Python algorithms for hardwired clustering systems

Oliver Tryding

Handledare: Marc Hellmuth
Examinator: Lars Arvestad
Inlämningsdatum: 2022-05-22

Abstract

A phylogenetic network is a directed acyclic graph often used in modelling evolutionary relationships where a tree does not suffice. A cluster is a subset of a set X, and a clustering system is a set of clusters. There are many properties of clustering systems from the literature, that are described mathematically but cannot currently be tested on actual data. In this thesis we provide polynomial-time algorithms in Python for properties, ***closed****,* ***L****,* ***hierarchy****,* ***weak hierarchy****,* ***paired hierarchy****,* ***N3O****,* ***pre-binary****,* ***binary****,* ***2-Inc****. With these algorithms, other research in phylogenetics and clustering systems can be preformed using real data. Here we provide algorithms for these properties, prove that they work and give their time complexity. In addition we provide an algorithm for constructing a Hasse diagram. The algorithms in this thesis allows users familiar with Python to easily test properties of phylogenetic networks using the corresponding clustering systems.*

Sammanfattning

Ett phylogenetiska nätverk är en riktad acyklisk graph oftast använd för att modullera evolutionära relationer där träd inte räcker. Ett kluster är en delmängd av en mängd X, och ett klustringssystem är en mängd kluster. Det finns många egenskaper hos klusteringssystem från literaturen, som är beskrivna matematiskt men som för nuvarande inte kan bli testade på verklig data. I denna avhandling ges Python algoritmer i polynomisk tid för egenskaper, ***closed****,* ***L****,* ***hierarchy****,* ***weak hierarchy****,* ***paired hierarchy****,* ***N3O****,* ***pre-binary****,* ***binary****,* ***2-Inc****. Med dessa algoritmer kan annan forskning inom phylogeni och klustringssystem utföras med verklig data. Här ger vi algoritmer för dessa egenskaper samt bevisar att de är korrekta och ger deras tidskomplexitet. Desutom ger vi en algoritm för konstruktionen av Hassediagram. Algoritmerna i denna avhandling tillåter användare bekanta med Python att enkelt testa egenskaper hos phylogenetiska nätverk genom deras korresponderande klustringssystem.*

Contents

1. Introduction

When modelling the evolutionary history and relationships of different organisms biologists often make use of a tree structure. Sometimes, however, like in the cases of for example horizontal gene transfer, recombination or hybridization, this is not enough [6]. The networks used to model these more complex relationships are often called phylogenetic networks. A phylogenetic network is typically a directed acyclic graph (DAG) with a unique root vertex and no vertices with only one parent- and one child-vertex. Here we will mainly focus on the corresponding clustering system, or what in the literature is more commonly known as the hardwired clustering system, and related properties [5]. In Python we represent this as a list of sets.

The purpose of this thesis is to research the possibility of giving efficient working implementations of algorithms that test if a given clustering system satisfies some key properties for clustering systems. For the readers convenience, Table 1.1, will contain descriptions of all properties of clustering systems for which an algorithm will be provided in the results section of the thesis.

This paper is organized in the following manner.

First, the preliminaries in chapter 2. Nothing novel will be provided here, only some relevant definitions and theorems commonly seen in graph theory, phylogenetics and the clustering literature. We begin the preliminaries in section 2.1 with giving basic definitions from graph theory. Then In sections 2.2 and 2.3 we give all the relevant definitions of networks and clustering systems as well as some conventional results from the clustering literature. We finish the preliminaries chapter with section 2.4 where we give a very brief introduction to and discussion of Python and NetworkX, the Python library that will be used to create graph objects.

Next, in chapter 3, comes the actual results of this contribution. Here we give algorithms complete with proofs of correctness and time complexity. Some, if not most, of the algorithms provided make use of certain subroutines or make assumptions about the input. These will be provided as separate algorithms complete with proofs and include: a subroutine for comparing two sets. This is necessary for a mergesort-like sorting algorithm which is needed since most other algorithms assume the clustering system is sorted. We also provide an algorithm for constructing an overlap graph which once constructed makes several other algorithms significantly simpler and some even trivial. Finally we will also provide an algorithm for reconstructing the phylogenetic networks from their clustering systems using Hasse diagrams.

Finally in chapter 4 we provide a summary of the main results and an additional appendix containing proofs for a lemma used by many of the other algorithms and an alternative implementation of one algorithm.

	definition	def.
closed	$\forall_{c_1,c_2} \in C \mid c_1 \cap c_2 \neq \emptyset \implies c_1 \cap c_2 \in C.$	2.3.2
L	$\forall_{c_1,c_2,c_3} \in C \vert c_1 \cap c_2 = c_1 \cap c_3$ where c_1 overlaps with c_2, c_3.	2.3.3
hierarchy	$\forall_{c_1,c_2} \in C \vert c_1 \cap c_2 \in \{\emptyset, c_1, c_2\}.$	2.3.11
weak hierarchy	$\forall_{c_1,c_2,c_3} \in C \vert c_1 \cap c_2 \cap c_3 \in \{c_1 \cap c_2, c_1 \cap c_3, c_2 \cap c_3, \emptyset\}.$	2.3.5
paired hierarchy	Every $c \in C$ overlaps with at most one other cluster.	2.3.12
N3O	C contains no three pairwise overlapping clusters.	2.3.13
pre-binary	$\forall_{x_1,x_2} \in X$ there is a unique inclusion minimal $c \in C$ s.t $x_1, x_2 \in c$.	2.3.7
binary	pre-binary, and $\forall_c \in C \exists x_1, x_2 \in X$ s.t c is the unique inclusion minimal cluster with $x_1, x_2 \in c$.	2.3.8
2-Inc	$\forall_c \in C$ there are at most 2 inclusion maximal and inclusion minimal clusters $a, b \in C$ s.t $a, b \subsetneq c$ respectively $a, b \supsetneq c$.	2.3.9

Table 1.1.: Properties of clustering system C with leaf set X.

2. Preliminaries and Methods

2.1. Basic Definitions

We begin by giving some basic definitions for graphs and related properties. As some familiarity with graph theory is assumed these definitions and properties will be listed without examples.

Definition 2.1.1. Two sets A and B overlap if $A \cap B \neq \emptyset$ where $A \nsubseteq B$ or $B \nsubseteq A$.

Definition 2.1.2. An undirected graph G consists of a vertex set V and an edge set $E = \{\{u, v\} |\, u, v \in V \,\&\, v \neq u\}$. Where $\{u, v\}$ denotes a set and is therefore unordered.

Definition 2.1.3. A directed graph G consists of a vertex set V and an edge set $E = \{(u, v) |\, u, v \in V \,\&\, v \neq u\}$. Where (u, v) denotes a tuple and is therefore ordered.

Definition 2.1.4. The degree of a vertex v, denoted by deg(v), is the number of edges that are incident to v.

Definition 2.1.5. For directed graphs the indegree of a vertex v, denoted indeg(v), is the number of edges incident to v such that any such edge $e = (u, v)$ for some $u \neq v$ and similarly the outdegree of v, denoted outdeg(v), is the number of edges incident to v such that any such edge $e = (v, u)$ for some $u \neq v$.

Definition 2.1.6. A path P in a directed graph G, is a subgraph of G such that P contains vertices $v_1, v_2, ..., v_n$ where $n \geq 2$ and for $v_i, v_{i+1} \in P$ there is and edge (v_i, v_{i+1}) in G.

Definition 2.1.7. A cycle C in a directed graph G, is a subgraph of G such that C contains vertices $v_1, v_2, ..., v_n$ where $n \geq 2$ and for $v_i, v_{i+1} \in C$ there is and edge (v_i, v_{i+1}) in G and G also contains an edge (v_n, v_1).

Something worth noting here is that for undirected graphs the definitions are equivalent except all ordered tuples would become ordered sets.

Definition 2.1.8. A directed acyclic graph, also denoted DAG, is a directed graph that does not contain any cycles.

Definition 2.1.9. A graph, G, with vertex set V, is connected if every vertex $v \in V$ is incident to a different vertex $u \in V$.

Definition 2.1.10. A graph, G, is biconnected if any vertex can be removed while G remains connected.

Definition 2.1.11. For some set of sets, C, a set $m \in C$ is inclusion minimal with respect to $s \in C$ if $s \subset m$ where there is no $m' \in C$ such that $s \subset m'$ and $m' \subset m$.

Definition 2.1.12. For some set of sets, C, a set $m \in C$ is inclusion maximal with respect to $s \in C$ if $m \subset s$ where there is no $m' \in C$ such that $m' \subset s$ and $m \subset m'$.

2.2. Networks

We are now ready to define concepts and properties specifically relevant to this paper. We begin by defining networks and phylogenetic networks.

Definition 2.2.1. [5] A network $N = (V, E)$ is a directed acyclic graph (DAG), such that there is a unique vertex with indegree zero called the root.

Definition 2.2.2. [5] A network N is phylogenetic if and only if it is a network and there is no vertex with exactly one parent and one child vertex.

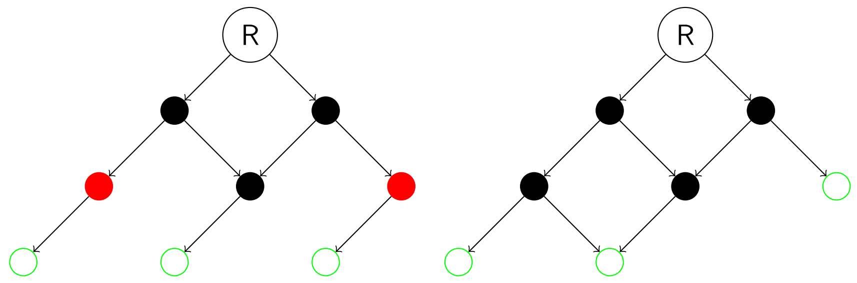

Figure 2.1.: Two networks. One non-phylogenetic network, with vertices marked in red having only one parent vertex and one child vertex, and one phylogenetic network. Leaf vertices are marked in green.

We also note that a vertex with an outdegree of zero is called a leaf and the set of all leaves is denoted with X.

Henceforth networks in this paper will be assumed to be phylogenetic and thus the phylogenetic part will not always be specified.

When studying networks a special type of vertex called hybrids can be of interest.

Definition 2.2.3. A vertex v is called hybrid if it has an indegree larger than or equal to two.

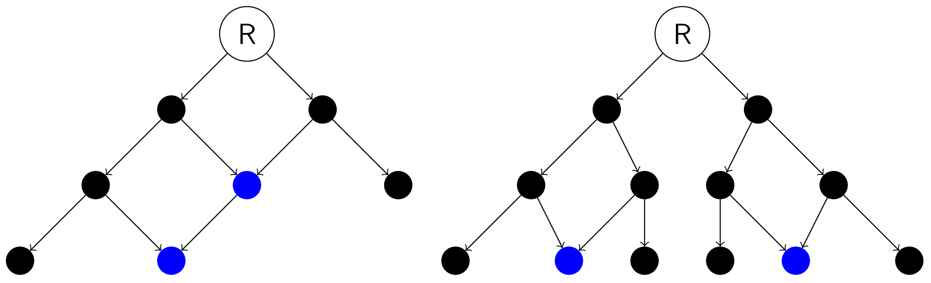

Figure 2.2.: Two network where hybrids are marked in blue.

With the definition of hybrids the level of a network can now be defined. Here we define level-k in the same way as [3]. This definition is equivalent to definition 3.24 from [5].

Definition 2.2.4. A level k network is a network where every maximal biconnected subgraph contains at most k hybrid vertices.

The network to the right in Figure 2.2 would then be a level-2 network as both of the hybrid vertices are contained in the same maximal biconnected subgraph. While the network to the left would be a level-1 network even though it has two hybrid vertices.

2.3. Clustering Systems

With all relevant definitions regarding graphs and networks, we can now define clustering systems.

Definition 2.3.1. [5] A (hardwired) clustering system C on a set X is a subset of the power set of X such that,

- $\emptyset \notin C$,
- $X \in C$,
- $x \in X$ implies $\{x\} \in C$.

It is also worth noting that an element in a clustering system is called a cluster.

Observe now that if X is the leaf set to a network, N, then a clustering system C_N, where for all $v \in N$ there is a $c \in C$ such that the elements of c are precisely the leaves to which there is a directed path from v, can be created. This will be how the relationship between clustering systems and networks are viewed in this paper.

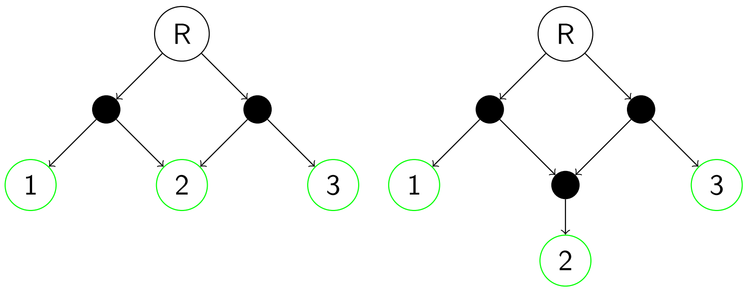

Figure 2.3.: Two different networks with the same clustering system $C = \{\{1, 2, 3\}, \{1, 2\}, \{2, 3\}, \{1\}, \{2\}, \{3\}\}$.

As can be seen from figure 4, not all clustering systems define a unique network. This is however not the purpose of this thesis and will not be explored further. Nor is it necessary, for the purposes of this thesis, to consider these networks as different since we will mostly deal with the corresponding clustering systems.

Now to the objective of this thesis; there are many properties of clustering systems that are of interest. The rest of this sections will contain some of these properties for which there are algorithms in the results section. We begin by defining the closed property.

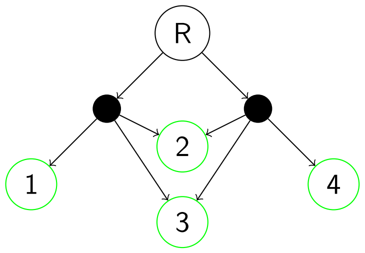

Figure 2.4.: A network, with clustering system $C_N = \{\{1, 2, 3, 4\}, \{1, 2, 3\}, \{2, 3, 4\}, \{1\}, \{2\}, \{3\}, \{4\}\}$, not satisfying the closed property.

Definition 2.3.2. A closed clustering system C is a clustering system where for all $c_1, c_2 \in C$ such that $c_1 \cap c_2 \neq \emptyset$ implies $c_1 \cap c_2 \in C$.

This is not the common definition of a closed clustering system but instead a well known result in clustering literature. It is however more practical for algorithmic purposes and so will be used as the definition of closed in this paper. For a more widely used definition and a proof that these are equivalent see Definition 3.31 and Lemma 3.32 from [5].

The next property to be defined is the property 'L'.

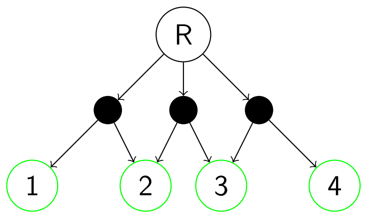

Figure 2.5.: A network, with clustering system $C_N = \{\{1,2,3,4\},\{1,2\},\{2,3\},\{3,4\}\{1\},\{2\},\{3\},\{4\}\}$, not satisfying the L property.

Definition 2.3.3. [5] A clustering system C satisfies property L if for all $c_1, c_2, c_3 \in C$: $c_1 \cap c_2 = c_1 \cap c_3$, where c_1 overlaps with c_2 and c_3.

If we wish to know when a clustering system is consistent with a level-1 network we can now do so. With both the closure property and L defined we can now give a theorem showing when this is the case.

Theorem 2.3.1. *[5] Let C be a clustering system. Then there is a level 1 network N such that $C_N = C$ if and only if C is closed and satisfies property L.*

For a proof of this theorem, see Theorem 7.38 in [5].

Definition 2.3.4. A Hasse diagram of a clustering system C is a DAG who's vertices are the elements in C. In the Hasse diagram an edge from vertex v to vertex w exist if an only if $w \subset v$ and there is no vertex u such that $w \subset u \subset v$.

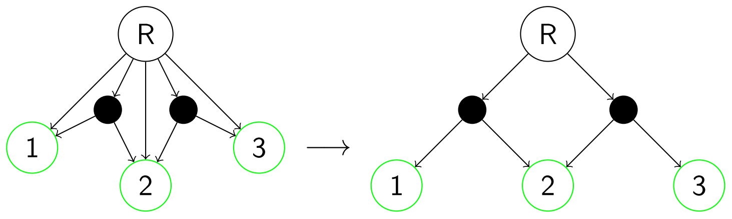

Figure 2.6.: A Hasse diagram being constructed from clustering system $C_N = \{\{1,2,3\},\{1,2\},\{2,3\},\{1\},\{2\},\{3\}\}$.

In figure 2.6 we see how a Hasse diagram can be constructed from a clustering system; first all edges between clusters contained in another are added, then any shortcuts are removed.

We can now define regular networks. For more of the properties of regular networks see section 3.4 and 3.5 in [5].

Definition 2.3.5. A network, N, is regular if and only if it is isomorphic the the Hasse diagram on its clustering system.

A theorem motivating the *is_level_1* algorithm is given in [5]. This theorem is the following.

Theorem 2.3.2. *For a closed clustering system satisfying property L, the corresponding Hasse diagram is a level 1 network.*

For a proof of this theorem see Proposition 7.36 from [5].

From Proposition 3.42 and Remark 1 in [5] we also have the following.

Lemma 2.3.3. *Every clustering system, C, has a corresponding unique regular network, N, that is isomorphic to the Hasse diagram on C.*

When a network is referred to as the corresponding network of some clustering system, C, it is the regular network isomorphic to the Hasse diagram on C.

The next property is that of 'weak hierarchy'. it is given separately from the other hierarchy properties only because the related algorithm does not use an overlap graph.

Definition 2.3.6. A clustering system, C, satisfies 'weak hierarchy' if for all $c_1, c_2, c_3 \in C$ it holds that $c_1 \cap c_2 \cap c_3 \in \{c_1 \cap c_2, c_1 \cap c_3, c_2 \cap c_3, \emptyset\}$.

It might be useful to recall the definitions of inclusion-minimal and inclusion maximal, from Definition 2.1.11 respectively Definition 2.1.12, for the next three properties. It is also important to note that for 'pre-binary' and 'binary' the leaves are not considered as clusters in the clustering system. The next property 'pre-binary' will be necessary for defining the 'binary' property.

Definition 2.3.7. [5] A clustering system, C, with leaf set X, satisfies 'pre-binary' if for every pair $x_1, x_2 \in X$ there is a unique inclusion-minimal cluster $c \in C$ such that $\{x_1, x_2\} \subseteq c$.

Definition 2.3.8. [5] A clustering system, C, with leaf set X, satisfies 'binary' if it is 'pre-binary' and for every $c \in C$ there is a pair $x_1, x_2 \in X$ such that c is the unique inclusion-minimal cluster containing x_1 and x_2.

Something to note here is that the 'binary' property on a clustering system in no way means that the corresponding network is binary in any sense. The true meaning of a binary clustering system will not be discussed in this thesis but for those interested it is discussed in depth in [1].

Definition 2.3.9. [5] A clustering system, C, satisfies '2-Inc' if for all clusters $c \in C$ there are at most two inclusion-maximal clusters $a, b \in C$ such that $a, b \subsetneq c$ and at most two inclusion-minimal cluster $a, b \in C$ such that $a, b \supsetneq c$.

The next three properties will all require the calculation of overlaps in the clustering system. Therefore in order to later simplify the algorithms for testing these properties we will make use of an auxiliary graph that shall be called an overlap graph.

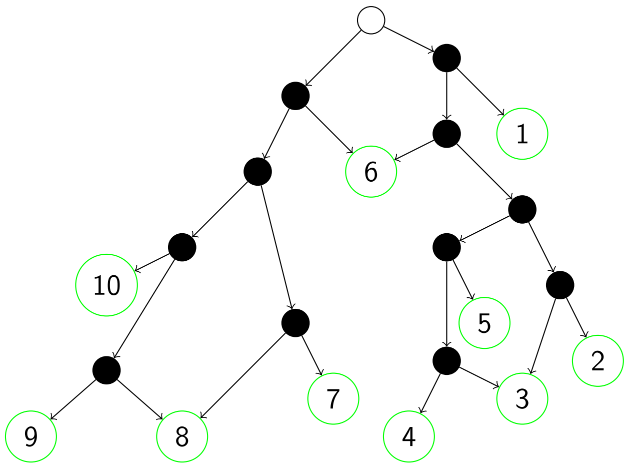

Figure 2.7.: A network with corresponding clustering system $C_N = \{\{1,2,3,4,5,6,7,8,9,10\},\{1,2,3,4,5,6\},\{6,7,8,9,10\},\{2,3,4,5,6\},\{7,8,9,10\},\{2,3,4,5\},\{3,4,5\},\{2,3\},\{3,4\},\{8,9,10\},\{7,8\},\{8,9\},\{1\},\{2\},\{3\},\{4\},\{5\},\{6\},\{7\},\{8\},\{9\},\{10\}\}$.

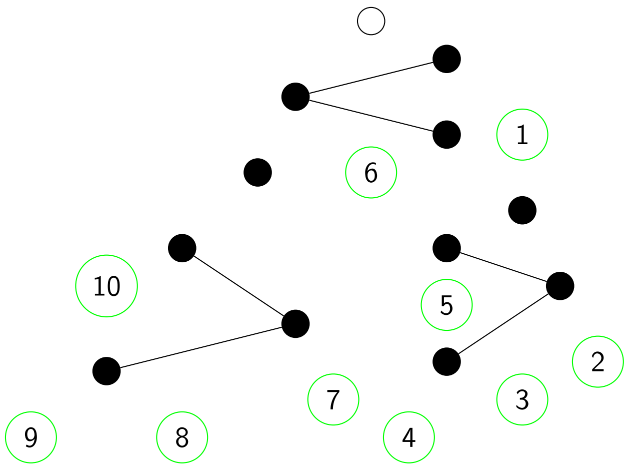

Figure 2.8.: An overlap graph with corresponding clustering system $C_N = \{\{1,2,3,4,5,6,7,8,9,10\},\{1,2,3,4,5,6\},\{6,7,8,9,10\},\{2,3,4,5,6\},\{7,8,9,10\},\{2,3,4,5\},\{3,4,5\},\{2,3\},\{3,4\},\{8,9,10\},\{7,8\},\{8,9\},\{1\},\{2\},\{3\},\{4\},\{5\},\{6\},\{7\},\{8\},\{9\},\{10\}\}$.

In figure 2.7 we see and example of a network for some clustering system and then in figure 2.8 we see and example of an overlap graph for the same clustering system.

Definition 2.3.10. An overlap graph to a clustering system C is an undirected graph $G = \{V, E\}$, whose vertices are elements in C, and for all vertices $v, w \in V$ there is an edge between v and w if and only if the corresponding clusters overlap.

The first of the property using overlaps to consider is that of hierarchy.

Definition 2.3.11. [5] A clustering system C satisfies hierarchy if for all $c_1, c_2 \in C$: $c_1 \cap c_2 \in \{\emptyset, c_1, c_2\}$.

It then follows trivially from definition 2.1.1 that

Corollary 2.3.4. *A clustering system C satisfies hierarchy if it contains no overlapping clusters.*

An expansion on hierarchy is the property paired hierarchy.

Definition 2.3.12. [5] A clustering system C satisfies paired hierarchy if every cluster in C overlaps with at most one other cluster in C.

It then follows that

Corollary 2.3.5. *A clustering system C satisfies paired hierarchy if the corresponding overlap graph has no vertices with a degree of more than 1.*

Another property of a clustering system is 'N3O'.

Definition 2.3.13. [5] A clustering system, C, satisfies N3O if it contains no three pairwise overlapping clusters.

It follows that this is equivalent to the corresponding overlap graph containing no triangles.

Corollary 2.3.6. *A clustering system, C, satisfies N3O if the corresponding overlap graph contains no cycles of length 3.*

Proof. It follows from the definition of the overlap graph that any three pairwise overlapping clusters, i, j, k, there will be edges (i, j), (j, k) and (k, i). This according to definition 2.1.7 means the graph contains a cycle of length 3. □

2.4. Python

Python is a powerful, high-level programming language [7]. Python has a multitude of libraries, contributors and packages that allow simple and flexible implementations of graph objects and related algorithms. A useful page from the Python documentation can be found here [9]; it documents the time complexity of some useful default Python functions and methods in a comprehensible table.

In this paper a cluster will be represented in Python as a set, while a clustering system will be represented as a list of sets. The reason for using a list is because it can be sorted. This means that we can search the list in $O(\log n)$ time. Python sets uses a hash algorithm so while a set could potentially be searched in constant time, it risks taken $O(n)$ time. An alternative implementation of *is_level_1* will therefore be given in the appendix. This one makes use of the itertools package [**van1995Python**].

One important note is that the time complexity for some of the algorithms will use the size of the leaf set. This is because it is often difficult to give the precise time complexity of several subroutines when the input is so varied. By definition of the clustering system, one cluster will contain precisely all of the leaves. This means that the run-time of functions on the clusters are certainly upper bound by the size of the leaf set.

To construct the graphs a package called 'NetworkX' will be used. NetworkX is a Python package meant for the study of graphs by mathematicians, biologists and computer scientists alike [4]. It provides many useful tools for the creation and manipulation of graphs. In this paper a NetworkX implementation of a transitive reduction will be used [8]. In Python NetworkX will be imported with the following code.

```
import networkx as nx
```

3. Results

The cluster cannot be assumed to be sorted. There are multiple ways in which the clustering system can be sorted. What sorting method is used is not particularly important for the purposes of this paper, and so any choice of sorting will be somewhat arbitrary. However the clustering system will need to be sorted and so an algorithm will still be needed.

3.1. set_comp

A function that compares two integer sets will be needed in order to sort the clustering system. First priority in the comparison is the number of elements in the set; A set with more elements should always come first in the sorted clustering set. If two sets are of equal length, then the set with smaller elements will be prioritized. How sets of equal length are compared turns out to not be very important as only the search algorithm makes use of it. As an example, when comparing the set $A = 2, 3, 4$ and $B = 1, 5, 6$, set B would be placed first since they are of equal length and 1 is smaller than 2. In Python this algorithm could look like the following.

```python
def set_comp(A,B):
    """Compares two sets.

    First priority in the comparison is the number of elements in the set;
    A set with more elements should come first in the sorted clustering set.
    If two sets are of equal length then
    the elements of each array are pairwise compared first to last
    and smaller elements are prioritised
    with the corresponding set placed first.

    Paramters
    -----------------
    A,B : sets
    Returns
    -----------------
    Boolean
        True if A would be prioritized over B.
        False otherwise.
    """

    if len(A) == len(B): #Longer array is prioritised
        for (a,b) in zip(A,B):
            if a < b: #Smaller numbers come first.
                return(True)
            elif a > b:
                return(False)
    elif len(A) > len(B):
        return(True)

    return(False)
```

No proof of correctness is given for this algorithm as it trivially works precisely as described.

Lemma 3.1.1. *If the sets A and B are of equal length then the time complexity of set_comp is in $O(m)$ where m is the length of the sets. Otherwise it is done in constant time. since 'len()' is a constant time function.*

Proof. To prove the time-complexity of *set_comp* we first consider the case that A and B are not of equal length. Then if A is longer than B the algorithm will return true. Otherwise, if B is longer than A, the algorithm will return false. Getting the length of a set in Python is done in constant time and so this part of the lemma is correct.

Next we prove time-complexity of *set_comp* if the sets are of equal length. There are m elements to iterate over in zip(A,B) and all operations within the for-loop is done in constant time. So if all numbers in the sets are the same then the for-loop will iterate through all m elements taking m time. After this the algorithm will either have returned true or the algorithm will return false.

The time complexity of the algorithm is therefore in $O(m)$. □

3.2. clusterSysSort

With a comparison algorithm we can now make a sorting algorithm. In order to sort the clustering system an algorithm similar to mergesort will be used. This algorithm will be called '*clusterSysSort*'.

```python
def clusterSysSort(clusterSys):
    """A sorting algorithm for clustering systems.
    Works similarly to mergesort.

    Paramters
    -----------------
    clusterSys : a list of sets representing a clustering system.

    Returns
    -----------------
    Null
    """

    if len(clusterSys) > 1:

        m = len(clusterSys) // 2
        L = clusterSys[:m]
        R = clusterSys[m:]
        clusterSysSort(L)
        clusterSysSort(R)

        i = j = k = 0
        while i < len(L) and j < len(R):
            if set_comp(L[i],R[j]):
                clusterSys[k] = L[i]
                i += 1
            else:
                clusterSys[k] = R[j]
                j += 1
            k += 1

        while i < len(L):
            clusterSys[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            clusterSys[k] = R[j]
            j += 1
            k += 1
```

Since it is difficult to say what the average input to *set_comp* will be we simply assume it is the largest element in the clustering system, i.e, the root cluster. This is possibly not very fair as the root can be much larger then other clusters.

Lemma 3.2.1. *clusterSysSort runs in $O(m \cdot n \cdot \log n)$, where m the is size of the leaf set and n is the number of clusters.*

Proof. We denote $T(n, m)$ as the time it takes the algorithm to terminate on an input of size n with leaf set of size m. Because the algorithm makes two recursive calls the total run-time of the algorithm is then $2 \cdot T(\frac{n}{2}, m) + f(n, m)$ where $f(n, m)$ is the time complexity of the three while-loops.

The run-time of the first while-loop will always be in $O(n \cdot m)$. This is because within the loop *set_comp* runs in $O(m)$ and the rest runs in $O(1)$. Because k is incremented at the end of the loop, for some value of k the run-time will then be in $\Theta(k \cdot m)$ and as k is upper bound by n the time complexity of the loop is in $O(n \cdot m)$. The other two while-loops iterate through the left and right halves of the input and the run-time of each iteration is constant so they combine to a run-time in $O(n)$. $f(n, m)$ is therefore in $O(n \cdot m)$.

The closed from is then $T(n, m) = 2^{\log n} + \log n \cdot O(n \cdot m) \in O(m \cdot n \cdot \log n)$.

This gives the final time complexity of *clusterSysSort* as $O(m \cdot n \cdot \log n)$. □

Theorem 3.2.2. *clusterSysSort correctly sorts an list containing numeric sets.*

Proof. To prove the correctness of *clusterSysSort* a proof by induction will be used.

In the base case assume we have a cluster of length one, such a cluster is trivially sorted.

For the induction step it is assumed that the algorithm sorts a cluster of length less than n correctly. Suppose now that a cluster of length n is sorted with *clusterSysSort*. This cluster will be split into two parts (L and

R) with lengths $\frac{n}{2}$. These will then recursively be put into the algorithm. By the induction hypotheses L and R will both be sorted correctly.

Since L and R are sorted the first element of either is the highest priority element in that list. A loop invariant condition on the first while-loop can now be used. This is that the output is sorted and contains only elements of higher priority than any in L or R. Before the loop this is true as the output is empty. Adding the highest priority element will then ensure the output is sorted after every iteration of the first while-loop. The condition therefore holds for every iteration of the while-loop. The algorithm will iterate until the entirety one of the two parts have been added to the sorted cluster. In which case all remaining elements are already sorted and must be of lower priority then all the elements added to the sorted clustering system. Adding these elements to the end of the sorted clustering system will then produce a complete sorted clustering system.

According to the principle of induction *clusterSysSort* correctly sorts a numeric clustering system. $\square$

Since we now have a sorting algorithm the input clustering systems will be assumed to be sorted.

3.3. includes

For the purposes of this thesis testing whether or not an cluster is included in the clustering system is necessary. Since a sorting algorithm was already constructed the clustering system can be assumed to be sorted. The following algorithm in Python is a binary search algorithm accomplishing this.

```python
def includes(clusterSys,A):
    """includes returns true if and only if set A is in the clustering system.

    Paramters
    -----------------
    clusterSys : a list of numeric sets representing a clustering system.
    A : a numeric set.
    Returns
    -----------------
    Boolean
        True if A is in clusterSys.
        False otherwise.
    """

    m = len(clusterSys) // 2
    M = clusterSys[m]
    L = clusterSys[:m]
    R = clusterSys[m:]

    if A == M:
        return(True)
    elif len(clusterSys) > 1:
        if set_comp(A,M):
            return includes(L,A)
        else:
            return includes(R,A)
    return(False)
```

Lemma 3.3.1. *includes returns true if and only if set A is in the clustering system.*

Proof. Proof by induction.

In the base case of a single set list the algorithm returns true if the sought after element A is in the list and false otherwise.

For the induction step it is assumed that the algorithm is correct for a list containing fewer than n elements. Suppose now that a clustering system of length n is searched with *includes*. If A is the middle set M than the algorithm correctly returns true. Since the list is sorted, if A is smaller than M then A is either not in the list or it is in the first half of the list. Likewise if A is larger than M then A is either not in the list or in the second half of it. In both cases a list with fewer than n elements will be searched through, but according to the induction hypotheses this will return the correct value.

By the principle of induction *includes* correctly returns whether or not a set is in a clustering system. □

Here it is again difficult to say what the average input to *set_comp* will be. So we simply assume it is the largest element in the clustering system, i.e the root cluster.

Lemma 3.3.2. *The time complexity of includes is in $O(\log_2(n) \cdot m)$, where n is the number of clusters in the clustering system and m is the size of the leaf set.*

Proof. Let $T(n, m)$ denote the time complexity of *includes* on an input of size n and average cluster size m. Then $T(n, m) = T(\frac{n}{2}, m) + f(m)$ where $f(m)$ is the time complexity of *set_comp*. In closed form this means $T(n, m) = \log n \cdot O(m) \in O(m \cdot \log n)$.

Giving a final time-complexity in $O(m \cdot \log n)$. □

3.4. test_weak_hierarchy

The first property for which an algorithm will be provided is 'weak hierarchy' from Definition 2.3.6. The algorithm works by simply getting every combination of three clusters and testing whether or not any of them contradict 'weak hierarchy'.

```
def test_weak_hierarchy(clusterSys):
    """Returns True if the network corresponding to
    clusterSys fulfills the property 'weak hierarchy'.

    Paramters
    -----------------
    clusterSys : a list of numeric sets representing a clustering system.

    Returns
    -----------------
    Boolean
        True if clusterSys is a weak hierarchy.
        False otherwise.
    """

    intersections = {}

    i = 0
    while i < len(clusterSys) - 2:
        j = i + 1
        while j < len(clusterSys) - 1:
            k = j + 1
            while k < len(clusterSys):
                if weak_hierarchy_help(clusterSys,intersections,i,j,k):
                    pass
                else:
                    return(False)
                k += 1
            j += 1
        i += 1
    return(True)
```

The help function *weak_hierarchy_help* is used in order to make this algorithm more legible.

```
def weak_hierarchy_help(clusterSys,intersections,i,j,k):
    if frozenset((i,j)) in intersections.keys():
        intersection_1 = intersections.get(frozenset((i,j)))
    else:
        intersection_1 = clusterSys[i].intersection(clusterSys[j])
        intersections[frozenset((i,j))] = intersection_1

    if frozenset((i,k)) in intersections.keys():
        intersection_2 = intersections.get(frozenset((i,k)))
    else:
        intersection_2 = clusterSys[i].intersection(clusterSys[k])
        intersections[frozenset((i,k))] = intersection_2

    if frozenset((j,k)) in intersections.keys():
        intersection_3 = intersections.get(frozenset((j,k)))
    else:
        intersection_3 = clusterSys[j].intersection(clusterSys[k])
        intersections[frozenset((j,k))] = intersection_3

    intersection = intersection_1.intersection(clusterSys[k])
    if not intersection or intersection in
        [intersection_1,intersection_2,intersection_3]:
        return(True)
    else:
        return(False)
```

Theorem 3.4.1. *test_weak_hierarchy will return true if and only if the input clustering system satisfies 'weak hierarchy'.*

Proof. Similarly to Lemma A.0.1 a loop invariant condition will be used to prove that the three loops iterate through every element in the clustering system. This condition is that for the first i elements in the list every combination of three containing that element will be iterated through.

Before the loop starts this is true as there are no combinations of zero elements.

Using Lemma A.0.1, we know that every combination of two elements larger than i will be iterated through. Thus every combination of three where i is the smallest element has been iterated through. By the loop invariant condition we already know that every combination containing an element smaller than i has been iterated through. These two together mean that every combination of three containing an element equal too or smaller than i will have been iterated through.

Thus the loop invariant condition is true before and after every loop and so every combination of i, j, k will be iterated through.

Now we need to show that *weak_hierarchy_help* in fact tests if any of these combinations contradicts 'weak hierarchy'. First the pairwise intersections are computed or retrieved if they have already been computed. Then the intersection of all three clusters is computed. Then the algorithm tests if the intersection of all three clusters is empty or equal to any of the pairwise intersections. Doing this for all combinations is precisely the definition for the 'weak hierarchy' property. □

Here it is difficult to say what the average time complexity of computing the intersections of three elements is exactly. Even though it is not very fair m will therefore be the size of the leaf set, which is the longest possible time it can take.

Lemma 3.4.2. *The time complexity of test_weak_hierarchy is in $O(n^3 \cdot m)$, where n is the number of clusters in the clustering system and m is the size of the leaf set.*

Proof. weak_hierarchy_help is done in $O(m)$ time because the intersection of the three clusters takes $O(m)$ time. Everything else is in $O(m)$ for n choose 2 of the calls and otherwise $O(1)$ because each pairwise intersection is only computed once.

It has already been proven that every combination of three elements will be iterated through by the algorithm. So it cannot make fewer than

n choose 3 calls to the '–Weak Hierarchy–' section of the code.

By the way i, j, k are initiated and incremented we have that $i < j < k$ at any time during the run-time of the algorithm.

For a proof by contradiction then. Suppose that the algorithm makes more than n choose 3 calls to *weak_hierarchy_help*. For every such call there is some i, j, k and since there are only n choose 3 such combinations at some point there will then be an i, j, k that has already been used. This is however impossible. For a fixed i, j, after every call to the '–Weak Hierarchy–' section k is always incremented by one. Meaning that for every i, j the combination i, j, k is only called once for every k such that $j < k \leq n$. From Lemma A.0.2 we also know that the first two loops only iterates through every combination of i, j once, where $0 \leq i, j \leq n - 1$. Therefore the three loops cannot iterate through any combination i, j, k more than once.

As no combination of i, j, k occurs twice and at least all combinations of i, j, k occur we can now say that exactly n choose 3 calls are made to *weak_hierarchy_help*. This means that the final time complexity is in $O(n^3 \cdot m)$. □

3.5. test_closed

The second property of a clustering system for which an algorithm will be provided is the 'closed' property from Definition 2.3.2. This algorithm works by iterating through all combinations of two from the clustering system. It then tests if any of them contradict the closed property.

```
def test_closed(clusterSys):
    """Returns True if clusterSys fullfills the 'closed' property.

    Paramters
    -----------------
    clusterSys : a list of numeric sets representing a clustering system.

    Returns
    -----------------
    Boolean
```

```
        True if clusterSys is closed.
        False otherwise.
    """

    i = 0
    while i < len(clusterSys)-1:
        j = i + 1
        while j < len(clusterSys):
            #----------------closed-----------------------
            intersection = clusterSys[i].intersection(clusterSys[j])

            if not intersection or includes(clusterSys,intersection):
                pass
            else:
                return(False)
            #----------------closed-----------------------
            j += 1
        i += 1
    return(True)
```

Theorem 3.5.1. *test_closed correctly computes if a clustering system is closed.*

Proof. Since retrieving the intersection of two elements is commutative, for *test_closed* to be correct it only needs to be shown that every combination of two elements are checked and that their intersection is empty or contained in the clustering system.

From lemma A.0.1 we already know that the two loop iterate through every combination of two elements in the clustering system.

The '–closed–' part of the algorithm returns true if and only if intersect returns an empty set or a set contained in the clustering system. Otherwise it returns false. Since the clustering system is assumed to be sorted it has already been proven that includes will correctly find if the set is included.

Therefore doing this for all combinations in the clustering system is precisely the definition of the 'closed' property. □

Lemma 3.5.2. *The time-complexity of test_closed is in $O(n^2 \log(n) \cdot m)$. Where n is the number if elements in the clustering system and m is the size of the leaf set.*

Proof. From Lemma A.0.2 we know that there will be exactly n choose 2 calls to the code in the '–closed–' section. Since the '–closed–' section is called only once for every combination of two elements the *intersection* algorithm will run in $O(m)$ time. The *includes* algorithm runs in $O(\log n \cdot m)$ time. So the total run-time of the '–closed–' section is in $O(\log n \cdot m)$. This is called n choose 2 times making the final time complexity of the algorithm $O(n^2 \log(n) \cdot m)$. □

3.6. overlaps

The next algorithm, '*overlaps*', checks whether or not the sets A and B overlap and returns true if they do but otherwise returns false. Recall Definition 2.1.1.

```
def overlaps(A,B):
    """returns True if sets A and B overlap.

    Paramters
    -----------------
    A,B : numeric sets.

    Returns
    -----------------
    Boolean
        True if A and B overlap.
        False otherwise.
    """
    if A.issubset(B):
        return(False)
    elif B.issubset(A):
        return(False)
    elif A.isdisjoint(B):
        return(False)
    else:
        return(True)
```

Lemma 3.6.1. *overlaps correctly tests if two sets overlap.*

Proof. It follows from the definition of overlapping sets that if one set is a subset of another they do not overlap and the algorithm correctly returns false if this is the case. Furthermore the two sets must have a non-empty intersection which they do not if they are disjoint. The algorithm correctly returns false if this is the case. Otherwise the algorithm correctly returns true. □

Lemma 3.6.2. *overlaps runs in $O(m)$ time, where m is the number of elements in the smaller set.*

Proof. The methods issubset and isdisjoint both run in $O(m)$ time, where m is the number of elements in the smaller set. Since they are called sequentially the time complexity of the algorithm is $O(m + m + m) = O(m)$. □

3.7. test_property_L

The next algorithm *test_property_L* takes a clustering system as input. It then, for every element i in the clustering system, checks if there is some combination of two elements, j, k, in the clustering system such that property L cannot be fulfilled. If so then it returns false. Otherwise it iterates through all elements in the clustering system and if none of them have returned false then the clustering system fulfills property L. Recall Definition 2.3.3.

```
def test_property_L(clusterSys):
    """Returns True if clusterSys fulfills the 'L' property.

    Paramters
    -----------------
    clusterSys : a list of numeric sets representing a clustering system.

    Returns
    -----------------
    Boolean
        True if clusterSys fulfills 'property L'.
        False otherwise.
    """

    intersections = {}
    i = 0
    while i < len(clusterSys):
        j = 0
        while j < len(clusterSys) - 1:
            k = j + 1
            while k < len(clusterSys):
                if property_L_help(clusterSys,intersections,i,j,k):
                    pass
                else:
                    return(False)
                k += 1
            j += 1
        i += 1
    return(True)
```

The help function *property_L_help* is used in order to make this algorithm more legible.

```
def property_L_help(clusterSys,intersections,i,j,k):
    if frozenset((i,j)) in intersections.keys():
        intersection_1 = intersections.get(frozenset((i,j)))
    else:
        intersection_1 = clusterSys[i].intersection(clusterSys[j])
        intersections[frozenset((i,j))] = intersection_1

    if frozenset((i,k)) in intersections.keys():
        intersection_2 = intersections.get(frozenset((i,k)))
    else:
        intersection_2 = clusterSys[i].intersection(clusterSys[k])
        intersections[frozenset((i,k))] = intersection_2

    if intersection_1 == intersection_2:
        return(True)
    elif overlaps(clusterSys[i],clusterSys[j]) and
         overlaps(clusterSys[i],clusterSys[k]):
        return(False)
    else:
        return(True)
```

Theorem 3.7.1. *test_property_L correctly computes if a clustering system fulfills property L.*

The proof of correctness for *test_property_L* is very similar to the one for test_closed, although this time there are three loops to consider.

Proof. The outer loop iterates through all elements, i, in the clustering system. We know from Lemma A.0.1 that the two inner loops iterate through all combinations of two elements, j, k.

Within the *property_L_help* help function, for every element i and combination of elements j, k from the clustering system, the algorithm returns false if and only if the overlaps between element i and elements j, k are identical or one of them does not exist. This is equivalent to the definition of property L and so *test_property_L* correctly tests if a clustering set fulfills property L. □

Lemma 3.7.2. *The time complexity of test_property_L is in $O(n^3 \cdot m)$, where n is the number of elements in the clustering system and m is the size of the leaf set.*

Proof. For every iteration of the outer loop i is incremented by one and it starts at 0. There are a maximum of n iterations of the outer loop. The two inner loops are therefore called n times. This together with Lemma A.0.2 means that the *property_L_help* help function is called n multiplied by n choose 2 times.

The *property_L_help* help function cannot break without computing the overlaps. Therefore the run-time of this section is at worst that of *overlaps* which is in $O(m)$. The rest of the code is either in $O(m)$ or in $O(1)$, both adding only a constant term.

The final time complexity of the algorithm is therefore in $O(n^3 \cdot m)$. □

3.8. is_level_1

The next algorithm, 'is_level_1', tests if a cluster can be represented as a level 1 (Definition 2.2.4) network, which according to Theorem 2.3.1 is equivalent to the cluster being closed and fulfilling property L. The algorithm could therefore be seen as a merge of the *test_closed* algorithm and the *test_property_L* algorithm.

```
def is_level_1(clusterSys):
    """Returns True if clusterSys fulfills the 'closed' and the 'L' properties
    and therefore are compatible with some level-1 network.

    Paramters
    -----------------
    clusterSys : a list of numeric sets representing a clustering system.

    Returns
    -----------------
    Boolean
        True if clusterSys fulfills 'closed' and 'property L'.
        False otherwise.
    """

    intersections = {}

    i = 0
    while i < len(clusterSys):
        j = 0
        while j < len(clusterSys)-1:

            if i < j:
                if closed_help(clusterSys,intersections,i,j):
                    pass
                else:
                    return(False)
            k = j
            while k < len(clusterSys):
                if property_L_help(clusterSys,intersections,i,j,k):
                    pass
                else:
                    return(False)
                k += 1
            j += 1
        i += 1
    return(True)
```

Similarly to *test_property_L* this function uses *property_L_help*. In addition to this it also uses another help function called *closed_help*.

```
def closed_help(clusterSys,intersections,i,j):
    if frozenset((i,j)) in intersections.keys():
        intersection = intersections.get(frozenset((i,j)))
    else:
        intersection = clusterSys[i].intersection(clusterSys[j])
        intersections[frozenset((i,j))] = intersection
    if not intersection or includes(clusterSys,intersection):
        return(True)
    else:
        return(False)
```

Theorem 3.8.1. *is_level_1 correctly computes if a clustering set is closed.*

The proof for this is a slight variation of the proof for Theorem 3.5.1 since we can no longer just use Lemma A.0.1.

Proof. For *is_level_1* to correctly test the closed property it only needs to be shown that every combination of two elements are checked and that their intersection is empty or contained in the cluster.

The last part is done within the *closed_help* help function for the same reason as in Theorem 3.5.1.

So it now only remains to be proven that the two loops iterate through every combination of elements in the clustering system. A loop invariant condition on the outer while-loop will be used to prove the correctness of this. This condition is that for the first i elements in the clustering system every combination containing i will have been iterated through.

Before the loop starts this is true as there are no combination of the first zero elements.

During the loop every element j in the clustering system will be iterated through and since, by the loop invariant condition, all combinations with an element smaller than i have already been iterated through, this implies that all combinations containing element i have been iterated through. So the condition holds after every loop.

Assuming *intersect* and *includes* are correct, the algorithm then tests if the clustering system is closed. □

Lemma 3.8.2. *is_level_1 correctly computes if a clustering set fulfills property L.*

Proof. The proof of this would be identical to the proof for Theorem 3.7.1 and will therefore be omitted. □

Theorem 3.8.3. *is_level_1 correctly tests if a clustering set is consistent with a level 1 network.*

Proof. According to Theorem 2.3.1 a clustering system is consistent with a level 1 network if and only if it is closed and fulfills property L. *is_level_1* has been proven to do both of these things. □

Lemma 3.8.4. *The time complexity of is_level_1 is in $O(n^3 \cdot m)$, where n is the number of elements in the input and m is the size of the leaf set.*

Proof. After every iteration of the outer loop i is incremented by 1 and starts at 0. So for any i the time lapsed is in $O(i \cdot r)$ where r is the time complexity of the inner two loops and the *closed_help* help function. Since i varies from 0 to n this gives the time complexity of the algorithm a run-time in $O(n \cdot r)$.

In Lemma 3.5.2 the '–closed–' section was proven have a time complexity in $O(m \cdot \log n)$. This section is equivalent to the *closed_help* help function, so it shares the same time complexity.

According to Lemma A.0.1, the inner two loops will make exactly n choose 2 calls to the *property_L_help* help function. In Lemma 3.7.2 we showed that this runs in $O(m)$ time. This gives the time complexity of the loops a run-time in $O(n^2 \cdot m)$. This is greater than the run-time for the *closed_help* help function.

Therefore the combined time complexity of *is_level_1* is in $O(n^3 \cdot m)$. □

3.9. clusterSys_to_Hasse

The next algorithm, *clusterSys_to_Hasse*, uses NetworkX to create a Hasse diagram from a clustering system. This is, according to Lemma 2.3.3, the corresponding network to the clustering system. It does this by iterating through all combinations of two clusters and adding an edge between the clusters if one is contained in the other. Then a transitive reduction is preformed [8], removing all shortcuts.

```
def clusterSys_to_Hasse(clusterSys):
    """Returns a networkX graph representing the corresponding
    Hasse diagram to the input clustering system.

    Paramters
    -----------------
    clusterSys : a list of numeric sets representing a clustering system.

    Returns
    -----------------
    NetworkX graph
    """

    size = len(clusterSys)

    g = nx.DiGraph()

    i = 0
    while i < size-1:
        j = i + 1
        while j < size:
            if clusterSys[i] > clusterSys[j]:
                g.add_edge(i,j)
            j += 1
        i += 1
    tg = nx.transitive_reduction(g)

    return(tg)
```

Lemma 3.9.1. *clusterSys_to_Hasse creates a Hasse diagram from an underlying clustering system.*

Proof. From Definition 2.3.4, we can infer that a Hasse diagram can be constructed by making each cluster in the clustering system a vertex in a graph and then adding an edge between to vertices if and only if one cluster is contained in the other. Then we preform a transitive reduction of the whole graph. As will be shown this is precisely what *clusterSys_to_Hasse* does.

According to Lemma A.0.1 the two loops iterate through every combination of two elements in the clustering system.

The algorithm then adds edges between two vertices if one is contained in the other. Since the cluster system is sorted by size, element i cannot be contained in element j since $i < j$ always holds. So it only needs to be tested if element j is contained in element i.

After all the edges have been added the transitive_reduction method is called on the graph. This method, as the name would suggest, preforms a transitive reduction on the graph.

Therefore *clusterSys_to_Hasse* does in fact construct a Hasse diagram from the underlying clustering system. □

Lemma 3.9.2. *clusterSys_to_Hasse has a time complexity in $O(n^3)$, where n is the number of elements in the clustering system, m is the average length of a cluster and $|E|$ is the number of times a cluster is contained in another.*

Proof. According to Lemma A.0.2 exactly n choose 2 calls are made to the code within the loops. Checking if a cluster is contained in another takes $O(m)$ time. Because every combination of elements is called exactly once, m is in fact the average length of the clusters.

The final time complexity of the loops is therefore $O(n^2 \cdot m)$. The transitive reduction is preformed in $O(n \cdot |E|)$ time. A maximum of one edge can be added for every combination of two clusters. Because of this $|E|$ must then be smaller than n choose 2 making the time complexity of the transitive reduction be in $O(n^3)$.

$|E|$ being bound by n is also tight. An example of a clustering system where this can easily be seen is the following. Assuming we have the entire leaf set, we start by add the leaf set as the root cluster. We then add a cluster containing all leaves but one. We continue adding clusters by removing one leaf from the previously added cluster.

We can say that $m < n$ because all elements of the longest cluster must be contained in the clustering system. This makes the final time complexity of the algorithm be in $O(n^2 \cdot m + n^3) = O(n^3)$. □

3.10. clusterSys_to_overlap

The next few algorithms will make use of an overlap graph to speed up computation. The overlap graph is defined in 2.3.10. The algorithm *clusterSys_to_overlap* is designed to create such a graph. It accomplishes this by retrieving every combination of two clusters and adding an edge if they overlap.

```python
def clusterSys_to_overlap(clusterSys):
    """Returns a overlap graph from a clustering system.

    Paramters
    -----------------
    clusterSys : a list of numeric sets representing a clustering system.

    Returns
    -----------------
    NetworkX graph
    """

    size = len(clusterSys)

    g = nx.Graph()

    i = 0
    while i < size-1:
        j = i + 1
        while j < size:
            if overlaps(clusterSys[i],clusterSys[j]):
                g.add_edge(i,j)
            j += 1
        i += 1

    return(g)
```

Theorem 3.10.1. *clusterSys_to_overlap correctly creates and overlap graph from a clustering system.*

Proof. From Lemma A.0.1 we know that the two loops will iterate through all combinations of clusters i, j. Since *overlaps* has already been proven correct we can say that the algorithm then adds an edge between cluster i and cluster j if they overlap. Otherwise it continues iterating through all combinations of i, j. This is precisely the definition of the overlap graph. □

Lemma 3.10.2. *The time-complexity of clusterSys_to_overlap is in $O(n^2 \cdot m)$ where n is the number of elements in the clustering system and m is the average length of the clusters.*

Proof. From Lemma A.0.2 we know that the two loops will make n choose 2 calls to the code within. Because every combination i, j is called exactly once the run-time of *overlaps* is in $O(m)$. The networkX *add_edge* method is done in constant time. This makes the final time complexity of *clusterSys_to_overlap* be in $O(n^2 \cdot m)$. □

3.11. hierarchy

We are now ready for the algorithms that test for properties in the clustering systems using the overlap graph. First of we have 'test_hierarchy' which takes a clustering system and tests if it fulfills the property 'hierarchy' (Definition 2.3.11). Using the overlap graph this is equivalent the edge-set being empty meaning the corresponding clustering system contains no overlaps between clusters.

```
def test_hierarchy(G):
    """Returns true if the overlap graph does not contain any edges,
    making the corresponding clustering system fulfill the 'hierarchy' property.

    Paramters
    -----------------
    G : a networkx graph where edges between vertices represent an overlap of
        the vertices.

    Returns
    -----------------
    Boolean
        True if G contains no edges,
        meaning that the corresponding clustering system has no overlaps.

        False otherwise.
    """
    if not G.edges:
        return(True)
    else:
        return(False)
```

Theorem 3.11.1. *test_hierarchy returns true if the clustering system corresponding to the input overlap graph fulfills the 'hierarchy' property.*

Proof. The algorithm returns true if and only if a the input overlap graph contains no edges. From Corollary 2.3.4 we know this means that no clusters in the corresponding clustering system overlaps with any other clusters. Therefore the algorithm returns true if the corresponding clustering system fulfills the 'hierarchy' property. □

When calculating the time complexity it is important to note that the time to create the overlap graph has not been taken into consideration. The actual time complexity of testing if a clustering system fulfills the 'hierarchy' property could therefore be the same as creating the overlap graph. That is to say in $O(n^2 \cdot m)$. However if testing other properties using the overlap graph anyways then it could be said that testing if property 'hierarchy' is fulfilled is done in constant time.

Lemma 3.11.2. *The time complexity of test_hierarchy is in $O(1)$.*

Proof. Testing if the graph contains no edges is done in constant time as a networkX graph contains a set with all its edges. The algorithm simply test if this set is empty. It returns true if so and false otherwise. These are all constant time operations. □

3.12. test_paired_hierarchy

The next algorithm again uses the overlap graph. This time the algorithm *test_paired_hierarchy* will test if the 'paired hierarchy' property is fulfilled (Definition 2.3.12). From Corollary 2.3.5 we know that, using the overlap graph, this is equivalent to no vertex having a degree of more than 1. This means that none of the clusters in the corresponding clustering system overlaps with no more than one other cluster.

```python
def test_paired_hierarchy(G):
    """Returns true if no vertices in the overlap graph
    has a degree of larger than 1.
    Meaning no cluster in the corresponding clustering system
    overlaps with more than one other cluster.

    Paramters
    -----------------
    G : a networkx graph where edges between vertices represent an overlap of
        the vertices.

    Returns
    -----------------
    Boolean
        True if every vertex in G as a degree of at most 1.
        False otherwise.
    """

    V = G.nodes()
    for v in V:
        if G.degree[v] > 1:
            return(False)
    return(True)
```

Theorem 3.12.1. *test_paired_hierarchy correctly returns true if and only if the clustering system corresponding to the input graph fulfills the 'paired hierarchy' property.*

Proof. The for-loop iterates through every vertex in the vertex-set. If the degree of the vertex is larger than 1 then false is correctly returned. Otherwise the algorithm continues iterating through the vertices. The algorithm returns true if and only if no vertex in the input graph has a degree larger than 1. As is said in Corollary 2.3.5, this means that every cluster in the corresponding clustering system overlaps with at most one other cluster. This is precisely the definition of 'paired hierarchy'. □

Just as with the *test_hierarchy* algorithm, the *test_paired_hierarchy* algorithm requires an overlap graph to function properly. It takes $O(n^2 \cdot m)$ time to make the overlap graph. So this could be seen as the actual time complexity of *test_paired_hierarchy*.

Lemma 3.12.2. *The time complexity of test_paired_hierarchy is in*

$O(n)$ where n is the number of vertices in the overlap graph or equivalently the number of clusters in the corresponding clustering system.

Proof. Getting the degree of a vertex using networkx is a constant time operation. The for-loop makes n such calls. The time complexity of the algorithm is therefore in $O(n)$. □

3.13. test_N3O

The next algorithm *test_N3O* will test if the 'N3O' property is satisfied. Recall Definition 2.3.13 and Corallary 2.3.6. The algorithm tests if the overlap graph contains triangles, or equivalently cycles of length 3.

```
def test_N3O(G):
    """Returns true if the overlap graph does not contain any triangles,
    making the corresponding clustering system fulfill the 'N3O' property.

    Paramters
    ------------------
    G : a networkx graph where edges between vertices represent an overlap of
        the vertices.

    Returns
    ------------------
    Boolean
        True if G contains no triangles,
        meaning that the corresponding clustering system
        has three pairwise overlaping clusters.

        False otherwise.
    """
    E = G.edges()
    for e in E:
        if not set(G[e[0]]).isdisjoint(set(G[e[1]])):
            return(False)
    return(True)
```

Theorem 3.13.1. *If the input graph is the overlap graph corresponding to a clustering system, C, then test_N3O correctly returns true if and only if C satisfies property 'N3O'.*

Proof. From Corollary 2.3.6 we know that if the underlying overlap graph to a clustering system does not contain cycles of length 3, that is triangles, then the clustering system satisfies property 'N3O'.

For any edge (i, j), if clusters i, j are part of a length 3 cycle then there must be some cluster k such that (i, k), (j, k) are edges in the overlap graph. This is also equivalent to saying i, j both share a neighbour k. This is precisely what the algorithm tests. If *test_N3O* returns true then property 'N3O' is satisfied by the corresponding clustering system.

This is also true in the reverse; if the overlap graph contains a length 3 cycle then there will be some edge (i, j), such that clusters i, j are part of a length 3 cycle. Therefore there must then exist some cluster k such that (i, k), (j, k) are edges in the overlap graph. The algorithm tests this for all edges and returns false if it can find such i, j, k. So if *test_N3O* returns false then property 'N3O' is not satisfied by the corresponding clustering system. □

The time complexity of this algorithm is actually in $O(n \cdot |E|)$, where n is the average number of neighbours each vertex has in the overlap graph and $|E|$ is the number of edges in the overlap graph. But since neither the number of edges or the average number of neighbours in the overlap graph are not directly apparent from the clustering system, we instead write the time complexity as the following.

Lemma 3.13.2. *The time complexity of test_N3O is in $O(n^3)$, where n is the number of clusters in the corresponding clustering system.*

Proof. Testing if two sets, s, t, are disjoint is done in $O(minlen(s), len(t))$ by Python. In this case the minimum length of the compared sets are dependant upon the number of neighbours. Since the number of neighbours a vertex can have in the overlap graph is bound by the total number of vertices in the graph we say the time complexity of this method is in $O(n)$.

The algorithm makes $|E|$ calls to the is disjoint method. Since there can only be one edge for every combination of two clusters $|E| \in O(n^2)$.

The final time complexity of *test_N3O* is therefore in $O(n^3)$. □

3.14. test_prebinary

In order to test if the 'prebinary' property is satisfied we will first create an algorithm that returns the unique inclusion minimal set with respect to a pair or returns an empty set if there is no such unique inclusion minimal set. We do this by iterating over every cluster, checking if the pair is contained. If it is Then we store that cluster as 'out' as a potential unique inclusion-minimal cluster. If we then find another cluster containing the pair, we need to check if this new cluster contains 'out'. If so then 'out' could still be unique. If not then 'out' cannot be unique and so we return an empty set.

```python
def inc_Min(clusterSys, pair):
    """Returns the unique inclusion minimal set
    in clusterSys with respect to a pair or
    returns an empty set if there is no such unique inclusion minimal set.

    Paramters
    -----------------
    clusterSys : a list of numeric sets representing a clustering system.

    Returns
    -----------------
    set : if possible, a unique inclusion minimal set
    in clusterSys for the pair. Otherwise an empty set.
    """

    k = len(clusterSys) - 1
    out = set()
    while k >= 0:
        C = clusterSys[k]

        if pair[0] in C and pair[1] in C:
            if not out:
                out = C
            elif out < C:
                pass
            else:
                return({})

        k -= 1
    return(out)
```

Theorem 3.14.1. *inc_min returns an inclusion minimal set, for the* *input pair of leaves, from the input clustering system if such a set exists. Otherwise it returns an empty set.*

Proof. The while-loop will iterate through the elements of clusterSys from last to first. Since clusterSys can be assumed to be sorted, this means that sets that are iterated through later must be longer than or of equal length to sets iterated through sooner.

Inside the loop, for every cluster it is tested if the pair is contained in the cluster. If no such pair is found then there cannot be any unique inclusion minimal sets that satisfy the requirements of the algorithm. This is however impossible if the pair is taken from the leaf set as the first element of the clustering system will by definition contain all the leaves.

If such a set, c_1, is found and the output is empty then it is added to the output. If another such set, c_2, is found it is a superset of c_1 then c_2 cannot be inclusion minimal for the pair and so the algorithm continues. Otherwise if c_2 is not a super set of c_1 then there are two inclusion minimal sets for the pair and therefore not a unique one. The algorithm then correctly returns an empty set. If no such set c_2 is found then and only then there is a unique inclusion minimal set for the pair in the clustering system and this set is correctly returned. □

Lemma 3.14.2. *The time complexity of inc_min is in $O(n \cdot m)$, where n is the number of clusters in the clustering system and m is the size of the leaf set.*

Proof. If there is one or no inclusion minimal clusters for the pair in the clustering system then the while-loop will not break until it iterates through all clusters. Therefore n calls are made to the code within the while-loop. Inside the loop everything is done in constant time except for testing if the output is a subset which is done in $O(m)$ time.

The final time complexity of *inc_min* is therefore in $O(n \cdot m)$. □

The next algorithm, *test_prebinary*, tests if the 'pre-binary' property is satisfied by the clustering system. Recall Definition 2.3.7. It does this by retrieving every combination of two leaves. Then it uses *inc_min* to check if there is a unique inclusion-minimal cluster for this pair.

```python
def test_prebinary(clusterSys, X):
    """Returns True if clusterSys fullfills the 'prebinary' property.

    Paramters
    -----------------
    clusterSys : a list of numeric sets representing a clustering system.
    X          : a list of numeric sets representing the leaves.

    Returns
    -----------------
    Boolean
        True if clusterSys fulfills 'prebinary'.
        False otherwise.
    """

    i = 0
    while i < len(X) - 1:

        x = next(iter(X[i]))

        j = i + 1
        while j < len(X):

            y = next(iter(X[j]))

            C = inc_Min(clusterSys, (x,y))
            if not C:
                return(False)

            j += 1
        i += 1
    return(True)
```

Theorem 3.14.3. *test_prebinary returns true if and only if the 'prebinary' property is satisfied by the clustering system.*

Proof. By Lemma A.0.1 we know that the first two loops will iterate over all of the combinations of two leaves. From Theorem 3.14.1 we also know that *inc_min* will return an empty set if and only if there is no unique inclusion minimal cluster for the pair. If there is not a unique inclusion minimal cluster for every pair then 'pre-binary' is not satisfied. So the algorithm correctly returns false if 'pre-binary' is not satisfied. This also means that the algorithm returns true if 'pre-binary' is satisfied. □

Lemma 3.14.4. *The time complexity of test_prebinary is in $O(m^3 \cdot n)$, where m is the number of leaves in the clustering system, n is the number of clusters in the clustering system.*

Proof. From Lemma A.0.2 we know that there will be exactly m choose 2 calls to *inc_min*. This is the only part that does not run in constant time. Since *inc_min* runs in $O(n \cdot m)$ time we then know that *test_prebinary* runs in $O(m^3 \cdot n)$. □

3.15. test_binary

The next algorithm, *test_binary*, will test if the clustering system fulfills the 'binary' property. Recall Definition 2.3.8. It works very similarly to *test_prebinary*. But it also stores all the clusters that are unique inclusion-minimal clusters for some pair of leaves, in the CS set. Doing this means you simply have to check that all clusters, at the end, are contained in CS.

```
def test_binary(clusterSys, X):
    """Returns True if clusterSys fullfills the 'binary' property.

    Paramters
    -----------------
    clusterSys : a list of numeric sets representing a clustering system.
    X          : a list of numeric sets representing the leaves.

    Returns
    -----------------
    Boolean
        True if clusterSys fulfills 'binary'.
        False otherwise.
    """

    CS = set()

    i = 0
    while i < len(X) - 1:
       x = next(iter(X[i]))

       j = i + 1
       while j < len(X):

           y = next(iter(X[j]))

           C = inc_Min(clusterSys, (x,y))
           if not C:
               return(False)
           else:
               CS.add(frozenset(C))

           j += 1
       i += 1
    if len(CS) == len(clusterSys) - len(X):
       return(True)
    else:
       return(False)
```

Theorem 3.15.1. *test_binary returns true if and only if clusterSys satisfies the 'binary' property.*

Proof. By Lemma A.0.1 we know that the first two loops will iterate over all of the combinations of two leaves. From Theorem 3.14.1 we also know that *inc_min* will return an empty set if and only if there is no unique inclusion minimal cluster for the pair.

Similarly to *test_prebinary*, ff there is not a unique inclusion minimal cluster for every pair then 'pre-binary' is not satisfied an thus neither is 'binary'. In this case the algorithm returns false. Otherwise 'pre-binary' is satisfied.

If *inc_min* instead returns a unique inclusion minimal cluster then this cluster is added to the CS set. After all pairs of leaves have been iterated through then the size of CS is compared to the number of non-leaf elements in the clustering system. Because only unique elements are in a set, if the length of this set is the same as the length of the clustering systems minus the leaf set, then every cluster is the unique inclusion-minimal set for some pair of leaves. This with 'pre-binary' is precisely the definition of the 'binary' property. □

Lemma 3.15.2. *The time complexity of test_binary is in $O(m^3 \cdot n)$, where m is the number of leaves in the clustering system, n is the number of clusters in the clustering system.*

Proof. From Lemma A.0.2 we know that there will be exactly m choose 2 calls to *inc_min*. This is the only part that does not run in constant time. Since *inc_min* runs in $O(n \cdot m)$ time we then know that *test_binary* runs in $O(m^3 \cdot n)$. □

A note here is that the 'add' method for sets is assumed to be in constant time. This is not always true as for some sets the hash function might fail. This wont change the time complexity as *inc_min* runs in $O(n \cdot m)$ time, but it should be kept in mind when using this algorithm.

3.16. test_2Inc

The next algorithm, *test_2Inc*, will test if the '2-Inc' property is satisfied by the clustering system. Recall Definition 2.3.9. In order to use this algorithm the Hasse diagram will first need to be constructed from the clustering system. This Hasse diagram is then used as the input.

```python
def test_2Inc(N):
    """Returns True if the corresponding clustering system
    fullfills the '2-Inc' property.

    Paramters
    -----------------
    N : A Hasse diagram corresponding to the clustering system.

    Returns
    -----------------
    Boolean
        True if the corresponding clustering system fulfills '2-Inc'.
        False otherwise.
    """

    V = N.nodes()

    i = 0
    while i < len(V):
        if N.out_degree[i] <= 2 and N.in_degree[i] <= 2:
            pass
        else:
            return(False)
        i += 1

    return(True)
```

Theorem 3.16.1. *For some clustering system, C, used with clusterSys_to_Hasse to create a Hasse diagram, N, test_2Inc returns true if and only if C satisfies the '2-Inc' property.*

Proof. From the definition of a Hasse diagram it follows that a vertex, v_1, in the Hasse diagram has a parent vertex, v_2, if and only if v_2 is an inclusion minimal cluster for v_1. Similarly it follows that a vertex, v_1, in the Hasse diagram has a child vertex, v_2, if and only if v_2 is an inclusion maximal cluster for v_1.

Therefore '2-Inc' is satisfied if and only if in the Hasse diagram corresponding to the clustering system, no vertex has an out- or indegree of more than two.

The first loop iterates through all vertices in the network and returns false if any of them has an out- or indegree of larger than 2. If this is not the case for any vertices then the algorithm returns true. □

When calculating the time complexity it is important to note that the time to create the Hasse diagram has not been taken into consideration. This only accounts for the additional time it takes to test '2-Inc'.

Lemma 3.16.2. *The time complexity of test_2Inc is in $O(n)$, where n is the number of elements in the clustering system.*

Proof. In the worst case scenario, the clustering system satisfies '2-Inc'. Then the while-loop will iterate through all n elements thus taking n time. All other operations are done in constant time and so the final time complexity of the algorithm is in $O(n)$. □

4. Conclusion

In this thesis we have used Python and NetworkX to provide working polynomial-time algorithms for testing key properties of clustering systems.

- The closed property is tested by *test_closed* which runs in $O(n^2 \log n \cdot m)$ time,
- property L is tested by *test_property_L* which runs in $O(n^3 \cdot m)$ time,
- if a clustering system is consistent with a level-1 network is tested by *is_level_1* which runs in $O(n^3 \cdot m)$ time,
- the hierarchy property is tested by *test_hierarchy* which runs in $O(1)$ time,
- the weak hierarchy property is tested by *test_weak_hierarchy* which runs in $O(n^3 \cdot m)$ time,
- the paired hierarchy property is tested by *test_paired_hierarchy* which runs in $O(n)$ time,
- the N3O property is tested by *test_N3O* which runs in $O(n^3)$ time,
- the pre-binary property is tested by *test_prebinary* which runs in $O(m^3 \cdot n)$ time,
- the binary property is tested by *test_binary* which runs in $O(m^3 \cdot n)$ time,
- the 2-Inc property is tested by *test_2Inc* which runs in $O(n)$ time.

Additionally we have provided an algorithm for reconstructing regular networks form a clustering system using the idea of a Hasse diagram with *clusterSys_to_Hasse* which runs in $O(n^3)$ time.

The source code for all to algorithms are available on Github via [10].

It should be noted here that there is a lack of large scale testing that has been done on any of the algorithms provided. The only testing done is on some networks of similar size to the example networks in the preliminaries. Mainly this is because there is at present a lack of algorithms that generate large networks and clustering systems. This is something that would be useful for future research into phylogenetic networks and clustering systems.

In the introduction of this thesis it was implied that the main uses of clustering systems and phylogenetic networks are found in evolutionary biology and phylogenetics. While this is true at the moment it must not necessarily hold for the future. With working implementations of several algorithms testing important properties of clustering systems and thereby phylogenetic networks using these as a tool to study other phenomena outside of biology might become easier. For an example in other fields of data analysis, clustering systems could become an alternative to pyramidal clustering [2].

There are also properties of clustering systems for which there are no working implementations or even theoretical algorithms. Such as testing if a clustering system is consistent with a level-2 network or even a general level-k network.

A. Additional Proof

Many of the algorithms in this thesis uses the same concept to iterate over combinations from a list. Therefore the proof that this part of the algorithms is correct will only be done once. When looking for every combination of two the algorithm may look like:

```
def 2combinations(list):
    i = 0
    while i < n - 1:
        j = i + 1
        while j < n:
            (code)
            j += 1
        i += 1
```

In this algorithm (code) means any code that does not interfere with the variables i, j, k or terminate the algorithm prematurely as that would obviously change functionality of the loops.

Lemma A.0.1. *2combinations gets all the combinations of two numbers from* 0 *to* n*.*

Proof. To prove that the two loops iterate through every combination of two elements in a list a loop invariant condition on the outer while-loop will be used. This condition is that for the first i elements in the list every combination containing that element will be iterated through.

Before the loop starts this is true as there are no combination of zero elements.

During the loop every element $j > i$ in the list will be iterated through and since, by the loop invariant condition, all combinations with an element smaller than i have already been iterated through, this implies that all combinations containing element i have been iterated through. So the condition holds after every loop.

The condition that was set now holds before and after every loop and therefore the algorithm iterates through every combination of two elements in the list. □

Lemma A.0.2. *2combinations makes exactly n choose 2 calls to the (code) section.*

Proof. It has already been proven that the algorithm iterates through all combinations of two elements so it cannot make fewer than n choose 2 calls to the (code) section.

The algorithm does not iterate through any size two permutation of i, j where $i = j$ because j starts at $i + 1$ and as it only increases we have that $i < j$ always holds.

We can now do a proof by contradiction. Suppose then that the algorithm makes more than n choose 2 calls to the (code) section. Since $0 \leq i, j \leq n$ there then must either 1. be some call to (code) where $i = a$ and $j = b$ but also one where $i = b$ and $j = a$ for some integers a, b or 2. some calls must share a combination i, j.

The first is impossible as $i < j$ has already been stated so therefore we would have $a < b$ and $b < a$, both of which cannot be true.

The second is impossible as for a fixed i we increment j by one after every call to (code). Meaning that for every i no same j is called twice. The only time j decreases is when the outer loop enters its next iteration. But then i is incremented by one and so no combination containing that i will be called again. Therefore we have that for every j no i is called twice.

This gives that no combination i, j is called twice and so the algorithms cannot make more than n choose 2 calls to the (code) section. □

B. Alternate Implementation

An implementation of *is_level_1* using a set of sets representation for clustering systems instead of a list of sets. The final time complexity is the same but checking if the intersect is contained in the clustering system is potentially done in constant time.

```
def is_level_1(clusterSys):
    """Returns True if clusterSys fullfills the 'closed' and the 'L' properties
    and therefore are compatible with some level-1 network.

    Paramters
    ------------------
    clusterSys : a set of numeric sets representing a clustering system.

    Returns
    ------------------
    Boolean
        True if clusterSys fulfills 'closed' and 'property L'.
        False otherwise.
    """


    for c in combinations(clusterSys,3):
        intersect1 = c[0].intersection(c[1])
        intersect2 = c[0].intersection(c[2])
        intersect3 = c[1].intersection(c[2])
        #---------------closed-----------------------
        if closed_help(clusterSys,intersect1,intersect2,intersect3):
            pass
        else:
            return(False)
        #---------------closed-----------------------

        #---------------property L-------------------
        if property_L_help(c,intersect1,intersect2,intersect3):
            pass
        else:
            return(False)
        #---------------property L-------------------
    return(True)
```

This version of *is_level_1* also uses two help functions for increased legibility. These are *closed_help* and *property_L_help*.

```
def closed_help(clusterSys,intersect1,intersect2,intersect3):
    if not intersect1 or (intersect1 in clusterSys):
        pass
    else:
        return(False)

    if not intersect2 or (intersect2 in clusterSys):
        pass
    else:
        return(False)

    if not intersect3 or (intersect3 in clusterSys):
        pass
    else:
        return(False)
    return(True)
```

```
def property_L_help(c,intersect1,intersect2,intersect3):
    if overlaps(c[0],c[1]) and overlaps(c[0],c[2]):
        if intersect1 == intersect2:
            pass
        else:
            return(False)

    if overlaps(c[1],c[0]) and overlaps(c[1],c[2]):
        if intersect1 == intersect3:
            pass
        else:
            return(False)

    if overlaps(c[2],c[0]) and overlaps(c[2],c[1]):
        if intersect2 == intersect3:
            pass
        else:
            return(False)
    return(True)
```

Lemma B.0.1. *is_level_1 correctly returns true if and only if the clustering system is closed.*

The proof for this is a slight variation of the proof for Theorem 3.8.1.

since we can no longer use Lemma A.0.1.

Proof. The itertools function *combinations* iterates through all combinations of size 3 from the clustering system. The algorithm then retrieves the pairwise intersections of three clusters. In the *closed_help* help function it then tests if the intersections exist and if they are contained in the clustering system. This is precisely the definition for closed.

Assuming *intersect* and *in* are correct, the algorithm will then test if the clustering system is closed. $\square$

Lemma B.0.2. *is_level_1 correctly returns true if and only if the clustering system satisfies L.*

Proof. The itertools function *combinations* iterates through all combinations of size 3 from the clustering system. The algorithm then retrieves the pairwise intersections of three clusters. In the *closed_help* help function, the algorithm checks if one cluster overlaps with the other two. If it does then the intersections with that cluster must be identical. This is precisely the definition of property L.

Assuming *intersect* and *overlaps* are correct, the algorithm will then test if the clustering system satisfies L. $\square$

Theorem B.0.3. *is_level_1 correctly tests if a clustering set is consistent with a level 1 network.*

Proof. According to Theorem 2.3.1 a clustering system is consistent with a level 1 network if and only if it is closed and fulfills property L. *is_level_1* has been proven to do both. $\square$

Lemma B.0.4. *The time complexity of is_level_1 is in $O(n^3 \cdot m)$, where n is the number of elements in the input and m is the size of the leaf set.*

Proof. The itertools function *combinations* iterates through all combinations of size 3 in the clustering system. Therefore the algorithm makes n choose 2 calls to the help functions. Assuming the hash function does not fail and *in* runs in constant time, both of the help functions run in $O(m)$ time. This gives a final time complexity for *is_level_1* in $O(n^3 \cdot m)$. $\square$

Bibliography

[1] Jean-Pierre Barthélemy. "Binary clustering". English. In: *Discrete Applied Mathematics* 156.8 (2008), pp. 1237–1250. DOI: 10.1016/j.dam.2007.05.024.

[2] Patrice Bertrand and Jean Diatta. "Prepyramidal Clustering and Robinsonian Dissimilarities: One-to-One Correspondences". In: *WIREs Data Mining and Knowledge Discovery* 3.4 (2013), pp. 290–297. DOI: https://doi.org/10.1002/widm.1096.

[3] Charles Choy et al. "Computing the Maximum Agreement of Phylogenetic Networks". In: *Electronic Notes in Theoretical Computer Science* 91 (Feb. 2004), pp. 134–147. DOI: 10.1016/j.entcs.2003.12.009.

[4] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. "Exploring Network Structure, Dynamics, and Function using NetworkX". In: *Proceedings of the 7th Python in Science Conference*. Ed. by Gaël Varoquaux, Travis Vaught, and Jarrod Millman. Pasadena, CA USA, 2008, pp. 11–15.

[5] Marc Hellmuth, David Schaller, and Peter F. Stadler. *Clustering Systems of Phylogenetic Networks*. 2022. DOI: 10.48550/ARXIV.2204.13466. URL: https://arxiv.org/abs/2204.13466.

[6] Sungsik Kong et al. "Classes of explicit phylogenetic networks and their biological and mathematical significance." In: *Journal of Mathematical Biology* 84 6 (2022), p. 47.

[7] Dave Kuhlman. *A Python Book: Beginning Python, Advanced Python, and Python Exercises*. URL: https://web.archive.org/web/20120623165941/http://cutter.rexx.com/~dkuhlman/python_book_01.html. (accessed: 05.22.2022).

[8] NetworkX. *Source Code for networkx.algorithms.dag*. URL: https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.dag.transitive_reduction.html#networkx.algorithms.dag.transitive_reduction. (accessed: 03.30.2022).

[9] Rolland Rusin. *Time Complexity*. URL: https://wiki.python.org/moin/TimeComplexity. (accessed: 03.30.2022).

[10] Oliver Tryding. *Source code for project*. URL: https://github.com/OliverTryding/Python-algorithms-for-clustering-systems.git.