A Privacy Score for Anonymous Databases

By

Lindsay A. White

Acknowledgements

This work was supported by Borealis AI through the Borealis AI Global Fellowship Award and by the Vector Institute through the Vector Scholarship in Artificial Intelligence (VSAI).

Abstract

In this thesis, we present a quantitative measure called the Database Privacy Score to assess the level of privacy in an anonymous database. Individuals in an anonymous database are still at risk of having personal information uncovered about them in a linkage attack. A privacy score is assigned to each individual in the database, measuring the risk of an adversary gaining new knowledge about them in a linkage attack. This requires looking at a set of attributes K and determining which additional attributes can be inferred from knowing K. This is where the bulk of the computational work occurs, and we present algorithms for computing this. C++ source code is included in the Appendix for all computations involved in computing the Database Privacy Score. We also show that under certain assumptions, applying k-anonymity to a database cannot worsen the privacy score, although there is no guarantee that it will improve the score. We also look at privacy from a topological perspective, and propose a solution for removing inferences that come from topological holes in the Dowker Complex representing our database.

Contents

1 Introduction **1**

2 Preliminaries **5**
2.1 Simplicial Complexes 5
2.2 Databases and Dowker Complexes 6
2.3 k-Anonymity 15

3 Database Privacy Score **17**
3.1 Desired Properties 17
3.2 Description of Score 23
3.3 Proof of Properties 24
3.4 Future Considerations 29

4 Algorithms **32**
4.1 Inference Sets 34
4.2 Intersection Poset 34
4.3 Computing $weight(S)$ 41

5 Connection to k-Anonymity **52**
5.1 Weights for k-Anonymity 52
5.2 Effect of k-Anonymity on Privacy Score 52

6 Topological Approach to Improve Privacy **56**
6.1 Topological Description of Privacy Loss 56
6.2 Modifying Databases to Preserve Privacy 57

7 Future Work **62**

8 Conclusion **64**

A C++ Code **65**

Chapter 1

Introduction

Even when personally identifiable data is removed from a database, it can still be possible for a person to be identified. In [Swe02], it was established that 87% of the U.S. population in 1990 could likely be identified using only the three properties of gender, date of birth and zip code. k-anonymity is a condition on a database which aims to prevent an individual from being uniquely identified in a database. In k-anonymity, a group of attributes (called a *quasi-identifier*) is identified that could, in combination, be used to uniquely identify individuals. The database is then modified to ensure that each individual has identical attributes to at least $k-1$ others, within this group of attributes. However, even if k-anonymity can be achieved, there can still be privacy loss for individuals during a linkage attack. In a linkage attack, an adversary has access to two databases, one that is anonymized, and one that contains personally identifiable information. They then look at common attributes between the databases to try to find a unique match to connect an individual in the anonymous database to one they can identify by name (or email, phone number, etc.) in the other database. For a database where k-anonymity has been achieved, an adversary may not be able to determine which row corresponds to a specified individual (this is what k-anonymity aims to prevent), but they may still be able to gain additional knowledge about the individual in question. If they knew an individual had the attributes *is_male* and *age_30*, and every individual with these attributes in the database also have the attribute *has_cancer*, then the attacker can still gain the additional knowledge of the attribute *has_cancer*, without needing to explicitly determine which row of the database corresponded to the given individual. This is what is called a homogeneity attack [KG06]. In this case, we are not uniquely identifying the individual in the database, yet we

are still acquiring additional information about them. In order to make this type of inference, we would first need to know some information about the individual. This could mean that we personally know an individual who is in the database and know certain facts about them, but more likely it will mean that we have another database from some other source, and are trying to gain more information about an individual using the knowledge from the other database.

The limitations of k-anonymity have been previously observed, and other techniques for measuring privacy have been introduced, such as l-diversity [KG06] and t-closeness [LLV07]. While l-diversity is an improvement from k-anonymity, it was still shown in [LLV07] to be subject to skewness and similarity attacks. In a skewness attack, if 85% of individuals who have the attribute *age_60* also have the attribute *has_diabetes*, an adversary is still able to infer potentially accurate information about an individual. t-closeness further improves upon l-diversity by requiring that within each group of individuals with identical quasi-identifier, the probability of having a sensitive attribute within this group should differ from the probability within the database as a whole by no more than some given threshold.

All of the above conditions can be measured for a given database, but they are either a yes or no condition - a database satisfies it, or it does not. They also all require grouping attributes into either identifying attributes, quasi-identifier attributes, sensitive attributes, or other attributes. In this thesis, we present a metric for measuring privacy loss that gives a database a score on a scale from 0 to 1, where 0 means there is no risk of information leakage and 1 indicates a very high risk. We measure privacy loss by considering the possible inferences that could occur for each individual, and assigning a numerical risk factor to each individual (again on a scale from 0 to 1). This allows a comparison of the risk before and after an anonymization technique is applied. In contrast to the conditions listed above, we do not group the attributes into different types, but instead assign a weight to each attribute to measure both its sensitivity level and the likelihood that an adversary may have knowledge of that attribute. By doing so, we are considering *all* possibilities. Certain inferences may be more likely to occur if the knowledge needed beforehand is easier to acquire, while some inferences may be more dangerous than others, depending on the sensitivity of the attributes that were inferred. All of these factors are considered in constructing the database privacy score. In this thesis, we consider only absolute inferences in constructing our score, but propose that this should be extended to include probabilistic inferences (to account for skewness attacks)

for future work.

The intent behind creating a database privacy score is to allow companies to assess the risk to individuals before releasing an anonymous database. As well, having a quantitative measure to assess privacy also allows a company to measure privacy both before and after an anonymization technique is applied. Without such a measure, there is no way to assess whether anonymization did in fact improve privacy. The use of the word anonymization often implies that there is no privacy risk. Under our definition of privacy, this would mean no new knowledge can be obtained about an individual. This could only occur if no inferences are possible, which is an unlikely scenario in general. We therefore need the database privacy score to give a precise measurement of how secure the database is. This score could also be used to prove theoretical results on how a particular anonymization technique affects privacy, by analyzing its effect on the database privacy score. We will see an example of this regarding k-anonymity in Section 5.2.

Other metrics have been introduced to measure privacy loss in other settings, such as hiding failure in [OZ03], which measures privacy loss on datasets consisting of transactional records. In this setting, our dataset is a list of transactions, such as purchases in a store, and data analysis is looking for frequent patterns that appear, such as groups of items purchased together. To measure privacy loss, hiding failure is the percentage of sensitive patterns in the modified database compared to that in the original. In contrast, we are considering a generic database consisting of individuals and attributes, and assume an adversary is looking for any type of knowledge they can acquire, as opposed to frequent patterns.

In this thesis, we are motivated by topological approaches to privacy. Topology can be used to describe the relationships between attributes in a database through the construction of a Dowker complex. Closure operators can be defined between the face posets of these complexes, and the inferences in our database can be derived from these closure operators. We can also use topology to describe how inferences occur in a database. Some inferences are the result of a topological hole in the Dowker complex. In Section 6.2, we propose a technique to remove the holes from the database in a way that eliminates the inferences coming from this hole, and does not create any new inferences.

In Chapter 7, we propose several avenues of further research. In modifying a database to improve privacy, the database privacy score gives us a way to measure whether this improvement occurs. However, improving privacy often

involves adding in false data, which can affect the usefulness of the data. For this, we need a way to meausre data usability as well. We propose that a Data Usability Score should be introduced, and both the Data Privacy Score and Data Usability Score should be used when assessing the effectiveness of a given anonymization technique.

In Appendix A, we include C++ source code for all of the algorithms needed to compute the Database Privacy Score. This can be used on a real dataset once the attributes are converted to binary ones.

Chapter 2

Preliminaries

In this chapter, we review some key definitions that will be used throughout. We refer the reader to [Hat02] and [EH10] for a thorough background on algebraic topology and simplicial complexes. We also refer the reader to [Wac06] for further background on simplicial complexes, faces and face posets, and to [Erd17] for more on the Dowker complexes and closure operators.

2.1 Simplicial Complexes

Definition 2.1. An **n-simplex** (geometrically) is the convex hull of $n+1$ vertices. It consists of $n+1$ vertices, all edges between each pair of vertices, all triangles between each group of 3 vertices, and so on. It is n-dimensional.

0-simplex 1-simplex 2-simplex 3-simplex

Definition 2.2. A **face** τ of an n-simplex σ is any simplex contained in σ (τ may be σ itself). We denote this by $\tau \leq \sigma$.

Definition 2.3. A **simplicial complex** S is a finite collection of simplices satisfying the following conditions:

1. If $\sigma \in S$ and $\tau \leq \sigma$, then $\tau \in S$ as well.
2. If $\sigma, \tau \in S$, then either $\sigma \cap \tau = \emptyset$, or $\sigma \cap \tau \leq \sigma$ and $\sigma \cap \tau \leq \tau$.

Example 2.4. The second condition above means that in a simplicial complex, we connect up simplices on edges, vertices, etc. We do not have an edge from one simplex slicing through the edge of another.

Acceptable *Not Acceptable*

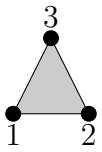
We can also represent simplices and simplicial complexes using sets.

Definition 2.5. In an **abstract simplicial complex** S, we represent each simplex σ in S by a set consisting of all the vertices of σ. S is then the collection of these sets.

Example 2.6. Suppose we label the vertices of an n-simplex as $\{1, 2, \ldots, n+1\}$. To represent an n-simplex as an abstract simplicial complex, we can take all non-empty subsets of $\{1, 2, ..., n+1\}$.

$S = \{\{1\}\}$

$S = \{\{1\}, \{2\}, \{1, 2\}\}$



$S = \{\{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$

In other words, we have that $S = \mathcal{P}(\{1, \ldots, n+1\}) \setminus \emptyset$ for an n-simplex S, where $\mathcal{P}(X)$ denotes the power set of the set X.

2.2 Databases and Dowker Complexes

Definition 2.7. Let X and Y be sets. A **relation** on $X \times Y$ is a set $R \subseteq X \times Y$. If $X = Y$, we say that R is a relation on X. If $(x, y) \in R$, we often denote this by xRy.

Definition 2.8. A **dataset** consists of a list of individuals with their attributes. The set of individuals is denoted by **Ind**, and the set of attributes is denoted by **Att**, so that our dataset **R** is a relation on $Ind \times Att$. We often represent this as a binary matrix.

Example 2.9. Let $Ind = \{1, 2, 3, 4\}$ and $Att = \{a, b, c, d\}$. We can represent a dataset R by a matrix, such as:

R	a	b	c	d
1	•	•		
2		•	•	
3	•		•	•
4		•	•	•

Throughout, we will typically use $\boldsymbol{K}$ to denote a set of attributes, and $\boldsymbol{J}$ to denote a set of individuals. When trying to determine inferences in a database, we will be thinking of K as the set of known attributes, hence the choice of notation. We represent a simplex by the set of its vertices. For example, the triangle containing vertices a, b, c would be represented by the set $\{a, b, c\}$. Sometimes we will shorten this to abc for easier readability.

Definition 2.10. From a relation R, we can construct two simplicial complexes called **Dowker Complexes**:

1. $\boldsymbol{\Phi_R} = \{K \subseteq Att \mid \exists i \in Ind \text{ s.t. } (i, x) \in R \ \forall x \in K\}$
2. $\boldsymbol{\Psi_R} = \{J \subseteq Ind \mid \exists x \in Att \text{ s.t. } (i, x) \in R \ \forall i \in J\}$

The simplices in Φ_R are the sets of attributes that at least one individual in our dataset has. From our matrix representation for R, the sets $K \in \Phi_R$ are the rows in the matrix, along with all subsets of each row. In our example relation R above, we would have:

$$\Phi_R = \{\{a, b\}, \{a\}, \{b\}, \{b, c\}, \{c\}, \{a, c, d\}, \{a, c\}, \{a, d\}, \{c, d\}, \{d\}, \{b, c, d\}, \{b, d\}\}$$

The simplices in Ψ_R are the sets of individuals who share at least one attribute. The sets $J \in \Psi_R$ consist of the columns in R, as well as all subsets of the columns in R. In our example relation R above, we would have:

$$\Psi_R = \{\{1, 3\}, \{1\}, \{3\}, \{1, 2, 4\}, \{1, 2\}, \{1, 4\}, \{2, 4\}, \{2\}, \{4\}, \{2, 3, 4\}, \{2, 3\}, \{3, 4\}\}$$

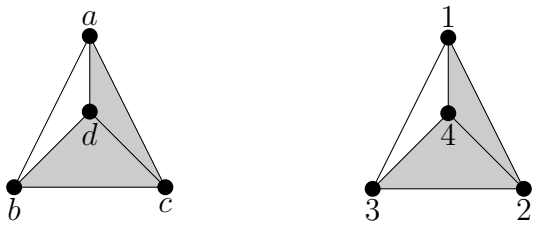


Figure 2.1: Φ_R on the left and Ψ_R on the right

Theorem 2.11 (Dowker Duality, [Dow52])**.** *Let R be a relation on $Ind \times Att$, with Φ_R and Ψ_R as above. Then Φ_R and Ψ_R are homotopy equivalent.*

For the purpose of this thesis, we will not use the fact that these complexes are homotopy equivalent. However, the maps that describe this homotopy will be important, and we describe them below. We first will need some other definitions.

Definition 2.12. Let R be a relation on a set X. R is a **partial order** if it is:

1. Reflexive: $aRa\ \forall a \in X$
2. Antisymmetric: If aRb and bRa, then $a = b$.
3. Transitive: If aRb and bRc, then aRc.

Definition 2.13. A set X along with a partial order R is called a **partially ordered set (poset)**.

Definition 2.14. The **face poset** of a simplicial complex S, denoted $F(S)$, is the poset with elements being the simplices of S and ordering being set inclusion.

Elements in the face poset are the simplices in our simplicial complex. For Φ_R, these simplices are subsets of Att, and for Ψ_R, these simplices are subsets of Ind. We now describe two important maps on the face posets.

Definition 2.15. Let $i \in Ind$ be an individual, and let $R \subseteq Ind \times Att$ be a dataset. Then Att_i is the set of attributes i has in R.

$$\boldsymbol{Att_i} = \{x \in Att | (i, x) \in R\}$$

This is the row corresponding to the individual i in our matrix representation of R.

Definition 2.16. Let $x \in Att$ be an attribute, and let $R \subseteq Ind \times Att$ be a dataset. Then Ind_x is the set of individuals who have the attribute x.

$$\boldsymbol{Ind_x} = \{i \in Ind | (i, x) \in R\}$$

This is the column in corresponding to the attribute x in our matrix representation of R.

Definition 2.17. The two maps describing the homotopy equivalence between Φ_R and Ψ_R are as follows:

1.
$$\boldsymbol{\phi_R} : F(\Psi_R) \to F(\Phi_R)$$
$$J \mapsto \bigcap_{i \in J} Att_i$$

2.
$$\boldsymbol{\psi_R} : F(\Phi_R) \to F(\Psi_R)$$
$$K \mapsto \bigcap_{x \in K} Ind_x$$

1. ***ϕ_R sends a set of individuals to the set of attributes they share***
2. ***ψ_R sends a set of attributes to the individuals who share them***

Observe that ϕ_R and ψ_R are both inclusion-reversing maps. That is, we have the following:

1. If $J_1 \subseteq J_2$, then $\phi_R(J_2) \subseteq \phi_R(J_1)$.
2. If $K_1 \subseteq K_2$, then $\psi_R(K_2) \subseteq \psi_R(K_1)$.

This fact follows from intersection of sets being inclusion-reversing.

The maps we are particularly interested in are the compositions $\phi_R \circ \psi_R$ and $\psi_R \circ \phi_R$, which turn out to be closure operators.

Definition 2.18. Let P be a poset, and let $f : P \to P$ be an order-preserving map. Then f is a **closure operator** if:

1. $x \leq f(x)\ \forall x \in P$
2. $f(f(x)) = f(x)\ \forall x \in P$

Lemma 2.19. *([Erd17]) $\psi_R \circ \phi_R$ and $\phi_R \circ \psi_R$ are closure operators.*

Let's consider the map $\phi_R \circ \psi_R : F(\Phi_R) \to F(\Phi_R)$. Let $\overline{K} = (\phi_R \circ \psi_R)(K)$. Since $\phi_R \circ \psi_R$ is a closure operator, this tells us that:

1. $K \subseteq \overline{K}$ for any $K \subseteq Att$
2. $(\phi_R \circ \psi_R)(\overline{K}) = \overline{K}$ for any $K \subseteq Att$

Let K denote a set of attributes, and let I_K denote the attributes we infer from K in our dataset R. We claim that $\overline{K} = K \cup I_K$ (where this is a disjoint union). To see this, we look at $\overline{K} = (\phi_R \circ \psi_R)(K)$:

$$\left\{ \begin{array}{c} \textit{set of} \\ \textit{attributes} \\ (K) \end{array} \right\} \mapsto \left\{ \begin{array}{c} \textit{set of} \\ \textit{individuals} \\ \textit{who share them} \\ (J) \end{array} \right\} \mapsto \left\{ \begin{array}{c} \textit{set of} \\ \textit{atrributes} \\ \textit{shared by} \\ \textit{those individuals} \\ (\overline{K}) \end{array} \right\}$$

Clearly, $K \subseteq \overline{K}$, since every individual in J has all attributes in K. However, it is possible that every individual in J also has additional attributes outside of A. If $\overline{K} \setminus K \neq \emptyset$, then $\overline{K} \setminus K = I_K$ is the set of additional attributes we can infer from knowing someone has all the attributes in K. Thus, $\overline{K} = K \cup I_K$, as desired.

Now that we know that $\overline{K} = K \cup I_K$, condition (2) above tells us that if $S = \overline{K}$ for some $K \subseteq Att$, then $S = \overline{S}$ and no new attributes can be inferred from S.

For simplicity, we will often use that $\overline{K} = \bigcap_{i \in J} Att_i$, where J is the set of all individuals in our dataset who have the attributes in K.

Example 2.20. We illustrate the map $\phi_R \circ \psi_R$ with an example. Consider the following relation R:

R	a	b	c	d
1	•	•		
2		•	•	
3	•		•	•
4		•	•	•

Here, we have:

$$\{d\} \mapsto \{3, 4\} \mapsto \{c, d\}$$

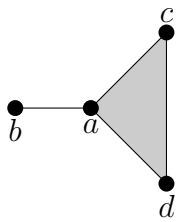
This means that everytime we know someone has attribute d, we also know they must have attribute c.

We can also make inferences of a different sort from the closure operator $\psi_R \circ \phi_R$. These are not relevant here, but we refer the interested reader to [Erd17] for more informtaion.

Definition 2.21. A simplex M of a simplicial complex S is **maximal** if there is no simplex $M' \in S$ such that $M \subsetneq M'$.

Definition 2.22. A simplex F of a simplicial complex S is said be a **free face** of S if it is contained in *exactly one* maximal simplex of S.

Example 2.23. Consider the following simplicial complex:



In this example, the simplices $\{a, b\}$ and $\{a, c, d\}$ are both maximal. Only the simplex $\{a\}$ is not free (it is in both maximal simplices; all others are only in one of the two).

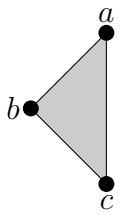
We now consider how inferences relate to free faces.

Notice that some information is lost when translating from our dataset to the Dowker complex. In the example above, we know there must be an individual with $Att_i = \{a, c, d\}$. This is because we only add a simplex to our Dowker complex if there is an individual in the dataset with all of those attributes, and since $\{a, c, d\}$ is not contained in a larger simplex, we must have someone with $Att_i = \{a, c, d\}$. However, we do not know whether an individual exists with $Att_j = \{a, c\}$. Since individual i has the attributes $\{a, c\}$, $\{a, c\}$ will be in our Dowker complex. It is however possible that another individual exists with $Att_j = \{a, c\}$. This can only be determined by looking at the original dataset. In other words, we cannot tell from the Dowker complex whether there are any individuals in our database with $Att_j \subseteq Att_i$.

Example 2.24. Let's consider a free face in Φ_R with two possible datasets:

R_1	a	b	c
1	•	•	•

R_2	a	b	c
1	•	•	
2	•	•	•



Here, ab is a free face, as it is only contained in the one maximal simplex abc. In R_1, we have the inference $ab \implies c$, but in R_2 we have no inference coming from ab.

This illustrates that in general, we cannot determine whether or not there will be an inference from a free face. However, if we make the assumption that $Att_j \not\subseteq Att_i$ for any $i, j \in Ind$, then our databases will look more like R_1, where we do have inferences.

Lemma 2.25. *Let R be a dataset, and suppose that there are no individuals $i, j \in Ind$ with $Att_j \subseteq Att_i$. If K is a free face in Φ_R and K is not maximal, then $\overline{K} = K$. In particular, this means that $I_K \neq \emptyset$, and there are attributes we can infer from I_K.*

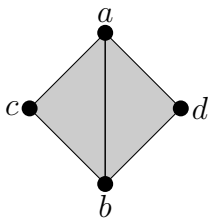
Proof. Since $Att_j \not\subseteq Att_i$ for any $i, j \in Ind$, we have that the maximal simplices in Φ_R are precisely the attributes sets Att_i for each $i \in Ind$. (By construction of Φ_R, the only way Att_j would not be a maximal simplex is if it were contained in another attribute set.) Since K is free and not maximal, then $K \subset Att_j$ for exactly one maximal simplex Att_j. Since $\overline{K} = \bigcap_{i \in J} Att_i$, where J is the set of all individuals in our dataset who have the attributes in K, and K is only contained in Att_j, we have that $\overline{K} = \bigcap_{i \in \{j\}} = Att_j$. Thus, $\overline{K} = Att_j$, and since K was not maximal, we have $\overline{K} \neq K$, as desired. □

In general however, we cannot conclude whether or not there is an inference from a free face.

Example 2.26. Now we consider a non-free face, again with two possible datasets.

R_1	a	b	c	d
1	•	•	•	
2	•	•		•

R_2	a	b	c	d
1	•	•	•	
2	•	•		•
3	•		•	



Here, a is a non-free face as it is in both of the maximal simplices abc and abd. In R_1, we have the inference $a \implies b$. However, in R_2, we have no inference coming from a. Again, this is caused by having an individual with $Att_3 \subseteq Att_1$.

Notice however that we also have the non-free face ab. In both R_1 and R_2, there is no inference coming from ab. Here, ab is the intersection of the maximal simplices it is contained in.

Lemma 2.27. *Let $K \subseteq Att$ and let $Att_1, \dots Att_n$ be the attribute sets that K is contained in within a dataset R. If $K = \bigcap_{i=1}^{n} Att_i$, then $\overline{K} = K$ and no inferences can be made from K.*

Proof. Let $J = \{1, \dots, n\}$ denote the set of individuals which have the attributes K. Then we have $\overline{K} = \bigcap_{j \in J} Att_j = \bigcap_{i=1}^{n} Att_i = K$, so $\overline{K} = K$, as desired. □

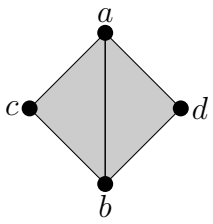
While in our example above, we saw that there was no guarantee whether or not there was an inference from the non-free face a (without putting conditions on the dataset), it turns out that if there are no free faces at all in Φ_R, then we cannot have any inferences at all.

Theorem 2.28 ([Erd17])**.** *Let Ind and Att be non-empty, and let R be a relation on $Ind \times Att$. If there are no free faces in Φ_R, then no attribute inferences are possible.*

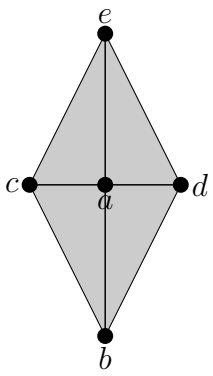
While we can have inferences coming from free or non-free faces in general, the existence of free faces is what causes the problem (even if the inference is actually coming from a non-free face).

Example 2.29. Let's revisit the example above, where we had the possible inference $a \implies b$.

R_1	a	b	c	d
1	•	•	•	
2	•	•		•



This inference is coming from the fact that a is sitting inside two free faces, ad and ac. If we make it so that a does not sit inside any free faces, we avoid inferences from a. For example, we could modify Φ_R as follows:



We now cannot have an inference from a. At a minimum, our dataset would need to contain the following:

R_1	a	b	c	d	e
1	•	•	•		
2	•	•		•	
3	•		•		•
4	•			•	•

It is clear from this dataset that there is no inference from a.

Lemma 2.30. *Let K be a non-free face in Φ_R that is not contained in any other non-free face. Then $\overline{K} = K$, and there no inference is possible from K.*

Proof. Suppose for a contradiction that there is an inference. Then there is a simplex K' such that $\overline{K} = K'$, where $K \subset K'$. If we can show that K' is a non-free face, then we have a contradiction to K not being contained in any other non-free face. Suppose then that K' is free. Then $K' \subseteq M$ for only one maximal face M. On the other hand, since K is non-free, we know that $K \subseteq M_1, M_2$ for at least two maximal faces. We have $K' = \overline{K} = \bigcap_{i \in J} Att_i$, where

J is the set of individuals who have the attributes in K. Since $K \subseteq M_1, M_2$ for two maximal faces, and we know the maximal faces must be attributes sets, let i_1 and i_2 denote the two individuals whose attribute sets corresponds to these two maximal faces. Then $i_1, i_2 \in J$ (since i_1, i_2 have all attributes in K). Since $K' = \bigcap_{i \in J} Att_i$, $K' \subseteq Att_i$ for each $i \in J$. Thus, $K' \subseteq Att_{i_1}, Att_{i_2}$, contradicting K' being free. Thus, K' is non-free, a contradiction to K not being contained in any other non-free face. Thus, we must have $\overline{K} = K$, as desired. □

2.3 k-Anonymity

k-anonymity is a condition on a database introduced in [Swe02] that is used to protect privacy. In k-anonymity, each individual will be identical to at least $k-1$ others across a particular set of attributes, where k is some integer greater than 1. The goal of k-anonymity is to prevent the ability of an adversary from being able to uniquely identify an individual in a database. In this thesis, we consider a stronger requirement for privacy. Instead of measuring privacy based on whether an individual can be tracked down to a unique row in the anonymous database, we measure privacy based on the ability of an adversary to gain any new knowledge about an individual that they did not already have beforehand.

Definition 2.31. A **quasi-identifer** is a collection of attributes that can (possibly) be used to uniquely identify an individual in a database.

For instance, the combination of the attributes {zip code, birthdate, sex} are often sufficient to uniquely identify each individual in a database. Usually, the attributes chosen for the quasi-identifier will be ones that are easily accessible in a public database. This allows an adversary to link the records in a public database to a private anonymous one. If there is a unique individual in the anonymous database with the quasi-identifier attribute values, they can be matched to the corresponding individual in the public database. This then allows an adversary to de-anonymize this individual and match them to a name.

Definition 2.32. A database is said to be **k-anonymous** if for each individual in the database, their attribute values corresponding to the quasi-identifier are identical to those of at least $k-1$ other individuals in the database.

This means that if an adversary were to try to link a public database to a k-anonymous private database, they would be unsuccessful in finding a unique

match. However, finding a unique match is not the only possible way for an adversary to gain additional information about an individual.

Example 2.33. Consider the following database, and suppose $\{a, b, c\}$ is the quasi-identifier, and $\{d, e, f\}$ are sensitive attributes that we do not want linked back to a particular individual.

R	a	b	c	d	e	f
1	•	•		•	•	
2	•	•		•		•
3	•		•		•	•
4	•		•	•	•	

This database is 2-anonymous with respect to the quasi-identifier $\{a, b, c\}$. Now let's assume an adversary has another database where they have an individual with attributes a and b. They won't be able to determine whether this person is individual 1 or 2. However, since everyone in the database with attributes a and b also have attribute d, they will be able to infer that the individual also has attribute d. This is still a loss in privacy, as the adversary was able to gain new (possibly sensitive) knowledge about the individual that they did not previously have. This form of privacy loss is not considered in the k-anonymity model.

Chapter 3

Database Privacy Score

In assigning a privacy score to a database, we want to assess the risk level of each individual in the database. Therefore, we must first consider a privacy score for an *individual* in a database. We can then choose to assign the score to the database by either taking the average, maximum, or some other combination of the scores of the individuals.

Before we decide how to assign a score to an individual, or to the database as a whole, we need to consider what properties we want the score to have. That is, we need to consider how the score should behave (or change) in various circumstances.

3.1 Desired Properties

The basis of our score will be in assessing the risk of inference for an individual. Careful consideration is needed to ensure we accurately assess the risk. Some factors to consider are:

1. What are all of the possible inferences that could be made for the individual?

2. For a given inference:

   (a) How likely is it that an adversary knows all of the attributes needed to make the inference?

   (b) How dangerous is it for an adversary to learn these new attributes?

These factors suggest that simply tallying up the number of inferences that occur for an individual will not suffice. Some inferences may be more likely to

occur than others, while some may be more dangerous to infer than others. This suggests that each inference should be weighted according to both the danger of learning the inferred attributes and the likelihood of an adversary having the knowledge necessary to make the inference.

Let K denote a set of attributes. We will think of K as a ″knowledge set″ or ″known set″ that an adversary has knowledge of ahead of time. Let I_K denote the new attributes we can infer from knowing K. For each possible K, we need to assign a weight $w(K)$ that captures both the likelihood we know K in the first place, as well as the danger from learning I_K. For each individual i in our database, an adversary could have knowledge of any set $K \subseteq Att_i$, where Att_i is the set of attributes for individual i. We will need to establish how to use these $w(K)$ values to assign a score to an individual. For this, we need to determine what properties our individual privacy score (denoted by $priv(i)$) should have. We will assign our score to be in the range $[0, 1]$, where 1 will be the worst for privacy, and 0 will be the best. Similarly, we will assign our weights to be values in the range $[0, 1]$ as well.

We would like $priv(i)$ to satisfy the following conditions:

1. If $Att_i \subseteq Att_j$ for some individuals i and j, then $priv(i) \leq priv(j)$.
   We would like this to occur since any inference that could be made for individual i could also be made for individual j. This is because j has all of the attributes i has. It is however possible that additional inferences could be made for individual j.

Suppose we were to define $priv(i)$ by taking the average of the $w(K)$ for $K \subseteq Att_i$. Consider an example where we have:

$$Att_i = \{a, b\}$$

$$Att_j = \{a, b, c\}$$

and suppose we have the inferences $a \implies b$ and $b \implies a$, each with weight 1. Further, assume that all other inferences have weight 0. Recall that we assign a $w(K)$ for each $K \subseteq Att_i$, so for individual i there will be 4 subsets, and for individual j there will be 8 (we include $K = \emptyset$ as a possibility). If we were to use an average of these weights, we would have:

$$priv(i) = \frac{1 + 1 + 0 + 0}{4} = 0.66$$

$$priv(j) = \frac{1+1+0+0+0+0+0+0}{8} = 0.29$$

The lower score for individual j gives the appearance that j is safer. However, the same inferences ($a \implies b$ and $b \implies a$) occurred for both i and j, with no other inferences being dangerous for either. This indicates that i and j should have the same risk. However, the addition of more 0-weight inferences caused j's score to decrease. This is something that we want to make sure our score avoids, and hence we should not take the average of the weights.

Notice that in the above, i and j share the same worst-case inferences. This leads to the decision that we should be taking a *maximum* of the $w(K)$'s instead of the average.

Definition 3.1. Let i be an individual in a database D. Then the **privacy score** for i is defined as:

$$priv(i) = \max_{K \subseteq Att_i} w(K)$$

where Att_i is the set of attributes for individual i, and $w(K)$ is the risk of inferring I_K from K.

Let us verify that this is a reasonable assessment of privacy. We are trying to assess the risk of individual i from being in the database. Assuming we have properly assessed the risk of inferring I_K from K (in the computation of $w(K)$), the maximum of these weights gives us the worst possible scenario that could occur. Since $w(K)$ will reflect the *likelihood* of the inference being made as well as the danger of inferring those attributes, the highest $w(K)$ is the worst inference that is likely to be made. A high $w(K)$ means its a dangerous inference *and* its likely to occur.

We need to assess whether taking the maximum is the best way to assess the privacy of an individual in the database. For this, we need to consider whether there is another way to combine the $w(K)$s to better reflect the risk. For instance, suppose we have two individuals i and j. Suppose individual i has $w(K_{i_1}) = 0.9$, and all other $w(K_{i_l}) = 0$. Suppose individual j has $w(K_{j_1}) = 0.9$, $w(K_{j_2}) = 0.5$, and all other $w(K_{j_l}) = 0$. Should individual j be given a higher score than i? By taking the maximum $w(K)$, both i and j are given the same score. We claim that this is an accurate reflection of the risk, and that j should not be given a higher score.

The inclination to say that j should have a higher score comes from thinking: ″What if an adversary had knowledge of *both* K_{j_1} and K_{j_2}?″ Here, the thinking

is that we should somehow be *adding* the risk weights of 0.9 and 0.5. However, if an adversary has knowledge of K_{j_1} and K_{j_2}, then they have knowledge of the set $K_{j_1} \cup K_{j_2}$. Since both $K_{j_1} \subseteq Att_j$ and $K_{j_2} \subseteq Att_j$, then $K_{j_1} \cup K_{j_2} \subseteq Att_j$ as well. This means $K_{j_1} \cup K_{j_2}$ was already in the list of sets we took the maximum over. There is no need to *add* any scores together, as by considering *all* possible subsets of Att_j (when we take $\max_{K \subseteq Att_j} w(K)$), we are already accounting for combined knowledge of sets. Thus, taking the maximum $w(K)$ really is assessing the worst possible case for the individual.

Now we list some properties that we would like for $w(K)$:

1. If $K \subseteq K'$ and $I_K = I_{K'}$, then $w(K) \geq w(K')$.
   We would like this to be true because if both K and K' infer the same set I_K, we are able to infer I_K from a *smaller* set when we infer from K. This means we can infer the same set with less knowledge, so this should be considered worse.

2. If $I_K = \emptyset$, $w(K) = 0$.
   If no inference is made, there is no risk.

Definition 3.2. If K is a set of attributes in a database D, then the **risk weight** of a set K, denoted $w(K)$, is

$$w(K) = lk(K) * dg(I_K)$$

where $lk(K)$ is the likelihood of knowing K and $dg(I_K)$ is the danger from inferring I_K.

To determine $lk(K)$ and $dg(I_K)$, we need to now look at individual attributes. For each attribute x, we assign two numbers:

$l(x)$: the likelihood (probability) of knowing attribute x $(0 \leq l(x) \leq 1)$
$d(x)$: danger rating for inferring x $(0 \leq d(x) \leq 1)$

A higher $l(x)$ means that we are more likely to know x, and a higher $d(x)$ means that it is more dangerous to infer x. We note that it *may* be reasonable to require that $l(x) + d(x) = 1$. If it is likely that an adversary knows an attribute, then it *might* indicate that it is something shared publicly and this *might* mean that it is not dangerous to infer. However, this may not always be a valid choice to make.

If we assume that our attributes are independent random variables, then it is straightforward to compute $lk(K)$ for a set K:

Definition 3.3. Let K be a set of attributes in a database D. The **likelihood of knowing** K is then:

$$lk(K) = \prod_{x \in K} l(x)$$

where $l(x)$ is the likelihood of knowing the attribute x.

We will assume that we do not have any dependent variables in our database. For instance, we cannot have one attribute that is age, and another that is age + 1, as we would have a dependence between attributes here. Note that there may be databases that include dependent variables, but for the purposes of this thesis we will assume that our database consists of a set of independent attributes. Future work could revise the likelihood function to deal with the possibility of dependent variables in our database.

Computing $dg(I_K)$ is not quite as straightforward. This is because we need a few considerations. We need that:

1. If $I_{K'} = I_K \cup \{x\}$, where $d(x) = 0$, then $dg(I'_K) = dg(I_K)$.
   In other words, if we infer an additional attribute that has a danger weight of 0, our danger score should not change. This then eliminates the possibility of multiplying our scores together, as multiplying by 0 would give us 0 even if other attributes had risk values.

2. If $d(x) = 0$ for all $x \in I_K$, then $dg(I_K) = 0$.

3. If $I_K \subseteq I_{K'}$, then $dg(I_K) \leq dg(I_{K'})$.
   This is because if $I_{K'}$ is inferred, then all attributes from I_K have also been inferred, so $dg(I_{K'})$ should be at least as high as $dg(I_K)$.

Since multiplication of the $d(x)$ values will not work, we need to consider addition instead. However, we need to scale $dg(I_K)$ to be between 0 and 1. We want the score to get worse when we add new attributes to I_K (i.e., when we infer more). If we consider a scenario where all attributes have weight 1, we want $dg(I_K)$ to get worse as I_K grows in size. We thus propose the following:

Definition 3.4. Let I_K be a set of attributes in a database D. Then the **danger**

score for inferring I_K is defined as:

$$dg(I_K) = \frac{\sum_{x \in I_K} d(x)}{1 + \sum_{x \in I_K} d(x)}$$

where $d(x)$ is the danger weight for attribute x.

Note that $dg(x) \neq d(x)$. This is because of our need to have $dg(I_K)$ worsen as I_K grows in size, and our inability to multiply scores. $d(x)$ should be chosen with this in mind.

We add 1 in the denominator to ensure that $dg(I_K)$ is between 0 and 1. It is possible to choose a number other than 1 here, and this may depend on the typical size of I_K that we will be looking at. For instance, suppose our sum in the numerator were 5. Using 1 in the denominator, we would have $dg(I_K) = 0.83$, but if we used 0.1 in place of 1, we would get 0.98. The choice of this value may depend on what we want to consider to be a ″bad″ sum of danger scores, and is something that should be discussed in future work.

Above, we had stated that we do not want the score to change if we add in an attribute with $d(x) = 0$. Notice that if we were to add in such an x, it would contribute 0 to both the numerator and denominator, so the score $dg(I_K)$ would be left unchanged, as desired.

We can also see in our scenario where all x have $d(x) = 1$, that as I_K grows, so does $dg(I_K)$. More specifically, as $\sum_{x \in I_K} d(x) \to \infty$, we have $dg(I_K) \to 1$.

Now that we have considered our individual privacy score, we need to determine how to assign a score to a database as a whole. For our database score, we would like that:

1. If we have two databases D and D', and the same set I_K is inferred in each, $dg(I_K)$ is the same in each. That is, $dg(I_K)$ is independent of the database we are currently looking at.

2. The score should not be relative to the database. A bigger database with larger (more dangerous) inferences should have a worse score than a small database with fewer dangerous inferences.

3.2 Description of Score

Here, we summarize the steps in computing the score that were derived in the previous section. We also introduce some options for computing a privacy score on the database as a whole.

Let K be a set of attributes in a database D. Let I_K be the set of attributes we can infer from knowing K.

Recall from Definitions 3.3 and 3.4 that the **likelihood** of knowing K, denoted $lk(K)$, and the **danger** of inferring I_K, denoted $dg(I_K)$ are:

$$lk(K) = \prod_{x \in K} l(x)$$

and

$$dg(I_K) = \frac{\sum_{x \in I_K} d(x)}{1 + \sum_{x \in I_K} d(x)}$$

respectively, where $l(x)$ is the likelihood of knowing the attribute x and $d(x)$ is the danger weight for attribute x. Note that $dg(\{x\}) \neq d(x)$.

From Definition 3.2, we had that the **risk weight** for a set of attributes K is $w(K)$ is:

$$w(K) = lk(K) * dg(I_K)$$

In Definition 3.1, we defined the **individual privacy score**, denoted by $priv(i)$, for any individual i in a database D, as:

$$priv(i) = \max_{K \subseteq Att_i} w(K)$$

where Att_i is the set of attributes that i has in D.

Definition 3.5. For a database D, the **average database privacy score**, denoted $priv_{avg}(D)$, is defined as:

$$priv_{avg}(D) = \frac{\sum_{i \in Ind} priv(i)}{n}$$

where Ind is the set of all individuals in D and $n = |Ind|$.

This assesses the *average* risk of an individual in the database D. This may

be combined with other scores as listed below.

Definition 3.6. For a database D, the **threshold privacy count** is defined as:

$$priv_{ct}(D) = |\{i \mid priv(i) \geq t\}|$$

and the **threshold privacy score** is defined as:

$$priv_t(D) = \frac{|\{i \mid priv(i) \geq t\}|}{n}$$

where $0 \leq t \leq 1$ is some chosen threshold value, and n is the number of individuals in D.

The threshold privacy count gives a count of the total number of individuals in a databse with a privacy score over a certain threshold, while the threshold privacy score gives a percentage of the number of individuals in a database with a privacy score over a certain threshold.

3.3 Proof of Properties

Desired Properties for $lk(K)$.

Proposition 3.7. *$lk(K)$ satisfies the following conditions:*

(i) If $K \subseteq K'$, then $lk(K) \geq lk(K')$.

(ii) If $\overline{x} \in K$ and $l(\overline{x}) = 0$, then $lk(K) = 0$.

(iii) If $l(x) = 1$ for all $x \in K$, then $lk(K) = 1$.

Proof. **(i):** Since $0 \leq l(x) \leq 1$ and $K \subseteq K'$,

$$lk(K) = \prod_{x \in K} l(x) \geq \prod_{x \in K'} l(x) = lk(K')$$

(ii):

$$lk(K) = \prod_{x \in K} l(x) = 0 \cdot \prod_{x \in K, x \neq \overline{x}} l(x) = 0$$

(iii):

$$lk(K) = \prod_{x \in K} l(x) = \prod_{x \in K} 1 = 1$$

□

Desired Properties for $dg(K)$.

Proposition 3.8. *If $I_{K'} = I_K \cup \{y\}$ and $d(y) = 0$, then $dg(I_{K'}) = dg(I_K)$.*

Proof. We have:

$$dg(I_{K'}) = \frac{\sum\limits_{x \in I_{K'}} d(x)}{1 + \sum\limits_{x \in I_{K'}} d(x)}$$

$$= \frac{d(y) + \sum\limits_{x \in I_K} d(x)}{1 + d(y) + \sum\limits_{x \in I_K} d(x)}$$

$$= \frac{0 + \sum\limits_{x \in I_K} d(x)}{1 + 0 + \sum\limits_{x \in I_K} d(x)}$$

$$= \frac{\sum\limits_{x \in I_K} d(x)}{1 + \sum\limits_{x \in I_K} d(x)}$$

$$= dg(I_K)$$

as desired. □

Proposition 3.9. *If $I_K \subseteq I_{K'}$, then $dg(I_K) \leq dg(I_{K'})$.*

Proof. Suppose $I_{K'} = I_K \cup J$. We have:

$$dg(I_{K'}) = \frac{\sum\limits_{x \in I_{K'}} d(x)}{1 + \sum\limits_{x \in I_{K'}} d(x)}$$

$$= \frac{\sum\limits_{x \in I_K} d(x) + \sum\limits_{x \in J} d(x)}{1 + \sum\limits_{x \in I_K} d(x) + \sum\limits_{x \in J} d(x)}$$

Let $a = \sum\limits_{x \in I_K} d(x)$ and $b = \sum\limits_{x \in J} d(x)$. We then have:

$$dg(I_{K'}) = \frac{a + b}{1 + a + b}$$

We want to show that $dg(I_K) \leq dg(I_{K'})$. Since $dg(I_K) = \dfrac{a}{1 + a}$ and $dg(I_{K'}) = \dfrac{a + b}{1 + a + b}$, we need to show that

$$\frac{a}{1 + a} \leq \frac{a + b}{1 + a + b}$$

where $a, b \geq 0$. We have

$$\frac{a+b}{1+a+b}$$

$$= \frac{a(1+\frac{b}{a})}{(1+a)(1+\frac{b}{1+a})}$$

$$= \frac{a}{1+a} \cdot \frac{1+\frac{b}{a}}{1+\frac{b}{1+a}}$$

We have that $\frac{b}{a} > \frac{b}{a+1}$, so $\frac{1+\frac{b}{a}}{1+\frac{b}{1+a}} > 1$. This means we have

$$\frac{a+b}{1+a+b} \geq \frac{a}{1+a}$$

as desired. □

Proposition 3.10. *If $d(x) = 0$ for all $x \in I_K$, then $dg(I_K) = 0$.*

Proof. If $d(x) = 0$ for all $x \in I_K$, we have:

$$dg(I_K) = \frac{\sum\limits_{x \in I_K} d(x)}{1 + \sum\limits_{x \in I_K} d(x)} = \frac{\sum\limits_{x \in I_K} 0}{1 + \sum\limits_{x \in I_K} 0} = \frac{0}{1+0} = 0$$

□

Notice that our worst possible case is when all x have $d(x) = 1$, *and* when $|I_K|$ is very large. If $d(x) = 1$ for all x, we have:

$$dg(I_K) = \frac{\sum\limits_{x \in I_K} d(x)}{1 + \sum\limits_{x \in I_K} d(x)} = \frac{\sum\limits_{x \in I_K} 1}{1 + \sum\limits_{x \in I_K} 1} = \frac{|I_K|}{1+|I_K|}$$

and this approaches ∞ as $|I_K| \to \infty$.

Desired Properties for $w(K)$.

Proposition 3.11. *$w(K)$ satisifies the following conditions:*

(i) If $K \subseteq K'$ and $I_K = I_{K'}$, then $w(K) \geq w(K')$.

(ii) If $I_K = \emptyset$, then $w(K) = 0$.

Proof. **(i):** By Proposition 3.7(i), since $K \subseteq K'$, we have that $lk(K) \geq lk(K')$. Then we have:

$$\begin{aligned} w(K) &= lk(K) * dg(I_K) \\ &= lk(K) * dg(I_{K'}) \qquad \text{since } I_K = I_{K'} \\ &\geq lk(K') * dg(I_{K'}) \\ &\geq w(K') \end{aligned}$$

as desired. **(ii):** If $I_K = \emptyset$, then $dg(I_K) = 0$, so $w(K) = lk(K) * dg(I_K) = lk(K) * 0 = 0$, as desired. □

Desired Properties for $priv(i)$.

Proposition 3.12. *If $Att_i \subseteq Att_j$, then $priv(i) \leq priv(j)$.*

Proof. We have that

$$priv(i) = \max_{K \subseteq Att_i} w(K)$$

Since $Att_i \subseteq Att_j$,

$$\max_{K \subseteq Att_i} w(K) \leq \max_{K \subseteq Att_j} w(K)$$

so $priv(i) \leq priv(j)$. □

Proposition 3.13. *If $I_K = \emptyset$ for every $K \subseteq Att_i$, then $priv(i) = 0$.*

Proof. If $I_K = \emptyset$ for every $K \subseteq Att_i$, this is saying that no known set K will produce any inferences. Then $w(K) = 0$ for every $K \subseteq Att_i$ by Proposition 3.11(ii). Thus, $priv(i) = \max_{K \subseteq Att_i} w(K) = 0$, as desired. □

For the next property, we want that if $K \subseteq K'$ and $I_K = I_{K'}$, our score should not count these as separate inferences. Since K can infer I_K with less knowledge, this is the more dangerous inference to occur. The inference of I_K from K' is not a new inference, since it is really being inferred just from K. Thus, $w(K')$ should not contribute to $priv(i)$, but $w(K)$ may.

Proposition 3.14. *If $K \subseteq K'$ and $I_K = I_{K'}$, then $w(K)$ may contribute to $priv(i)$, but $w(K')$ should not.*

Proof. Since $priv(i) = \max_{K \subseteq Att_i} w(K)$, only one K will ultimately contribute. For K and K', only the one with higher weight may contribute. By Proposition 3.11(i), $w(K) \geq w(K')$, so $w(K')$ will not contribute, but $w(K)$ may. □

Desired Properties Across Two Databases.
Let D and D' be two databases that share some subset of their attributes. Thus, it is possible that an attribute set K or an inference set I may appear in both. Note that I may be inferred from different sets in D and D'.

Proposition 3.15. *If we have two databases D and D', and the same set of attributes I is inferred in each, $dg(I)$ is the same.*

Proof. We have

$$dg(I) = \frac{\sum_{x \in I} d(x)}{1 + \sum_{x \in I} d(x)}$$

This definition is independent of the choice of database. It depends only on the values assigned for $d(x)$. If the same $d(x)$ values are used, $dg(I)$ will be the same in any database. □

Proposition 3.16. *Individual scores should not be scaled relative to the size of the database.*

Proof. Since $priv(i)$ takes the *maximum* of the $w(K)$ weights (as opposed to an average or other measure), $priv(i)$ is not scaled relative to the size of the database. □

This means databases with larger inferences will result in larger $priv(i)$ scores, as desired.

3.4 Future Considerations

How to Assign Attribute Weights. More careful thought needs to go into establishing a reliable and consistent way of choosing the attribute likelihood and danger weights.

In assigning likelihood scores, it may be fair to say that higher scores will be assigned to more readily-available public data. In this case, it may also be fair to say that $l(x)+d(x) = 1$. In other words, if we share certain information publicly and frequently, the likelihood score will be high, but we may not consider it dangerous to infer.

On the other hand, there may be sensitive attributes that we want to assign a high $d(x)$ value to, but perhaps the likelihood of knowing the attribute is a bit higher than $1 - d(x)$. In this case, we may not want $l(x) + d(x) = 1$.

A standardization of weights for commonly seen attributes should be established, perhaps with an analysis of how easy they are to find in public databases.

Partial Inferences. In the privacy score presented in this thesis, we only considered ″complete″ inferences. In other words, we only considered I_K to be inferred from K if *everyone* who had the attributes in K also had the attributes in I_K. In future work, partial inferences should be considered. For instance, if 95% of those with the attributes in K also have those in I, this could be considered similar to the full inferences, but perhaps multiplied by 0.95. Then $w(K)$ may need to be computed by taking the maximum across all of these possible weightings. The reason for considering these inferences as well is that if an adversary sees that 95% of individuals with the set of attributes they are looking for also have the attributes in some set I, there is a 95% chance that they inferred correct information (I) about the individual they are looking at. This is still a privacy loss.

Which Database Score Should Be Used. Three different scores were given for the overall database. The average and threshold scores may be used together to give a better overall assessment of the risk. Future work should consider whether there are other ways to combine the individual scores to assess the overall risk of the database.

What is a Good Score? Future work will need to analyze the scores of various databases and come to a consensus on what an acceptable score is. This should be done both at an individual level, and for the database as a whole. Consideration should also be given as to what the ideal number to add in the denominator of the $dg(I_K)$ calculation should be.

How This Score Should Be Used. The intention behind this score is to allow companies to establish the privacy risk of individuals in their database before they share this information, either publicly, internally, or with third parties

for data analytics. Before sharing a database, if the score is too high, further anonymization techniques or removal of high risk individuals should be done.

This score also gives a quantitative measure in assessing whether applying a given anonymization technique has in fact improved the privacy of individuals in the database.

How Can We Improve the Score? Future work will need to look at existing and new anonymization techniques to establish effective ways to improve the privacy score of a database, while maintaining data usability.

Chapter 4

Algorithms

In computing the database privacy scores, we need to first compute the individual privacy scores. Recall that we have

$$priv(i) = \max_{K \subseteq Att_i} w(K)$$

where $w(K) = lk(K) * dg(I_K)$, and I_K is the set of attributes that we infer from K. The computation of $w(K)$ is straightforward once we know both K and I_K.

The biggest task then is determining I_K from K. Let us denote by $\overline{K}$ the set $K \cup I_K$, consisting of both K and the attributes inferred from knowing K. Recall that $\overline{K} = (\phi_R \circ \psi_R)(K)$. We will refer to $\overline{K}$ as our inference set.

By definition of $priv(i)$, the most obvious algorithm for computing it would involve enumerating all subsets K of Att_i, determining I_K, and then computing $w(K)$. If $|Att_i| = N$, we would then have 2^N subsets K to enumerate, making this algorithm exponential in the size of our Att_i set. We would also have the same set K appearing as a subset of more than one Att_i (which indicates we might want to store the $w(K)$s once computed to avoid repeating computations). This observation however does not avoid the 2^N enumeration of subsets for each Att_i set.

Let $n = |Ind|$ and $m = |Att|$. Then the brute force algorithm described above will have a worst-case complexity of $n \cdot 2^m \cdot l$, where l is the complexity of computing $w(K)$. For a dataset, n will often be in the $100s$ of $1000s$, while m may be significantly smaller. For instance, if $n = 80000$ and $m = 19$, we would have

$$n \cdot m \cdot l = 80000 \cdot 542288 \cdot l$$

We propose an alternative approach in which we do not need to consider

all possible subsets K for each Att_i set. Instead of starting with a set K and determining I_K, we start with the sets S that could possibly be a $\overline{K}$ for some K, and then determine all of the K for which $\overline{K} = S$. (We will see later that we actually do not need to determine all such K, but can further refine this to a smaller number of possible Ks that we find.)

If $K \subseteq Att_i$, we know that $\overline{K} \subseteq Att_i$ as well. We can organize all the possible sets S (that could be $\overline{K}$ for some K) into a poset ordered by set containment. We know that if we have a set S with $S = \overline{K}$ for some $K \subseteq Att_i$, then S must be a descendant of Att_i in our poset, as $S \subseteq Att_i$.

We can assign a weight $weight(S)$ for each possible inference set S, where

$$weight(S) = \max_{K \subseteq S} w(K)$$

Then we have that $priv(i) = weight(Att_i)$. Instead of computing this directly by listing out all possible subsets K, we will use the poset to more efficiently compute all of the $priv(i)$ values for all individuals $i \in Ind$.

Let's consider computing these weights as we traverse our poset from bottom to top. Notice that if $S' \subseteq S$, then any $K \subseteq S' \subseteq S$ has already had $w(K)$ computed before we reach S, and is accounted for in $weight(S')$. Thus, we can compute $weight(S)$ by taking the maximum value across the $weight(S')$s for any $S' \subseteq S$ (where S' is a direct child of S), along with considering new $w(K)$s where $K \subseteq S$ but $K \not\subseteq S'$ for any $S' \subseteq S$.

$$weight(S) = \max\{\max_{K \subseteq S \text{ and } K \not\subseteq S' \ \forall S' \subseteq S} w(K), \quad \max_{S' \subseteq S,\ S' \text{ a direct child of } S} weight(S')\}$$

There are thus three major tasks to consider:

1. Determine the possible sets S we can have as inference sets for some K.

2. Construct a poset with these S, starting from our original collection of sets $\{Att_1, \ldots, Att_n\}$.

3. For a given set S in our poset, determine the new sets K with $K \subseteq S$ and $K \not\subseteq S'$ for any $S' \subseteq S$, and determine $weight(S)$.

4.1 Inference Sets

We consider the first task of determining the possible sets S we can have as inference sets for some set K. To do this, we first make some useful observations.

By construction, $\overline{K}$ must be the intersection of some collection of Att_i sets. More precisely, recall that $\overline{K} = (\phi_R \circ \psi_R)(K) = \bigcap_{i \in J} Att_i$, where J is the set of individuals who have the attributes in K. This establishes that our inference sets S must be of the form $S = \bigcap_{i \in J} Att_i$ for some collection of individuals $J \subseteq Ind$.

Lemma 4.1. *For any set $K \subseteq Att$, we have that $\overline{K} = \bigcap_{i \in J} Att_i$ for some collection of individuals $J \subseteq Ind$.*

While this tells us that our inference sets take the form $S = \bigcap_{i \in J} Att_i$ for some collection of individuals $J \subseteq Ind$, it does not tell us whether each such set S will actually have some K with $\overline{K} = S$. However, recall Lemma 2.27, which states that if $K = \bigcap_{i \in I} Att_i$ for some collection of individuals $I \subseteq Ind$, then $\overline{K} = K$. In other words, when $K = S$, we have $\overline{S} = S$.

This answers our first question of determining the possible sets S we can have as inference sets for some K. More precisely, we know that:

1. An inference set S can only be of the form

$$S = \bigcap_{i \in J} Att_i$$

   for some collection of individuals $J \subseteq Ind$.

2. Any set of the form $S = \bigcap_{i \in J} Att_i$ will be the inference set of at least one set K. (i.e., when $K = S$)

4.2 Intersection Poset

Our second task was to construct a poset where the elements are the intersection sets of the form $S = \bigcap_{i \in J} Att_i$ for some $J \subseteq Ind$, and the ordering is set containment.

Definition 4.2. Let $A = \{A_1, \ldots, A_n\}$ be a collection of n sets. The **Intersection Poset** for A is the poset where the elements are sets of the form $\bigcap_{i \in I} A_i$ for

some collection $I \subseteq \{1, \ldots, n\}$, and the ordering is set inclusion.

We will illustrate how to construct this with an example.

Example 4.3. Consider the following relation:

R	a	b	c	d	e	f	g
1	•	•	•	•			
2	•	•	•		•		
3	•	•				•	•
4		•	•			•	•

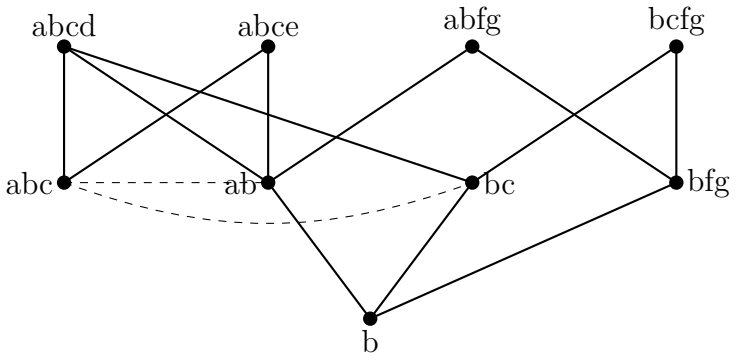
We denote the Att_i sets as follows:

$$Att_1 = abcd, Att_2 = abce, Att_3 = abfg, Att_4 = bcfg$$

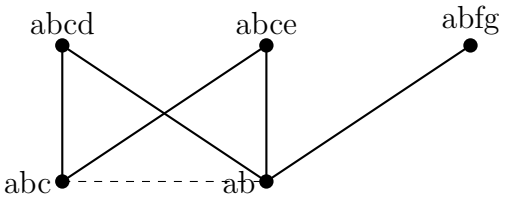
We want to take all intersections of the Att_i sets, and organize them into a poset (ordered under set containment). The top row will be the Att_i sets. Then we will take all pairwise intersections of the Att_i sets. These pairwise intersections will create the next ″row″ of the intersection poset. We can then take pairwise intersections of this new row, and continue on. (There will be some slight adjustments to this in order for the poset to be constructed properly.)

If we are taking the intersection of two sets S_1 and S_2, we then need to create a parent-child relationship in our poset between the parents (S_1, S_2) and the child ($S_1 \cap S_2$).

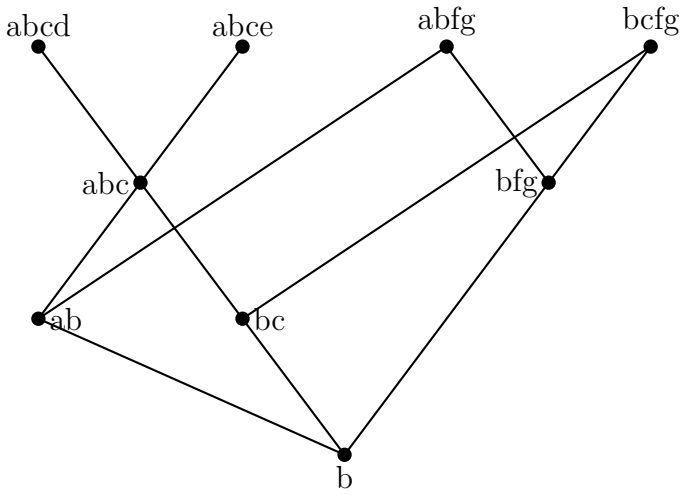
Proceeding as we have described so far, we get the following diagram for our poset:



We see that in our second row, we have two sets (ab, bc) who have a parent (abc) in the same row. To construct our poset properly, the two sets ab and bc should be on a lower level of the poset than abc is. We will also adjust parent-child relationships to reduce redundancy. For this, consider the following sub-diagram of our poset:



Looking at the diagram above, we see that both *abcd* and *abce* are parents for both *abc* and *ab*. However, since *abc* is a parent of *ab*, we do not need to keep $abcd, abce$ as parents of *ab* as well. (This is implied by transitivity of set containment.) While we could leave these parent-child connections in anyways, we actually will want them removed for when we consider the task of looking for the sets K that infer each S in our poset. After making adjustments to redundant parent-child relationships and moving subsets down in the poset, we get the following:



The first step in creating an algorithm for the process illustrated above is to generate all intersections by generating pairwise intersections row by row. We provide pseudocode for generating the pairwise intersections in Algorithm 1. In Algorithm 2, we provide the pseudocode for generating all intersections.

We now discuss the complexity of the algorithms, and explain why Algorithm 2 will in fact give us all possible intersections of the Att_i sets, and we confirm that the algorithm will terminate.

Theorem 4.4. *Algorithm 1 has complexity* $\mathcal{O}(k^2 \cdot m)$*, where* $= |A|$ *and* $m = |Att|$.

Algorithm 1 getPairwiseIntersections(A)

Input:

- $A = \{A_1, \ldots, A_k\}$ is a collection of k sets.
- Each set A_i contains elements from a set Att of m elements.

Output:

- $List$ contains all pairwise intersections of sets in A

```
List ← {}
for (i = 1 to k − 1) do
    for (j = 2 to k) do
        List ← List ∪ {A_i ∩ A_j}
    end for
end for
```

Proof. Algorithm 1 will compute the intersection of each unique pair of elements from A. Since there are k sets in A, there are $\binom{k}{2} = \frac{k \cdot (k-1)}{2}$ pairs. For each pair, we need to compute the intersection between two sets containing elements from S. These sets are represented as boolean vectors of length m, where we include a 1 if the element is in the set, and 0 if not. To compute the intersection, we use logical and on the boolean vectors. This gives m comparisons to get the intersection. Thus, this algorithm has complexity $\mathcal{O}(k^2 \cdot m)$. □

Note that k will be the size of a row in our poset. On the first iteration of Algorithm 2, when we call Algorithm 1, $k = n$. However, on future iterations, k may be either larger or smaller than n.

Algorithm 2 will terminate when no new intersections are added in a given iteration.

Theorem 4.5. *Algorithm 2 will terminate after* $\log n$ *iterations of the while loop, where* $n = |Ind|$ *is the number of sets in* $A = \{A_1, \ldots, A_n\}$.

The fact that this algorithm will terminate in $\log n$ iterations of the while loop follows from the following lemma, which establishes that after the k^{th} iteration, we have found all intersections of up to 2^k distinct sets from A. Thus, after $\log n$ iterations, we will have found all intersections of up to n distinct sets. Since there are only n distinct sets in A, we will have found all possible intersections after $\log n$ iterations.

Algorithm 2 getAllIntersections(A)

Input:

- $A = \{A_1, \ldots, A_n\}$ is a collection of n attribute sets, where $n = |Ind|$.
- Each set A_i contains elements from a set Att of m elements.

Output:

- $NewTotal$ contains all intersections of any finite collection of sets from A

$OldTotal \leftarrow \{\}$
$NewTotal \leftarrow A$
$NewSet \leftarrow A$
while $(NewTotal \neq OldTotal)$ **do**
  $OldTotal \leftarrow NewTotal$
  $NewSet \leftarrow getPairwiseIntersections(A)$
  $NewTotal \leftarrow OldTotal \cup NewSet$
end while

Let $A = \{A_1, \ldots, A_n\}$. Let S_k denote $NewSet$ after the k^{th} iteration of Algorithm 2. In other words, S_k consists of all pairwise intersections of sets from S_{k-1}.

Lemma 4.6. *Let $2^{k-1} + 1 \leq m \leq 2^k$. Then all intersections of m distincts sets from A are contained in S_k.*

Note that S_k may also contain intersections of a smaller number of distinct sets as well.

Proof. We prove this by induction on k.
Base Case: $k = 1$
When $k = 1$, we have $2^{1-1} + 1 \leq m \leq 2^1$, so $m = 2$. On the first iteration, we take all intersections of two sets from A, so S_1 contains all intersections of two distinct sets from A, as desired.
Induction Hypothesis: Assume that for any collection of m' distinct sets in A, where $2^{k-1} + 1 \leq m' \leq 2^k$, their intersection is in S_k.
Induction Step: We want to show that for any collection of m distinct sets in A, where $2^k + 1 \leq m \leq 2^{k+1}$, their intersection is in S_{k+1}.

Let $B = \bigcap_{i \in I} A_i$ be an intersection of m distinct sets from A, where $I \subseteq \{1, \ldots, n\}$, $|I| = m$, and $2^k + 1 \leq m \leq 2^{k+1}$. Since the elements of S_{k+1} are pairwise intersections of elements frm S_k, we want to show that $B = B_1 \cap B_2$ for some $B_1, B_2 \in S_k$, so that $B \in S_{k+1}$.

We thus need to find sets B_1 and B_2 that are the intersections of m_1 and m_2 sets respectively, where $2^{k-1} + 1 \leq m_1, m_2 \leq 2^k$. We will show that we can take our index set I for B and split it into two sets I_1 and I_2, where $I_1 \cup I_2 = I$, $|I_1| = m_1$, and $|I_2| = m_2$. We will then have $B_1 = \bigcap_{i \in I_1} A_i$ and $B_2 = \bigcap_{i \in I_2} A_i$, so that $B = B_1 \cap B_2$. We now establish that we can split I into two such sets.

Let $I = \{i_1, \ldots, i_m\}$. If m is even, let

$$I_1 = \{i_1, \ldots, i_{\frac{m}{2}}\}$$

$$I_2 = \{i_{\frac{m}{2}+1}, \ldots, i_m\}$$

Then we have $|I_1| = |I_2| = \frac{m}{2}$ and clearly $I_1 \cup I_2 = I$. Since $2^k + 1 \leq m \leq 2^{k+1}$ and m is even, we have $2^k + 2 \leq m \leq 2^{k+1}$, which gives $2^{k-1} + 1 \leq \frac{m}{2} \leq 2^k$. Thus, we have $2^{k-1} + 1 \leq |I_1|, |I_2| \leq 2^k$ as desired.

If m is odd, let

$$I_1 = \{i_1, \ldots, i_{\lfloor \frac{m}{2} \rfloor + 1}\}$$

$$I_2 = \{i_{\lfloor \frac{m}{2} \rfloor + 1}, \ldots, i_m\}$$

Then we have $|I_1| = |I_2| = \left\lfloor \frac{m}{2} \right\rfloor + 1$ and clearly $I_1 \cup I_2 = I$ (the two sets overlap on one element). Since $2^k + 1 \leq m \leq 2^{k+1}$ and m is odd, we have:

$$\begin{aligned} 2^k + 1 \leq m &\leq 2^{k+1} - 1 \\ \left\lfloor \frac{2^k + 1}{2} \right\rfloor \leq \left\lfloor \frac{m}{2} \right\rfloor &\leq \left\lfloor \frac{2^{k+1} - 1}{2} \right\rfloor \\ 2^{k-1} \leq \left\lfloor \frac{m}{2} \right\rfloor &\leq 2^k - 1 \\ 2^{k-1} + 1 \leq \left\lfloor \frac{m}{2} \right\rfloor + 1 &\leq 2^k \end{aligned}$$

Thus, we have $2^{k-1} + 1 \leq |I_1|, |I_2| \leq 2^k$ as desired. □

While the while loop will terminate after $\log n$ iterations, we also need to consider the complexity of what is being done at each iteration. In each itera-

tion, we are calling the function $getPairwiseIntersections(NewSet)$, which has complexity $\mathcal{O}(k^2 \cdot m)$, where k is the number of sets in $NewSet$, and $m = |Att|$ as before. m will be the same on each iteration, but k will vary. However, since Algorithm 2 is computing all intersections of n sets, we know this can have no more than 2^n unique sets generated. On the other hand, we also know these sets are subsets of Att, which has size m. There are at most 2^m unique subsets of Att. Thus, Algorithm 2 will produce at most $min\{2^m, 2^n\}$ sets. This gives us the following theorem:

Theorem 4.7. *Let $m = |Att|$ and $n = |Ind|$. Then Algorithm 2 will produce at most $min\{2^m, 2^n\}$ unique sets.*

In the context of databases, n will be the number of individuals in our database, and m will be the number of attributes in our database. There will often be a significant difference in the size of n compared to m. For instance, we may have $n = 80000$, while $m = 19$. Thus, if our algorithm does reach the worst case and produce 2^{19} sets, that is only 524288 sets, which is relatively small.

Lemma 4.8. *Algorithm 2 has approximate complexity of $\mathcal{O}(m \cdot 2^m)$, where $m = |Att|$.*

Proof. By Theorem 4.7, we have that Algorithm 2 generates at most $min\{2^m, 2^n\}$ unique sets, where the minimum will be 2^m in the context of datasets. Each set is represented as a boolean vector of length m, where we include a 1 if an attribute is in our set, or 0 otherwise. The intersection of two sets is computed using logical and on these two vectors, which has complexity $\mathcal{O}(m)$. Thus, Algorithm 2 has complexity $\mathcal{O}(m \cdot 2^m)$ if it generates *exactly* 2^m sets. However, some sets may be generated more than once, so this complexity is simply an approximation. □

To create our Intersection Poset, we need to make some modifications to our algorithm for generating pairwise intersections. We do not want to store the same set in more than one place, so one modification is that we will only add $A_i \cap A_j$ to $NewSet$ if $A_i \cap A_j$ has not been created previously. In either case, when we process $A_i \cap A_j$, we will also need to add parent-child relationships between A_i, A_j and $A_i \cap A_j$. Further, if $A_i \cap A_j \in A$, we will potentially need to adjust parent-child relationships involving $A_i \cap A_j$ for redundancy, as discussed in the example above. The pseudocode for this is provided in Algorithm 3. The complexity of Algorithm 3 is the same as that of Algorithm 2, as no major changes have been introduced that would change the complexity.

4.3 Computing $weight(S)$

We are aiming to compute

$$weight(S) = \max_{K \subseteq S} w(K) = \max\{ \max_{K \subseteq S \text{ and } K \not\subseteq S' \ \forall S' \subseteq S} w(K), \quad \max_{S' \subseteq S} weight(S') \}$$

for each S in our poset. This means that for any child $S' \subseteq S$, we consider the weights for these S', and only need to determine $w(K)$ for any $K \subseteq S$ if $K \not\subseteq S'$ for any $S' \subseteq S$. To compute $w(K) = lk(K) * dg(I_K)$, we need to determine I_K (or alternatively $\overline{K}$, where $\overline{K} = K \cup I_K$). We claim that for any $K \subseteq S$ where $K \not\subseteq S'$ for any $S' \subseteq S$, $\overline{K} = S$.

Lemma 4.9. *For a set of attributes $K \subseteq Att$, $\overline{K} = \bigcap_{i \in I} Att_i$ is the unique smallest (with respect to set containment) intersection of attribute sets for which $K \subseteq \bigcap_{i \in I} Att_i$.*

Proof. We have that $\overline{K} = (\phi_R \circ \psi_R)(K) = \bigcap_{i \in J} Att_i$, where J is the set of individuals who have the attributes in K. We need to show that there is no set $S' = \bigcap_{j \in J'} Att_j$ with $K \subseteq S' \subseteq \overline{K}$, and we also need to show that there cannot be two such minimal $\overline{K}$s.

First, let $\overline{K} = \bigcap_{i \in J} Att_i$. We need to show that there is no $S' = \bigcap_{j \in J} Att_j$ with $K \subseteq S' \subset \overline{K}$. We have that $\overline{K} = (\phi_R \circ \psi_R)(K) = \bigcap_{i \in J} Att_i$. If $K \subseteq S' = \bigcap_{j \in J'} Att_j$ for some collection of individuals J', then all j in J' must share the attributes K. Thus, $J' \subseteq J$, so $\bigcap_{i \in J} Att_i \subseteq \bigcap_{j \in J'} Att_j$. Thus, $\overline{K} \subseteq S'$, contradicting $S' \subset \overline{K}$. Thus, $\overline{K}$ must be the smallest intersection set that K is contained in.

We now need to show that there is a unique smallest intersection set that K is contained in. Suppose for a contradiction that there are two distinct intersection sets S_1, S_2 with $K \subseteq S_1$, $K \subseteq S_2$, and $K \not\subseteq S_1'$, $K \not\subseteq S_2'$ for any $S_1' \subset S_1$ or $S_2' \subset S_2$. Notice that these conditions mean that $S_1 \not\subset S_2$ and $S_2 \not\subset S_1$.

Let $S_1 = \bigcap_{i \in I_1} Att_i$ and $S_2 = \bigcap_{i \in I_2} Att_i$. Then since $K \subseteq S_1, S_2$, this means that all attributes in K are shared by the individuals in I_1 and I_2, so $I_1 \cup I_2 \subseteq J$. Note that we must have $I_1 \cup I_2 \subsetneq J$. Otherwise, if $I_1 \cup I_2 = J$, then $\overline{K} = S_1 \cap S_2$. But since $S_1 \not\subseteq S_2$, we must have that $S_1 \cap S_2 \subsetneq S_1$, this contradicts that there is no intersection set S_1' with $K \subset S_1' \subset S_1$, as $S' = S_1 \cap S_2$ satisfies this. Thus,

$I_1 \cup I_2 \subsetneq J$. Then we have

$$\overline{K} = (\phi_R \circ \psi_R)(K) = \bigcap_{i \in J} Att_i \subset \bigcap_{i \in I_1 \cup I_2} Att_i$$

This gives that $\overline{K} \subset S_1, S_2$, a contradiction. Thus, there is a unique smallest intersection set $\bigcap_{i \in I} Att_i$ with $K \subseteq \bigcap_{i \in I} Att_i$.

□

This means that in order to determine the sets K for which $\overline{K} = S$ for a given intersection set $S = \bigcap_{i \in I} Att_i$, we want to find the subsets $K \subseteq S$ for which $K \not\subseteq S'$ for any other intersection set $S' \subseteq S$. In other words, S must be the smallest (with respect to subset containment) intersection set that K is contained in.

Let $POSET$ denote our intersection poset.

Theorem 4.10. *Let $K \subseteq Att$ and $S \in POSET$. $\overline{K} = S$ if and only if $K \subseteq S$ and $K \not\subseteq S'$ for any $S' \subseteq S$.*

Recall that

$$weight(S) = \max\{ \max_{K \subseteq S \text{ and } K \not\subseteq S' \ \forall S' \subseteq S} w(K), \qquad \max_{S' \subseteq S} weight(S') \}$$

By Theorem 4.10, the sets $K \subseteq S$ with $K \not\subseteq S'$ are precisely the sets K with $\overline{K} = S$. This means that (once we have the desired sets K) we can easily compute $w(K)$ for these as $w(K) = lk(K) * dg(S \setminus K)$. It would be helpful if we could further narrow down which of these Ks have the potential to produce a maximal $w(K)$, and avoid considering any K for which we know $w(K)$ cannot be maximal. In other words, if we know a certain K' has $w(K') < w(K)$ for some other K, we would like to eliminate K' from our search to begin with.

Lemma 4.11. *If $K \subseteq K'$ and $\overline{K} = \overline{K'}$, then $w(K) \geq w(K')$.*

Proof. To show that $w(K) \geq w(K')$, we need to show that $lk(K) * dg(I_K) \geq lk(K') * dg(I_{K'})$, where $I_K = \overline{K} \setminus K$ and $I_{K'} = \overline{K'} \setminus K'$.

By Proposition 3.7(i), if $K \subseteq K'$, then $w(K) \geq w(K')$. Since $K \subseteq K'$ and $\overline{K} = \overline{K'}$, we then have that $I_K \supseteq I_{K'}$. By Proposition 3.9, since $I_{K'} \subseteq I_K$, we have that $dg(I_{K'}) \leq dg(I_K)$. Since $lk(K) \geq lk(K')$ and $dg(I_K) \geq dg(I_{K'})$, we then have:

$$w(K') = lk(K') * dg(I_{K'}) \leq lk(K) * dg(I_K) \leq w(K)$$

as desired. □

Suppose we are considering a given set S in our poset, and suppose that both K and K' have S as their inference set, with $K \subseteq K'$. We thus only need to be able to find the minimal (with respect to set containment) such sets K and compute their weights. These are the only sets that could potentially produce a maximal $w(K)$.

Thus, for each S in our poset, we need to find the minimal K with $\overline{K} = S$. These Ks must satisfy two conditions:

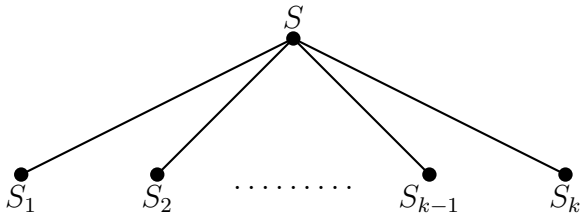
1. $K \subseteq S$ and $K \not\subseteq S'$ for any $S' \subset S$, where $S, S' \in POSET$.
2. There is no smaller $K' \subset K \subseteq S$ for which K' also satisfies Condition (1).

Condition (1) establishes that $\overline{K} = S$ and Condition (2) establishes that there is no smaller $K' \subseteq K$ with $\overline{K'} = S$ as well.

Finding Ks With Condition 1.
First, we will determine how to enumerate the sets K satisfying (1). Let

$$\mathcal{K}_S = \{K \subseteq S \mid K \not\subseteq S_i \text{ for any } S_i \subseteq S\}$$

In other words, $\overline{K} = S$ for all $K \in \mathcal{K}_S$. In our poset, we would have:



Since our poset is ordered by set containment, this means $S_i \subseteq S$ for $i = 1, \ldots k$.

We may also have other sets $S_i' \subseteq S_i \subseteq S$. However, for determining which sets are in $\mathcal{K}_S$, if we state that $K \not\subseteq S_i$, then $K \not\subseteq S_i'$ follows immediately for any $S_i' \subseteq S_i$. Thus, it is enough to consider just the S_i sets that are direct children of S in the poset when trying to establish (for a given K) that $K \not\subseteq S_i$ for any $S_i \subseteq S$.

For K to be in $\mathcal{K}_S$, we need that $K \subseteq S$ and $K \not\subseteq S_i$ for any $i = 1, \ldots k$, where $S_1, \ldots S_k$ are the direct children of S in the poset. Now, let $S_i^c = S \setminus S_i$. Then the condition that $K \subseteq S$ and $K \not\subseteq S_i$ means that $K \cap S_i^c \neq \emptyset$ for each $i = 1, \ldots, k$.

This means that K must contain at least one element from each S_i^c set. We can then rephrase Condition (1) as:

1. K contains at least one element from each S_i^c for $i = 1, \ldots, k$, where $S_1, \ldots, S_k$ are the direct children of S in the poset.

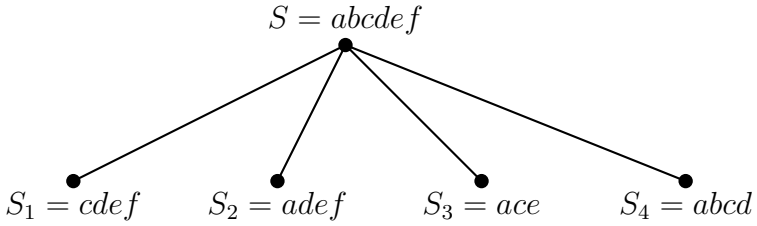
The pseudocode for finding all of the sets K satisfying condition 1 for a given S can be found in Algorithms 4 and 5. Observe that the complexity will be $|S_1{}^c| \cdot |S_2{}^c| \ldots \cdot |S_k{}^c|$, where k is the number of direct children of S. Note that we need to do this for *each* S in our poset. However, since we only need the minimal Ks, we will not actually implement these algorithms. Instead, we will use the algorithms presented in the next section.

Finding Minimal Ks.

We now need to determine how to find the sets K that will also satisfy Condition (2). These are the sets that are minimal (with respect to set containment) amongst the sets satisfying Condition (1). For Condition (1), we needed K to contain *at least* one element from each S_i^c. If we want a minimal K, to start, we should restrict this to sets that contain *exactly* one element from each S_i^c.

Now, some of the S_i^c sets will have elements in common, so choosing exactly one element from each will not always guarantee a minimial K. We illustrate this with an example.

Example 4.12. Consider the following set S and direct children S_1, S_2, S_3, S_4 within our poset.



We then have:

$$\begin{array}{lcccccccc} S_1^c & = & \text{a} & \text{b} & & & & & \\ S_2^c & = & & \text{b} & \text{c} & & & & \\ S_3^c & = & & \text{b} & & \text{d} & & \text{f} & \\ S_4^c & = & & & & & \text{e} & \text{f} & \end{array}$$

Now, if we chose $abde$, that uses one element from each S_i^c. However, since b is common to the first three sets, choosing be will also select one element from each and $\{b, e\} \subseteq \{a, b, d, e\}$. Thus, $abde$ is not a minimal K, so it would not satisfy Condition (2).

The example above suggests that we need to choose the elements that occur in the highest frequency first, and then continue to choose elements appearing in the highest frequency amongst the remaining sets that we have not yet chosen an element from. We illustrate what this process would look like, again with the example above. Here, we will find *all* of the minimal K, satisfying both conditions (1) and (2).

Example 4.13. We start by listing all of our S_i^c sets, and first choose an element occurring with the highest frequency.

$$\begin{array}{lllllll} S_1^c & = & a & b & & & \\ S_2^c & = & & b & c & & \\ S_3^c & = & & b & & d & f \\ S_4^c & = & & & & e & f \end{array} \tag{4.1}$$

Here, there is only one element occurring with the highest frequency, so we choose it (b). This means we have now chosen an element from S_1^c, S_2^c, S_3^c. We now only need to choose an element from S_4^c. There are two choices here, which give us two minimal Ks:

$$be, bf$$

At this point, we have now exhausted all minimal Ks that contain b in them. We now remove b from consideration and proceed as before:

$$\begin{array}{lllllll} S_1^c & = & a & & & & \\ S_2^c & = & & c & & & \\ S_3^c & = & & & d & & f \\ S_4^c & = & & & & e & f \end{array} \tag{4.2}$$

Here, f is our only letter occurring with maximal frequency in what we have left, so we choose it. We see that in the remaining rows, we have only one option for S_1^c and one for S_2^c, so the only K we get from here is:

$$acf$$

We now remove f from consideration and repeat:

$$\begin{array}{lllll} S_1^c & = & a & & \\ S_2^c & = & & c & \\ S_3^c & = & & & d \\ S_4^c & = & & & e \end{array} \tag{4.3}$$

Here, we have four options to choose from for our maximal frequency letter. It does not matter which we choose. In this case, the only option is to take the

single element available from each, which gives us the following K:

$$acde$$

To see how this process ends, normally we would proceed by removing a at the next step.

$$\begin{array}{rcccc} S_1^c & = & & & \\ S_2^c & = & c & & \\ S_3^c & = & & d & \\ S_4^c & = & & & e \end{array} \tag{4.4}$$

We now have a row with no elements. This means we cannot proceed further, as we need to select an element from each row. Thus, our algorithm stops at this point. We have then found the following desired minimal sets K:

$$be, bf, acf, acde$$

In Algorithm 7 we outline the algorithm for generating all of our minimal Ks, and in Algorithm 6, we provide pseudocode for the algorithm which finds the next element that we are looking for at any stage in the process illustrated above.

Theorem 4.14. *Algorithm 6 has at best constant time complexity, and at worst $\mathcal{O}(m)$, where m is the number of attributes.*

Proof. In Algorithm 6, there are two cases. In the first case, we have not finished finding a set K, and thus our next element will be found simply by looking at the next position in our traversal vector. This is the first part of the if statement, and is thus constant time complexity.

If we are in the else part of our if statement, this means we have found all possible sets K with the elements we have chosen so far. This means we need to go back a step and change our last element to the next one in its traversal vector. In this case, we pop all of our stack variables, and try again. In the worst case, we will need to work our way back to the very beginning of the stacks. The stacks will hold at worst m elements, where m is the number of attributes we are working from. Thus at most, it will take m steps to get our next element.

Thus, the worst case complexity is $\mathcal{O}(m)$. □

The implementation of $getTraversal$ and $getNewRows$ can be found in the appendix. $getTraversal$ requires sorting a vector of size m. The sort function

in $C++$ has linearathmic complexity, $\mathcal{O}(m * log2(m))$ [cpl]. $getNewRows$ is linear in the number of rows we started with, so it is $\mathcal{O}(k)$ complexity.

In Algorithm 7, during each iteration of the while loop, $getNewRows$, $getTraversal$ and $findNextElement$ are each called once. These three functions have complexity $\mathcal{O}(k)$, $\mathcal{O}(m * log2(m))$ and somewhere between constant and $\mathcal{O}(m)$ complexity, respectively, where m is the number of attributes and k is the number of rows we originally started with. Here, the number of rows is the number of direct children of our set S. Thus, if Algorithm 7 outputs l sets, then it will have complexity $\mathcal{O}(l \cdot \max\{m * log2(m), k\})$. This gives us the following theorem:

Theorem 4.15. *Algorithm 7 will be polynomial in the number of sets it outputs.*

Algorithm 3 createIntersectionPoset(A)

Input:

- $A = \{A_1, \ldots, A_n\}$, a list of Att_i sets

Notes:

- Combines and modifies Algorithms 1 and 2 so that if an intersection set was generated in a previous iteration, we do not add to $NewSet$ on future iterations. It also adds in parent-child relations.
- $flagSet$ flags the sets we need to move down a level in the POSET

$POSET \leftarrow \{\}$
$OldTotal \leftarrow \{\}$
$NewTotal \leftarrow A$
$OldSet \leftarrow A$
while $(NewTotal \neq OldTotal)$ **do**
  $NewSet \leftarrow \{\}$
  $flagSet \leftarrow \{\}$
  for $(i = 1$ to $|OldSet| - 1)$ **do**
    for $(j = 2$ to $|OldSet|)$ **do** ▷ Let $S_i = OldSet[i], S_j = OldSet[j]$
      if $(S_i \cap S_j \notin OldSet)$ **then**
        $NewSet \leftarrow NewSet \cup \{S_i \cap S_j\}$
      end if
      if $(S_i \cap S_j \in OldSet)$ **then**
        $flagSet \leftarrow flagSet \cup \{S_i \cap S_j\}$
      end if
      $(S_i \cap S_j).parents \leftarrow (S_i \cap S_j).parents \cup S_i \cup S_j$
      $S_i.children \leftarrow S_i.children \cup (S_i \cap S_j)$
      $S_j.children \leftarrow S_j.children \cup (S_i \cap S_j)$
    end for
  end for
  $OldSet \leftarrow OldSet \setminus flagSet$
  $NewSet \leftarrow NewSet \cup flagSet$
  $POSET \leftarrow POSET \cup \{OldSet\}$ ▷ Add $OldSet$ not $NewSet$
  $OldTotal \leftarrow NewTotal$ ▷ Set up for next iteration
  $NewTotal \leftarrow OldTotal \cup NewSet$
  $OldSet \leftarrow NewSet$
end while
$POSET \leftarrow POSET \cup \{OldSet\}$

Algorithm 4 getAllK

Finds all K satisfying condition 1. That is, $K \subseteq S$ and $K \not\subseteq S'$ for any $S' \subseteq S$.

Input:

- Set S from $POSET$
- $SC\_Children = \{S_1^c, \ldots S_k^c\}$, the complements in S of the direct children of S.

$AllK \leftarrow generateCrossProd({S_1}^c, \ldots {S_k}^c)$

Algorithm 5 generateCrossProd

Generates all elements of $X_1 \times \cdots \times X_k$.

Input:

- Sets $X_1, \ldots, X_k$

$AllElements \leftarrow \{\}$
if $(n = 1)$ **then**
  $AllEls \leftarrow X_1$
  return
end if
for $(i = 1$ to $|X_1|)$ **do**
  $ToMerge \leftarrow generateCrossProd(X_2, \ldots, X_k)$
  for (each $\overline{x}$ in $ToMerge$) **do**
    $\overline{x} \leftarrow X_1[1] \times \overline{x}$
  end for
end for

Algorithm 6 findNextElement

Input:

- *ElsChosen*, a stack variable that contains a list of the attributes selected for the current K, in order
- *RowsSelected*, a stack variable where each element on the stack is a list of the additional rows selected when we choose a new attribute for K
- *Traversal*, a stack variable where each element is a vector listing the attributes from highest to lowest frequency in the remaining rows
- *Pos*, a stack variable that tells us where in the corresponding traversal vector we have worked up to
- *EliminatedEls*, a stack variable where each element contains a list of the attributes we have exhausted all K for at the current stage

Conditions:

- There are m attributes and k rows.
- *new_rows* will be any rows *new_el* is in that were not already in *RowsSelected*
- This algorithm will find the next element in our set K. The traversal vectors will be modified after returning from this function.
- Returns 1 if all K have been found.

if $(pos = \emptyset)$ **then** ▷ All Ks found
  return 1
end if
$curr\_pos \leftarrow pos.top()$ ▷ Let $new\_el = traversal.top()[curr\_pos]$
if ($curr\_pos < m$ and $new\_el \notin ElsChosen$ and $new\_el \notin EliminatedEls$) **then**
  $ElsChosen.push(new\_el)$
  $pos.top() + +$
  $pos.push(0)$ ▷ Sets pos to 0 for next traversal
  return ▷ Next element has been found
else
  $pos.pop()$
  if $(pos = \emptyset)$ **then** ▷ All Ks found
    return 1
  end if
  $traversal.pop()$
  $RowsMatched.pop()$
  $ElsChosen.pop()$
  $EliminatedEls.push(ElsChosen.top())$
  $findNextElement(ElsChosen, RowsSelected, Traversal, Pos, EliminatedEls)$
end if

Algorithm 7 getAllMinimalK

Conditions:

- $getTraversal$ finds the next traversal vector
- $getNewRows$ finds the next set of matched rows and adds to the $RowsSelected$ stack

$pos, traversal, ElsChosen \leftarrow \emptyset$
$RowsSelected, EliminatedEls \leftarrow \emptyset$
$pos.push(0)$
$traversal \leftarrow getTraversal$
$done \leftarrow findNextElement$
while (not done) **do**
    $next\_el \leftarrow ElsChosen.top()$
    $RowsSelected \leftarrow getNewRows$
    if $(|RowsSelected| = k)$ **then** ▷ All rows selected
        output K
    end if
    $traversal \leftarrow getTraversal$
    $done \leftarrow findNextElement$
end while

Chapter 5

Connection to k-Anonymity

5.1 Weights for k-Anonymity

In k-anonymity, it is assumed that the attributes in our quasi-identifier are commonly found in public databases, so we can assign a likelihood weight of 1 to these attributes. We may also choose to assign a danger weight of 0 to these attributes since we assume they are public knowledge and (possibly) not dangerous to acquire.

5.2 Effect of k-Anonymity on Privacy Score

One way of turning a database into a k-anonymous one is by adding additional attributes to certain individuals in order to make them identical to at least $k-1$ others (within the set of quasi-identifier attributes). For non-binary attributes such as age, these attributes may be generalized to larger ranges. For instance, someone with the age 26 might end up with the age $25-30$ after generalization. This can be seen as being equivalent to adding new attributes. Essentially, this individual kept their original attribute of 26 but gained the new attributes $25, 27, 28, 29$ and 30. In the following, we will consider the effect on the database privacy score when k-anonymity is achieved through adding additional attributes within the quasi-identifier.

We first consider an example to illustrate that it is not immediately clear whether k-anonymity will improve privacy. While k-anonymity is intended to prevent unique identification of an individual in a database, it is possible that new inferences will appear in our database.

Example 5.1. Consider the following small database, before and after making it 2-anonymous.

A	B	C	D	E
•		•	•	
	•	•		•

A	B	C	D	E
•	•	•	•	
•	•	•		•

Here, we are assuming that our quasi-indentifier is $\{A, B\}$. Notice that in our original database, we could not infer anything from C, but in our new database, $C \implies AB$. While this is a new inference, it actually will not affect the privacy score, since it will have a weight of 0 (assuming we have a danger weight of 0 assigned to our quasi-identifier attributes). In any new inference we see, if the new attributes inferred are within our quasi-identifier, this will not affect our privacy score. Notice that we also have new sets appearing such as BD that were not in an attribute set for any individual in our original database. Here, $BD \implies AC$, so the inference does include attributes outside the quasi-identifier. However, observe that $D \implies AC$ as well, and $D \subseteq BD$ and $D \implies AC$ was an inference in our original database. Thus, by Proposition 3.11(i), $w(BD) \leq w(B)$, so BD will not contribute to the privacy score.

Let D be our original database, and let D' be our new database after k-anonymity has been achieved by giving individuals additional attributes within the quasi-identifier. For a given attribute set $K \subseteq Att$, let I_{K_D} and $I_{K'_D}$ denote the set of attributes inferred from K in the databases D and D', respectively. Let $\overline{K_D} = K \cup I_{K_D}$ and let $\overline{K'_D} = K \cup I_{K'_D}$. Alternatively, let $\overline{K_D} = (\phi_D \circ \psi_D)(K)$ and $\overline{K'_D} = (\phi'_D \circ \psi'_D)(K)$ denote the attribute closure operators in the databases D and D', respectively. For a given individual $i \in Ind$, let Att_{D_i} and $Att_{D'_i}$ denote the set of attributes for individual i in the databases D and D', respectively. Finally, let $w_D(K)$ and $w_{D'}(K)$ denote the weight of K in the databases D and D', respectively.

Theorem 5.2. *Let D be a database, and D' be a k-anonymized version of D, where the only alterations to D were adding additional attributes (within the quasi-identifier) to individuals. Assume that $d(x) = 0$ for all x in the quasi-identifier. Then $w_{D'}(K) \leq w_D(K)$ for any set $K \subseteq Att$.*

Proof. We have that

$$w_D(K) = lk(K) * dg(I_{K_D})$$

$$w_{D'}(K) = lk(K) * dg(I_{K_{D'}})$$

Thus, to show that $w_{D'}(K) \leq w_D(K)$, we need to show that $dg(I_{K_{D'}}) \leq dg(I_{K_D})$.

Let $Att = Q \cup V$, where Q is our quasi-identifier, and V consists of the remaining attributes. In computing $dg(I_K)$, any $x \in Q$ will contribute 0 to $dg(I_K)$ since $d(x) = 0$ by assumption. Thus, $dg(I_K) = dg(I_K \cap V)$. This means that in order to show that $dg(I_{K_{D'}}) \leq dg(I_{K_D})$, we need to show that $dg(I_{K_{D'}} \cap V) \leq dg(I_{K_D} \cap V)$. Recall from Propisition 3.9 that if $I_K \subseteq I_{K'}$, then $dg(I_K) \leq dg(I_{K'})$. Thus, to show that $dg(I_{K_{D'}} \cap V) \leq dg(I_{K_D} \cap V)$, we will show that $I_{K'_D} \cap V \subseteq I_{K_D} \cap V$. More precisely, we will show that $\overline{K_{D'}} \cap V \subseteq \overline{K_D} \cap V$.

Recall that $\overline{K_D} = K \cup I_{K_D} = (\phi_D \circ \psi_D)(K)$. Now, $\psi_D(K)$ is the set of individuals in D sharing the attributes in K. Since attributes are only ever added to individuals in D', we must have that $\psi_D(K) \subseteq \psi_{D'}(K)$. We have that

$$\overline{K_{D'}} = \bigcap_{i \in \psi_{D'}(K)} Att_{D'_i}$$

$$\overline{K_D} = \bigcap_{i \in \psi_D(K)} Att_{D_i}$$

Since the only new attributes given to an individual in D' are quasi-identifier attributes, we have that $Att_{D'_i} = Att_{D_i} \cup Q_i$, for some set of quasi-identifier attributes $Q_i \subseteq Q$. Thus, $Att_{D'_i} \cap V = Att_{D_i} \cap V$. We then have:

$$\overline{K_{D'}} \cap V = \bigcap_{i \in \psi_{D'}(K)} \widetilde{Att_{D_i}}$$

$$\overline{K_D} \cap V = \bigcap_{i \in \psi_D(K)} \widetilde{Att_{D_i}}$$

where $\widetilde{Att_{D_i}} = Att_{D_i} \cap V$. Now, $\psi_D(K) \subseteq \psi_{D'}(K)$, so $\overline{K_{D'}} \cap V \subseteq \overline{K_D} \cap V$ (as long as $\psi_D(K) \neq \emptyset$), and this is what we wanted to show.

We now consider the case when $\psi_D(K) \neq \emptyset$. This happens when K was not a subset of any individual's attribute set in D. If $\psi_{D'}(K) = \emptyset$ (in other words, K is also not a subset of any individual's attribute set in D'), then $\overline{K_{D'}} \cap V \subseteq \overline{K_D} \cap V$ as both are $\emptyset$. We now consider the case where $\psi_D(K) = \emptyset$, but $\psi_{D'}(K) \neq \emptyset$. Since $\psi_D(K) = \emptyset$, we have that $K \not\subseteq Att_{D_i}$ for any $i \in Ind$. On the other hand, since $\psi_{D'}(K) \neq \emptyset$, there is at least one i for which $K \subseteq Att_{D'_i}$. The only way we can have $K \not\subseteq Att_{D_i}$ and $K \subseteq Att_{D'_i}$ for some i is if K contains some attributes from the quasi-identifier. When we take $K \cap V$, we must have $\psi_D(K \cap V) \neq \emptyset$, as $K \cap V$ must have individuals with tje attributes in $K \cap V$ in the original database for an individual in D' to have these (non-quasi-identifier) attributes.

Now, $\overline{K \cap \overline{V}} \cap V = \overline{K} \cap V$, so the above argument can now be applied on $K \cap V$ instead of K.

Thus, we have that $\overline{K_{D'}} \cap V \subseteq \overline{K_D} \cap V$, so $w_{D'}(K) \leq w_D(K)$. □

Theorem 5.3. *Assume that $d(x) = 0$ for all attributes x in the quasi-identifier. Applying k-anonymity to a database by only adding in new attributes (within the quasi-identifier) for an individual cannot increase the database privacy score.*

This follows immediately from Theorem 5.2.

Chapter 6

Topological Approach to Improve Privacy

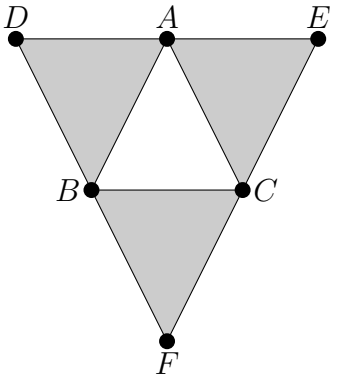
6.1 Topological Description of Privacy Loss

If we return to looking at the Dowker complexes of a database, we can explain how some privacy loss occurs from a topological perspective. This is discussed in [Erd17]. There are at least two ways we see an inference occurring in a database. The first is coming from a free face (a simplex contained in exactly one maximal simplex) inferring the maximal simplex that it is contained in. The second is coming from within the intersection of a set of simplices. (A simplex within the intersection inferring the intersection). We illustrate each in the examples below.

Example 6.1. Here, we consider an inference from a free face. This can occur from a simplex on the boundary of our Dowker complex, or from a hole within. Consider the following dataset:

R	a	b	c	d	e	f
1	•	•		•		
2	•		•		•	
3		•	•			•

This gives the following Dowker complex:

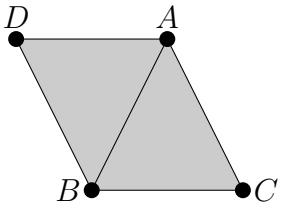


We can see that we have inferences coming from free faces in the outer boundary, such as $\overline{AE} = ACE$. We also have inferences coming from free faces in the boundary of the hole in the middle, such as $\overline{AC} = ACE$.

Example 6.2. Here, we consider an inference coming from within an intersection of simplices. Consider the following dataset:

R	a	b	c	d
1	•	•	•	
2	•	•		•

This gives the following Dowker complex:



We can see from our dataset that we have the inferences $A \implies B$ and $B \implies A$ coming from within the intersection of the two triangles. If A or B were in another simplex that did not contain the other, we would no longer have this inference. Thus, this will not occur *every* time we have an intersection of simplices. (Recall that to get $\overline{K}$ we needed to intersect *all* attribute sets containing K. These are the intersection sets for which this inference from all subsets will occur.)

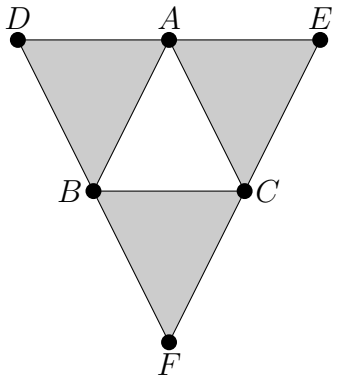
6.2 Modifying Databases to Preserve Privacy

We will consider the case where the inference is a result from a topological hole in the Dowker complex. In order to eliminate this inference, we can fill in this

hole. We can do this in more than one way. We need to consider how useful the data will be after doing this as well. Let's return to our example to see how we could modify our database.

Example 6.3. Here, we have the following database and Dowker complex:

R	A	B	C	D	E	F
1	•	•		•		
2	•		•		•	
3		•	•			•



In order to fill in the hole ″ABC″, we need at least one individual in the database to have all three attributes. One option is to take any individual who has attributes contained in ABC and give them all three attributes. This may not be ideal for data usability, and could also introduce higher dimensional holes as our simplicial complex will have new higher dimensional simplices added to it. If we were to do this, our database would look like this:

R	A	B	C	D	E	F
1	•	•	•	•		
2	•	•	•		•	
3	•	•	•			•

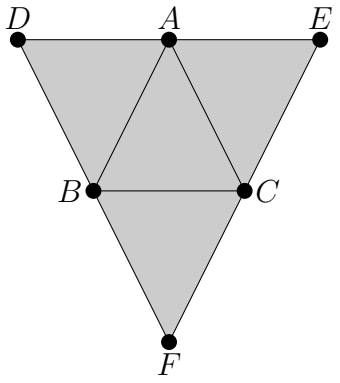
We no longer have any inferences from within $\{A, B, C\}$ to other attributes, but we can still have inferences to $\{A, B, C\}$. This looks similar to k-anonymity, in that if the attributes in $\{A, B, C\}$ all have danger weights of 0, these new inferences will not affect the privacy score. Notice for instance, that originally we had $BD \implies A$, but now we have $BD \implies AC$. The new attributes inferred will all be attributes contributing to the hole.

We notice that this may not be an ideal way to resolve the hole, as the attributes may not all have danger weights of 0, and further, this may be undesirable to do in terms of data usability.

A second option for filling the hole is simply to add a new individual into the database who has all of the attributes in the hole. This avoids higher-dimensional simplices from being added, and prevents any new inferences from being introduced. In our example, the new database would look like this:

R	A	B	C	D	E	F
1	•	•		•		
2	•		•		•	
3		•	•			•
4	•	•	•			

The resulting Dowker complex would be:



Note that this is not what the Dowker complex would be under our first option. In that option, there would be higher-dimensional simplices added on top of this.

We see here now that any inferences from within $\{A, B, C\}$ are removed. We do not have any new inferences introduced either. In terms of data usability, this is something that would need to be analyzed more closely. If we only needed to fill in one hole in a database with 500 people, it seems less likely to affect data usability than in the small example we have here. We would need to also consider how many holes and how many false individuals we need to introduce to our dataset.

In the following, M will denote our topological hole.

Theorem 6.4. *Let D be a database. Let $M = \{a_1, \ldots, a_n\} \subseteq Att$. Suppose that for each subset $F \subset M$ with $|F| = n - 1$, there is an individual $i \in Ind$ with $F \subseteq Att_i$ but $M \not\subseteq Att_i$. Then, for any subset $K \subset M$, $I_K \cap M = \emptyset$.*

In other words, if $I_K \cap M = \emptyset$, the inference from K must only contain attributes outside of the hole M.

Proof. Let $M = \{a_1, \ldots, a_n\}$ and suppose that for any subset $F \subseteq M$ of size $n-1$, there is an individual $i \in Ind$ with $F \subseteq Att_i$, but $M \not\subseteq Att_i$. This means that each of these individuals have exactly $n-1$ of the n attributes in M. Thus, for each such individual, there is exactly one attribute $x \in M$ that they do not have.

Let $K \subseteq M$. We want to show that $I_K \cap M = \emptyset$. In other words, any attribute we infer from K must be outside of M.

To compute I_K, we are looking at the additional attributes (outside of K) shared by all individuals who share K. $I_K \cap M$ is the set of these additional attributes that are also in M.

Consider the set $M \setminus K$. We know that $I_K \subseteq M \setminus K$, as $I_K \cap K = \emptyset$, by definition. For each $x \in M \setminus K$, we have an individual $i \in Ind$ with $F = M \setminus \{x\} \subseteq Att_i$. Thus, for each $x \in M \setminus K$, we have an individual with $K \subseteq F$ but $x \notin Att_i$. This means that x cannot be in I_K. Therefore, no $x \in M \setminus K$ is in I_K, which means there are no elements of M in I_K. Thus, $I_K \cap M = \emptyset$, as desired. □

Theorem 6.5. *Let D be a database. Let $M = \{a_1, \ldots, a_n\} \subseteq Att$. Suppose that for each $F \subset M$ with $|F| = n-1$, there is an individual $i \in Ind$ with $F \subseteq Att_i$ but $M \not\subseteq Att_i$. Suppose we modify D to a database D' by adding an individual j with $Att_j = M$. Then for each $i \in Ind$, $priv_{D'}(i) \leq priv_D(i)$. We also have that $priv_{D'}(j) = 0$.*

In other words, adding this individual to the database will either improve or maintain privacy for the other individuals. The score itself will actually improve, but note that this strict improvement only comes from adding a false individual with 0 score, so it is not necessarily improving everyone's individual scores.

Proof. Let ${I_K}^D$ denote I_K in the database D, and let ${I_K}^{D'}$ denote I_K in the database D'. First, we show that $priv_{D'}(j) = 0$. Since j has $Att_j = M$, for any $K \subseteq M$, we must have $\overline{K} \subseteq M$. By Theorem 6.4, for any $K \subset M$, ${I_K}^D \cap M = \emptyset$ in D. In D', I_K could only change due to the new individual j with $Att_j = M$. Since ${I_K}^D = \bigcap_{i \in I} Att_i \setminus K$ (for some set of individuals I), in D' we have ${I_K}^D = (\bigcap_{i \in I} Att_i \cap M) \setminus K$. Thus, we have ${I_K}^{D'} \subseteq {I_K}^D$. As observed above, ${I_K}^D \cap M = \emptyset$, so ${I_K}^{D'} \cap M \subseteq \emptyset$. Thus, we have ${I_K}^{D'} = \emptyset$. We also have that $I_M = \emptyset$, as j has attribute set M and no other attributes to infer. Since $I_K = \emptyset$

for any $K \subseteq M$, we have that $w(K) = 0$ in D'. Since $priv_{D'}(j) = \max_{K \subseteq M} w(K)$, $priv_{D'}(j) = 0$.

Now we show that $priv_{D'}(i) \leq priv_D(i)$ for any individual i in D. No attributes for i have changed in D', but the inferences may have. Since $Att_j = M$, the only time that I_K could change in D' from what it was in D is if $K \subseteq M$. In other words, the only time we will have an additional individual sharing K is if that individual is j. However, we showed above that $I_K = \emptyset$ in D' for any $K \subseteq M$. The weights of any other sets will remain unchanged. Thus, $w_{D'}(K) = 0 \leq w_D(K)$ for any $K \subseteq M$. Thus, $priv_{D'}(i) \leq priv_D(i)$. □

In the proof of the previous theorem, we proved the following:

Corollary 6.6. *Let D be a database. Let $M = \{a_1, \ldots, a_n\} \subseteq Att$. Suppose that for each $F \subset M$ with $|F| = n - 1$, there is an individual $i \in Ind$ with $F \subseteq Att_i$ but $M \not\subseteq Att_i$. Suppose we modify D to a database D' by adding an individual j with $Att_j = M$. Then for any $K \subseteq M$, $I_K = \emptyset$ in D' and $w_{D'}(K) = 0$. In other words, there is no inference in D' from any set $K \subseteq M$.*

The above results tell us that if we fill in a hole using the method of adding an individual with the missing attributes, our privacy score of the new database will improve, if we had inferences coming from the original hole.

Chapter 7

Future Work

Data Usability Score. As discussed in the last section, while we suggest potential ways to improve the privacy score of a database, we need a way to measure data usability as well. Thus, future work should involve creating a Data Usability Score. When a technique to anonymize a database is created, both the Data Privacy Score and Data Usability Score should then be taken into consideration in assessing the usefulness of the technique. In [KK12], it was stated that there was no standard metric for measuring data usability that has been widely accepted.

Improving Database Privacy. The main focus of this thesis was on computing the database privacy score. However, more work needs to be done in determining what can be done to improve privacy (while maintaining data usability). In the previous section, we offered one suggestion of removing some inferences from the database. More work needs to be done to assess whether this is a useful technique to use in terms of data usability. If so, algorithms would also need to be developed to find the topological holes in the dataset.

In [Sam01], an algorithm was presented to modify a database into a minimally k-anonymous one, in order to minimize the risk of losing data utility. Another task would be to create an algorithm with a similar goal of minimizing the amount of changes to the database needed to achieve a privacy score under a given threshold.

Algorithms. Future work should involve running the algorithms presented on real-world datasets. In particular, we need to establish that it is feasible to do, and if not, improve the algorithms further so that they can be run.

Another future direction should involve parallelizing the algorithms for further improvements.

Probabilities. All of the inferences considered here needed to be absolute. Future work should generalize the database privacy score to include when an inference occurs some percentage of the time as well. This non-absolute inference may still allow accurate inferences to be uncovered.

Data Analysis. While the focus here was in creating a database privacy score, the algorithms created to find the inferences could also be used on datasets (where privacy is not a concern) to make these inferences. This is also something that may be combined with the probability work listed above.

We also propose investigating how the techniques presented here could be used to solve (supervised) classification problems. The setup here would be to include the output result as an attribute in the dataset, and then look for minimal subsets that imply the output result.

Other Types of Data Input. In this thesis, we only considered binary attributes. We can generalize this to numeric or categorical variables. For categorical variables, if there are 5 options, we can create 5 binary attributes for these. Each individual would have 1 of the 5 attributes. For numeric variables, we can first convert to ranges so it becomes categorical, and from there convert to binary attributes. However, more work needs to go into determining if there is a better way to proceed with non-binary attributes. The condition that each individual can only have 1 out of 5 attributes may also affect how we proceed with algorithms, or how we can make inferences.

Generalization to String Data. In this thesis, we looked at *sets* of attributes. We mentioned above the potential of using this for data analysis to find inferences. Future work should also look at how to generalize the techniques for computing inferences to strings. More precisely, we would like to determine in a given string database whether all strings with a particular substring also always contain other substrings. This could be used in applications such as DNA analysis and cryptography. We propose using substrings as our attributes in this context. Here, our Dowker complex would then use all possible substrings as the vertices.

Chapter 8

Conclusion

In this thesis, we presented a Database Privacy Score which measures the risk of information leakage in a linkage attack on an anonymous database. We presented algorithms for computing this score, as well as source code in $C++$ in the appendix.

We also explored how k-anonymity affects the Database Privacy Score, and saw that it cannot worsen the privacy score, as long as we assume the quasi-identifier attributes are not dangerous to infer.

As well, we considered some topological explanations as to how inferences end up occurring in a database, and presented a method to remove those inferences from the database.

Future work will need to consider how to measure data usability so that both privacy and data usability can be measured when assessing the usefulness of anonymization techniques.

Appendix A

C++ Code

```
// Database Privacy Score

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
#include <numeric>
#include <vector>
#include <array>
#include <stack>
#include <boost/unordered_map.hpp>
#include <boost/dynamic_bitset.hpp>
// From https://github.com/boostorg/dynamic_bitset/issues/34 : hash function for dynamic_bitset
// put in namespace boost if using unordered_map from boost, but change to std if using std map
#include <boost/functional/hash.hpp>
namespace boost {
  template<> struct hash<boost::dynamic_bitset<>> {
    std::size_t operator()(const boost::dynamic_bitset<>& bs) const {
      std::string h;
      boost::to_string(bs, h);
      return std::hash<std::string>{}(h);
    }
  };
}

using namespace std;
using namespace boost;


// Intersection POSET Creation functions:
void createIntersectionPOSET(vector<unordered_map<dynamic_bitset<>, vector<unordered_map<dynamic_bitset<>, int>>>>& POSET,
                             unordered_map<dynamic_bitset<>, vector<unordered_map<dynamic_bitset<>, int>>>& A);

void getPairWiseIntersections(unordered_map<dynamic_bitset<>, vector<unordered_map<dynamic_bitset<>, int>>>& A,
                              unordered_map<dynamic_bitset<>, vector<unordered_map<dynamic_bitset<>, int>>>& new_set,
                              vector<unordered_map<dynamic_bitset<>, vector<unordered_map<dynamic_bitset<>, int>>>>& POSET);

void processFlagged(unordered_map<dynamic_bitset<>, vector<unordered_map<dynamic_bitset<>, int>>>& A,
                    unordered_map<dynamic_bitset<>, vector<unordered_map<dynamic_bitset<>, int>>>& new_set,
                    vector<unordered_map<dynamic_bitset<>, vector<unordered_map<dynamic_bitset<>, int>>>>& POSET,
                    unordered_map<dynamic_bitset<>, int> flag);

void modifyParentChild(unordered_map<dynamic_bitset<>, vector<unordered_map<dynamic_bitset<>, int>>>& set_a,
                       unordered_map<dynamic_bitset<>, vector<unordered_map<dynamic_bitset<>, int>>>& set_for_parents,
                       dynamic_bitset<> Atti, dynamic_bitset<> Attj, dynamic_bitset<> a);

// Privacy Scores from POSET:
void getPrivi(vector<unordered_map<dynamic_bitset<>, vector<unordered_map<dynamic_bitset<>, int>>>>& POSET,
              unordered_map<dynamic_bitset<>, float>& last_row_weights,
              vector<float> lk, vector<float> dg);

void createM(unordered_map<dynamic_bitset<>, int> children, vector<int> Sind, vector<dynamic_bitset<>>& M);

void getIndices(dynamic_bitset<> S, vector<int>& Sind);

float getAvgPrivD(unordered_map<dynamic_bitset<>, float>& last_row_weights);

// getAllK functions:
void getAllK(vector<dynamic_bitset<>> &M, dynamic_bitset<> TK, vector<int> TKIndices,
             vector<float> lk, vector<float> dg, float *max_weight);

void column_sum_dynamic_bitset(int m, int n, vector<dynamic_bitset<>> &M, vector<int> &sums);

bool findNextEl(int m, int n, stack<int>& pos, stack<vector<int>>& sum, stack<vector<int>>& traversal,
                stack<dynamic_bitset<>>& rows_matched, stack<dynamic_bitset<>>& rows_so_far, stack<int>& els_so_far, dynamic_bitset<

void getNewRows(int m, int n, int new_el, vector<dynamic_bitset<>> M, stack<dynamic_bitset<>>& rows_matched, stack<dynamic_bitset<>>

void getNextSum(int m, int n, stack<vector<int>>& sum, stack<dynamic_bitset<>> rows_matched, vector<dynamic_bitset<>> M);

bool isComplete(int m, dynamic_bitset<> so_far);

float processK(int n, int num_features, dynamic_bitset<> chosen_els, dynamic_bitset<> TK, vector<int> TKIndices,
               vector<float> lk, vector<float> dg);

void getTraversal(int n, vector<int> curr_sum,
                  stack<vector<int>>& traversal);


// Weight Calculations:
float likelihood(dynamic_bitset<>  K, vector<float> lk);
float danger(dynamic_bitset<> IK, vector<float> dg);
float weight(dynamic_bitset<>  K, dynamic_bitset<>  IK,
             vector<float> lk, vector<float> dg);

// Convert from length num_features to shorter ones:
void getTKIndices(dynamic_bitset<> TK, int num_features, vector<int> &TKIndices);

int main(int argc,char **argv){

    // Set Up for A, POSET:
```

```
    unordered_map<dynamic_bitset<>, vector<unordered_map<dynamic_bitset<>, int>>> A;
    vector<unordered_map<dynamic_bitset<>, int>> pc_vec;
    unordered_map<dynamic_bitset<>, int> parents;
    unordered_map<dynamic_bitset<>, int> childs;
    pc_vec.push_back(parents);
    pc_vec.push_back(childs);

    // Initially, we will leave the vector of parent/child hash tables empty:
    A.insert({dynamic_bitset<> (string("111100")), pc_vec});
    A.insert({dynamic_bitset<> (string("111010")), pc_vec});
    A.insert({dynamic_bitset<> (string("111001")), pc_vec});
    A.insert({dynamic_bitset<> (string("110110")), pc_vec});
    A.insert({dynamic_bitset<> (string("110000")), pc_vec});

    // Our POSET will be a list of these "A"s; each "A" is a row in the POSET.
    // **We don't add A to the POSET until we've run through and modified it to add both parents and children.
    vector<unordered_map<dynamic_bitset<>, vector<unordered_map<dynamic_bitset<>, int>>>> POSET;
    createIntersectionPOSET(POSET, A);

    // Set up lk and dg weights:
    vector<float> lk(6, 1); //length 6, all 1s for testing
    vector<float> dg(6, 1);

    // Now traverse POSET to get Priv_i scores (they will be stored in last_row_weights):
    unordered_map<dynamic_bitset<>, float> last_row_weights;
    getPrivi(POSET, last_row_weights, lk, dg);

    // Print all individual Priv_i Scores:
    auto it_lrw = last_row_weights.begin();
    for (it_lrw = last_row_weights.begin(); it_lrw != last_row_weights.end(); it_lrw++){
        cout << "Atti: " << (*it_lrw).first << " Weight: " <<  (*it_lrw).second << endl;
    }

    // Compute and print Average Score:
    cout << "Average Score: " << getAvgPrivD(last_row_weights) << endl;

}
/************************************POSET Creation Functions*****************************************************************/

void createIntersectionPOSET(vector<unordered_map<dynamic_bitset<>, vector<unordered_map<dynamic_bitset<>, int>>>& POSET,
                unordered_map<dynamic_bitset<>, vector<unordered_map<dynamic_bitset<>, int>>& A){
    /*

         Creates the Intersection POSET from A, where the elements of POSET are all possible intersections between sets from A.
         The ordering is standard set inclusion.
         Each element in the POSET is represented as an unordered map, where the key is the set (as a dynamic bitset),
         and the "value" is a vector with two hash tables. The first one is a list of all parents of the given set.
         The second is a list of all children of the given set.
         Parents and children are wrt set containment. We only take the minimal number of these needed.
         ie. if A->B->C, we only store A->B and B->C; we do not include a parent/child relation for A->C.

         *We don't add A (a row) to the POSET until we've taken all intersections from it. This is because
         we may drop elements from A as they're pushed down in the POSET.

         Assumption: This assumes that we do not have any individual with Atti contained in Attj.
                    If this were the case, the "top row" (after POSET creation) would no longer contain Atti, as
                    it would be pushed down lower in the POSET.
                    If this case needs to be considered, the code could be modified to flag these sets in the top row
                    and keep track of their weights to add to the last_row_weights hash table later.

        Input:  A = {Att1,...,Attn} our original set of attributes
        Output: POSET (we create this)

        USES:   <boost/unordered_map.hpp>
                <boost/dynamic_bitset.hpp>
                <vector>

        Functions: getPairWiseIntersections

    */

    unordered_map<dynamic_bitset<>, vector<unordered_map<dynamic_bitset<>, int>>> new_set;

    getPairWiseIntersections(A, new_set, POSET);
    POSET.push_back(A);
    while (!new_set.empty()){
        A = new_set;
        new_set.clear();
        getPairWiseIntersections(A, new_set, POSET);
        POSET.push_back(A);
    }

}


void getPairWiseIntersections(unordered_map<dynamic_bitset<>, vector<unordered_map<dynamic_bitset<>, int>>& A,
                              unordered_map<dynamic_bitset<>, vector<unordered_map<dynamic_bitset<>, int>>& new_set,
                              vector<unordered_map<dynamic_bitset<>, vector<unordered_map<dynamic_bitset<>, int>>>& POSET){

    /*
```

```
    Creates all pairwise intersections from the sets in A and adds to new_set.

    A will be our initial list of sets (values are two hash tables - one for parents, one for children).
    new_set will initially be empty, but added to throughout this function.
    We go through A and take every pairwise intersection, and add these to new_set.
    If the intersection was already in A, we move that set (and associated parent/children) into new_set
    (and remove from A).
    Both A and new_set will be revised. A will be added to POSET after this function call.

    In the first call, A is our set {Att1, Att2,...,Attn}.
    Each element Atti is a set of attributes, stored as a dynamic bitset.
    A is stored as a hash table:
        -The keys are the Atti sets.
        -The values consist of a vector of two hash tables:
                -the first hash table is a list of the parents to Atti
                 ( this is accessed by .at(0) )
                -the second hash table is a list of the children to Atti
                    -the keys are the Attj sets that are the parents/children
                    -the values are simply integers '1' (not really needed)
                 ( this is accessed by .at(1) )

    Let a be an intersection set we found.
    If a is already in A, we will remove it from A and add to new_set instead.
    (ie. a needs to be on a lower level of the POSET)
    We also may modify parent/child relationships.
    eg. if we had a -> B -> C and a->C, we drop the a -> C relationship
    In other words, if B is a parent of a, and a and B have a common parent,
    we keep that parent for B, but remove it as a parent from a.
    Thus, only minimal parental relationships are kept.

    ALGORITHM:
    Let's suppose A={Att1,...,Attn}.
    We traverse through all pairs of Atti sets in A.
    For each i,j (i<j), we take the intersection of Atti with Attj - call this a.
    For each intersection a, we need to do the following:
            -if a is not in A already, we add it to new_set
             (at the end, A will be appended to POSET; this is essentially the next "row" in our POSET;
              new_set will be added in the next iteration)
            -either way, we need to add Atti as a parent of a (as long as a!=Atti), and same for Attj
            -we also need to add a as a child of Atti and Attj (if a is not equal to it)
            -if a is in A already, then we need to flag it
            -the flag will be dealt with later (we will need to potentially adjust parents)
            -flag will be a hashtable (if anything is in there, it's been flagged; we store a 1 as value)

    NOTES:
    -must use pointer to parent_child vector in order to modify
    -(*parent_child_ptr).at(0) is the hash table for parents
    -(*parent_child_ptr).at(1) is the hash table for children

    Inputs: -A  (list of original sets, as a hash table)
                (key is the set, value is a vector with the first entry a parent hash table,
                 second entry a child hash table)
            -POSET


    Outputs:    -new_set (list of the pairwise intersections + their parent/child hash tables)
                -A will be modified (possibly elements removed, or parent/child relations modified)

    USES:   <boost/unordered_map.hpp>
            <boost/dynamic_bitset.hpp>
            <vector>

    Functions: modifyParentChild
               processFlagged


*/

// Can only do pairwise intersections if at least 2 elements:
if (A.size() < 2){
    return;
}

vector<unordered_map<dynamic_bitset<>, int>> *parent_child_ptr, *parent_child_Atti_ptr, *parent_child_Attj_ptr, parent_child_new
unordered_map<dynamic_bitset<>, int> flag, parent_a, child_a;
dynamic_bitset<> Atti, Attj, a;

// Traverse through all pairs in A:
auto it1 = A.begin();
int i=1,j=0;
for ( it1 = A.begin(); it1 != A.end(); ++it1 ){
    auto it2=it1;
    j=i;
    for ( it2++; it2 != A.end(); it2++){
        j++;

        // Two bitsets we're intersecting:
        Atti=(*it1).first;
        Attj=(*it2).first;
        a = (Atti&Attj);
```

```
            /*
                If a is not already in A, we add it to the new set.
                If a already in new_set, we only modify parent/child relations.
                If a is in A, we flag it, as we'll need to re-adjust parents after.
                ie. flagged means a has parents in the same "row" as it (ie. parents in A)
            */
            if (A.find(a)==A.end()){

                // a is not in A; add to new_set (or modify if already there)

                if (new_set.find(a)!=new_set.end()){

                    // a in new_set; modify parent/child

                   modifyParentChild(new_set, A, Atti, Attj,a);

                } else {

                    // a is not already in new_set ; add to new_set

                    parent_child_new.clear();
                    parent_a.clear();
                    child_a.clear();

                    if (a!=Atti){
                        parent_a.insert({Atti,1});
                        parent_child_Atti_ptr = &((A.find(Atti))->second); //gets ptr to vec with parent, child tables
                        ((*parent_child_Atti_ptr).at(1)).insert({a,1});

                    }

                    if (a!=Attj){
                        parent_a.insert({Attj,1});
                        parent_child_Attj_ptr = &((A.find(Attj))->second); //gets vec with parent, child table
                        ((*parent_child_Attj_ptr).at(1)).insert({a,1});

                    }

                    // Add parent and child hash tables into a vector. Then insert into new_set:
                    parent_child_new.push_back(parent_a);
                    parent_child_new.push_back(child_a);
                    new_set.insert({a, parent_child_new});
                }

            } else {

                // a is already in A. Modify parent/child in A:
                modifyParentChild(A, A, Atti, Attj,a);

                // We flag it (if not flagged already) so that once we are done we can go back and check parent/child relations to m
                if (flag.find(a)==flag.end()){
                    flag.insert({a,1});
                }
            }
            i++;
        }
    } // end traversal of all pairs

    processFlagged(A,new_set,POSET,flag);

}

void processFlagged(unordered_map<dynamic_bitset<>, vector<unordered_map<dynamic_bitset<>, int>>>& A,
                    unordered_map<dynamic_bitset<>, vector<unordered_map<dynamic_bitset<>, int>>>& new_set,
                    vector<unordered_map<dynamic_bitset<>, vector<unordered_map<dynamic_bitset<>, int>>>>& POSET,
                    unordered_map<dynamic_bitset<>, int> flag){

    /*
           We modify parent-child relations for any redundancy in the sets we flagged.
           eg. if a->b->c and a->c, we can drop a->c as it's a redundant relation
           (We need to drop these for our algorithm for getAllK to work).
           We also push down flagged sets in the POSET.
           (It's possible there was nothing flagged.)
           a will denote the set we flagged.
           We want to do the following:

           Let a be a flagged set in A.
           For each parent x of a:
               -extract the parents of x
               -compare these to the parents of a
               -if y is a parent of BOTH x and a, we need to:
                    -remove y from a's parent list
                    -remove a from y's child list
           After this, we want to remove a from A, and instead move a
           (with revised parents) into new_set.
           (So we are pushing it down in the POSET).
           NOTE: x,a will both be in A

           We will use B to denote another row in the POSET.

           Inputs:  -flag is a hash table of sets in A that appeared as intersections of other sets in A
```

```
   Inputs:  -flag is a hash table of sets in A that appeared as intersections of other sets in A
            -A is a set of sets
            -new_set is where flagged sets will be added to
            -POSET includes previous rows before A, new_set (parents may be in these rows)

   Outputs: -A, new_set, POSET may all be modified as the parent-child relationships are modified


   USES:   <boost/unordered_map.hpp>
           <boost/dynamic_bitset.hpp>
           <vector>

*/

unordered_map<dynamic_bitset<>, vector<unordered_map<dynamic_bitset<>, int>>>* B;
vector<unordered_map<dynamic_bitset<>, int>> *a_pc_ptr, *y_pc_ptr, x_pc; //parent-child vec (pointer except for x, which is
unordered_map<dynamic_bitset<>, int> *a_parent_ptr, *y_child_ptr, x_parent, x_child;
dynamic_bitset<> a, x, y, to_erase;

auto it_flag = flag.begin();
for ( it_flag = flag.begin(); it_flag != flag.end(); it_flag++){
    a = (*it_flag).first;

    // We want a ptr so we can adjust, but then we will move into new_set:
    a_pc_ptr = &((A.find(a))->second);
    a_parent_ptr = &((*a_pc_ptr).at(0));

    // For each parent of a:
    auto it_parent = (*a_parent_ptr).begin();
    for (it_parent = (*a_parent_ptr).begin(); it_parent != (*a_parent_ptr).end(); it_parent++){
        // x is a parent of a
        x = (*it_parent).first;
        x_pc = (A.find(x))->second;
        x_parent = x_pc.at(0);
        auto it_aparents = (*a_parent_ptr).begin();
        for (it_aparents = (*a_parent_ptr).begin(); it_aparents != (*a_parent_ptr).end(); it_aparents++){

            // Erase any from last iteration (see below):
            (*a_parent_ptr).erase(to_erase);

            // y is a parent of x
            y = (*it_aparents).first;
            if (x_parent.find(y) != x_parent.end()){

                // y is a parent of both x and a

                // Remove y as a parent of a (we flag as to_erase and do on next iteration):
                // Because of incrementing on loop, we can't erase it here.
                to_erase = y;

                // Remove a as a child of y. y could be in any level of the POSET above a (or in A).
                // Check A first. If not, reverse backwards through the POSET.
                // NOTE: A is not stored in POSET yet.
                if (A.find(y) != A.end()){

                    // y is in A

                    // Remove a as a child of y:
                    y_pc_ptr = &((A.find(y))->second);
                    y_child_ptr = &((*y_pc_ptr).at(1));
                    (*y_child_ptr).erase(a);

                } else {

                    // y is in some earlier level of the poset
                    auto it_poset_reverse = POSET.rbegin();
                    // B is a row in the POSET. B must be a pointer below; otherwise it won't update.
                    for (it_poset_reverse = POSET.rbegin(); it_poset_reverse != POSET.rend(); it_poset_reverse++){
                        B = &(*it_poset_reverse);
                        if ((*B).find(y) != (*B).end()){
                            // y is in B

                            // Remove a as a child of y:
                            y_pc_ptr = &(((*B).find(y))->second);
                            y_child_ptr = &((*y_pc_ptr).at(1));
                            (*y_child_ptr).erase(a);

                            // Now that we have found it, we need to stop searching POSET:
                            break;
                        }

                    }
                }
            }
        }
        (*a_parent_ptr).erase(to_erase); //erase one from last iteration
    }

    // Now we need to move a into new_set, and erase from A:
    new_set.insert({a, *a_pc_ptr});
    A.erase(a);
}
```

```
}

void modifyParentChild(unordered_map<dynamic_bitset<>, vector<unordered_map<dynamic_bitset<>, int>>>& set_a,
                       unordered_map<dynamic_bitset<>, vector<unordered_map<dynamic_bitset<>, int>>>& set_for_parents,
                       dynamic_bitset<> Atti, dynamic_bitset<> Attj, dynamic_bitset<> a){


    /*
        (Possibly) modifies the parents of a and children of Atti, Attj.
        a is the intersection of Atti and Attj, and we want to add a as a child of Atti, Attj.
        (And Atti, Attj as parents of a).
        a is in (or will be in) set_a, and the parents are in set_for_parents (possibly the same).

        PARENT-CHILDREN: *(parent_child_ptr).at(0) is the hash table for parents of a
                         *(parent_child_Atti_ptr).at(1) is the hash table for children of Atti
                         We add Atti as a parent of a, and a as a child of Atti, as long as a!=Atti and the relation
                         is not already there.
                         We do the same for Attj.

        Inputs: -Atti, Attj, a (a is intersection of Atti and Attj)
                -set_a (set a is or will be in), set_for_parents (set Atti, Attj in)

        Outputs: -set_a, set_for_parents are modified
                 -a will be added as a child of Atti, Attj; Atti, Attj added as parents of a
                  (unless Atti=a, or the relation is already there)


        USES:   <boost/unordered_map.hpp>
                <boost/dynamic_bitset.hpp>
                <vector>


    */

    vector<unordered_map<dynamic_bitset<>, int>> *parent_child_Atti_ptr, *parent_child_Attj_ptr;
    vector<unordered_map<dynamic_bitset<>, int>>* parent_child_ptr = &((set_a.find(a))->second);

    if (a!=Atti && ((*parent_child_ptr).at(0)).find(Atti)==((*parent_child_ptr).at(0)).end()){ //parent not already in there
        ((*parent_child_ptr).at(0)).insert({Atti,1});
        parent_child_Atti_ptr = &((set_for_parents.find(Atti))->second);
        ((*parent_child_Atti_ptr).at(1)).insert({a,1});
    }
    if (a!=Attj && ((*parent_child_ptr).at(0)).find(Attj)==((*parent_child_ptr).at(0)).end()){ //parent not already in there
        ((*parent_child_ptr).at(0)).insert({Attj,1});
        parent_child_Attj_ptr = &((set_for_parents.find(Attj))->second);
        ((*parent_child_Attj_ptr).at(1)).insert({a,1});
    }
}

/************************************Privacy Scores from POSET ********************************************************************/
void getPrivi(vector<unordered_map<dynamic_bitset<>, vector<unordered_map<dynamic_bitset<>, int>>>>& POSET,
              unordered_map<dynamic_bitset<>, float>& last_row_weights,
              vector<float> lk, vector<float> dg){

    /*

        Traverses through POSET from bottom to top.
        Recall that priv(i) is the max of w(K) for any K that infers Atti.

        For each S in POSET, determines the maximal weight from any subset K of S.
        ie. -we can find which subsets K will infer S (using getAllK)
            -among these, we want to find the one that produces the highest w(K) score
            -we then compare this to the highest score from the children of S
            -we will associate the highest score from all of these to S and store in the hash-table "weights"
            -we ultimately only need to keep the weights at the last/top row (connected to the individual i)
            -however, we do need to keep the others for comparison as we traverse the POSET
             (we store this in the hash table "weights")
        The "last row" (top row) is the set of individuals in the database (ie. the Atti sets).
        These weights are the priv(i) scores.
        last_row_weights is modified and when returned will contain all the priv(i) scores.

        Assumption: This assumes that we do not have any individual with Atti contained in Attj.
                    If this were the case, the "top row" (after POSET creation) would no longer contain Atti, as
                    it would be pushed down lower in the POSET.
                    If this case needs to be considered, the code could be modified to flag these sets in the top row
                    and keep track of their weights to add to the last_row_weights hash table later.

        NOTE:   We use S to denote an element of the POSET (this is an intersection of Atti's).
                In our getAllK functions, we use TK (total K; ie. K u I_K) instead of S.
                For getAllK, we need a matrix M. This will be a matrix containing the complements of all the children of S.
                This is needed to determine our minimal K (which give maximal weights).

        Inputs: -POSET (stored as a vector, where each entry is a hash table representing a ROW of the POSET)
                -lk, dg likelihood and danger weights (each of length num_features)

        Outputs: -last_row_weights is a hash table with all of the maximal weights from the last/top row
                 -these are all the priv(i) scores
```

```
        USES:   <boost/unordered_map.hpp>
                <boost/dynamic_bitset.hpp>

        Functions: getAllK
                   getIndices

    */


    unordered_map<dynamic_bitset<>, vector<unordered_map<dynamic_bitset<>, int>>>* row_ptr;
    unordered_map<dynamic_bitset<>, int> children;
    unordered_map<dynamic_bitset<>, float> weights;
    dynamic_bitset<> S, child;
    vector<dynamic_bitset<>> M;
    vector<int> Sind;
    int row_number = 0;
    float var=0, *max_weight;
    max_weight = &var;


    // Traverse POSET from bottom to top (rbegin means we start at the END of the POSET):

    auto it_prows = POSET.rbegin();
    for (it_prows = POSET.rbegin(); it_prows != POSET.rend(); it_prows++){ // traverse rows
        row_number++;
        row_ptr = &(*it_prows);

        auto it_row = (*row_ptr).begin();
        for (it_row = (*row_ptr).begin(); it_row != (*row_ptr).end(); it_row++){ // traverse sets in row
            S = (*it_row).first;
            getIndices(S,Sind);

            // Get children of S:
            children = ((*it_row).second).at(1);
            createM(children, Sind, M);
            auto it_children = children.begin();

            // Get M:
            *max_weight = 0;
            getAllK(M, S, Sind,lk,dg, max_weight);

            // Now compare max_weight against all children of S:
            for (it_children = children.begin(); it_children != children.end(); it_children++){
                child = (*it_children).first;
                if (weights.at(child) > *max_weight){
                    *max_weight = weights.at(child);
                }
            }

            // Add in S with weight to weights hash table:
            if (row_number < POSET.size()){
                weights.insert({S,*max_weight});
            } else {
                last_row_weights.insert({S,*max_weight});
            }
        }

    }

}

void createM(unordered_map<dynamic_bitset<>, int> children, vector<int> Sind, vector<dynamic_bitset<>>& M){


    /*

        From a set S in our POSET, we need to create M.
        M is a matrix where each row is the complement of a child of S.
        However, we need the bitsets (rows) of M to be the same length as the number of elements in S.
        (rather than of length num_features).
        eg. -if num_features=6 and S=110100 (abd), we want M to be of size mx3, not mx6
            -instead of a child being 100000 (a), we would have 100
             (to denote the set with a but not b or d)

        To create M, we then need the children of S.
        We will also need Sind (indices of non-zero elements of S).
        Both of these are done outside of the function, as both will be needed for other steps as well.

        Therefore, we use children and Sind as input to the function, instead of S.
        As M will have been used in other calls, we need to clear it before we start.

        Inputs: -children is a hash table of the children of the set S we are working with
                 (S is not input to the function, as we need to re-use children, Sind in multiple places, so
                 there is no need to input S here if we use them instead)
                -Sind is a vector containing the indices of non-zero elements of S
                 (S is a bitset, so entries are either 1 or 0)

        Outputs: -we will return M, which is a matrix with the complements of children of S
                 -M will be size mxn, where n is the number of non-zero entries in S (ie. length of Sind)
                 -M is stored as a vector of length m, where each "row" is an entry in the vector (stored as a bitset)
```

```
        USES:   <boost/unordered_map.hpp>
                <boost/dynamic_bitset.hpp>
                <vector>

    */

    dynamic_bitset<> child, new_child;
    M.clear();

    auto it_children = children.begin();
    for (it_children = children.begin(); it_children != children.end(); it_children++){
        child = (*it_children).first;

        /*
            Create shortened version of child.
            Go in reverse because of push_back.
            Flip bits at end because we want the complement.
        */

        new_child.clear();
        auto it_vec = Sind.rbegin();
        for (it_vec = Sind.rbegin(); it_vec != Sind.rend(); it_vec++){
            new_child.push_back(child[*it_vec]);
        }
        M.push_back(new_child.flip());
    }
}


void getIndices(dynamic_bitset<> S, vector<int>& Sind){

    /*

        Takes the bitset S and returns the vector Sind with the indices/locations of all 1s in S.
        Since Sind will be filled and returned, we need to clear it before we fill it, as it will be filled from a previous call.

        USES:   <boost/dynamic_bitset.hpp>
                <vector>
    */

    Sind.clear();

    size_t index = S.find_first();
    while(index != boost::dynamic_bitset<>::npos)
    {
            Sind.push_back(index);
            index = S.find_next(index);
    }


}


float getAvgPrivD(unordered_map<dynamic_bitset<>, float>& last_row_weights){

    /*

        Computes the average database privacy score.
        This is the average of the priv_i scores over all individuals i in the database D.

        USES:   <boost/unordered_map.hpp>
                <boost/dynamic_bitset.hpp>
    */

    float sum = 0;
    int count = 0;
    auto it = last_row_weights.begin();
    for (it = last_row_weights.begin(); it != last_row_weights.end(); it++){
        sum += (*it).second;
        count++;
    }
    return (sum / count);
}

/************************************getAllK Functions*****************************************************************************/
void getAllK(vector<dynamic_bitset<>> &M,
             dynamic_bitset<> TK,
             vector<int> TKIndices,
             vector<float> lk,
             vector<float> dg,
             float *max_weight){

    /*  M is an mxn matrix, represented as a length m vector of n-length bitsets.
        n is the number of elements in the original set we are looking at! (not total # attributes)
        M should be a list of the complements of all children of an element from our POSET.
        min_weight is returned with the max weight for the set represented by M.

        -TK is our original set from our POSET, of length n (S above)
        -TKIndices indicate which elements from our full set of features were in TK
                -eg. if TK=[1101001011], TK_Indices=[013689]

        Finds all K which are:
```

```
Finds all K which are:
    (a) subsets of N={0,1,...,n-1}
    (b) contain at least one element from each row in M
        (ie. each row represents a subset of N - 1 for an element in it, 0 if not)
    (c) minimal amongst all of these
        ie. there is no K' contained in K that also satsifies (a) and (b)

M is constructed outside of this function, but here is the original problem:
Finds all K which are:
    (1) subsets of N={0,1,...,n-1}
    (2) NOT subsets of any row  in S, a list of subsets of N
        (a row is a binary 1/0 representation of some subset of N;
         S is mxn so each row has n entries)
    (3) minimal amongst all of these
        ie. there is no K' contained in K that also satsifies (1) and (2)
K are not returned, but are processed as they're found.
processK function decides what we do with the K - to be modified later.
For now, we just output K as we find them.

ALGORITHM DISCUSSION:
To satisfy (2), we are looking for subsets of N that are not subsets of a list of other subsets of N.
These other subsets are each of the rows in S.
Example: N=abcd, S={ab, bcd}
         Subsets of N that are not subsets of any from S:
               ac, ad, abc, abd, acd, abcd
To satisfy (3), we want the minimal sets K (ordered by containment) from the above list.
Example: ac, ad (all others have ac or ad as a subset)

To satisfy (2):
Let N be our original set and S={S1,...,Sm} be our list of subsets of N.
If K is a subset of N, then K \subseteq N.
If K is NOT a subset of Si, then there is at least one element of K not in Si.
Thus, K n (N\Mi) is non-empty.
For each i=1,...,m this must happen.
So we need to choose at least one element from the complement of S1 in N, the complement
of S2 in N,....,up to the complement of Sm in N.

To satisfy (3):
If we are ultimately looking for minimal ones, a first step is to just choose one element
from each of these complements.
However, this will not guarantee a minimal K.
Example: N=abcde, S={ab, bcd}
         Subsets of N that are not subsets of any from S:
         e, ac, ad, ae, be, ce, de, abc, abd, abe, acd, ace, bce, bde, + any 4 or 5 element subset
         Minimal K:
         e, ac, ad
Our complements of sets in S are C={cde, ae}. If we took one element from each, we could take
d from the first, and e from the second, to get de. But this is not minimal because e is smaller!
e was available to choose in BOTH complements - so if we choose e in one, we should choose it in
the other.

Our algorithm then will look at all of the elements in the complements. M will be our set of these complements.
This algorithm then finds the minimal subsets of N that contain at least one element from each row in M.

ALGORITHM:
Since our rows in M are all represented as binary vectors, each column corresponds to an element in our set N.
For column a, the sum of 1/0 tells us how many rows (subsets) have the element a.
We first sum all of these up. Our first choice of element for a minimal set K will be one with the largest sum.
We choose this element, and this is our choice of element for each row that had it.
We then look at the remainder of the rows that were not matched, re-calculate the sums, and choose an element
with the largest sum amongst these.
We continue until we have chosen an element for each row. At this point we have a K.
Once we have found a K, we remove our last choice of element. We drop it from our list of sums and choose an element
with the highest sum from what's left (if possible). We again continue forwards until we have a K. When there
is no element left (or the elements left have a sum of 0 or less (ie. they don't match any new rows), we move back a
step and try again. Eventually, we use up everything, and move back to the beginning, at which point we're done.

We use the following variables to keep track of all this:

STACK VARIABLES:

sum - this is the sum of the rows in M
    - we add our revised sums here, rather than modifying a single sum vector
    - this allows us to move back a step and see our sum from the previous stage, which is needed
traversal - order in which to traverse the elements at a given stage
          - goes from highest matches in modified M to lowest (ie. order elements by sum high to low)
          - sum[traversal[i]] tells you the number of row matches for the corresponding traversal number
          - technically above should really be sum.top(), traversal.top() to get a single vector out
pos - tells you which spot in the traversal vector we're currently at
rows_matched - set of new rows we matched at this stage
             - not the same as all rows matching the element!
             - only the ones we hadn't already matched; important for when we take a step back
             - adds a vector at each stage, so we can see the matches we had when we take a step back
rows_so_far  - set of all rows matched up to this stage
             - adds a vector at each stage, so we can see the matches we had when we take a step back
els_so_far - list of elements we have so far
           - adds a vector at each stage, so we can see the matches we had when we take a step back

OTHER:

chosen_els - elements chosen so far
           - this is will be a bitset y/n for each element
```

```
                - this is will be a bitset y/n for each element
                - we use this to output K since we can't traverse through a stack
                - since bitset, be careful with indices!! [0] is the LAST element, not the first!
        M - our matrix of complements
        [m,n] = number of rows, columns in M

        USES:   <stack>
                <vector>
                <boost/dynamic_bitset.hpp>

        Functions used: column_sum_bitset
                        getTraversal
                        getNewRows
                        getNextSum
                        findNextEl
                        isComplete
                        processK

    */

    int m = M.size();

    *max_weight = 0;
    // If M is empty, we compute weight for the empty set:
    if (m==0){
        *max_weight = danger(TK, dg);
        return;
    }
    int n = (M.at(0)).size();
    int num_features = TK.size();
    float wk;


    stack<int> pos,els_so_far;
    stack<vector<int>> sum, traversal; //length n vectors
    stack<dynamic_bitset<>> rows_matched, rows_so_far; // binary vecs - 1 if matched, 0 if not ; length m
    dynamic_bitset<> chosen_els (n); // default initializes to 0s ; length n
    vector<int> first_sum (n); //length n

    /* is_match tells us if we have a complete K
       If done=1, we have found all K.
       next_el gives us our next element; we use this to find new rows matched.
    */
    bool is_match, done=0;
    int next_el;

    // Initialize first sum for stack:
    column_sum_dynamic_bitset(m,n,M,first_sum);
    sum.push(first_sum);

    // Get first traversal order, and initialize position to our first entry (0):
    getTraversal(n, sum.top(), traversal);
    pos.push(0);


    // Loop until done=1 (we're done entirely).
    done = findNextEl(m, n, pos, sum, traversal, rows_matched, rows_so_far, els_so_far, chosen_els);

    while (done!=1) {

        next_el = els_so_far.top();

        // Get new rows matched:
        getNewRows(m, n, next_el, M, rows_matched, rows_so_far);

        // Get new sum:
        getNextSum(m, n, sum, rows_matched, M);


        //Check if we're done:
        is_match = isComplete(m, rows_so_far.top());
        if (is_match){
            wk = processK(n, num_features, chosen_els, TK, TKIndices, lk, dg);
            if (wk > *max_weight){
                *max_weight = wk;
            }
        }


        getTraversal(n, sum.top(), traversal);

        // Proceed to next element:
        done = findNextEl(m, n, pos, sum, traversal, rows_matched, rows_so_far, els_so_far, chosen_els);

    }


}

void column_sum_dynamic_bitset(int m, int n, vector<dynamic_bitset<>> &M, vector<int>& sums){

    /*  Sums the columns of the mxn bitset M.
```

```
    /*  Sums the columns of the M bitset matrix.
        Returns in sums array.
        NOTE 1:     Since values in M are bits not ints, we cannot add them directly.
                    Instead, add 1 when bit is 1.
        NOTE 2:     We have j as the inner loop to process elements consecutively in memory.

        USES:   <boost/dynamic_bitset.hpp>
                <vector>

    */

    int i,j;

    // Intialize all entries in sums to 0:
    sums.assign(n,0);


    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++) {
            if (M[i][j]){
                sums[j]++;
            }

        }
    }
}

bool findNextEl(int m, int n,
                stack<int>& pos,
                stack<vector<int>>& sum, //vecs length n
                stack<vector<int>>& traversal, //vecs length n
                stack<dynamic_bitset<>>& rows_matched, //bitsets length m
                stack<dynamic_bitset<>>& rows_so_far, //bitsets length m
                stack<int>& els_so_far,
                dynamic_bitset<>& chosen_els){


    /*  Finds the next element in K. Adds it to els_so_far and chosen_els (flip 0 to 1 here).
        Returns 1 if we're done (no new elements to find - ALL K have been found), 0 otherwise.
        *1 means all K have been found, not just our most recent one! This is when we end our search for Ks.
        NOTE: bitset indices go from R to L (BUT the row order in a bitset matrix/stack is correct)
              if we really want index i, that will be n-1-i in a bitset of length n (index 0,..,n-1)

        ALGORITHM:
        The element in traversal[curr_pos] is our next element.
        This is an acceptable element if:
            -we are not at the end of a traversal vector (if our index is n, we've finished the vector)
            -corresponding sum is >0 (if it's 0 or less, there are no new matches for this element)
            -if sum is <=0, since we traverse in decreasing order based on sums, every element after it
             in traversal will also be <=0
            -nothing more to try here, so we pop everything

        If we have an acceptable element, we add it and then set up for the next call to this function.
        This means increment our position by 1 (the next time we get back to choosing an element at this level,
        this will be the position in traversal to check), and push a new 0 to the position stack. This 0 is
        because we have chosen an element at our current level, and now need to choose a next element at a
        new level.
        We will create new traversal, rows_matched (etc) vectors to
        match with this new 0 position AFTER returning from this function.

        If we don't have an acceptable element, we have exhausted all K we can get with the elements we have chosen so far.
        Suppose we've chosen r elements so far (e_1,...,e_r). We then need to pop everything to go back a step to the previous r-1 e
        We then need to find the next rth element to choose.
        Since we have exhausted all possible K with elements e_1,...,e_r, we no longer want to have e_r available to choose later
        along with e_1,...,e_{r-1} ((ie. by "later" we mean choosing it as an (r+1)st or later element). Thus we "zero out"
        e_r from our last sum vector. By turning its sum to 0, any subtraction of rows will result in a value <=0 for e_r,
        so it won't be considered. Note that by "zeroing it out" at this stage, we are not preventing it from being chosen again
        if we go back a further step and choose a different (r-1)st element (we want it possible at this point).

        NOTE: n-1-i gives the index for what we see as i in bitset since it reads R to L

        USES:   <stack>
                <vector>
                <boost/dynamic_bitset.hp>

    */

    int curr_pos, new_el;

    // If all positions have been cleared, we are done.
    if (pos.empty()){
        return 1;
    }

    /*  Otherwise, we proceed.

        ACCEPTABLE ELEMENT:
        If our curr_pos is <n, then we can still index in to our latest traversal vector.
        If we also have a sum >0, this is an acceptable element.
        sum.top()[traversal.top()[curr_pos]] gives the sum corresponding to our chosen element.
        We can only define this once we're sure curr_pos < n!
        By using &&, it won't evaluate the second part until we've verified our curr_pos is valid.
```

```
        We add the new element and modify the position for the next stage.
        Return 0 since we are not finished checking for elements.

        UNACCEPTABLE ELEMENT:
        If our curr_pos=n, we finished a traversal vector, and thus need to pop everything.
        If curr_pos is not n, but we have a sum <=0, we also have gone as far as we can in a
        traversal vector, and again need to pop everything.
        We also need to "zero out" our sum and modify our chosen elements. Before we do this, we
        need to verify that we haven't popped back to empty sets (this indicates we are done
        entirely).
        We then need to try to find the next element.

    */

    curr_pos = pos.top();

    if (curr_pos < n && sum.top()[traversal.top()[curr_pos]] > 0){
        new_el = traversal.top()[curr_pos];
        els_so_far.push(new_el);
        chosen_els[n-1-new_el] = 1;
        pos.top()++;
        pos.push(0);
        return 0;

    } else {

        pos.pop();
        // If this is empty, we're done.
        // WARNING: We need to check this before we pop rows_matched, so_far, because they could be
        //          empty already. (Since we modify them AFTER we return from this function).
        if (pos.empty()){
            return 1;
        }
        sum.pop();
        traversal.pop();
        rows_matched.pop();
        rows_so_far.pop();


        // Zero out last element from previous sum (now at the top):
        sum.top()[els_so_far.top()] = 0;

        // Flip last chosen_els to 0 along with popping from els_so_far:
        // chosen_els is a bitset, so n-1-i is really our usual i index
        chosen_els[n-1-els_so_far.top()]=0;
        els_so_far.pop();

        return findNextEl(m, n, pos, sum, traversal, rows_matched, rows_so_far,els_so_far,chosen_els);

    }



}

void getNewRows(int m, int n,
                int new_el,
                vector<dynamic_bitset<>> M, // mxn; m vector entries, each an n-length bitset
                stack<dynamic_bitset<>>& rows_matched, // bitset size m
                stack<dynamic_bitset<>>& rows_so_far){ //bitset size m

    /*  Given new_el, determines the NEW rows matched, and adds (pushes) to the rows_matched list.
        New rows matched are the rows in M with a 1 in column new_el, that were not already matched previously.
        Also adds (pushes) a running total of rows matched so far to the rows_so_far list.

        ALGORITHM:
        Let col denote the column of M corresponding to new_el.
        If col[i]=1, row i has been matched (by new_el).
        It's possible this row was already matched. We only want a list of the new matches.
        If it was matched before, it would have a 1 in our most recent rows_so_far vector.
        So we want col[i]=1 and rows_so_far.top()[i]=0 for row i to be a new match.

        NOTE:   & is logical AND for bitsets
                | is logical OR for bitsets
                ~ is logical NOT for bitsets

        USES:   <stack>
                <vector>
                <boost/dynamic_bitset.hpp>
    */

    int i;
    dynamic_bitset<> col (m);
    dynamic_bitset<> so_far (m);

    // Get column from M:
    for (i=0; i<m; i++){
        col[i] = M[i][new_el];
    }

    /*  If rows_so_far is empty, we're at the initialization stage, so our new rows matched will just be
```

```
        col. Our running total (our next vector for rows_so_far) will also be col.
        NOTE:   If rows_so_far is empty, we cannot do rows_so_far.top() or we'll get a segmentation fault.
                So, we need to treat this case separately.
    */

    if (rows_so_far.empty()){
        rows_matched.push(col);
        rows_so_far.push(col);
        return;
    }

    /*  Otherwise, our new matches will be col AND ( NOT so_far).
        We only want new matches, so we need a match in col, and not a match in so_far.
        Our running total will be anything matched in either col or so_far (or both), so we
        can take col OR so_far.
    */

    so_far = rows_so_far.top();
    rows_matched.push(col & (~so_far));
    rows_so_far.push(col | so_far);
    return;

}

void getNextSum(int m, int n,
                stack<vector<int>>& sum, // length n vectors
                stack<dynamic_bitset<>> rows_matched,  //length m bitsets
                vector<dynamic_bitset<>> M){ // mxn; m vector entries, each an n-length bitset

    /*  Creates the next sum vector, adds (pushes) to sum list.
        This is the new sum after choosing a new element. The previous sum is revised by subtracting newly matched rows.
        The resulting sum is the sum for each column in M, ignoring elements in rows we've already matched an element with.
        ie. It's the sum of the rows we have yet to match an element to.
        Assumes rows_matched has already been updated with the new rows matched from the latest element.

        ALGORITHM:
        Take the last sum vector.
        Subtract each row we've newly matched with. The rows are bits, so we cannot subtract from ints directly.
        rows[i]=1 means it's a row we want to subtract. Then we subtract M[i][j] from our jth sum coordinate.

        USES:   <boost/dynamic_bitset.hpp>
                <vector>
                <stack>
    */

    int i,j;
    vector<int> new_sum = sum.top(); // initialize to the previous sum
    dynamic_bitset<> rows = rows_matched.top(); // list of newly matched rows (1 means newly matched, 0 not)

    for (i=0; i<m; i++){
        if (rows[i]){
            for (j=0; j<n; j++){
                if (M[i][j]){
                    new_sum[j]--;
                }
            }
        }
    }

    sum.push(new_sum);

}


bool isComplete(int m, dynamic_bitset<> so_far){


    /*  Checks if we have a complete K.
        Returns 1 if we do, 0 if not.

        Input:  most recent rows_so_far vector (bitset of length m)
                this keeps track of which rows we have matched (1/0 bitset format)

        ALGORITHM:
        We have a complete K if we have matched all rows.
        This is when so_far is all 1s.

        USES:   <boost/dynamic_bitset.hpp>

    */

    int i;

    for (i=0; i<m; i++){
        if (!so_far[i]){
            return 0;
        }
    }

    return 1;

}
```

```
}

float processK(int n, int num_features,
               dynamic_bitset<> chosen_els, // length n
               dynamic_bitset<> TK, // length num_features
               vector<int> TKIndices, // length n
               vector<float> lk, // length num_features
               vector<float> dg){ // length num_features

    /*  Once we have a complete K, we need to compute weight(K).
        Returns weight(K).

        Inputs:
            -chosen_els is our K; this is a list of length n
                -1 means the element is in the set, 0 means it is not.
                -chosen_els[i] means element n-1-i chosen
                -However, these indices indicate which elements from within TK are in K.
                -To compute weight(K), we need to convert K back to a bitset of length num_features.
            -TK is our original set from our POSET, of length n
                -K is a subset of TK
            -TKIndices indicate which elements from our full set of features were in TK
                -eg. if TK=[1101001011], TK_Indices=[013689]
            -lk, dg are the weights for each attribute needed to compute weight(K)

        To compute weight(K), we need to convert K back to a vector of length num_features. This is
        what we will store in the variable K.

        NOTE: bitset reads R to L so we output in reverse order.
        (i is really n-1-i)



        USES: <boost/dynamic_bitset.hpp>
              <iostream> (for now, because of cout)
    */

    int i;
    float wk;
    dynamic_bitset<> K (num_features);

    for (i=0; i<n; i++){
        if (chosen_els[i]){
            K[TKIndices[i]] = 1;
        }
    }

    // Now we can compute weight(K):
    wk = weight(K, TK, lk, dg);
    return wk;

}

void getTraversal(int n, vector<int> curr_sum,
                  stack<vector<int>>& traversal){

    /*  curr_sum and traversal (elements) are vectors of length n.

        Sorts curr_sum from largest to smallest.
        The corresponding indices to this sort tell us the order to traverse (choose elements) in.
        These are placed in a vector next_traversal, which is pushed onto our traversal list.

        ALGORITHM:
        Initialize next_traversal array to be the indices 0,1,...,n-1 (use iota).
        We get the new index order when we sort curr_sum.
        We use sort on next_traversal (our indices), but our comparison function is actually
        looking at our curr_sum values to decide the new order of indices.

        USES: <numeric> (for iota)
              <algorithm> (for sort)
              <stack>
              <vector>
    */

    int i;
    vector<int> next_traversal (n);

    // Initialize indices of next_traversal to 0,1,...,n-1:
    iota(next_traversal.begin(), next_traversal.end(), 0);

    // Sort and get new indices:
    // Reference for this part: https://stackoverflow.com/questions/1577475/c-sorting-and-keeping-track-of-indexes
    sort(next_traversal.begin(), next_traversal.end(), [&](int i,int j){return curr_sum[i]>curr_sum[j];});

    // Add to stack:
    traversal.push(next_traversal);

}

/*****************************WEIGHT CALCULATIONS*************************************************************/

float likelihood(dynamic_bitset<> K, vector<float> lk){
```

```
    /*  Computes the likelihood that we know K.
        Input:
            - K as a bitset of length NUM_FEATURES.
            - lk array of the likelihood values (between 0 and 1) for knowing each attribute

        WARNING: When we call this from processK, in all of the getAllK functions, we are using a shorter
                 length bitset to denote K. We need to convert back to one of length NUM_FEATURES first.

        ALGORITHM: Multiply all lk(x) for each x in K.
                   This means multiply lk(x) if K(x)=1.
                   Since K is stored as a bitset, indices are reversed, so x is really NUM_FEATURES-1-x.

        USES:   <boost/dynamic_bitset.hpp>
                <vector>

    */

    float lk_K = 1;
    int num_features = K.size();
    int i;

    for (i=0; i<num_features; i++){
        if (K[i]){
            lk_K *= lk[num_features-1-i];
        }
    }

    return lk_K;
}


float danger(dynamic_bitset<> IK, vector<float> dg){

    /*  Computes the danger from inferring IK.
        Input:
            - IK as a bitset of length NUM_FEATURES.
            - IK should be the NEW attributes we can infer from knowing K.
            - dg array of the likelihood values (between 0 and 1) for knowing each attribute

        WARNING: When we call this from processK, in all of the getAllK functions, we are using a shorter
                 length bitset to denote K. We need to convert back to one of length NUM_FEATURES first.

        ALGORITHM:  dg(IK) = {sum of dg(x) for each x in IK}/{1 + this sum}
                    We could add another argument if we wanted to choose a number other than 1 to add
                    in the denominator.

        USES:   <boost/dynamic_bitset.hpp>
                <vector>

    */

    float dg_sum = 0;
    int num_features = IK.size();
    int i;


    for (i=0; i<num_features; i++){
        if (IK[i]){
            dg_sum += dg[num_features-1-i];
        }
    }

    return (dg_sum / (1+dg_sum));

}


float weight(dynamic_bitset<> K, dynamic_bitset<> TK,
             vector<float> lk, vector<float> dg){

    /*  Computes weight(K)=likelihood(K) * danger(IK).
        IK is the set of NEW inferences from K, whereas TK is the full set, including K.
        IK = TK \ K.
        TK is the set we started with during our getAllK call.
        ie. K => TK, but IK are the new inferences outside of K.

        ALGORITHM:  We want IK to be elements in TK but not in K.
                    Thus IK = TK & (~K) (~ for not in bitsets)

        USES:   <boost/dynamic_bitset.hpp>
                <vector>

    */

    dynamic_bitset<> IK = TK &(~K);
    return (likelihood(K, lk) * danger(IK, dg));

}


/*******************Convert Between K, TK, IK of Length Num_Features vs. Length =to that of TK*************************/
```

```
void getTKIndices(dynamic_bitset<> TK, int num_features, vector<int> &TKIndices){

    /*  Takes the TK bitset of length num_features as input.
        Fills TKIndices with the indices where TK is 1.
        This indicates the elements of the set represented by TK.
        NOTE: These indices correspond to reading TK from R to L!
        They go in reverse order to correspond to the order we see things in.
    */

    int i;

    for (i=num_features-1; i>=0; i--){
        if (TK[i]){
            TKIndices.push_back(num_features-1-i);
        }
    }


}
```

Bibliography

[cpl] std::sort.

[Dow52] C.H. Dowker. Homology groups of relations. *Annals of Mathematics*, 56:84–95, 1952.

[EH10] Herbert Edelsbrunner and John Harer. *Computational Topology - an Introduction.* American Mathematical Society, 2010.

[Erd17] Michael Erdmann. Topology of Privacy: Lattice Structures and Information Bubbles for Inference and Obfuscation. *arXiv e-prints*, page arXiv:1712.04130, December 2017, 1712.04130.

[Hat02] Allen Hatcher. *Algebraic topology.* Cambridge University Press, Cambridge, 2002.

[KG06] Daniel Kifer and Johannes Gehrke. l-diversity: Privacy beyond k-anonymity. In *In ICDE*, page 24, 2006.

[KK12] P. Kiran and P. KavyaN. A survey on methods, attacks and metric for privacy preserving data publishing. *International Journal of Computer Applications*, 53:20–28, 2012.

[LLV07] Ninghui Li, Tiancheng Li, and Suresh Venkatasubramanian. t-Closeness: Privacy Beyond k-Anonymity and l-Diversity. In *23rd International Conference on Data Engineering (ICDE 2007)*, pages 106–115. IEEE, 2007.

[OZ03] Stanley Oliveira and Osmar Zaane. Privacy preserving frequent itemset mining. *Proceedings of the IEEE ICDM Workshop on Privacy, Security and Data Mining*, 06 2003.

[Sam01] Pierangela Samarati. Protecting respondents' identities in microdata release. *IEEE Trans. Knowl. Data Eng.*, 13(6):1010–1027, 2001.

[Swe02] L. Sweeney. k-anonymity: A model for protecting privacy. *IEEE Security and Privacy*, 10:1–14, 01 2002.

[Wac06] Michelle L. Wachs. Poset Topology: Tools and Applications. *arXiv e-prints*, page arXiv:0602226, February 2006, 0602226.