28	2023443032	174.68	–	–	–	–	–
29	5469566585	657.46	–	–	–	–	–

Complexity comments

- Considering small graphs, $n \leq 40$
- So assume uniform cost - constant time string operations
- Runtimes of the functions approximately proportional to the number of trees returned but
  - Extra time for sequences skipped until $\mathbf{a} \succcurlyeq \mathbf{b}^{\#}$
  - This increases runtimes per string by less than 5%
  - Less time per tree for even n as faster for bicentroidal trees
  - Compilation overhead increases the times per tree for small n
  - Much slower with the CPython interpreter but less fluctuation
- Generation times increasing slightly with n
  - In reality, string operation times depend on string length
  - Dividing by n gives more stable generation times
  - Increases for larger n possibly due to hardware cache size

Generation time per tree in nanoseconds

n	No. of trees	BRFE wt. seq	WROM level seq	BRFE list	WROM list	BRFE matrix	WROM matrix
18	123867	403.66	1776.10	1695.37	2664.15	1453.17	3067.81
19	317955	251.61	912.08	912.08	1666.90	817.73	2170.12
20	823065	145.80	643.93	571.04	1385.07	534.59	1919.65
21	2144505	130.57	512.94	433.67	1221.73	391.70	1781.30
22	5623756	94.24	469.44	359.19	1207.38	330.74	1799.51
23	14828074	108.58	438.36	359.45	1210.54	314.27	1843.80
24	39299897	86.77	450.89	345.04	1231.81	314.00	1920.87
25	104636890	105.99	475.26	386.77	1238.28	356.38	1978.56
26	279793450	86.64	455.01	374.06	1304.00	365.13	2124.39
27	751065460	106.86	448.12	425.94	–	383.96	–
28	2023443032	86.33	–	–	–	–	–
29	5469566585	120.20	–	–	–	–	–

Generation times per tree divided by n

n	No. of trees	BRFE wt. seq	WROM level. seq	BRFE list	WROM list	BRFE matrix	WROM matrix
18	123867	22.43	98.67	94.19	148.01	80.73	170.43
19	317955	13.24	48.00	48.00	87.73	43.04	114.22
20	823065	7.29	32.20	28.55	69.25	26.73	95.98
21	2144505	6.22	24.43	20.65	58.18	18.65	84.82
22	5623756	4.28	21.34	16.33	54.88	15.03	81.80
23	14828074	4.72	19.06	15.63	52.63	13.66	80.17
24	39299897	3.62	18.79	14.38	51.33	13.08	80.04
25	104636890	4.24	19.01	15.47	49.53	14.26	79.14
26	279793450	3.33	17.50	14.39	50.15	14.04	81.71
27	751065460	3.96	16.60	15.78	–	14.22	–
28	2023443032	3.08	–	–	–	–	–
29	5469566585	4.14	–	–	–	–	–

Concluding remarks

- We have presented new canonical representations for ordered, rooted and free trees: the weight sequence
- We constructed recursive algorithms for generating all rooted trees and all free trees of order n using these representations
- We made a number of improvements to the algorithms and their Python implementations, in particular, caching the lists of rooted trees of small order
- The runtimes for the new algorithm are less than a quarter of those for the WROM algorithm, with similar results for adjacency lists and matrices