Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Automatizálási és Alkalmazott Informatikai Tanszék

Garai Richárd

DATA ANONYMIZATION IN GO

a practical implementation of an approximation algorithm

SUPERVISOR

Dr. Dudás Ákos

Budapest, 2019

Contents

Chapter 1

Introduction

1.1 Privacy and the collection of personal data

A few years ago in Minneapolis an angry man walked into a retail store called *Target* and demanded to see the manager. *"My daughter got this in the mail!"* he exclaimed, showing some coupons in his hand. *"She is still in high school, and you're sending her coupons for baby clothes and cribs? Are you trying to encourage her to get pregnant?"* [12, 1] The coupons were indeed addressed to the man's daughter and contained advertisements for maternity clothing, nursery furniture and pictures of smiling infants. The manager apologized, and called a few days later to apologize again.

On the phone, though, the father was somewhat abashed. *"I had a talk with my daughter".* he said. *"It turns out there has been some activities in my house I haven't been completely aware of. She's due in August. I owe you an apology."* [1] But how did the retail store know that the girl is pregnant before her own father? As a matter of fact *Target* was observing the purchasing habits of its customers through their membership cards. In this case, a change like purchasing scent-free lotions and soap, extra-big bags of cotton balls, hand sanitizers, washcloths and supplements containing calcium magnesium and zinc indicated a likely pregnancy to the retail store's cleverly constructed advertising algorithm. In fact, the algorithm was so advanced, that it could even predict the due date with a relatively small time window so that the store can send coupons timed to very specific stages of pregnancy.

In our modern age data is power. Data is now a commodity — arguably the world's most valuable commodity, dethroning oil from its former number one position [20]. Big software companies have recognized this a long time ago and offer convenient, easy to use software and services *"for free."* In reality however, nothing is free. Users of these services are paying with their personal data, ranging from simple personal identifiers to more complex usage statistics and metrics. This data is most often cross-referenced with other similarly collected databases to draw conclusions and offer a more personalized user experience — including advertisements.

As a result, we are now subject to a greater level of surveillance than in any point in history, and we hand over most of our data willingly [18]. It can be expected, that the increasing amount of *internet-of-things* devices and the introduction of *5G networking* will result in an even bigger surge in the amount and types of data collected.

1.2 Regulation

Even the Big Tech companies (who not surprisingly are also the biggest collectors and consumers of personal data) are beginning to acknowledge that personal data collection needs to be regulated [19]. Recent privacy scandals of Google (Google+) [11], Facebook, Cambridge Analytica [2], Amazon [3] and others [21] seem to support the importance of the need for a stricter regulation as well.

More recently governments are slowly catching up on the regulation front. One prime example is the *"General Data Protection Regulation" (GDPR)* of the European Union. The *GDPR* got some critique [5, 17], but it has a solid foundation. It regulates how the collection of personal data is disclosed to the data subjects, and demands that data stored on people in the EU undergo either an ***anonymization*** or a ***pseudonymization*** process [6, 4]. The main difference between the two processes is, that pseudonymized data can be restored to its original state by re-adding identifiers, while anonymized data can not be restored. In this document we will mostly be discussing anonymization, however it is not hard to imagine how one would construct a pseudonymization algorithm given a working anonymization algorithm.

1.3 Data anonymization

In the previous chapters we briefly introduced the sociological and political aspects of privacy. Now we will proceed to discuss the concept of ***anonymization*** and its technical aspects. So how exactly can we define anonymization?

Definition DATA ANONYMIZATION has been defined as a process by which personal data is irreversibly altered in such a way that a data subject can no longer be identified directly or indirectly, either by the data controller alone or in collaboration with any other party [4, 13].

A typical *data controller* can be a hospital storing medical data about its patients. The data is then released to third parties or government entities, for example to draw conclusions in an epidemic outbreak. We can see, that some parts of the data like address, age, race, gender — although personal — can be necessary to draw accurate statistical conclusions. To allow the extraction of useful statistics while also protecting the privacy of its patients, the hospital will need to alter the data before handing it over.

1.4 k-anonymity

1.4.1 Insufficiency of identifier truncation

Before diving into the definition of k-anonymity, let's take the previous example a little bit further, and consider the following patient data taken from the hospital's database:

SSN	**Name**	**Age**	**Race**	**Gender**	**Zip Code**	**Disease**
1001	John Smith	34	White	Male	18500	Flu
1002	Eva Brown	27	Asian	Female	18200	Cancer
1003	Anne Parker	17	White	Female	18500	Hypertension
1004	Mark Wayne	34	Black	Male	18200	Flu

Figure 1.1: Example patient data

In order to attempt to remove personal information from the data, one — albeit naïve — approach the hospital could choose is to simply *truncate* all fields that contain identifiers. In the above example they will be the *social security number* and *name* fields. The resulting table will look something like this:

Age	**Race**	**Gender**	**Zip Code**	**Disease**
34	White	Male	18500	Flu
27	Asian	Female	18200	Cancer
17	White	Female	18500	Hypertension
34	Black	Male	18200	Flu

Figure 1.2: Truncated patient data

Unfortunately by using an external public database — for example census records or a phone book — an attacker could still piece together which person was diagnosed with which disease, thus rendering the hospital's data protection method useless. This is called a ***table join attack***:

Age	**Name**	**Zip Code**
34	John Smith	18500
27	Eva Brown	18200
17	Anne Parker	18500
34	Mark Wayne	18200

Figure 1.3: Census data

Name	Age	Race	Gender	Zip Code	Disease
John Smith	34	White	Male	18500	Flu
Eva Brown	27	Asian	Female	18200	Cancer
Anne Parker	17	White	Female	18500	Hypertension
Mark Wayne	34	Black	Male	18200	Flu

Figure 1.4: Patient data restored with join attack

One might think, that this does not usually happen outside of fabricated examples. Unfortunately, as Sweeney observed [27] for 87% of the U.S. population the combination of date of birth, gender and zip code corresponded to a unique person [23] — which could easily make the above example reality.

1.4.2 Data model

Most anonymization algorithms work on data tables with a concrete *schema* so we will adopt this model as well. Tables can also be viewed as n row vectors, or *tuples*. Each tuple represents an individual piece of data, and consists of m attributes $(c_1, c_2, \ldots, c_m)$. Attributes are sometimes also referred to as *dimensions*.

Attribute types

identifiers: these attributes by themselves alone can be used to identify the person whom the data belongs to without the need to reference any other attributes. It is easy to see, that anonymization algorithms will always need to truncate or suppress all of these attributes from the final anonymized data set.

quasi-identifiers: as illustrated by the examples in Section 1.4.1 some attributes are not identifiers themselves but can still be used in conjunction with external data sources to identify the concrete person. These attributes are *quasi-identifier* attributes, and will be subject to some level of generalization or suppression based on the algorithm being used (refer to the next section called *‘Data hiding techniques‘* for a more precise explanation of generalization and suppression).

non-identifiers: attributes, which the *data controller* deems irrelevant in the context of anonymization are non-identifiers. We can often extract useful statistics from their values, but they are meaningless without pairing them with additional attributes.

Data hiding techniques

In order to remove sensitive data from a database, anonymization algorithms use common data hiding techniques. In this section we will briefly introduce two commonly used, and illustrate how they work on simple examples.

suppression is very similar to truncating all sensitive data values. When looking at a single attribute (like age) in a database, truncating it would redact *all* values. Suppression on the other hand redacts *some* of the values, while retaining the rest [23]. Which values are suppressed and which values are retained is decided by the anonymization algorithm, which considers several different factors.

Figure 1.5 shows the anonymization of a table using suppression only, along the quasi-identifiers Name, Age and Gender:

Name	Age	Gender	Salary
John	25	Male	50000
Mark	30	Male	65000
Jill	24	Female	45000
Jane	24	Female	70000

Name	Age	Gender	Salary
*	*	Male	50000
*	*	Male	65000
*	24	Female	45000
*	24	Female	70000

Figure 1.5: Suppression

generalization replaces values with less specific, but semantically consistent values from the same domain. For example a date can be made more generic by omitting the days and even more so by reducing it to the year only. (Note, that suppression is basically a special case of generalization — we can create a special generalization function, which maps every value to a suppressed value in one step.)

Figure 1.6 shows the anonymization of a table using generalization along the quasi-identifiers Name, Age and Gender:

Name	**Age**	**Gender**	**Salary**
John	25	Male	50000
Mark	30	Male	65000
Jill	24	Female	45000
Jane	24	Female	70000

Name	**Age**	**Gender**	**Salary**
*	[20...30]	Male	50000
*	[20...30]	Male	65000
*	24	Female	45000
*	24	Female	70000

Figure 1.6: Generalization

In most cases we can assume, that for each attribute a ***generalization hierarchy*** exists [23]. Each level of the generalization hierarchy corresponds to a partition of the attribute domain, and a refinement over the partitions on the previous level [26]. On the two opposing ends of the hierarchy we have singleton sets corresponding to the lack of generalization, and a single set containing all values from the domain corresponding to the highest level of generalization.

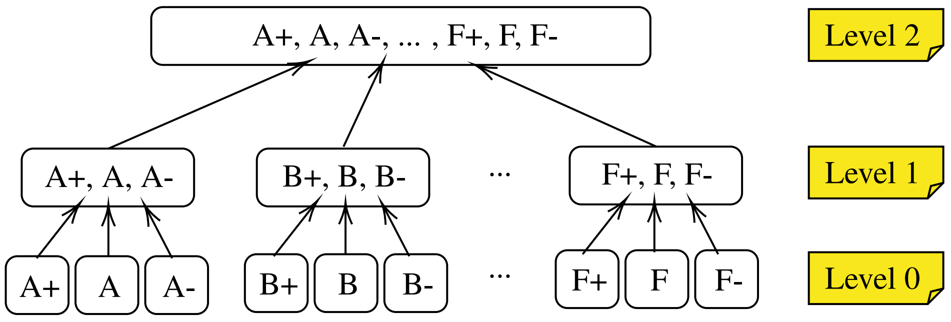


Figure 1.7: Generalization hierarchy for school grades

The generalization hierarchy is a useful model, but it is not without restrictions — especially when considering very large or infinite value domains (for example integer numbers). In later chapters we will see how we can implement more flexible generalization hierarchy for our anonymization algorithm.

1.4.3 Definition of k-anonymity

The *k-anonymity* model was originally proposed by Samarati and Sweeney [26, 28]. The idea is to generalize some of the data in the input table to ensure, that *for each tuple in the anonymized table there are at least $k-1$ other tuples in the table that are identical to it along the quasi-identifier attributes*. While achieving this, the anonymization algorithm should also minimize the *cost of generalization*.

A k-anonymized data table will be ‘immune’ to join attacks or *record linkages* even if the attacker has access to all quasi-identifying attributes of all the individuals represented in the table [23]. This is because each individual tuple is hidden among $k-1$ other identical tuples. Note however that it is the responsibility of the data controller the correctly identify all quasi-identifiers in the table and provide a sufficient generalization hierarchy for them.

Selecting an appropriate *k* value is also the data controller’s responsibility. While picking a large *k* value will ensure a bigger level of privacy, it also reduces the amount of information left in the data set. The *k* value should always be selected by considering the needs of the application.

Name	**Status**	**Gender**	**Age**	**Kids**	**Income**	**Grade**
*	employee	*	[18..36]	[0..2]	[10000..50000]	[A]
*	client	female	[18..36]	[0..2]	[15000..15624]	[A-]
*	employee	*	[30]	[2]	[30000..39999]	[A, A+, A-]
*	employee	*	[18..36]	[0..2]	[10000..50000]	[A]
*	client	male	[18..36]	[2]	[40000..44999]	[A, A+, A-]
*	client	female	[18..36]	[0..2]	[15000..15624]	[A-]
*	employee	male	[18..36]	[2]	[40000..44999]	[A, A+, A-]
*	client	*	[30]	[2]	[30000..39999]	[A, A+, A-]

Figure 1.8: An anonymized table for k=2 with various data types

Now we can formally define k-anonymity [23]:

Definition K-ANONYMITY WITH SUPPRESSION:
given $x_1, \ldots, x_n \in \Sigma^m$ and the anonymity parameter *k*, obtain a k-anonymous suppression function *t* so that *c (t)* is minimized.

A **k-anonymous suppression function** *t* maps each x_i to $\bar{x_i}$ by replacing some components of x_i by $*$, so that every $\bar{x_i}$ is identical to at least $k-1$ other $\bar{x_j}$s.

The **cost of *t* suppression function** *c (t)* is the total number of hidden entries ($*s$) in all the $\bar{x}_i$s.

Definition K-ANONYMITY WITH GENERALIZATION:
given $x_1, \ldots, x_n \in \Sigma^m$ and the anonymity parameter k, obtain a k-anonymous generalization function h so that *c (h)* is minimized.

Let the j^{th} attribute have domain D^j and l_j levels of generalization. Let the partition corresponding to the h^{th} level be denoted by $g_{h(y)}$. A **generalization function** h is a function that maps a pair $(i, j), i \leq n, j \leq m$ to a level of generalization $h(i, j) \leq l_j$.

Let $h(x_i)$ denote the *generalized* vector corresponding to x_i, i.e.

$$h(x_i) = (g_{h(i,1)}(x_1[1]), \ldots, g_{h(i,m)}(x_i[m])).$$

h is a **k-anonymous generalization function** if for every $i, h(x_i)$ is identical to $h(x_j)$ for at least $k-1$ values of $j \neq i$.

Consider a k-anonymous generalization function h. It incurs a **cost** of r/l_j whenever it generalizes a value for the j^{th} attribute to the r^{th} level. The **total cost incurred by the generalization function** h is defined as the sum of the costs incurred over all the entries of the table, i.e. $c(h) = \Sigma_i \Sigma_j h(i, j)/l_j$.

While this definition seems complicated at first, it basically just states, that we are looking for the *'cheapest'* generalization (or suppression) function which partitions the rows in such a way, that at least k identical rows are present in each partition after applying the function — this is in-line with the rule of k-anonymity stated earlier.

The complication comes from calculating the cost incurred by the function. For a suppression function we simply need to count the number of suppressed entries. For a generalization function however we need to consider the maximum level of generalization possible for the attribute's domain, and the level of generalization applied by the function in order to find the generalization cost for a single entry. This cost for a single entry will always be a rational number between $[0..1]$. The total cost of applying the generalization function is the sum of all of these costs.

It is also worth noting, that the problem of **k-anonymity with suppression** is a special case of the problem of **k-anonymity with generalization**.[23] This can be proven by using a special generalization function which only has one level of generalization for each attribute domain which corresponds to completely hiding the element value. It can easily be seen, that this special generalization function is equivalent to a suppression function for the same k anonymity parameter.

1.4.4 NP-hardness of k-anonymity

Theorem k-anonymity with suppression is NP-hard even for a ternary alphabet ($\Sigma = 0,1,2$).[23] . The complete proof will not be presented in this document, but the core idea is, that the NP-hard problem *Edge Partition Into Triangles* (Kann, 1994) can be reduced into the problem of k-anonymity with suppression for $k = 3$.[23] Furthermore, by reduction from *Edge Partition Into r-Cliques* (Kann, 1994) the proof can be extended to any $k \geq 2$ integer value.

Let's pause for a moment and consider the consequences of the above theorem. A decision problem H is NP-hard, when for every problem L in NP, there is a polynomial-time reduction from L to H.[25] In other words problems in this class are *"at least as hard as the hardest problems in NP"*.[15] As a consequence if $P \neq NP$, then NP-hard problems cannot be solved in polynomial time.[14, 15] . In the next chapter we will outline a polynomial **approximation** algorithm for the k-anonymity problem. This means, that we won't always get the *optimal* solution — i.e. the cost of anonymization is not always *minimal* — but we get one, that's "good enough".

Chapter 2

The Algorithm

This chapter will introduce an approximation algorithm for the **k-anonymity with generalization** problem. The algorithm was published in 2005, in *Journal of Privacy Technology*. [23] The original publication is credited to the following authors: Gagan Aggarwal, Tomas Feder, Krishnaram Kenthapadi, Rajeev Motwani, Rina Panigrahy, Dilys Thomas and An Zhu.

2.1 Overview

This algorithm is a graph based algorithm. It works for any $k \geq 2$ parameter and an arbitrary alphabet size and gives an $\mathscr{O}(k)$-approximation solution.

The input of the algorithm is given in a form of *row vectors*: $x_1, x_2, \ldots, x_n \in \Sigma^m$. Note, that the row vectors can have the same amount of components, and the corresponding elements can have the same data types, therefore representing an $n \times m$ data table with a fixed *schema*. While this is a very common use-case (and also the route we are taking with the **go implementation**) it is not necessarily required — the algorithm could function on schema-less data.

As mentioned above, the algorithm also takes a k integer value — the anonymity parameter. In addition, the input also needs to contain any generalization hierarchies required to properly generalize all dimensions as specified in Section 1.4.2. In this section we will simply assume that all input is fully given and define the technicalities and possible input formats more precisely in the next chapter, when we describe the **go implementation**.

2.2 Algorithm outline

The outline of the algorithm can be summarized as follows:

1. construct a cost-graph from the table
2. build a forest which partitions the graph into $c \geq k$ components
3. gradually decompose "oversized" components until the approximation criteria is reached

Once the last step is finished the algorithm terminates, and the output will be a *forest* where each component represents a set of rows which should be generalized into the same partition.

2.2.1 A note on the algorithm output

The careful reader may notice, that the output of the above algorithm is not an anonymized data table, but a graph. If we recall the definition of *k-anonymity with generalization* from Section 1.4.3 it actually stated, that we are looking for a *k-anonymous generalization function*.

Producing the generalization function from the output graph is fortunately rather straightforward. The components of the output graph tell us exactly which nodes (rows) will belong to the same partition after generalization, while the cost graph will assist us in knowing the exact level of generalization needed. Since the generalization hierarchies for each dimension were given as part of the input, the generalization function can simply be composed of applying the g generalization function to the given level for each row vector.

Given the k-anonymous generalization function h produced with the method above, it is a trivial matter to obtain the generalized data table: the function should be applied to each of the input data vectors: $[h(x_1), h(x_2), \ldots, h(x_n)]$.

2.3 Algorithm details

2.3.1 Cost-graph construction

The very first step of the algorithm is the construction of a cost-graph. Given the input row vectors $x_1, x_2, \ldots, x_n \in \Sigma^m$ we create an edge-weighted complete graph $G = (V, E)$ [23]. The vertex set V is constructed from the row vectors by mapping each row vector to a separate vertex.

Next we need to calculate the edge weight between every two vertices. The assigned weight needs to be proportional to the generalization cost of bringing the two connected nodes into the same partition.

Unscaled Generalization cost

Given two row vectors a and b from the input we will now define the unscaled generalization cost for each corresponding components of the vectors. Let $h_{a,b}(j)$ refer to the lowest level of generalization for attribute j for which the j^{th} components of both a and b are in the same partition [23]. Or in other words, the lowest level of generalization for the corresponding components, for which their generalized values are equal.

Scaled Generalization cost

If we scale down the above generalization cost for an attribute pair with the *total number of levels* in the generalization hierarchy for the given attribute domain we get the scaled generalization cost. The resulting value will be in the range $[0..1]$. Assuming the total number of levels of generalization for the j^{th} attribute is l_j, the scaled generalization cost can be formalized as $h_{a,b}(j)/l_j$ [23].

Edge weight formula

Using the scaled generalization cost for each component, the weight of an edge between the nodes representing *a* and *b* row vectors can be calculated with the following formula:

$$w(e) = \sum_j h_{a,b}(j)/l_j.$$

Example Consider the example data on Figure 2.1:

Score	Grade	Gender
4	C-	male
7	B+	male
9	A+	female
8	A	male

Figure 2.1: Example data

Let's assume the domain for the *score* attribute is integer numbers with bounds $[0 \ldots 9]$, and that its generalization hierarchy can be represented with the following tree:

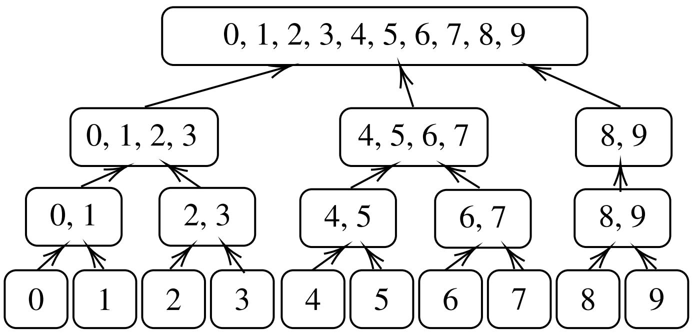


Figure 2.2: Hierarchy for the score attribute

This is a valid generalization hierarchy, because each partition at the leaf nodes contain only a single element, the top level contains a single set with all elements and finally the union of all sets on every level of the hierarchy is equal to the full domain of values.

Let's calculate the generalization cost for the first two rows.

In order to transform the values of the first attribute pair $(4,7)$ into the same partition, they need to be generalized exactly two times until their values become [4, 5, 6, 7]. This means, that the unscaled generalization cost for the score attribute $h_{a,b}(0) = 2$. The scaling factor corresponds to the highest level in the hierarchy: $l_0 = 3$, so the scaled generalization cost will be $h_{a,b}(0)/l_0 = 2/3$.

The edge cost between the nodes representing the first two rows will be the sum of the scaled generalization costs of $(4,7)$, $(C-,B+)$ and $(male, male)$. Using the grade hierarchy introduced in Chapter 1.4.2 in figure 1.7 and considering that the values of the gender attribute are already in the same partition, so their generalization cost will be 0:

$$w(e_{a,b}) = 2/3 + 1 + 0 = 5/3.$$

If we repeat the above calculation for each row pair, we can determine the cost-graph for this table. This process is left to the reader, but after finishing it the resulting graph should be isomorphic to the graph on figure 2.3.

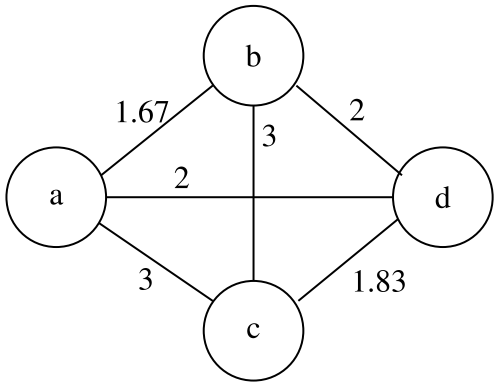


Figure 2.3: Example cost-graph

2.3.2 A note on the limitations of the graph representation

Unfortunately the graph representation has some limitations, as the original authors also note in their article [23]. When the input data table is transformed into its graph equivalent, some information is lost which results in the algorithm only being able to achieve the $\mathcal{O}(k)$ approximation factor.

This can be demonstrated by constructing two input tables on a binary alphabet ($\Sigma = 0, 1$) for a given k value in a way, that the optimal anonymization cost and the total cost incurred by the graph representation differs by a factor of $\mathcal{O}(k)$:

Let $l = 2^{k-2}$.

Table A should be $k \times kl$, and for each row i contains the value **1** in positions $[(i-1)l + 1 \dots il]$ and **0** otherwise. The anonymization cost for this table is $k^2 l$.

$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

Figure 2.4: Example for type A table and k=3, l=2

Table B should be $k \times 4l$. Its i^{th} row is broken down into 2^i equal-sized blocks. Every value in odd blocks is **0**, and every value in even blocks is **1**. The anonymization cost for this table is $4kl$.

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}$$

Figure 2.5: Example for type B table and k=3, l=2

Note, that both tables are represented by a k-clique with all edge costs being $2l = 2^{k-1}$, while their respective anonymization cost differs by exactly $\mathcal{O}(k)$.

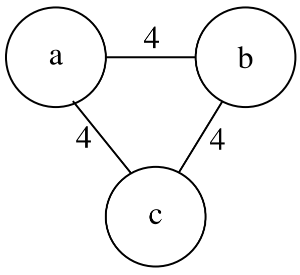


Figure 2.6: Graph representation of both Table A and Table B

2.3.3 Forest building algorithm

The next step in the overall k-anonymity algorithm is to produce a forest. (Section 2.2) We will now use the nodes and calculated weights from the cost-graph, but in this step we will be constructing a directed graph which will ultimately be used to represent a partitioning of the original row vectors.

Algorithm FOREST BUILDING [23]

Invariant:

- The chosen edges do not create any cycle.
- The out-degree of each vertex is at most one.

Steps:

1. Start with an empty edge set so that each vertex is in its own connected component.
2. Repeat until all components are of size at least k:

   Pick any component T having size smaller than k. Let u be a vertex in T without any outgoing edges. Since there are at most $k-2$ other vertices in T, one of the $k-1$ *nearest neighbors* of u, say v, must lie outside T. We add the edge $\vec{uv}$ to the forest. *(Observe that this step does not violate any of the invariants.)*

In simple terms we start from the original graph without any edges. In every step we extend a selected tree ($size < k$) in the forest. We add a new vertex to the tree by selecting an u "leaf" (without any outgoing edges) in the current tree, and connecting it with its lowest cost neighbor v.

Lemma The forest produced by the Forest building algorithm has a minimum tree size at least k and has cost at most *OPT*, where *OPT* denotes the cost of an optimal k-anonymity solution [23].

Example Forest building ($k = 3$)

For this example let's assume that the cost-graph has already been computed from the input table, and is isomorph to the graph shown on figure 2.7.

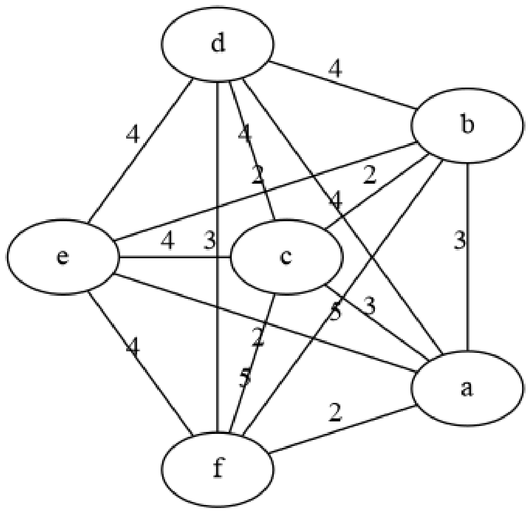
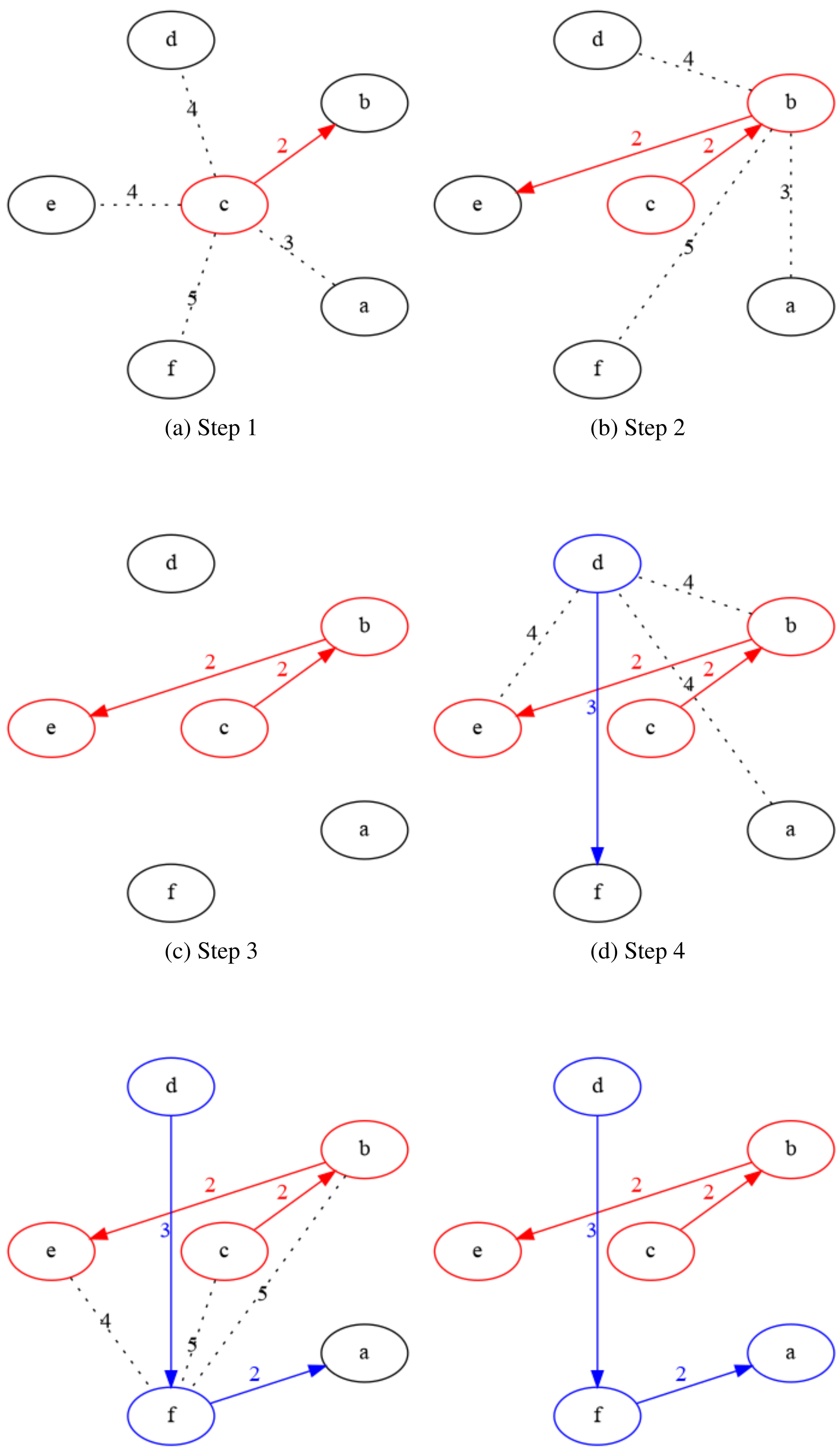


Figure 2.7: Example cost graph

The starting position has all nodes in their separate component. From there, the steps of the algorithm can be traced on figure 2.8.

The forest building algorithm starts from an empty edge set. Each node is in a separate component. Since at this point the size of each component is smaller than k, we pick a component to extend at random.

In Step 1 of the example we picked the component which contains the c node. Inspecting the outgoing edges we find that the closest one is the edge going to b. In Step 2 we continue extending the same component, since its size is still under the threshold. Node b is now part of the component, and has no outgoing edges, so we try to find an outgoing


(a) Step 1 (b) Step 2

(c) Step 3 (d) Step 4

(e) Step 5 (f) Step 6

Figure 2.8: Forest Building Algorithm

edge from that. Closest is the component containing *e*, so we add it.

At this point, the component's size has reached $k = 3$, so we move on to another component to extend. In Step 4 we have picked *d*. Note, at this point when considering the possible edges we also considered edges between *d* and nodes in the previously created component (*b* and *e*). This is an example of how this part of the algorithm could end up with component sizes greater than *k*. In this example however, the closest neighbor is *f*, and by Step 6 we can cleanly finish the partitioning with two 3-node partitions.

2.3.4 Algorithm to decompose oversized components

In the example in Section 2.3.3 we fortunately ended up with two perfectly sized components, which defined a good partitioning for anonymization. This will not always be the case however, and there may be oversized components in the forest.

In this section we show an algorithm to break any component with size $s \geq \max\{2k-1, 3k-5\}$ into two components each of size at least *k*. (The following algorithm treats the component as an *undirected* graph.)

Algorithm DECOMPOSE COMPONENT [23]

1. Pick any vertex *u* as the candidate vertex.

2. Root the tree at the candidate vertex *u*. Let U be the set of sub-trees rooted at the children of *u*. Let the size of the largest subtree of *u* be ϕ, rooted at vertex *v*. (See figure 2.9) If $s - \phi \geq k - 1$, then we do one of the following partition and terminate:

   A. If $\phi \geq k$ and $s - \phi \geq k$, then partition the tree into the largest subtree and the rest.

   B. If $s - \phi = k - 1$, partition the tree into a component containing the subtrees rooted at the children of *v* and the rest. To connect the children of *v* create a *Steiner's Vertex* (see below).

C. If $\phi = k-1$, then partition into a component containing the subtree rooted at v along with the vertex u and the rest. In order to connect the children of u create a *Steiner's Vertex*.

D. Otherwise, all subtrees have size at most $k-2$. In this case, we create an empty partition and keep adding subtrees of u to it until the first time its size becomes at least $k-1$. Put the remaining subtrees into the other partition. If one of the partitions has size equal to $k-1$, add u to that partition. Otherwise add u to the first partition. In the partition not containing u add a *Steiner's Vertex* to keep it connected.

3. Otherwise, pick v as the new candidate vertex and go to Step 2.

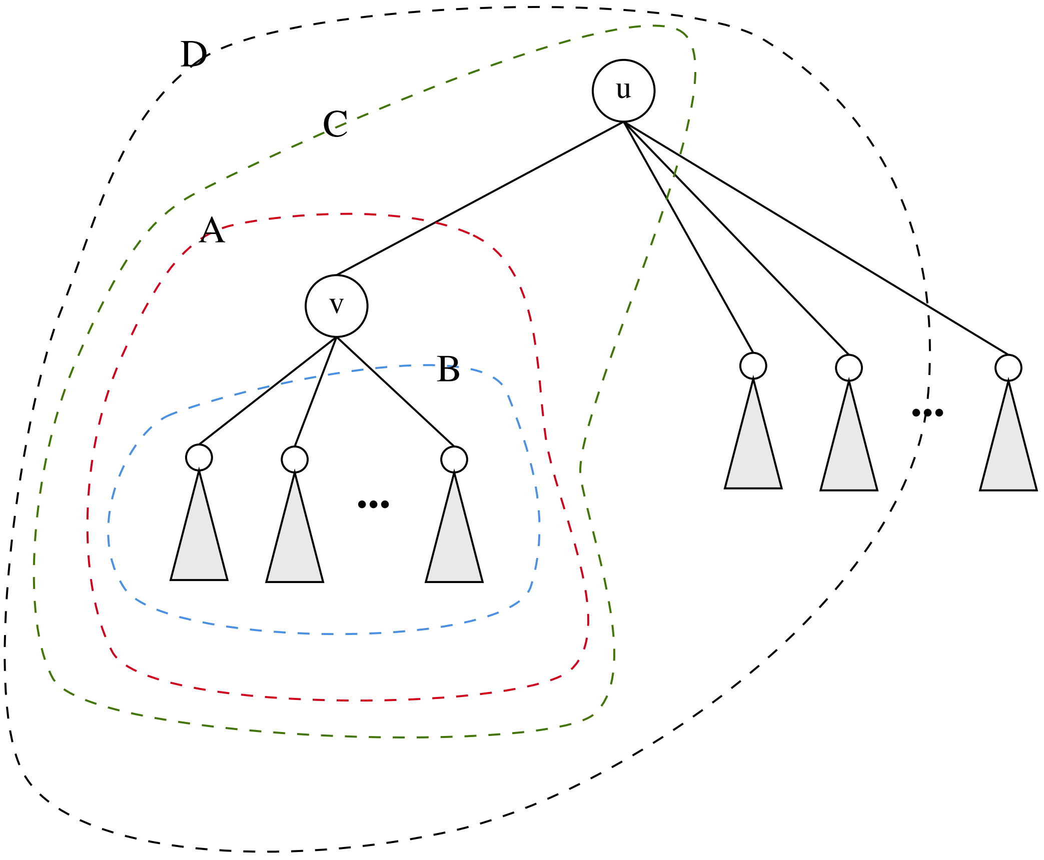


Figure 2.9: Different cut types for the decompose algorithm

The Steiner's Vertex is a dummy vertex inserted to structurally hold the component together, but does not contribute to the size of the component during the algorithm.

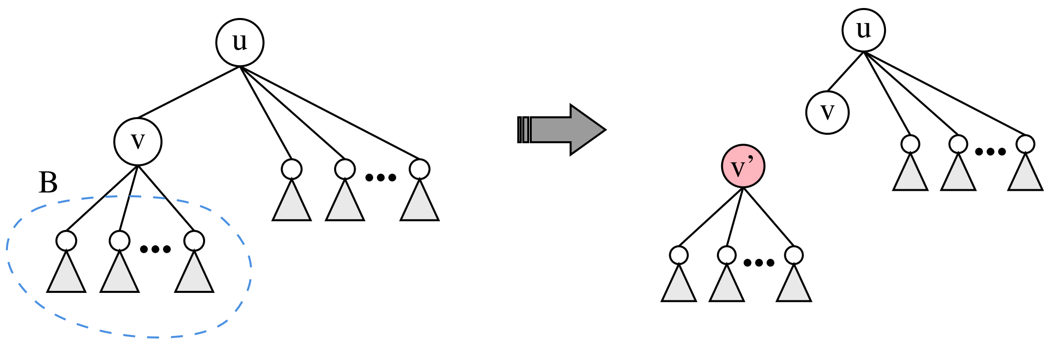


Figure 2.10: Steiner's Vertex for cut type B

Example decompose

Let's assume, that the forest building algorithm yielded the graph on figure 2.11, and the respective u starting vertex has already been selected.

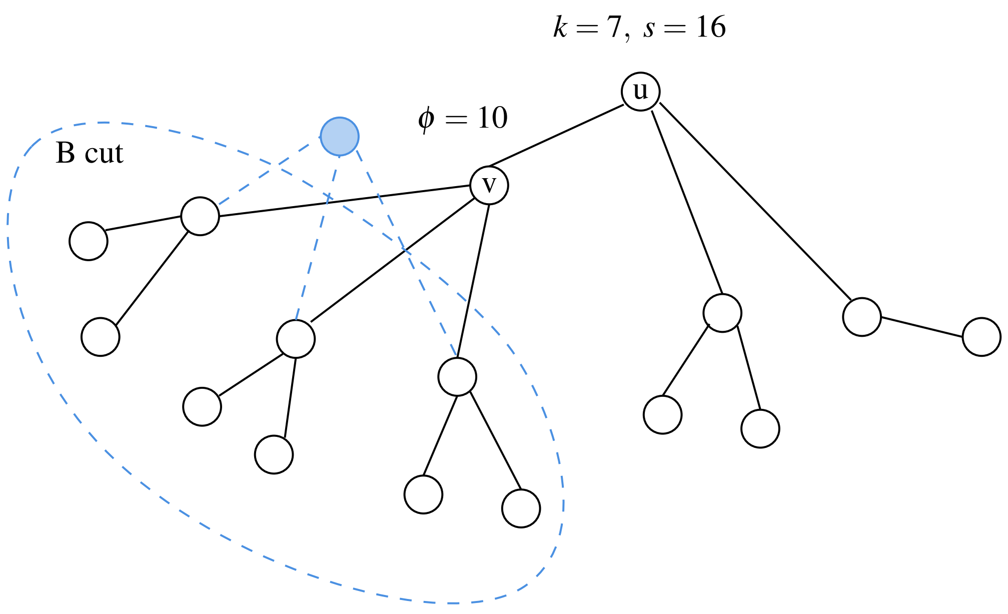


Figure 2.11: Example decompose step

The component size is $s = 16$, and the anonymization parameter is $k = 7$. $s \geq \max\{2k -$

$1, 3k-5\} = \max\{13, 16\}$ is true, so the component is considered oversized and needs to be cut. The root of the largest sized subtree is v, and its size is $\phi = 10$.

Since $s - \phi \geq k - 1$, we can proceed with one of the cut types. In fact, in this case $s - \phi = k - 1 = 6$, so we will need to apply cut type B as highlighted by the dotted line in figure 2.11.

In order to keep the nodes in the cut partition together, we insert a *Steiner's Vertex*. This vertex is also highlighted on figure 2.11 with dotted edges.

The resulting two partitions are of size 7 and 9 — note, that the *Steiner's Vertex* does not count into the component's size. Both partitions are of proper size now.

2.3.5 Polynomial-time approximation algorithm

Theorem There is a polynomial-time algorithm for the k-Anonymity problem, that achieves an approximation ratio of $\max\{2k-1, 3k-5\}$ [23].

Proof First, use algorithm FOREST to create a forest with cost at most OPT and minimum tree size at least k. Then repeatedly apply algorithm DECOMPOSE COMPONENT to any component that has size larger than $\max\{2k-1, 3k-5\}$. Note that both these algorithms terminate in $\mathcal{O}(kn^2)$ time [23].

2.4 Summary

As we have seen in this chapter, it is possible to give a polynomial-time approximation algorithm for the k-anonymity with generalization problem, and the algorithm gives an approximation ratio of $\max\{2k-1, 3k-5\}$.

We have started by defining the input table as a set of row vectors, and transforming the input data into its graph representation. We have discussed the limitations of the graph representation, and why we can only achieve an $\mathcal{O}(k)$ approximation.

Next we have introduced the three main steps in the algorithm: cost-graph calculation, the FOREST BUILDING algorithm, and the DECOMPOSE COMPONENT algorithm. We have introduced each of them in great detail with examples, and finally have proven that the polynomial-time approximation algorithm exists by giving its formalized description.

In the next chapter we will introduce a software library written in the Go programming language by the author, which implements the approximation algorithm discussed in this chapter.

Chapter 3

Implementation in Go

3.1 About Go

3.1.1 The Go language

The Go programming language (also known as “Golang”) has first appeared around 9 years ago, but it wasn’t until a couple of years ago that it gained a large boost in popularity. Go is a statically typed, compiled language. It has a C-like syntax, but with an emphasis on simplicity and safety[7]. It is easy to learn, and the number of keywords and features are small.

In terms of paradigms, the language is mostly imperative, with some functional and a few object oriented features. There is no type hierarchy or inheritance in the language. Instead, it emphasizes the usage of composition and interfaces. There are no classes either, but one can use structs and receivers on functions to achieve a somewhat similar effect.

Even though Go has a static type system and pointers, it also features a highly optimized garbage collector and memory safety.

3.1.2 Key differences

Below we list a few interesting differences compared to traditional object oriented languages. The list is not exhaustive, but gives a relatively good overview of what to expect when coming from another language, like Java.

Combined declaration/initialization: writing (`s:=3`) will not only declare and initialize the `s` variable, but also infer its type.

Multiple return values: functions can return multiple values. In fact, this is very common when handling errors.

Type system: apart from the standard types, and pointers one can use arrays and *slices* (dynamic array), and *structs* for custom types.

Error handling: there are no try-catch blocks and exceptions. An *error* can be returned from functions in case of an exceptional execution path and callers are expected to check for it, and either handle it or return an error themselves.

Interfaces: runtime polymorphism is provided by interfaces. They are very similar to *protocols* from other programming languages. They can be implemented implicitly. Conformance to an interface is checked statically by the Go compiler — this is referred to as *structural typing*. The empty `interface{ }` can refer to any type, and is similar to *Object* from other languages. Interfaces can be converted to other types at runtime with *type assertion*.

Packages: packages in Go represent a path, and references to other packages must always be prefixed with the name of the other package.

Visibility: publicly exposed members have a *Capitalized* name, and should be documented properly.

Concurrency: the language has a built-in toolkit in the form of *goroutines* and *channels* to deal with concurrent execution.

3.1.3 Advantages

One of the most important advantage of Go is its speed. Go programs run almost as fast as C, but also compile very quickly. The language is also cross-platform. The built-in compiler can target almost any major platform and architecture (including ARM). An additional benefit is the garbage collector. There is no need to deal with manual memory management, which makes it easy to focus on implementing business logic. Finally, testing is a first-class citizen in go. It has a built-in testing framework, which supports hierarchical unit tests, testable examples and benchmarks.

Based on the above listed features, Go proved to be an ideal candidate to implement the graph based anonymization algorithm with.

3.2 Bird's eye view

In this section we present a high-level overview of how our anonymization library works. The flowchart on Figure 3.1 illustrates what happens to the input data after it is passed into the algorithm.

The terminal states are denoted by rounded rectangles: the anonymization can terminate with an error if the input is not valid or not in the correct format (*error* state). Otherwise it will terminate by returning the anonymized data table (*done* state).

Processing steps are represented with rectangles, while data in the different stages is represented by diamond shapes. We can observe, that the input data is first transformed into its graph representation, then it goes through the anonymization process discussed in the previous chapter. The anonymization process yields partitions. Finally, the partitions are be mapped back to data rows. Rows in the same partition are generalized until they are equivalent, and the output anonymized data table is constructed.

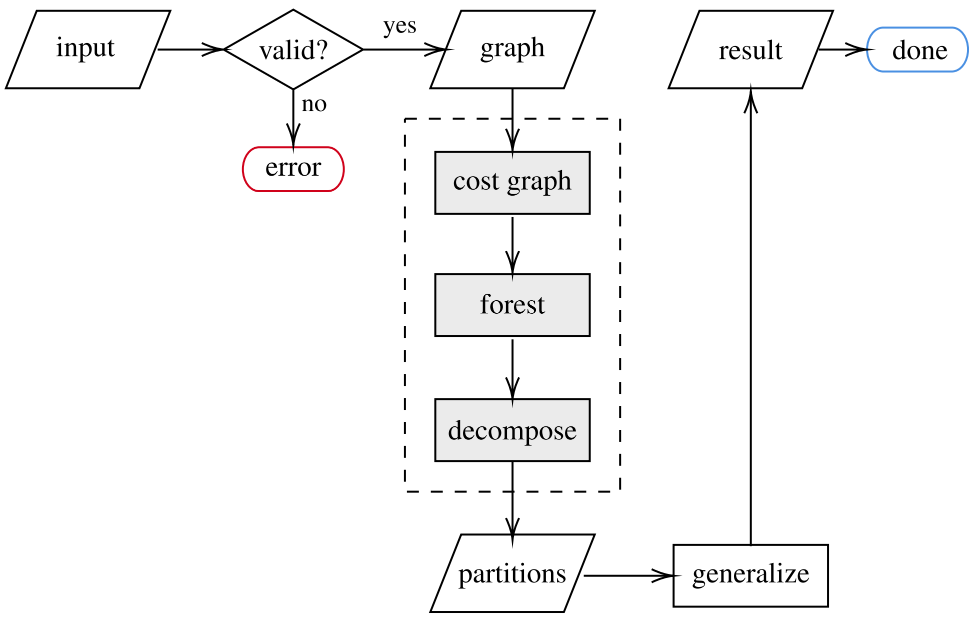


Figure 3.1: Bird's eye view of the program

3.3 Input format

The algorithm can only accept structured data, meaning that the input needs to have a static *schema*. The schema defines the name and position of the columns in the data table. Recall from Chapter 2.3.1, that in order to calculate the cost graph, the algorithm needs to summarize the *scaled generalization cost* for each dimension. This implies, that for each column we also need to provide the corresponding *generalizer*. Note, that as part of the input validation, the generalizer is also validated.

Once the schema is established, we can fill up the data table by providing the row data. Finally, by providing the k anonymity parameter, the input is fully defined.

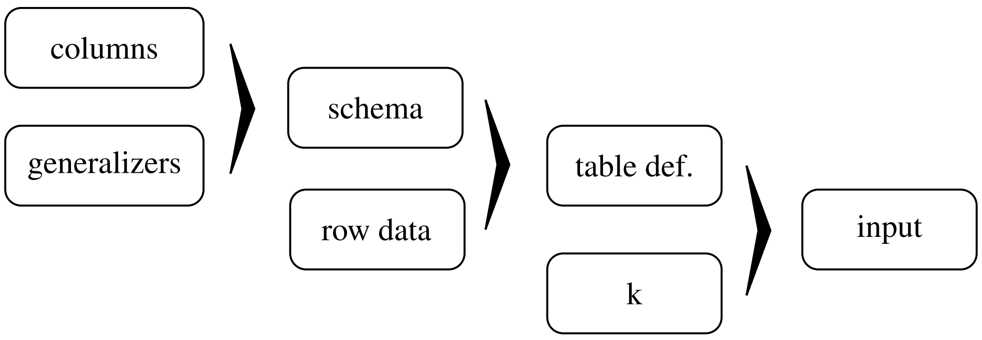


Figure 3.2: Input declaration

```
// g map contains a generalizer for each column
var g map[string]generalization.Generalizer
t := NewTable(&Schema{
Columns: []*Column{
    NewColumn("Name",    g["Name"]),
    NewColumn("Age",     g["Age"]),
    NewColumn("Salary",  g["Salary"]),
    NewColumn("Disease", g["Disease"]),
    },
})
t.AddRow("Joe", 30, 60000, "Flu")
t.AddRow("Eve", 27, 58000, "Sore throat")
```

Listing 3.1: Input data table declaration

Example input declaration

Listing 3.1 shows an example usage of how to declare an input table for the anonymizer algorithm. For simplicity's sake we assume, that a g map exists, which contains an appropriate generalizer for each column we use in the schema. (We will discuss how exactly generalizers are built in Section 3.4).

3.4 Generalization

Generalization is a core concept when talking about the *k-anonymity with generalization* problem. The algorithm described in Chapter 2 uses generalization at its core: both to determine which rows to put into the same partition, and during the final step when the original rows are anonymized.

3.4.1 Generalization API requirements

The library needs to provide a way for the API user to define how to generalize certain data types in their schema. The implementation also needs to be flexible, but convenient and easy-to-use at the same time. We want to make sure, that the implementation can satisfy at least the following requirements:

- supports **generalization hierarchy**, with convenient declaration
- supports **suppression**

- supports **sets, ranges**
- supports **text**
- supports **non**-hierarchy based generalization
- custom generalizers can be created by **implementing an interface**

Until now we have only discussed *hierarchy*-based generalization, and for the theoretical discussion of the anonymization algorithm it was perfectly sufficient. You can create a generalization hierarchy for basically any data type — even though the number of levels and the number of partitions on a given level may end up very large, or even infinite.

It is important to note however, that in real-life scenarios this is not always feasible. Storage, operating memory and computing time is finite, which means that for some dimensions (like *text*) we will need to apply a different type of abstraction.

3.4.2 Generalization implementation

Based on the requirements outlined in 3.4.1 we can define the interface of a `Generalizer`. Note, that since Go is not a classic object oriented language, a traditional class-hierarchy is not present in the design. What we do have in our toolkit are *interfaces* and *composition*.

We define the `Generalizer` interface with the following operations (refer to Figure 3.3):

1. generalize a partition *n* times
2. get the maximum number of levels of generalization

Remember, that a partition can either contain a single value, or a set of values, so it makes sense that the `Generalize` method takes and returns a `Partition`. It is also essential to know the maximum levels of generalization in advance, otherwise we wouldn't be able to compute the scaled generalization cost when comparing two partitions. There is a third convenience method on the interface which defines how to create a `Partition` from a raw value. The actual implementation of this may vary in each type that implements the

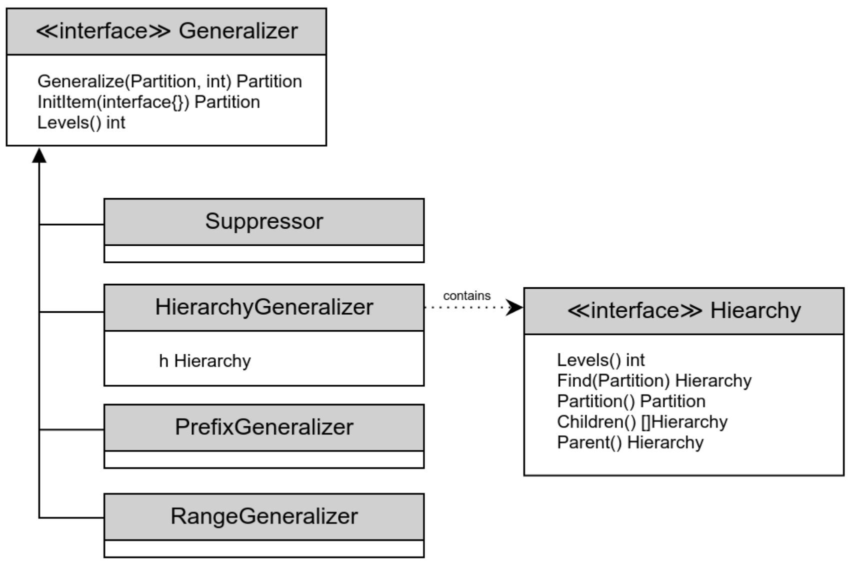


Figure 3.3: The generalization package

interface.

3.4.3 Suppressor

Based on the above `Generalizer` interface we can easily implement the `Suppressor` generalizer, which has only one level, and when generalized returns the suppressed value token: ‘*‘.

Example Listing 3.2 shows the code implementing a simple `Generalizer`, that can suppress values.

3.4.4 HierarchyGeneralizer

`HierarchyGeneralizer` represents generalization hierarchies in the classic sense that has been outlined in 1.4.2. This generalizer will *contain* an actual `Hierarchy` — as seen on Figure 3.3. The `Hierarchy` interface has functions to navigate the partitions within the hierarchy based on the parent-child relationships, a function to return the number of levels within the hierarchy and additional helper functions. The `hierarchy` package also contains helper functions to make it easy for the end-user to build it from a set of items.

```go
type Suppressor struct {
}

func (s *Suppressor) Generalize(p partition.Partition, n int) partition.Partition {
    if n == 0 {
        return p
    }
    if n == 1 {
        return s.InitItem("*")
    }
    return nil
}

func (s *Suppressor) Levels() int {
    return 2
}

func (s *Suppressor) InitItem(item interface{}) partition.Partition {
    return partition.NewItem(item)
}
```

Listing 3.2: Suppresor implementation

Manual hierarchy: We will now show how to manually declare a simple grade hierarchy similar to the one introduced on Figure 1.7, albeit with a lesser number of grades for simplicity's sake. The full code listing can be seen on Listing 3.3.

```go
func GetGradeHierarchy() Hierarchy {
    h, _ := Build(
        partition.NewSet("A+", "A", "A-", "B+", "B", "B-", "C+", "C", "C-"),
            N(partition.NewSet("A+", "A", "A-"),
                N(partition.NewSet("A+")),
                N(partition.NewSet("A")),
                N(partition.NewSet("A-"))),
        N(partition.NewSet("B+", "B", "B-"),
            N(partition.NewSet("B+")),
            N(partition.NewSet("B")),
            N(partition.NewSet("B-"))),
        N(partition.NewSet("C+", "C", "C-"),
            N(partition.NewSet("C+")),
            N(partition.NewSet("C")),
            N(partition.NewSet("C-"))))
    return h
}
```

Listing 3.3: Grade Hierarchy

In order to build the hierarchy, we need to describe it from top to bottom. Recall, that the top level of the hierarchy contains a single partition with every item in the domain. We then add the children of this partition as new sets, continuing until we reach the lowest level, where each partition contains only a single item.

Automatic hierarchy: another, more convenient way to build a hierarchy is by using the included `AutoBuild` function. It takes two parameters: the amount of children a node should have, and a variadic array of the items to partition. The function will build a valid hierarchy on-the-fly by splitting the set of items to the specified child partitions on each level, until only singleton partitions remain. (Note, that we may still get an error in case there is a problem with the arguments, for example when the desired child count is larger than the available items.) The builder may create partitions with less than the specified amount of children in cases when there aren't enough items. This often happens when the original number of items is not divisible by the desired child count.

We now show how to build the same grade hierarchy automatically. Listing 3.4 demonstrates how to specify the child count and the items. By passing in *3* as the desired child count we achieve the exact same result as the previous example.

```
h, err := hierarchy.AutoBuild(3, "A+", "A", "A-", "B+", "B", "B-", "C+", "C", "C-")
```

Listing 3.4: Hierarchy with the AutoBuild function

Validation A hierarchy will also be *validated* when it is built. Validation will enforce, that:

- an item can only appear once on a given level
- the superset of partitions on each level contains all items from the domain

3.4.5 RangeGeneralizer

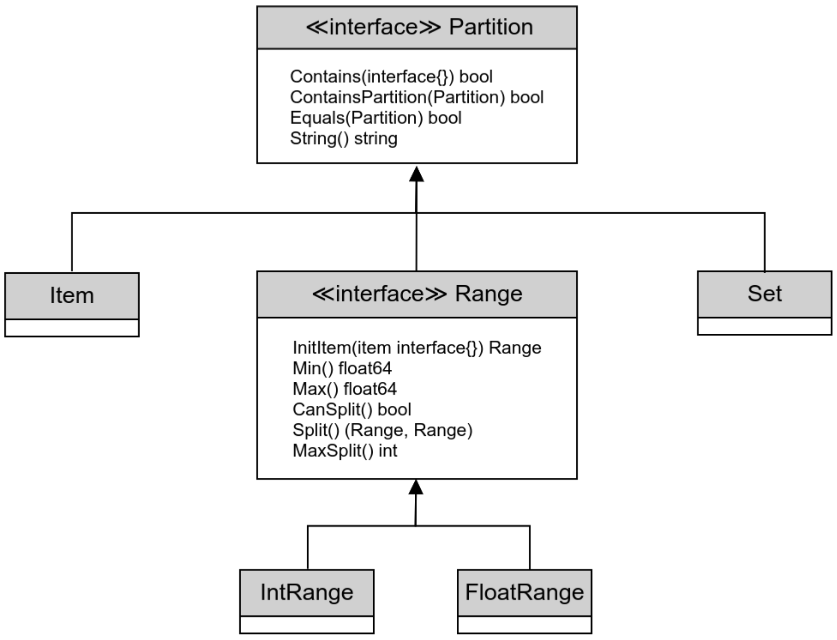


Figure 3.4: Partition hierarchy

We have previously seen how partitions are the building blocks of generalization. They are very similar to *sets* in the mathematical sense. In a standard generalization hierarchy we only needed to use `Partition`, `Item` and `Set`. (See Figure 3.4.)

Representing larger ranges of numbers with the above strategy however would be very tedious. We would not only have to write up each number on every level of the hierarchy, but also need to determine at what number the ranges are split up into partitions, and assign the proper parent-child relationships. It is too much work, and in case of floating point numbers not even possible. One option could be to generate a hierarchy with a `HierarchBuilder` as mentioned in 3.4.4. This removes the manual work of building up the partitions, but still has the need to store the whole hierarchy in the memory. For very large sets this could be inefficient. We need a more feasible way of describing these types of dimensions. This is why the `RangeGeneralizer` has been built into the library.

As it can be seen from Figure 3.4, a `Range` interface can be derived from `Partition` which will deal with representing and splitting the ranges. Its two implementations are `IntRange` and `FloatRange`.

The basic idea is, that after a number range is declared with its minimum and maximum boundaries, we can dynamically compute the partitions at the required level *on the fly*. Instead of keeping the full hierarchy in the memory, the `RangeGeneralizer` will `Split ()` the initial range the requested `n` number of times to arrive at the necessary generalization level.

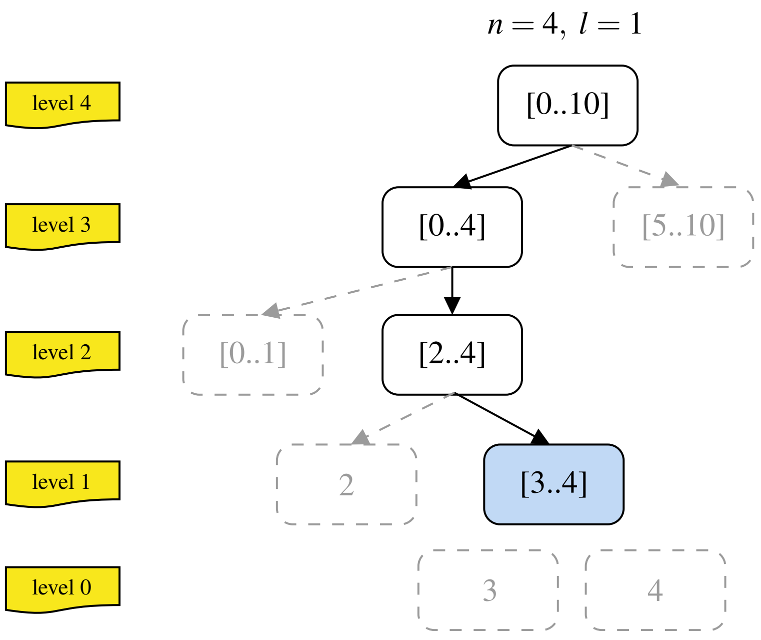


Figure 3.5: Range generalizer example

Figure 3.5 shows an example of this in action. (Nodes with gray dotted border represent only a visual aid for the reader, the algorithm does not need to compute them.) Let's assume we are generalizing a range of integers $[0\ldots10]$. If we are interested in what partition the value $n = 4$ would be generalized into on level $l = 1$, we need to split the range $s = l_{\max} - l - 1 = 3$ times. After each step we need to keep splitting the partition in which the requested element exists. After the required number of splits we arrive at the result (partition $[3\ldots4]$) without having to worry about constructing the actual generalization hierarchy at any point. Note, that the distribution of the items in the partitions is actually handled by how the `Split ()` function is implemented for the corresponding `Range`. In this case (and in the library) we are using a uniform distribution by splitting each range at the median value.

With some modification this strategy can implemented for floating point ranges as well, but the code will need to take floating point precision into account, and compare with a delta.

3.4.6 PrefixGeneralizer

The `PrefixGeneralizer` is another built-in generalizer, that does not rely on a fixed hierarchy. It deals with generalizing plain text data. As the name suggests, it will try to use different sized prefixes to see if two different strings can be collapsed into the same partition.

The input text will be interpreted as a series of words. The generalizer has a `MaxWords` setting, and it cannot handle text that has a larger word count. Note, that the actual length of words in characters is *irrelevant* for this generalizer, and is only bounded by operative memory.

Example: the example on Figure 3.6 shows how the prefix generalizer transforms two different sentences when different levels of generalization l are selected. We assume, that $MaxWords = 5$ for this example.

The first sentence only contains 3 words, so the `PrefixGeneralizer` will pad the last two slots. This makes it easy to compare the two sentences word-by-word. On levels 0, 1 and 2 the string prefixes are not equal. On level 3 the two strings are equal. Calculating the generalization cost for the two values would also be pretty simple: we divide the lowest level of generalization which assigns the two items into the same partition by the maximum levels of generalization: $h/l_{\max} = 3/5$.

Figure 3.7 illustrates what happens when the two strings differ in all prefixes. In this case

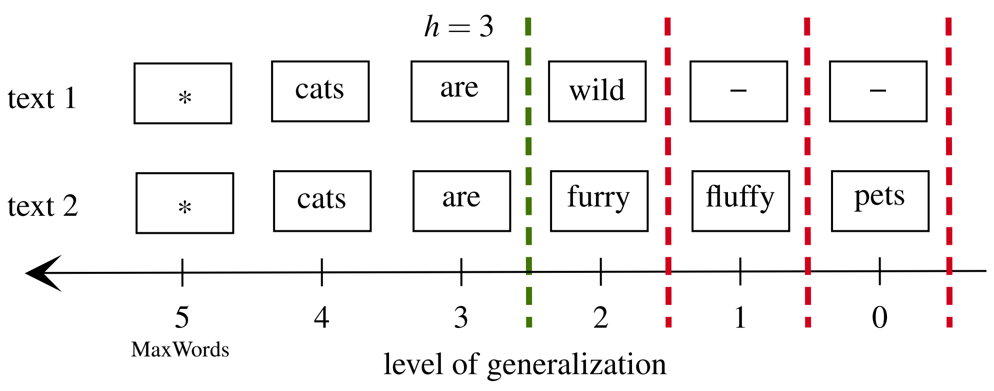


Figure 3.6: Prefix generalizer (h=3)

the cost of generalization will be $h/l_{\max} = 1$, and both of the two strings are *suppressed* (replaced with the “*” token).

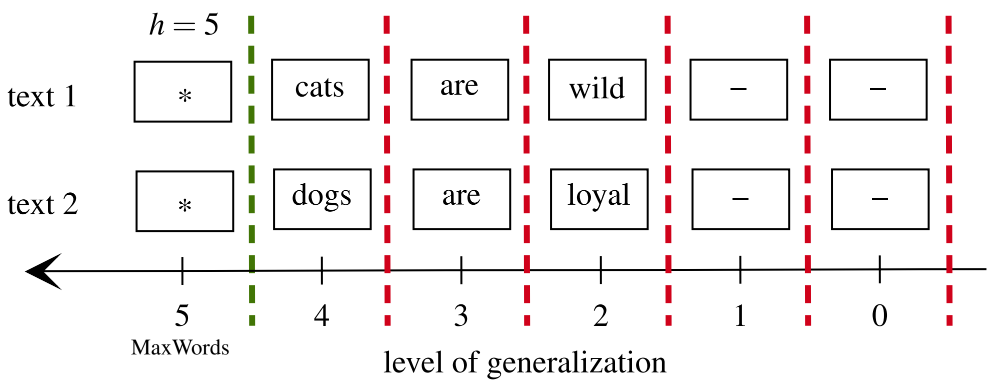


Figure 3.7: Prefix generalizer (h=5)

3.4.7 Custom Generalizers

Finally, if none of the built-in generalization techniques are adequate for a certain use case, the user of the library can still implement the `Generalizer` interface to provide a custom logic.

Recall from Section 3.4, that in order to properly implement the `Generalizer` interface, one has to implement the following methods:

`Levels () int`:

This method should return the maximum number of times the generalizer can generalize the values. In hierarchy based generalizers it is usually the number of levels in the tree.

It is important to note, that the levels are zero-indexed, which needs to be taken account when calculating the number of levels with a non-hierarchy based logic.

`Generalize (Partition, int) Partition`:
This method should generalize the input partition n times (providing that $n \leq Levels())$ and return the resulting partition. The result will be used by the core anonymization algorithm when calculating the cost graph.

`InitItem (interface{ }) Partition`:
Implement this method to tell how to initialize a raw value. In most cases you just want to wrap it into an `Item` or `Set`.

3.5 Graph algorithm

After discussing the input format and the various built-in generalization methods we can now get to the meat of the anonymization algorithm. As Figure 3.1 illustrated, the pipeline will closely resemble the steps described in the anonymization algorithm in Chapter 2.

3.5.1 Graph representation

The anonymization algorithm heavily uses graphs, so we need to either implement or pick an existing graph library for Go, that can handle all the required graph operations efficiently. In this library we have chosen the latter, and selected a third-party component.

Graph library requirements

- handle both directed and undirected graphs
- handle edge weight/cost
- find cycles
- get connected components

At the time of implementing the library, the available open source contenders were

```
func buildEmptyCostGraph(t *model.Table) *simple.WeightedUndirectedGraph {
    g := simple.NewWeightedUndirectedGraph(0, math.MaxFloat64)
    for i := range t.GetRows() {
        node := simple.Node(i)
        g.AddNode(node)
    }
    return g
}
```

Listing 3.5: Empty cost graph creation

gonum [9], **yourbasic-graph** [22] and **go-graph** [16]. After briefly evaluating each library it turned out that **gonum** provides the richest feature set, supporting all of the required functions — including topological ordering.

Gonum is a set of numeric libraries, containing libraries not only for graphs, but for matrices, statistics, linear algebra, probability distributions, network analysis and more [10]. In our anonymization library however we only used its graph based functionality.

3.5.2 Cost graph

The first step of the anonymization algorithm was to construct a cost graph. For this, we need to convert the input table into its graph based representation. Recall from Section 2.3.1, that the cost graph is a weighted full graph, where the nodes are the table rows and the edge weights between them are the generalization cost of bringing the two rows into the same partition.

The cost graph creation is handled by the `cost_graph.go` file, while the edge weight calculation is implemented in `cost.go`. The first part of the algorithm, creating the "empty" cost graph can be seen on Listing 3.5.

The code is using the `NewWeightedUndirectedGraph` function from the *gonum* package. Nodes are uniquely identified with an index, which will be the number of the row they are representing in the input table.

The essence of the code that calculates edge weights can be seen on Listing 3.6.

```
func calculateCostFraction(p1, p2 partition.Partition,
            g generalization.Generalizer) (float64, error) {
    maxLevels := g.Levels()
    for level := 0; level < maxLevels; level++ {
        g1 := g.Generalize(p1, level)
        g2 := g.Generalize(p2, level)
        if g1 != nil && g1.Equals(g2) {
            return float64(level) / float64(maxLevels-1), nil
        }
    }
    return 0, fmt.Errorf(fmt.Sprintf("data cannot be generalized
                into same partition: %v, %v", p1, p2))
}
```

Listing 3.6: Generalization cost calculation

```
func BuildAnonGraph(table *model.Table, k int) (graph.Directed, error) {
    costGraph, err := BuildCostGraph(table)
    if err != nil {
        return nil, err
    }
    g := buildEmptyAnonGraph(table)
    for {
        components := UndirectedConnectedComponents(g)
        c := pickComponentToExtend(components, k)
        if c == nil {
            break
        }
        u := pickSourceVertex(g, c)
        v := pickTargetVertex(g, c, u, costGraph)
        g.SetEdge(g.NewEdge(u, v))
    }
    return g, nil
}
```

Listing 3.7: Forest building algorithm

The function will try to generalize the input partitions starting from 0 to `maxLevels`, until both partitions are equal. If the two partitions are still different when generalizing them to `maxLevels` it is an error — the Generalizer is invalid, as by definition all values should be in the same partition on the highest level.

3.5.3 Forest building

The forest building algorithm is responsible for constructing the directed unweighted graph, that represents the nodes (before cutting) that belong to the same partition. The algorithm is in `anon_graph.go`. Listing 3.7 shows the outline of the algorithm.

```go
func (d *Decomposer) Decompose() {
    threshold := d.getThreshold()
    for {
        c := d.pickComponent(threshold)
        if c == nil {
            break
        }
        d.partitionComponent(c)
    }
}
```

Listing 3.8: Decompose main loop

The function will first build up the cost graph (see 3.5.2), then create an empty directed graph, which contains the nodes representing the table rows. It will then start to iterate, and picks a component to extend (based on the conditions outlined in 2.8). If there is no component to extend, the iteration has finished. Otherwise it picks a target vertex and connects the source and target vertex with a new directed edge.

3.5.4 Decomposition

The file `decompose.go` is concerned with cutting the oversized components to smaller ones, and implements all the different cut types.

The main loop of the decompose logic can be seen on Listing 3.8. At a high-level it is very simple: we look for a component to cut that is over the threshold, and partition it. If there is no component, we exit the loop.

The essential part of the code for the partitioning logic is listed in 3.9. After finding the u and v root vertices, and calculating the s and ϕ values (see Section 2.3.4) it will decide which cut type to perform. The actual implementation of the different cut types rely heavily on the gonum package and will not be listed here.

3.6 Example run

In this section we look at a basic, but full example run.

```go
func (d *Decomposer) partitionComponent(component []graph.Node) {
    u, v, t := d.getSplitParams(component)
    s := d.calculateSize(component)
    if t >= d.k && s-t >= d.k {
        d.performCutTypeA(u, v)
    } else if s-t == d.k-1 {
        d.performCutTypeB(u, v)
    } else if t == d.k-1 {
        d.performCutTypeC(u, v)
    } else {
        d.performCutTypeD(u, v, component)
    }
}
```

Listing 3.9: Partition cutting

```go
table := model.NewTable(&model.Schema{
    Columns: []*model.Column{
        model.NewColumn("Name", &generalization.Suppressor{}),
        model.NewColumn("Status", nil),
        model.NewColumn("Age", generalization.NewIntRangeGeneralizer(0, 150)),
        model.NewColumn("Grade", generalization.GradeGeneralizer()),
        model.NewWeightedColumn("Motto",
            &generalization.PrefixGeneralizer{MaxWords: 10}, 0.1),
    },
})
table.AddRow("Joe",  "employee", 25, "A",  "cats are wild")
table.AddRow("Jane", "employee", 29, "A+", "cats are wild")
table.AddRow("Jack", "client",   35, "A-", "dogs are loyal")
table.AddRow("Jill", "client",   47, "A",  "cats and dogs are pets")

a := &Anonymizer{ K: 2, Table: table }
a.Anonymize()
```

Listing 3.10: Example run

As a first step we define a table schema, and populate it with a few data rows. Next we create an `Anonymizer` instance, and pass in the reference to the table, and the *k* anonymity parameter. Finally we call the `Anonymize` method.

There are a couple of interesting things in the example that haven't been mentioned before.

First, we can use a `nil` value instead of passing in a generalizer when adding a new column to the schema. This will actually result in the column to be entirely ignored by the algorithm, as if it were a *non-identifier* attribute.

The second notable thing is, that we can also use a `WeightedColumn` instead of a normal one. In the example this is done for the *Motto* column. When calculating the cost graph of a weighted column, the algorithm will scale the generalization cost with the specified weight whenever a value from this columns is used. This can be handy, if we want to suppress some columns more than others. What will actually happen is that the cost of these columns will be artificially increased or reduced during cost calculation, thus influencing the final scaled generalization cost and skewing the result in favor of generalizing some columns to higher levels than others.

After the `Anonymize` method finishes, the table passed into the anonymizer is processed in place, and we can access the result through the `table` reference. Printing the result will yield a table similar to Figure 3.6 (random elements in the algorithm can cause the result to end up slightly differently each time it is run). Note, that as requested by the input declaration, the *Status* column is skipped, and not anonymized.

Name	**Status**	**Age**	**Grade**	**Motto**
*	employee	[0..74]	[A]	cats
*	employee	[27..36]	[A, A+, A-]	*
*	client	[27..36]	[A, A+, A-]	*
*	client	[0..74]	[A]	cats

Figure 3.8: Example run anonymized result

3.7 Continuous mode

Continuous mode is a special mode in which the Anonymizer can be used. In continuous mode it is possible to feed the rows of the data table in multiple different segments instead of adding them all at once. It is very useful, when the data arrives in a “stream” format and needs to be processed either row-by-row or in smaller chunks.

3.7.1 Implementation

The core algorithm outlined earlier can be used with very little modification. The only notable restriction is, that the row count of the first data chunk must be at least the size of the *k* anonymity parameter.

The anonymizer will process the first batch normally. When the second batch of data arrives, it will merge the data rows into the *already anonymized* data table, and run the algorithm again. During the cost-graph calculation there will be rows from the first iteration, which are already in the same partition. The cost of these edges in the cost-graph will all be 0. Note however, that this does not have an effect on the rest of the algorithm. In the forest building algorithm a new row will either be partitioned into a partition created in the first iteration, or a completely new partition created from new rows based on the scaled generalization cost in the cost graph. The decompose step is completely unaffected.

Example Let’s assume, that the continuous anonymizer receives the first chunk of data and creates the anonymized table as illustrated onFigure 3.9.

Name	Age	Pet
John	27	cats
Susan	38	dogs
Sarah	30	cats
Mark	40	dogs

Name	Age	Pet
*	[25..30]	cats
*	[38..40]	dogs
*	[25..30]	cats
*	[38..40]	dogs

Figure 3.9: First iteration

Now let’s see what happens when the second chunk of data arrives and is added to the already anonymized data table (Figure 3.10).

Name	Age	Pet
*	[27..36]	cats
*	[37..55]	dogs
*	[27..36]	cats
*	[37..55]	dogs
Jack	38	cats
Jill	40	dogs
Lilly	27	cats

Name	Age	Pet
*	[25..30]	cats
*	[38..40]	dogs
*	[25..30]	cats
*	[38..40]	dogs
*	[38..40]	dogs
*	[38..40]	dogs
*	[25..30]	cats

Figure 3.10: Second iteration

3.7.2 Effect on the output

Both solutions are correct in terms of *k-anonymity*, but would the order of partitioning change if we knew all rows in advance? The answer is *sometimes*. Figure 3.11 shows a result from anonymizing the above 7 rows normally, in one go. If the *random* order in which the graph algorithm picks components to extend is more favorable, we might get a more efficient result (with less information loss) compared to when anonymizing in multiple chunks.

Name	Age	Pet
*	[25..30]	cats
*	[38]	dogs
*	[25..30]	cats
*	[40]	dogs
*	[38]	dogs
*	[40]	dogs
*	[25..30]	cats

Figure 3.11: Results knowing all rows in advance

So how does the continuous mode affect the characteristics of the anonymization? Obviously *k-anonymity* is not violated in continuous mode either. However, we can see that the "optimality" of the solution may decrease based on the order in which the rows arrive and the size of the data chunks. If two or more rows are anonymized into the same partition in the first iteration, some of the data is lost. There is no way to tell, if it might have been *cheaper* (in terms of anonymization cost) to partition a certain row with another from the second batch instead.

```
// create the table and schema as usual
table := model.NewTable(&model.Schema{
    Columns: []*model.Column{
        model.NewColumn("Name", &generalization.Suppressor{}),
        model.NewColumn("Age", generalization.NewIntRangeGeneralizer(0, 50)),
        model.NewColumn("Pet",
            &generalization.PrefixGeneralizer{MaxWords: 10}),
    },
})

// add the first batch of rows
table.AddRow("John", 27, "cats")
table.AddRow("Susan", 38, "dogs")
table.AddRow("Sarah", 30, "cats")
table.AddRow("Mark", 40, "dogs")

// create an anonymizer and run the anonymization
anon := &Anonymizer{
    Table: table,
    K:     2,
}
anon.Anonymize()

// add the second batch of rows
table.AddRow("Jack", 38, "dogs")
table.AddRow("Jill", 40, "dogs")
table.AddRow("Lilly", 27, "cats")

// anonymize again
anon.Anonymize()

// print the results
fmt.Printf("%v", anon.Table)
```

Listing 3.11: Continuous anonymization

This means, that the continuous version of the algorithm will generalize each dimension to *at least* the same generalization level as the normal version, but sometimes it will generalize them more. This will result in somewhat more information loss than when anonymizing the same data in the standard mode.

3.7.3 Usage

Listing 3.11 shows how to actually write code that uses the continuous anonymization mode. It is not much different from the regular mode: we only need to call the `Anonymize` method after each data chunk arrives, and add a new batch of rows using the table reference afterwards.

3.8 Summary

In this chapter we have briefly introduced the Go programming language, and why we think it is a good choice for implementing the anonymization algorithm.

Then we took a high-level overview of the algorithm implementation, including the different pieces, and how they work together. After which we discussed in what format the algorithm will accept the input data. We gave a detailed description of how generalization works, and how one can use the built-in generalizers or implement a custom one.

Next we justified our decision on the selected graph library used by the program, and went through the most important steps of the graph algorithm one-by-one: cost graph calculation, forest building and decomposition.

Then we have shown a code listing for a full run of the anonymizer program for a small sample data table, and illustrated a possible result. Finally we briefly looked at how the continuous mode works for the anonymizer, and when we can take advantage of it.

In the next chapter we will talk about testing, performance and benchmarks.

Chapter 4

Tests, Benchmarks

4.1 Testing in Go

In Go testing is built into the language and the core tooling. There is no need to depend on third party libraries, one can simply use the `testing` package. Tests can be invoked with the `go test` command which automates execution of any function which matches the required naming convention[8] of `func TestXxx (*testing.T)`. (See Listing 4.1 for a basic example).

A test suite can be crated by putting the test functions into a file whose name ends with `_test.go`. This file needs to be in the same package as the one being tested. All tests will

```go
package demo

import "testing"

func TestFactorial(t *testing.T) {
    fact := Factorial(5)
    if fact != 120 {
        t.Errorf("got: %d, want: %d.", fact, 120)
    }
}
```

Listing 4.1: A basic Go test

be excluded from regular package builds[8], meaning that production artifacts will not be polluted with test files.

Actual test functions will rely on the `*testing.T` argument passed into the test function to interact with the testing framework. The testing toolkit follows the general philosophy of Go, being minimal, simple to use yet performant and robust.

4.1.1 Verifying expectations

When compared to testing libraries in other languages, the most obvious difference is the *absence of assertions*. Code that checks expectations should be written explicitly, and the test should be terminated with a respective function call from `t` when expectations aren't met. (For example `t.Error` or `t.Fail`.)

But why isn't `assert` part of Go's testing package? One of the most important reasons is that the creators of the language wanted to avoid a common problem with other testing frameworks, namely that tests feel like they are written in a different language[24]. They often have their own special syntax (like *Mocha, RSpec*) which reads very differently from "normal" code.

4.1.2 Special test types

Sub-test

The testing package contains a `Run` method on the `testing.T` and `testing.B` types, that allows the creation of sub-tests and sub-benchmarks. Sub-tests result in a more readable test code and provide a way to handle common setup for test cases.

Table tests

Table driven tests run the same logical test code for multiple similar test cases. The input is often referred to as *table*. In Go, table tests are implemented by declaring a slice of test cases, then looping over them and running each iteration as a sub-test. This is called a table-driven test. (See Listing 4.2.)

```go
func TestFactorial(t *testing.T) {
    tests := []struct { n, want int }{
        {0, 1},
        {1, 1},
        {2, 2},
        {3, 6},
    }
    for _, test := range tests {
        t.Run("factorial test", func(t *testing.T) {
            // test logic
        })
    }
}
```

Listing 4.2: Table-driven test

```go
func BenchmarkFactorial(b *testing.B) {
    for i:=0; i < b.N; i++ {
        // run the function
    }
}
```

Listing 4.3: Go Benchmark

Benchmarks

Benchmarks are special tests, with the goal of measuring the performance of a given function or code block. They are placed into `_test.go` files, just like the normal tests, but their names start with the `Benchmark` word, and they get the `*testing.B` type as parameter. A typical benchmark runs the function `N` times, which value is provided by the benchmark runner. (It will increase the value until the spread is within a given margin of error.) The runner will also take care of calculating and printing the average execution time (ns/op). A skeleton of a typical benchmark is shown in Listing 4.3. Note, that benchmarks can be combined well with table tests to perform a detailed analysis of a function for different kinds of arguments. We will use this feature heavily in Section 4.3 when measuring the performance of the anonymization algorithm.

4.1.3 Coverage

The Go testing framework has built-in support for statement coverage. It can be used in conjunction with `go test` by passing on the `-cover` flag. While higher statement coverage is usually better, a 100% statement coverage does not mean that there can be no errors. We aim to use coverage as a metric to highlight edge-cases and missed branches during testing.

4.2 Algorithm Correctness

In this section we will reason about the *correctness of the implementation*.

We will be relying on the unit and integration test suite that have been created along with the algorithm. Unit tests will not give a mathematical proof of correctness or total correctness (the former proves input-output correctness, while the latter also includes proof of termination). A well-tested codebase however can achieve a similar result with a high degree of confidence.

4.2.1 Generalization tests

For each built-in generalizer implementation a corresponding test suite verifies the correct behavior. Without getting too much into the details, Figure 4.1 shows a summary of the tests and their locations.

File	Functionality
`suppressor_test.go`	verifies basic suppression functionality
`hierarchy_test.go`	basic & auto builder, including edge cases for non-full trees and invalid input
`hierarchy_generalizer_test.go`	generalization to different levels, and related edge cases, benchmarks
`range_generalizer_test.go`	tests for int & float ranges, including range validation, virtual hierarchy splitting and levels
`prefix_generalizer_test.go`	tests the prefix generalization logic, including validation of input

Figure 4.1: Generalizer Tests

4.2.2 Core algorithm tests

Cost graph tests

These tests cover the building and edge-cost calculation of cost graphs.

One test suite deals with verifying the cost calculation formula. These tests are located in `cost_test.go`. The cost calculation algorithm is tested with several different data types.

The second test suite tests how the cost-graph is built. In these tests the input is always a *table*, and we verify, that the structure and edge-costs of the output graph is as expected. Tests also cover edge cases, like invalid data in the table, invalid generalizer for the column and empty input. The cost graph test suite is in `cost_graph_test.go`.

Forest building tests

The tests for the forest building algorithm are located in `anon_graph_test.go`, and they verify the following aspects of the algorithm:

- a correct component should be picked for extension
- a correct vertex should be picked as source vertex
- a correct vertex should be picked as target vertex
- forest (tree) properties are not violated after the new edge is added

Decompose tests

The decompose step is a rather complex step in the core algorithm. The test suite in the file `decompose_test.go` tries to deal with this complexity. With the proper set of helper methods however, the tests in it are still readable and relatively easy to follow. An example can be seen on Figure 4.4.

Other tests in the suite verify the following:

- oversized threshold calculation logic

```
//             ------- 0 ------
//            |                |
//       ---- 1 ----           6
//      |     |     |          |
//      2     3     4          7
//            |
//            5
//
//   u := 0, v := 1, k := 4, s > 2k-1
//   s-t == k-1
func TestPartition_TypeBCut(t *testing.T) {
    g := GetUndirectedTestGraph2()
    d := NewDecomposer(g, 4)
    u := g.Node(0)
    v := g.Node(1)
    d.performCutTypeB(u, v)
    testutil.AssertVertexReplaced(t, g, 1, 8, 2, 3, 4)
}
```

Listing 4.4: A test for Decompose Cut type B

- selected component is actually oversized
- cut types A through D
- handling of Steiner's vertices
- the algorithm stops when there are no more oversized components in the forest

Aonoymizer tests

Last, but not least the tests in `anonymize_test.go` check, that all the above work well together. They call the `Anonymizer` for a given test input table and verify, that *k-anonymity* on the output is not violated (at least $k-1$ rows must be the same for any given row).

4.2.3 Examples

Several of the above tests also have *testable examples*. Examples in Go are special tests, which contain a runnable snippet of code, that demonstrates API usage or functionality. They will be run by the testing framework along with the rest of the tests, and their example output will be verified automatically. A mismatch in the example output will cause the test to fail.

Each provided testable example is listed on Figure 4.2.

File	Functionality
`anonymize_example.go`	demonstrates basic anonymization
`continuous_example.go`	demonstrates continuous anonymization

Figure 4.2: Testable examples

4.2.4 Stats & metrics

Table 4.3 shows some interesting metrics about the project.

Metric	Value
Number of source files	44
Total LOC	4370
Number of Unit Tests	257
Number of Benchmarks	12
Number of Examples	2
Statement Coverage	90.5%

Figure 4.3: Metrics

4.3 Performance, Benchmarks

In this section we will present the results of the benchmarks executed on the anonymization algorithm, and compare the results to the expected algorithm characteristics introduced earlier in Chapter 2. As mentioned in Section 4.1.2 benchmarks in Go are special tests, which can be executed with the testing framework. Go will help us run the analyzed function until the measure is *stable*, and is able to precisely calculate the average execution time.

4.3.1 RangeGeneralizer benchmarks

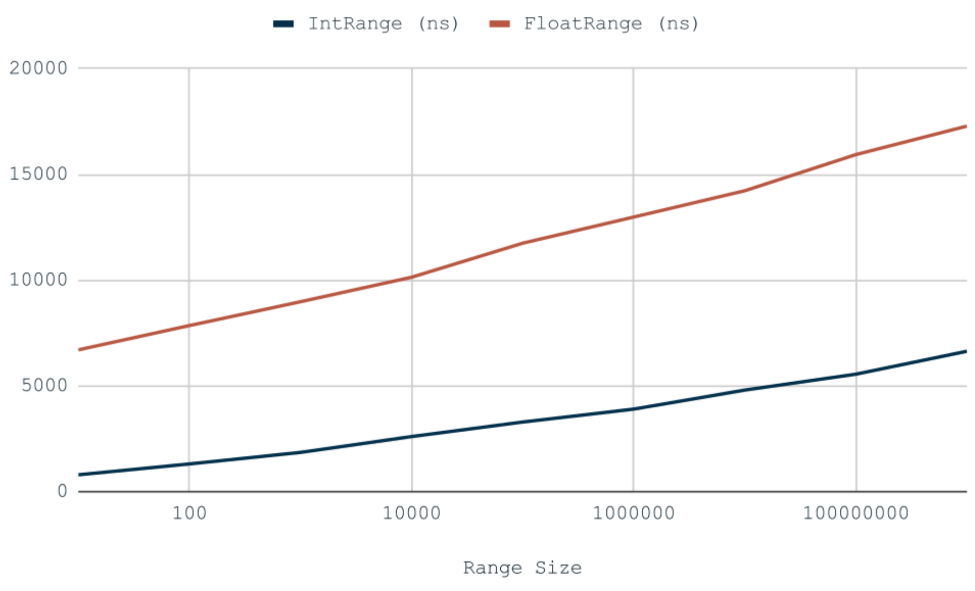


Figure 4.4: Range Generalizer Benchmarks

4.3.2 PrefixGeneralizer benchmarks

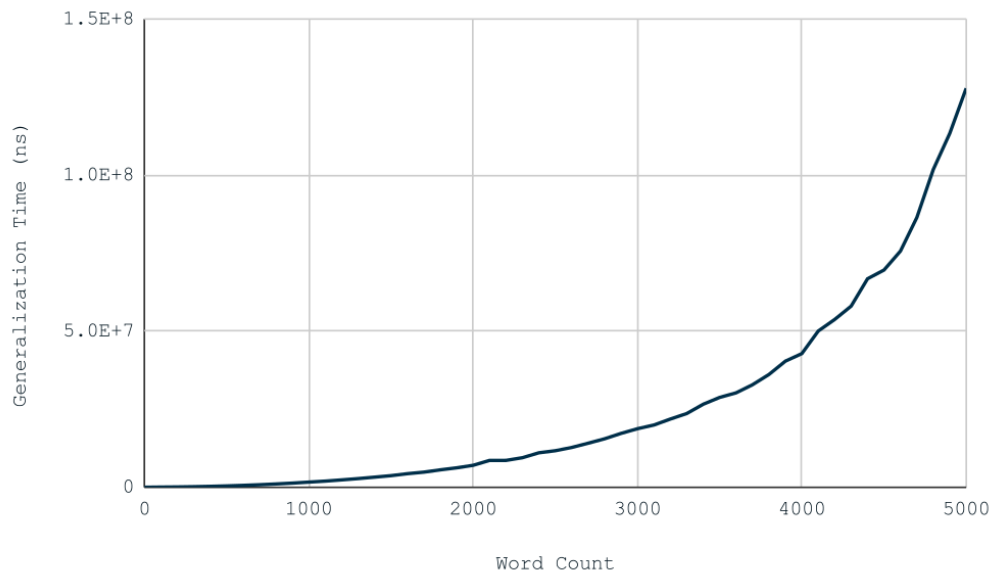


Figure 4.5: Prefix Word Count Benchmark

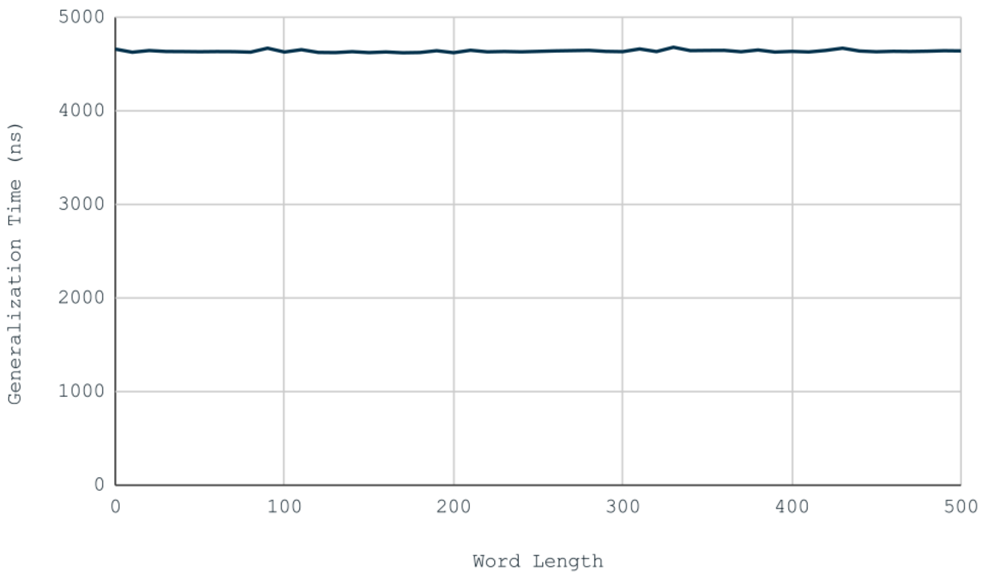


Figure 4.6: Prefix Word Length Benchmark

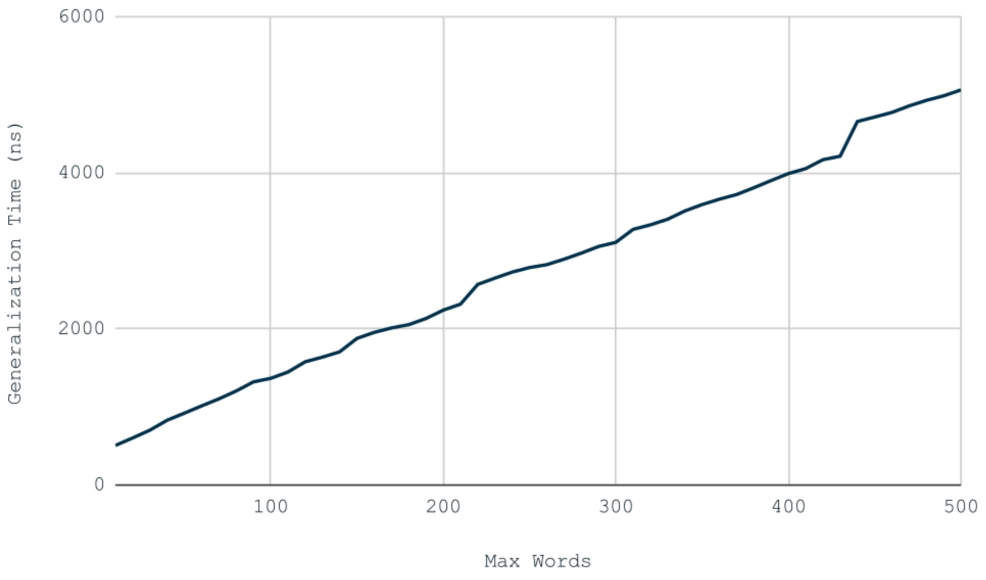


Figure 4.7: Prefix Max Words Benchmark

4.3.3 HierarchyGeneralizer benchmarks

4.3.4 Comparison of generalizers

4.3.5 Anonymization benchmarks

Chapter 5

Integration

Chapter 6

Improvements, Future Work

Appendix A

Appendix

A.1 Raw benchmark data

Bibliography

[1] https://www.nytimes.com/2012/02/19/magazine/shopping-habits.html. Accessed: 2019-08-14.

[2] Cambridge analytica. https://en.wikipedia.org/wiki/Cambridge_Analytica. Accessed: 2019-04-30.

[3] Criticism of amazon. https://en.wikipedia.org/wiki/Criticism_of_Amazon. Accessed: 2019-04-30.

[4] Data anonymization. https://en.wikipedia.org/wiki/Data_anonymization. Accessed: 2019-04-30.

[5] The first real test of gdpr. https://clearcritique.com/the-first-real-test-of-gdpr/. Accessed: 2019-08-10.

[6] General data protection regulation. https://en.wikipedia.org/wiki/General_Data_Protection_Regulation. Accessed: 2019-04-30.

[7] Go programming language. https://en.wikipedia.org/wiki/Go_(programming_language). Accessed: 2019-09-18.

[8] golang.org/testing. https://golang.org/pkg/testing/. Accessed: 2019-09-30.

[9] gonum. https://github.com/gonum/gonum. Accessed: 2019-09-22.

[10] gonum.org. https://www.gonum.org/. Accessed: 2019-09-22.

[11] Google's top 35 privacy scandals. http://precursorblog.com/?q=content/googles-top-35-privacy-scandals. Accessed: 2019-04-30.

[12] How people disappear. https://www.youtube.com/watch?v=CPBJgpK0Ulc. Accessed: 2019-08-14.

[13] Iso 25237:2017. https://www.iso.org/standard/63553.html. Accessed: 2019-04-30.

[14] Np-hardness. `https://en.wikipedia.org/wiki/NP-hardness)`. Accessed: 2019-08-13.

[15] P vs np problem. `https://en.wikipedia.org/wiki/P_versus_NP_problem)`. Accessed: 2019-08-13.

[16] Steplg/go-graph. `https://github.com/StepLg/go-graph`. Accessed: 2019-09-22.

[17] Top five concerns with gdpr compliance. `https://legal.thomsonreuters.com/en/insights/articles/top-five-concerns-gdpr-compliance`. Accessed: 2019-08-10.

[18] The vanishing state of privacy. `https://magazine.factor-tech.com/factor_winter_2017/richard_stallman_and_the_vanishing_state_of_privacy`. Accessed: 2019-04-30.

[19] The wired guide to your personal data. `https://www.wired.com/story/wired-guide-personal-data-collection`. Accessed: 2019-04-30.

[20] The world's most valuable resource is no longer oil, but data. `https://www.economist.com/leaders/2017/05/06/the-worlds-most-valuable-resource-is-no-longer-oil-but-data`. Accessed: 2019-04-30.

[21] Worst internet privacy scandals. `https://www.technadu.com/worst-internet-privacy-scandals/30236`. Accessed: 2019-04-30.

[22] yourbasic/graph. `https://github.com/yourbasic/graph`. Accessed: 2019-09-22.

[23] Gagan Aggarwal, Tomas Feder, Krishnaram Kenthapadi, Rajeev Motwani, Rina Panigrahy, Dilys Thomas, and An Zhu. Approximation algorithms for k-anonymity. *Journal of Privacy Technology*, 2005.

[24] Brian W. Kernighan Alan A. A. Donovan. *The Go Programming Language*. Addison-Wesley Professional, 2015.

[25] Jan van Leeuwen. Algorithms and complexity. *Handbook of Theoretical Computer Science. Vol. A*, 1998.

[26] L. Sweeney P. Samarati. Generalizing data to provide anonymity when disclosing information (abstract). *Proceedings of the 17th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, 1998.

[27] L. Sweeney. Uniqueness of simple demographics in the u.s. population. *Technical Report LIDAP-WP4, Laboratory for International Data Privacy, Carnegie Mellon University*, 2000.

[28] L. Sweeney. k-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-based Systems*, 2002.

Acknowledgments

I would like to thank…

Ákos Dudás my supervisor, for providing guidance and ideas.

Morgan Stanley my employer, for facilitating flexible working hours which allowed me to finish my studies.

My Girlfriend for the support during the long nights of studying and coding.

The results presented in the thesis were established in the framework of the professional community of Balatonfüred Student Research Group of BME-VIK to promote the economic development of the region. During the development of the achievements, we took into consideration the goals set by the Balatonfüred System Science Innovation Cluster and the plans of the "BME Balatonfüred Knowledge Center", supported by EFOP 4.2.1–16–2017–00021.

The research has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2–16–2017–00013, Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunications).