Distinct Distance Forests

Dakotah Kurtz

November 21, 2024

1 Introduction

A *tree* is an undirected graph with the property that any two vertices are connected by only one path. By assigning a weight (or distance) to each edge of a tree, we obtain a *weighted tree*. It is then possible to consider the distance of a path, obtained by summing the weights on each edge of the path. Various sub-categories of weighted trees are of interest to mathematicians and have applications in a range of fields. This paper discusses the results of a computer search for instances of weighted trees and sets of trees, called *weighted forests*, that meet various criteria, and the algorithm used to do so.

2 Background

2.1 Definitions and Inspiration

A weighted tree where every path has a unique weight is a *distinct distance tree* (DDT). A *perfect distance tree* (PDT) on n vertices is a weighted tree that contains every weighted path in $\{1, 2, ..., \binom{n}{2}\}$. The following image, obtained from Calhoun and Polhill [1], show the only known instances of PDT.

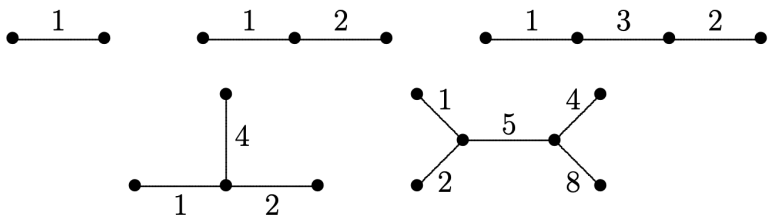

Figure 1: Perfect Distance Trees

In [1], Calhoun and Polhill proceed to define a *minimal distinct distance tree* as a distinct distance tree with a *maximal distance* M(n). Here, I define maximal distance in the following way.

Definition: Maximal Distance - Let F be a distinct distance forest, with P the set of distances that can be reached through paths in F. The *maximal distance* of F is the largest value in P.

Definition: Minimal Distinct Distance Tree - Let F be a distinct distance tree on n vertices, with maximal distance m. F is a *minimal distinct distance tree* iff for all distinct distance trees F' on n vertices with maximal distance m', $m \leq m'$.

Then M(n) is the maximal distance of a minimal distinct distance tree on n vertices.

This project achieved the following goals:

1. Rule out the existence of perfect distance trees on $2 \leq$ n ≤ 16 vertices.
2. Determine M(n) for $2 \leq$ n ≤ 16.
3. Find the minimum number of trees required to form a perfect distance forest on $2 \leq$ n ≤ 16 vertices.

4. For $2 \leq n \leq 16$ vertices, find the distinct distance tree on n vertices that maximizes the smallest missing distance, which is here called a *nearest perfect distance tree.*

Definition: Perfect Distance Forest -

Let F be a distinct distance forest containing k distinct distance trees. Let $T = \{t_1, t_2, ..., t_k\}$ be those disjoint trees. Let $N = \{n_1, n_2, ..., n_k | n_i$ is the number of vertices in $t_i\}$. F is a *perfect distance forest* on $\sum_{i=1}^{k} n_i$ vertices if F contains a path of every distance $\{1, 2, ..., \sum_{i=1}^{k} \binom{n_i}{2}\}$.

Definition: Nearest Perfect Distance Tree -

For a *distinct distance tree* T, let P be the set containing the distance of each path in T. T is a *nearest perfect distance tree* iff for all distinct distance trees T' on n vertices with P' the set of distances, $\min(\{1, 2, ..., \binom{n}{2}\} / P) \geq \min(\{1, 2, ..., \binom{n}{2}\} / P')$

2.2 Algorithm Outline

In general, the Distinct Distance Forest Generating Algorithm (DFGA) is a brute-force recursive algorithm. The primary data structure used is a linked-list based stack that follows first-in-last-out protocol. A sketch of the DFGA is provided here, with details in the upcoming sections mentioned. Beginning with the simplest possible forest F on two vertices (Figure 2), DFGA performs the following steps: **Note:** we can consider a tree as a forest with only one disjoint set of vertices.

1. Determine if F meets any of the desired criteria (see 3.1).
2. Determine the smallest integer w that cannot be achieved with a path through the F (see 3.3).
3. Generate every possible distinct distance forest (DDF) that can be obtained by adding to F a new vertex attached to the tree by an edge weighted w (see 3.2.1).
4. If F contains two or more disjoint trees, generate every possible DDF that can arise by connecting the vertices of two disjoint trees in F by an edge weighted w (see 3.2.2).
5. Generate a new forest by adding to F a tree containing only two vertices connected by edge weighted w (see 3.2.1).
6. Add all of the newly generated DDFs to the stack. Let F now denote the most recently added DDF and return to step 1.

As mentioned above, DFGA operates on DDF with a FILO protocol. With that in mind, it is clear that while (Figure 2) below gives a sample of the first few cycles of DDF generation, it does not accurately show the order in which DDF are generated.

3 Implementation Details

When the Distinct Distance Forest Generating Algorithm (DFGA) acquires a distinct distance forest F from the stack, the first step is to determine the smallest distance that cannot be reached through a path in F. This is easily achieved, since the list of possible distances of F is updated upon every change to F. This also allows for easily verifying distinctness of path distances. To properly segment the description of DFGA, the following steps assume an updated list containing all possible path distances achievable in F. Details of updating the path distances of F can be found in Section 3.3.

3.1 Meeting Criteria

There are billions of distinct distance forests (DDF) on $n \leq 16$ vertices. Rather than store them all, only DDF that meet one of the criteria listed in 2.1 for some number of vertices are recorded. That is, for each search criteria allocate a two-dimensional array to store forests; one dimension distinguishes the number of vertices and the other to track cases where multiple DDF with the same number of vertices equally fulfill some given goal.

Perfect Distance Forest: Minimize Number of Trees

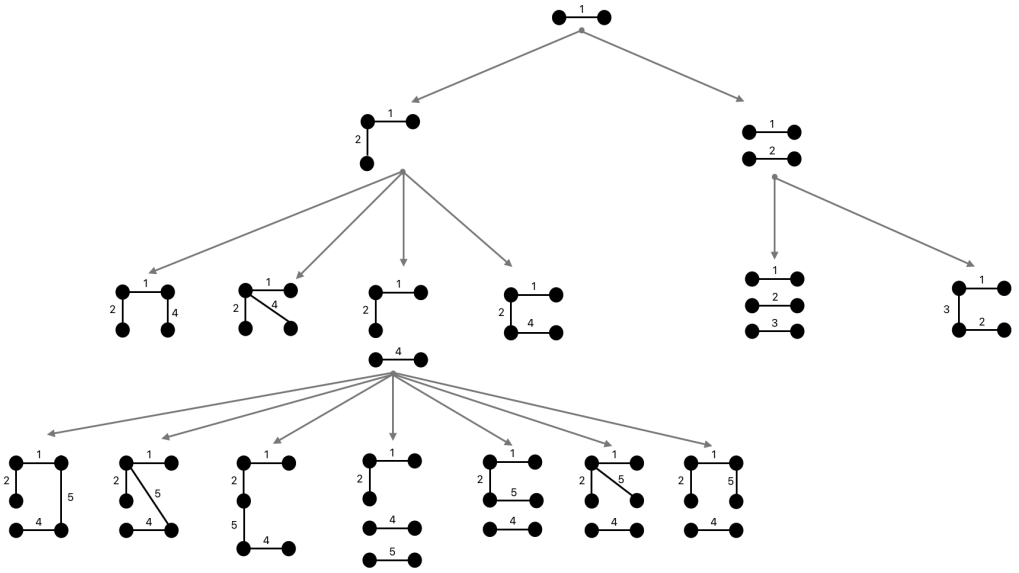

Figure 2: Distinct Distance Forest Generation

A perfect distance forest F on n vertices will contain every path distance in $\{1, 2, ..., l = \sum_{i=1}^{k} \binom{n_i}{2}\}$. If F has no paths with distance $> l$, then F is a perfect distance forest. Since we are searching for the minimum number of trees needed to form a PDF, if F consists of the same or fewer distinct distance trees as the currently stored PDF on n vertices, then F meets the criteria and is stored.

Minimize the Maximum Distance

Let F be a distinct distance forest on n vertices containing only one tree. The maximal distance of F, denoted m, can be obtained through the maintained list of path lengths. If m is less than or equal to the maximal distance of the current lowest maximal distance found on n vertices, then F meets the criteria and is stored.

Nearest Perfect Distance Tree

Let F be a distinct distance forest on n vertices containing only one tree. The updated list of possible path lengths in F is iterated through to find the smallest missing value m. If m is greater than or equal to the current smallest missing value in a forest on n vertices, then F is stored.

3.2 Generating Forests

Following the FILO policy, DFGA obtains a distinct distance forest F, with some smallest value w that can't be achieved in any path through F. Let $T = \{t_1, t_2, ..., t_k\}$ be the set of k disjoint distinct distance trees in F. Let $V = \{V_1, V_2, ..., V_k | V_i$ be the vertices in $t_i\}$. In implementation, these sets are stored as attributes/fields of F. Let $n_i = |V_i|$, and index each vertex $V_{i_j} \in V_i$, noting $1 \leq j \leq n_i$.

Note: The following description of DFGA, while detailed, does omit edge cases and is not a perfect representation of the algorithm regardless. For further details, see attached code.

3.2.1 Adding Vertices

For each $t_i \in T$, iterate through the n_i vertices V_i in t_i. For each $1 \leq j \leq n_i$, create a new forest F', where $F' = F$. Connect a new vertex to v_{i_j} by an edge weighted w. Update the list of possible path distances in F' with the procedure found in Section 3.3, and add F' to the stack, provided the modified path distances are all still distinct.

A distinct distance forest F' can also be extended from F by adding a new disjoint set t_{k+1} to T in F'. t_{k+1} contains two vertices connected by an edge weighted w, and no other V_i in V are incident

with V_{k+1}. For this reason, F' constructed in this way from a DDT will always be a DDT and the path distances in F' can be updated simply by adding w to the list from F.

3.2.2 Connecting Disjoint Trees

Additionally, so long as $k \leq 2$, DDT F can be built upon by adding an edge e weighted w between any two vertices with no path between them. In each of the following iterations, a new forest $F' = F + e$ is formed.. For each t_i, t_l in F where $i \text{ ¡ } l \leq k$, connect each vertex $\{V_{i_1}, V_{i_2}, ..., V_{i_{n_i}}\}$ to each vertex $\{V_{l_1}, V_{l_2}, ..., V_{l_{n_l}}\}$ with e. That is, generate $F_{i_a,l_b} = F + e$, where e connects vertex V_{i_a} in t_i to vertex V_{l_b} in t_b. Several iterations are shown in concert in (Figure 3) below, namely the potential DDF that may be formed by connecting V_{1_2} to each vertex in V_2.

Not all forests generated in this manner are DDF. Before adding F' to the stack for criteria testing and further propagation, the reachable path distances must be updated and distinctness ensured. Section 3.3 provides details regarding updating the path distances after connecting two disjoint DDF.

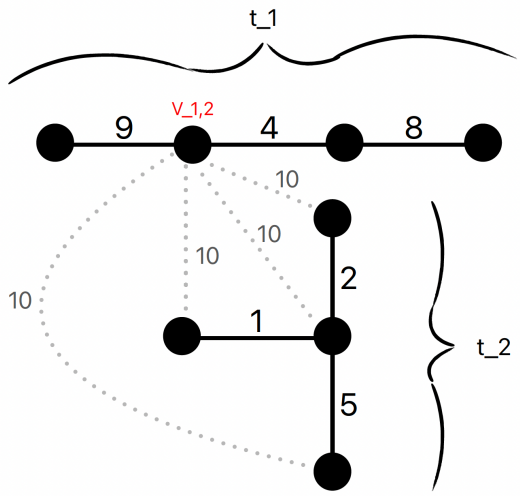

Figure 3: Connecting Disjoint Trees

3.3 Updating Path Distances

The description of the generation algorithm in Section 3.2 assumes a list of all possible path distances in a given forest is available. This mirrors the behavior of DFGA; updating path distances occurs before a forest is put back on the stack to eventually form the root of new DDF in some later iteration. There are three different cases for updating the paths in a DDF. In the trivial case (adding a disjoint tree with a single edge) adding the weight of that new edge to the path distances suffices. The other two cases deserve further explanation.

3.3.1 Adding Vertex - Updating Distances

Section 3.2.1 discusses the process of forming new potential DDFs by adding a new vertex to an existing DDF. Let F be the original DDF, with $L = \{l_1, l_2, ..., l_k\}$ the list of reachable path distances in F. Let w be the minimum missing weight in L. Then let F' be the forest reached by adding an edge weighted w to F by connecting a new vertex to some vertex v in some disjoint tree t in F. The path distances in F' different from L are exactly paths that include w. Let A represent those paths. By definition, w is in A. The subsequent values in A are determined by a breadth-first graph traversing algorithm, outlined as follows.

1. Let $v' = v$.

2. v' is connected to $k \geq 0$ vertices $\{v_1, v_2, ..., v_k\}$ that have **not yet been traversed**, by edges with weights $\{d_1, d_2, ..., d_k\}$. If $k = 0$, this branch of the recursion ends. Otherwise, for each v_i, add $w + d_i$ to A. Let $w = w + d_i$ and $v' = v_i$, then return to the beginning of step 2.

Once the recursive search has ended, the reachable path distances in F' can be found by $L + A$. This process is illustrated in (Figure 4).

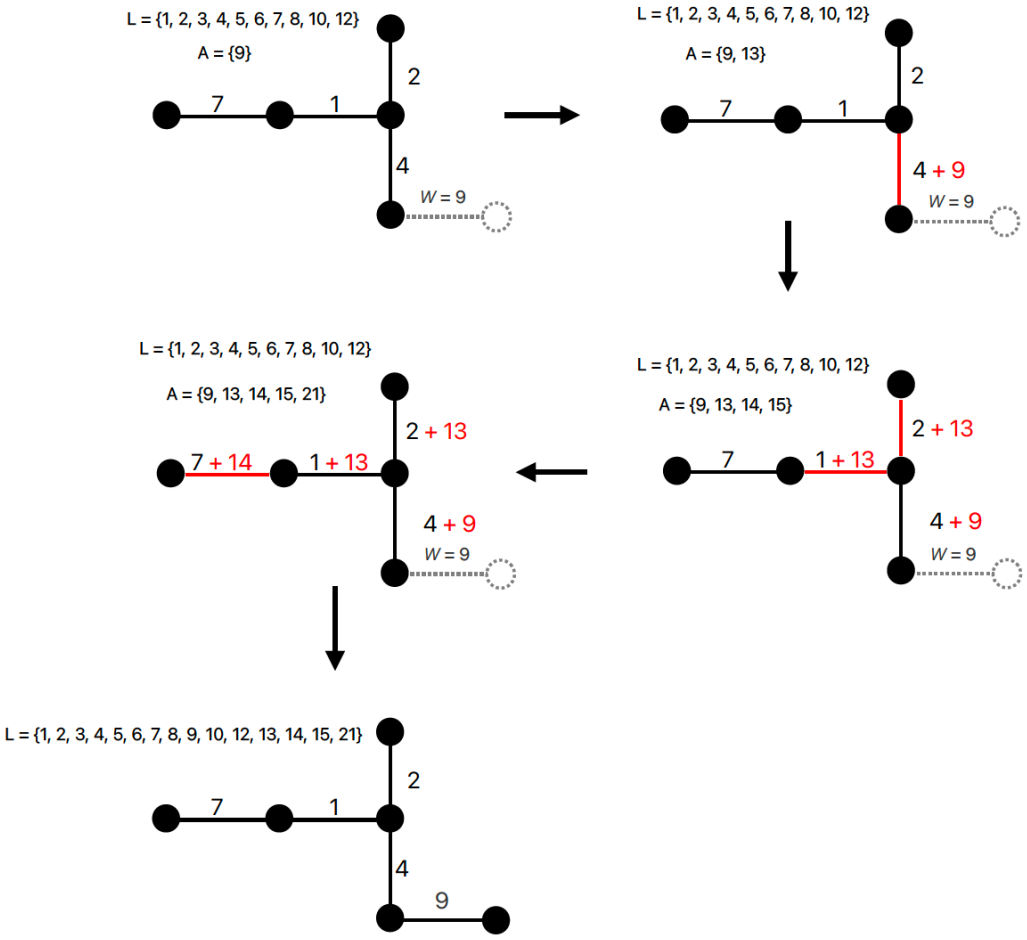

Figure 4: Updating Reachable Paths After Adding Vertex

3.3.2 Combining Trees - Updating Distances

Section 3.2.2 covers the generation of potential DDF by joining two disjoint trees. The following method can be used to update the path distances of a forest after such a joining. Let F be the initial DDF, with path distances L, and suppose F' is formed by connecting trees t_1, t_2 in F by adding an edge e weighted w between vertices v_1 in t_1 and v_2 in t_2. We need to choose a tree to start; without loss of generality choose t_2. Then, the general strategy will be to traverse t_2, finding the total distance of all paths in t_2 that end on v_2. These distances, denoted $D = \{d_1, d_2, ..., d_k\}$, can be obtained by applying the breadth-first traversal in Section 3.3.1, letting $v' = v_2$.

Let B be a list to store newly obtainable path distances. v_2 is connected to v_1 by an edge weighted w. Therefore, w is in B, and for all d_i in D, $d_i + w$ is in B. For each weight b_i in B, apply the breadth-first traversal algorithm from Section 3.3.1, using b_i as the weight and v_1 as the initial point. Let the resultant A from that algorithm be denoted A_i. Then the totality of new path distances in F' is simply

$$A = D \cup B \cup \bigcup_{i=1}^{k+1} A_i = \{x \mid x \in A_i \text{ for some } i \in I\}$$

Figure 5 somewhat outlines this process, joining two disjoint trees with $w = 13$. However, the graphic is inaccurate in that it shows the breadth-first algorithm being applied to all of B at once.

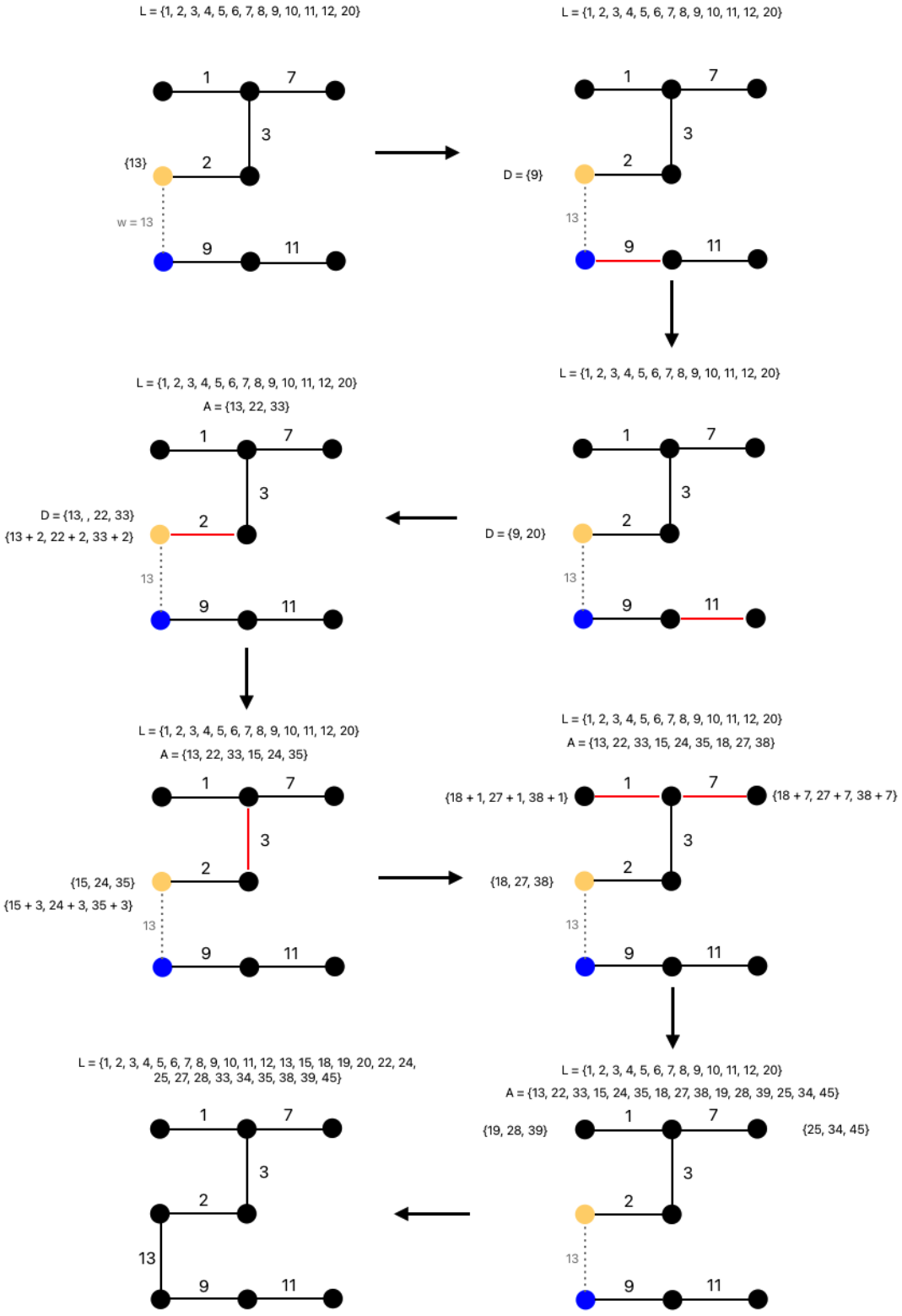

Figure 5: Updating Reachable Paths After Joining - (Blue Vertex to Gold, weight added: 13)

4 Results

Run-time constraints limited the search to forests on ≤ 16 vertices. Calculating the theoretical time complexity for an algorithm of this size is well beyond me, so I attempted to determined the runtime as a function of the number of vertices experimentally. The data measured is shown in Table 1.

Table 1: Algorithm Runtime

Number of Vertices	3	4	5	6	7	8	9
Time (s)	.001	.001	.004	.016	.071	.105	.218
Number of Vertices	10	11	12	13	14	15	16
Time (s)	.86	4.24	34	272	7049	53971	413994

Considering the nature of graph algorithms, it is reasonable to expect the time to grow exponentially with the number of vertices. Under that assumption, a line of best fit

$$f(n) = 2.857e^{2.0379n} * 10^{-9}$$

can be determined using the least squares method. If accurate, this would imply that my computer would take a little under a year working through $n = 17$.

This is disappointing; my results on the minimum number of disjoint sets required for perfect distance forests don't extend beyond that found in [1] by Calhoun and Polhill. Table 2 shows the minimum number of trees required to form a perfect distance forest on $3 \leq n \leq 16$ vertices, as well as how many unique instances of PDFs on n vertices exist.

Table 2: Perfect Distance Forests

Number of Vertices	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Minimum Trees	1	1	2	1	2	2	2	2	2	2	2	3	3	3
Instances	1	2	2	1	10	3	2	14	2	1	2	231	112	203

Figure 6 contains four examples of perfect distance forests. Forests labeled A and C on $n = 16$ vertices, contain paths with all weights in $[1, 43]$ and $[1, 37]$ respectively. Forests B and D are on $n = 13$ vertices, with all distances in $[1, 42]$ and $[1, 36]$ respectively reachable.

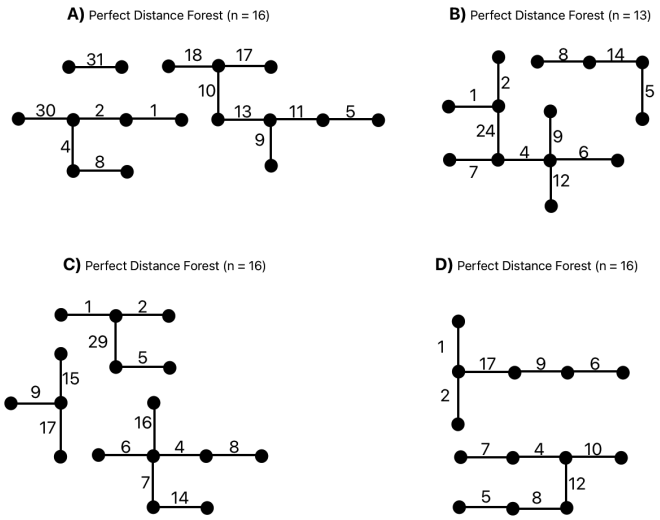

Figure 6: Perfect Distance Forests

Recall the working definition of minimal distinct distance tree; a forest F on n vertices is a minimal distinct distance tree if there are no forests on n vertices with maximal path less than that of F. By exhaustive computer search, M(n) and the number of minimal distinct distance trees on n vertices was determined for $2 \leq n \leq 16$. Of course, if $\mathrm{M(n)} = \binom{n}{2}$, then there exists a perfect distance tree on n vertices. This data is provided in Table 3.

Table 3: Minimum Maximal Distances

Number of Vertices	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\binom{n}{2}$	3	6	10	15	21	28	36	45	55	66	78	91	105	120
M(n)	3	6	11	15	22	30	39	52	68	76	103	125	150	172
Instances	1	2	5	1	1	1	1	1	3	1	1	1	1	1

The final distinct distance tree examined maximizes the lowest missing path distance in a tree of size n, here called nearest perfect distance trees. Similar to the previous case, if the minimum missing distance d in a forest of size n is such that $d = \binom{n}{2+1}$, then there exists a perfect distance tree of size n.

Table 4: Nearest Perfect - Smallest Missing Path

Number of Vertices	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\binom{n}{2} + 1$	4	7	11	16	22	29	37	46	56	67	79	92	106	121
Smallest Missing	4	7	10	16	21	26	31	38	46	48	61	58	69	69
Instances	1	2	3	1	1	1	2	2	1	1	1	1	1	4

It is worth noting that for $11 \leq n \leq 16$, the nearest perfect distance tree on n vertices is a weighted *path graph* on n vertices. That is, two vertices of the graph have degree one, and the others have degree two. Examples of this are shown in (Figure 7) below.

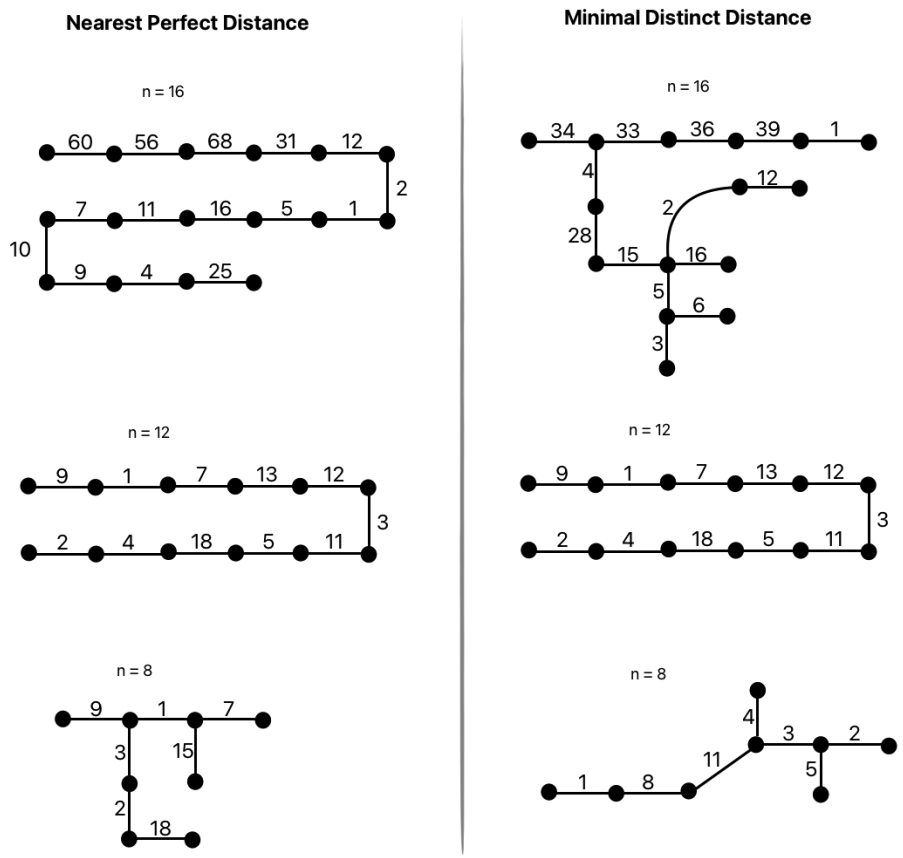

Figure 7: Distance Tree Comparison - Nearest Perfect and Minimal Distinct

References

[1] W. Calhoun, J. Polhill, Perfect distance forests, Australasian J. of Combinatorics 42 (2008), 211-222.