Computing Optimum Covers of Functional Dependencies

Guangrui Wang

Department of Computer Science
The University of Auckland

Supervisor: Professor Dr. Sebastian Link

A thesis submitted in partial fulfilment of the requirements for the degree of Master of Professional Studies in Data Science, The University of Auckland, 2020.

Abstract

Functional dependencies are important in relational databases as they formulate the constraints between different sets of attributes. There are different but equivalent ways to represent the same set of functional dependencies which are as known as covers. The performance of many data processing tasks such as data cleaning and query optimization deeply replies on how small the size of the cover is. In this article, we will talk about different types of ”smallest covers” (optimum covers), how they can be calculated, how about the time consumption and how small they are.

Acknowledgement

I would like to express my deepest appreciation to my supervisor Prof. Sebastian Link for his excellent guidance, suggestions and encouragement not only in this dissertation but also in other courses.

I would also like to extend my deepest gratitude to Dr Ziheng Wei who helped a lot to review and guide the implementation.

I am also grateful to my parents and my wife, for their unconditional love and support especially during this tough period against COVID-19.

Contents

Chapter 1

Introduction

The term ”relational database” was first introduced by Codd [5] from IBM in 1970. In the relational database model, the data are described in the form of relations. A relation can be represented as a table. Each row (except the header) indicates a relationship between a tuple of elements. Each column indicates a set of elements called an attribute.

A	B	C
a_1	b_1	c_1
a_1	b_2	c_1
a_2	b_1	c_2
a_1	b_2	c_1

Table 1.1: Some example of table

Functional dependencies (FDs) are used to constraint relations. Let’s say we have a relation r and a functional dependency $f : X \to Y$, r satisfies f only if each set of tuples with equal X values has equal Y values [13]. Table 1.1 shows some example table, it satisfies the functional dependency $f : AB \to C$, because when the values of A and B are determined, the value of C can also be uniquely determined.

A	B	C
a_1	b_1	c_1
a_1	b_2	c_1
a_1	b_2	c_1
a_2	b_1	c_2

Table 1.2: Some example of table (sorted by A and B)

A set of functional dependencies are usually called a cover. Given a cover, we may have different but equivalent representations. For example $\{A \rightarrow B, A \rightarrow C\}$ can be also represented as $\{A \rightarrow BC\}$.

There are many scenarios where the size of functional dependencies is very important.

1) Data validation. Functional dependencies are used as integrity constraints to validate if the updates of a database are meaningful. If any update operation leads some violation of any functional dependency, then the update should be rejected, or at least some warning should be issued. Therefore, database management system must verify after every update whether the resulting database still satisfies all the functional dependencies that have been specified. To implement such a relational database system which supports functional dependencies, a simple idea could be to iterate each functional dependency f and read related attributes of a relation after updating this relation. Obviously, the smaller the cover is, the less time it takes for the validation.

2) Data mining. For many datasets, the functional dependencies are unknown and almost unable to be discovered manually. Many algorithms trying to discover potential functional dependencies from a given dataset [1, 7, 11, 17, 23, 24]. But the output of these algorithms can be large. It is therefore important to represent the output as concisely as possible.

3) Data normalization. Some data normalization algorithms may use different notions of covers [4, 8]. So it is also nice to know what is the difference between these different kinds of covers.

These use cases strongly motivate the question: What do the different notions of covers achieve? To be more specific, the question can be divided into two parts: 1) How do the output sizes of the different cover algorithms compare against each other? 2) How do the times to compute the different covers compare against each other?

So far, the literature has not brought forward any experimental study focusing on these questions about different covers. Therefore, the main purpose of this article is trying to go deeper into the algorithms of different kinds of ”minimum” covers and experimentally analysis the ratio between the time consumption of these algorithms and the reduction of the size of attributes. In Chapter 2., we will show some literature review of the notions of different covers. In Chapter 3., we will introduce the algorithms to compute different covers. In Chapter 4., we will show the experiment result and experimental comparison to quantify the trade-off.

Chapter 2

Literature Review

In this chapter, we will introduce the formal definitions in relational database model including relations, functional dependencies, closures and covers. Then we'll talk about the notions of different kinds of covers from the previous literature.

2.1 Relational Database Model

2.1.1 Relations

The term relation was introduced by Codd [5] in 1970. He gave a mathematical sense to describe the Large Shared Data Banks as a Relational Model of Data.

Definition 2.1.1 *Given sets of attributes $S_1, S_2, \cdots, S_n$, a* ***relation*** *r on these attribute sets is a set of n-tuples where the $i-th$ element $\in S_i$.*

2.1.2 Functional Dependencies

The functional dependency can be seen as a special type of relationship between two sets of attributes under a given relation [4].

Definition 2.1.2 *Let's say X and Y are two sets of attributes chosen from a universe U in a given database D, a* ***functional dependency*** *$f : X \to Y$ is such a constraint between X and Y in a relation r of the database D. r satisfies $f : X \to Y$ if and only if there are no such two entities e_1 and e_2 in r where the X values are equal but the Y values are not equal.*

2.1.3 Closure

Definition 2.1.3 *For a given set of functional dependencies F, the* ***closure*** *of F is written F^+ which contains all the functional dependencies which can be influenced by F.*

The closure can be calculated based on Armstrong's axioms.

Theorem 2.1.4 (reflexivity) $Y \subseteq X \Rightarrow X \to Y \in F^+$.

Theorem 2.1.5 (projectivity) $X \to Y \in F^+ \wedge X \to Z \in F^+ \Rightarrow X \to YZ \in F^+$.

Theorem 2.1.6 (transitivity) $X \to Y \in F^+ \wedge Y \to Z \in F^+ \Rightarrow X \to Z \in F^+$.

2.1.4 Covers

Definition 2.1.7 *For a given set of functional dependencies F, we say G is a* ***cover*** *of F if and only if $G^+ = F^+$. And we also say two sets of functional dependencies F and G are* ***equivalent****, written $F \equiv G$, if $F^+ = G^+$.*

2.2 Different Types of Covers

2.2.1 Non-redundant Covers

Definition 2.2.1 (Maier [12]) *We say a cover G is* ***non-redundant*** *if there is no such functional dependency $f \in G$ where $(G - \{f\}) \equiv G$.*

2.2.2 Canonical Covers

Definition 2.2.2 (Paredaens [19]) *We say G is a* ***canonical*** *cover if G is* ***non-redundant*** *and for each functional dependency $f : X \to Y \in G$:*

- *the size of the left side $|Y| = 1$.*
- *there is no such X' where $X' \subset X$ and $X' \to Y \in G^+$.*

2.2.3 Mini Covers

Definition 2.2.3 (Peng & Xiao [20]) *We say G is a* ***mini*** *cover if:*

- *for each functional dependency $f : X \to Y \in G$, the size of the right side $|Y| = 1$.*
- *with the first constraint, G has the fewest number of functional dependencies.*
- *with the first two constrains, G has the fewest number of attributes (repetitively counted).*

2.2.4 Minimum Covers

Definition 2.2.4 (Maier [12]) *We say G is a* ***minimum*** *cover if there is no such H where $H \equiv G$ and $|H| < |G|$.*

2.2.5 L-Minimum Covers

Definition 2.2.5 (Maier [12]) *We say G is a **L-minimum** cover if:*

- *G is **minimum**.*
- *for each functional dependency $f : X \to Y \in G$, there is no such $X^{'}$ where $X^{'} \subset X$ and $X^{'} \to Y \in F^{'}$.*

2.2.6 LR-Minimum Covers

Definition 2.2.6 (Maier [12]) *We say G is a **LR-minimum** cover if:*

- *G is **L-minimum**.*
- *for each functional dependency $f : X \to Y \in G$, there is no such $Y^{'} \subset Y$ where $(G - \{f\} + \{X \to Y^{'}\}) \equiv G$.*

2.2.7 Optimal Covers

Definition 2.2.7 (Maier [12]) *We say G is an **optimal** cover if there no such H where $H \equiv G$ and H has a fewer number of attributes (repetitively counted).*

2.3 Relationships between Different Types of Covers

Canonical, L-minimum, LR-minimum and optimal covers are the covers with only necessary attributes in the left sides. Canonical, LR-minimum and optimal covers are the covers with only necessary attributes in the right sides. Figure 2.1 shows the relationships between different definitions of covers. In the next section, we'll start to talk about the algorithms to compute all these kinds of covers. Computing optimal cover should be the most difficult task. And in the next section, we'll find that Non-redundant, Minimum, L-Minimum, LR-Minimum and Canonical covers can be calculated in non-exponential time. Mini and Optimal cover problems are NP-Complete.

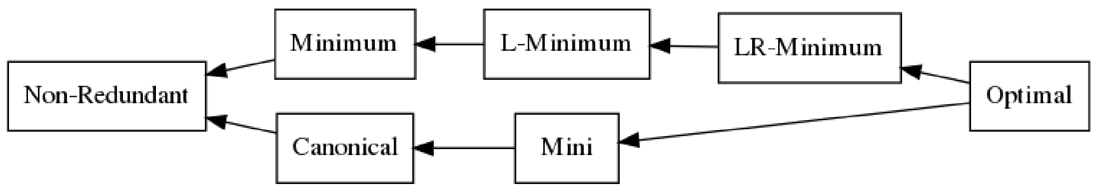


Figure 2.1: Relationships between different types of covers

2.4 Related Works

Maiver [12] introduced the definitions of most different optimum covers (minimum, L-minimum, LR-minimum and optimal) mentioned in this paper. He also introduced n square-time algorithm to find the minimum cover. Beeri & Bernstein [4] introduced a linear-time algorithm to deal with the depend and membership problem. They also gave a procedure to remove redundant attributes from the left side of each functional dependency. Peng & Xiao [20] introduced a new concept called Mini Cover and indicated that the computation of Mini Cover can be equivalent to the computation of Minimum Boolean Expression. They also indicated that an Optimal Cover can be calculated via an OPTIMIZE algorithm from a Mini Cover.

2.5 Summary

In this chapter, we provided some literature review about the basic definitions in the relational database model including relations, functional dependencies, closures and covers. And we also introduced the different notions of covers and the relationships between these covers.

Chapter 3

Algorithms

In this chapter, we'll first talk about some basic operations for computing covers including dependent set computation, membership determination, attributes equivalence determination and functional dependencies equivalence determination. Then, based on these basic operations, we'll introduce how to compute non-redundant cover, canonical cover, minimum cover, l-minimum cover and lr-minimum cover. After that, we'll move on the computation of the mini cover. It can be transformed (Delobel-Casey Transform) as a minimum boolean expression problem. We'll talk about how to use the Quine-McCluskey method to find candidate expressions and how to transform that problem again into a 0-1 integer programming model. Finally, based on the result of the mini cover, we will show the algorithm to compute optimal cover.

3.1 Dependent Set

Before introducing the algorithms of different kinds of optimum covers, we first talk about how to find the dependent set from a given set of attributes X under a given set of functional dependencies F. Algorithm 1. shows how to compute $\{y \mid X \to \{y\} \in F^+\}$ from Beeri & Bernstein [4].

Where U is the universe of attributes, F is the given set of functional dependencies and X is the given set of attributes. In this algorithm, we will first count the number of left side attributes in each functional dependency $count[i]$ and use a variable $attrlist[x]$ to indicate the indexes of functional dependencies where x is in the left side. The following is a BFS-like approach. If the left side attributes of a functional dependency f can be fully determined from X, we will add it's right side attributes into the queue and iterate this process.

Since each functional dependency will be consumed only once, if we regard the length of each functional dependency as a constant, the time complexity should be $O(|F|)$.

```
Algorithm 1: depend(U, F, X)
for F_i : X → Y ∈ F do
    count[i] = |X|;
    for x ∈ X do
        attrlist[x] = attrlist[x] ∪ {i}
    end
end
D = {x | x ∈ X};
Q = a queue initialized with X;
while Q is not empty do
    pop x from Q;
    for i ∈ attrlist[x] do
        count[i] = count[i] - 1;
        if count[i] = 0 then
            f : X → Y = F_i
            for y ∈ Y do
                D = D ∪ {y};
                push y into Q;
            end
        end
    end
end
return D;
```

3.2 Membership Determination

Based on the DEPEND algorithm, we can easily determine if $X \to Y \in F^+$. Algorithm 2. shows the process how to determine if a given functional dependency $X \to Y$ belongs to the closure of a given functional dependency set F. Since the membership algorithm only takes extra $O(|Y|)$ time to check if $Y \subseteq depend(U, F, X)$, the time complexity of membership algorithm should be $O(|F|)$ as same as the DEPEND algorithm.

Algorithm 2: membership(U, F, X, Y)

Y' = depend(U, F, X);
for $y \in Y$ **do**
  if $y \notin Y'$ **then**
    return *false*;
  end
end
return *true*;

3.3 Attributes Equivalence Determination

Given two sets of attributes X and Y, to determine if $X \equiv Y$ under a given set of functional dependency is equivalent to check if $X \to Y \in F^+$ and $Y \to X \in F^+$. Algorithm 3. shows this process. And the time complexity of determining attributes equivalence should be, as same as the time complexity of membership algorithm, $O(|F|)$.

Algorithm 3: attr_eq(U, F, X, Y)

---

if *membership(U, F, X, Y)* $\wedge$ *membership(U, F, Y, X)* **then**
  return *true*
else
  return *false*
end

3.4 Functional Dependencies Equivalence Determination

Given two sets of functional dependencies F and G, to determine if $F \equiv G$, we can iterate each $f \in F$ to check if $f \in F^+$ and each $g \in G$ to check if $g \in G^+$. After these two iterations, we can determine if $F^+ \subseteq G^+$ and $G^+ \subseteq F^+$, which is equivalent to $F^+ = G^+$. Algorithm 3.

shows this process. Since we need to run $|F|$ times membership algorithm among G and $|G|$ times membership algorithm among F, the overall time complexity should be $O(|F| \times |G|)$.

Algorithm 4: fd_eq(U, F, G)

```
for f : X → Y ∈ F do
    if ¬ membership(U, G, X, Y) then
        return False;
    end
end
for g : X → Y ∈ G do
    if ¬ membership(U, F, X, Y) then
        return False;
    end
end
return true;
```

3.5 Non-Redundant Cover

Given a set of functional dependencies F, to compute a non-redundant cover G where $G \equiv F$ and for each $H \subset G$, $H \not\equiv F$, we can iterate each functional dependency $f \in F$, if $f \in (F - \{f\})^+$, then f should be removed from the result G. Algorithm 5. shows this process. It takes membership algorithm for $|F|$ times; so the time complexity should be $O(|F|^2)$.

Algorithm 5: non_redundant(U, F)

```
G = ∅;
for f : X → Y ∈ F do
    X' = X;
    if ¬ membership(U, F − {f}, X, Y) then
        G = G ∪ {f};
    end
end
return G;
```

3.6 Canonical Cover

Algorithms 6. shows how we compute a canonical cover. To compute a canonical cover of a given set of functional dependencies F, first we should split the right side of each funtional dependency to make sure the size of right side is single and convert it to a non-redundant cover. Then we can use the algorithm introduced by Beeri & Bernstein [4] to remove left side extraneous attributes. Every time we can apply a linear-time membership algorithm to check if $(X - B) \rightarrow A \in F^+$. If so, we replace X with $X - B$. Since we may run membership algorithm at most $|F|$ times. The time complexity of canonical algorithm is $O(|F|^2)$.

Algorithm 6: canonical(U, F)

```
G = ∅;
for f : X → Y ∈ F do
    for y ∈ Y do
        G = G ∪ {X → {y}};
    end
end
G = non_redundant(G);
H = ∅;
for g : X → Y ∈ G do
    X' = X;
    for x ∈ X do
        if membership(U, F, X' − {x}, Y) then
            X' = X' − {x};
        end
        H = H ∪ {X' → Y};
    end
end
return H;
```

3.7 Minimum Cover

In Maier 1980 [12], the author introduced an $O(|F|^2)$ algorithm to compute the minimum cover. Algorithm 7. shows the process of how we compute a minimum cover under a given universe U and a functional dependency set F. The main idea is that we first convert the functional dependency set F to a non-redundant cover G. And then, we iterate all pairs of functional dependency $X_i \rightarrow Y_i \in G$ and $X_j \rightarrow Y_j \in G$, if we find such pair of functional dependencies where $X_i \leftrightarrow X_j$ and $X_i \rightarrow Y_j \in G^+$, we can replace the two functional dependencies with a new one: $X_j \rightarrow Y_iY_j$. Since we may iterate all pairs of functional dependencies, the time complexity of the minimum cover algorithm is $O(|F|^2)$.

Algorithm 7: minimum(U, F)

```
G = non_redundant(F);
for X → Y ∈ G do
    D[X] = depend(U, G, X);
end
for X_i → Y_i ∈ G do
    for X_j → Y_j ∈ G do
        M[i][j] = X_j ⊆ X_i;
    end
end
for X_i → Y_i ∈ G do
    D' = depend(U, G, X_i);
    for X_j → Y_i ∈ G do
        if i ≠ j ∧ M[i][j] ∧ M[j][i] ∧ X_j ⊆ D' then
            G = G − X_i → Y_i;
            G = G − X_j → Y_j;
            G = G ∪ X_j → Y_iY_j;
        end
    end
end
```

3.8 L-Minimum Cover

The computation of L-minimum cover is based on the computation of minimum cover. By using the same algorithm [4] mentioned in **3.6 canonical** to remove extraneous left side attributes, we can convert a minimum cover to an L-minimum cover. Algorithm 8. shows the process of L-minimum cover algorithm. It first converts the given functional dependency F to a minimum cover G. And then every time it applies a linear-time membership algorithm to check if $(X - B) \rightarrow A \in F^+$. If so, replace X with $X - B$. The time complexity of computing L-minimum cover should be as same as the time complexity of the minimum cover algorithm, $O(|F|^2)$.

Algorithm 8: l_minimum(U, F)

G = minimum(U, F);
H = ∅;
for $g : X \rightarrow Y \in G$ **do**
  $X^{'}$ = X;
  for $x \in X$ **do**
    if *membership*$(U, F, X^{'} - \{x\}, Y)$ **then**
      $X^{'}$ = $X^{'} - \{x\}$;
    end
    H = $H \cup \{X^{'} \rightarrow Y\}$;
  end
end
return H;

3.9 LR-Minimum Cover

The computation of LR-minimum cover is based on the computation of L-minimum cover. Algorithm 9. shows the process of computing LR-minimum cover. For a functional dependency $f : X \rightarrow Y \in F$ if there is $Y^{'} \subset Y$ where $f \in (F - \{f\} + \{X \rightarrow Y^{'}\})^+$, then we should replace Y with $Y^{'}$. The time complexity of this algorithm is also $O(|F|^2)$, as same as the previous two algorithms.

Algorithm 9: lr_minimum(U, F)

G = lminimum(U, F);
H = ∅;
for $f : X \to Y \in G$ **do**
  $Y^{'} = Y$;
  for $y \in Y$ **do**
    $G^{'} = G - \{f\} + \{X \to (Y^{'} - \{y\})\}$;
    if *membership*($U, G^{'}, X, Y$) **then**
      $Y^{'} = Y^{'} - \{y\}$;
    end
  end
  $H = H \cup \{X \to Y^{'}\}$;
end
return H;

3.10 Mini Cover

3.10.1 Delobel-Casey Transform

The concept of mini cover was introduced by Peng & Xiao [20]. And the authors introduced an idea that the computation of mini cover is equivalent to the minimum boolean expression problem.

By using the first Delobel-Casey transform [6], a functional dependency $f : X_1X_2 \cdots X_m \to Y_1Y_2 \cdots Y_n$ can be transformed to a boolean expression $X_1X_2 \cdots X_m\overline{Y_1} + \cdots + X_1X_2 \cdots X_m\overline{Y_m}$. And a set of functional dependencies F can be transformed to the boolean sum of the transforms of each functional dependency.

For example, the Delobel-Casey transform of $F_1 : \{AB \to C, AB \to D, C \to D\}$ should be $b_1 : AB\overline{C} + AB\overline{D} + C\overline{D}$. Since $AB\overline{D} = ABC\overline{D} + AB\overline{C}\overline{D}$, $ABC\overline{D}$ can be covered by $C\overline{D}$ and $AB\overline{C}\overline{D}$ can be covered by $AB\overline{C}$. $AB\overline{D}$ is redundant in the boolean expression b_1. So the minimum boolean expression of b_1 should be $b_2 = AB\overline{C} + C\overline{D}$. And b_2 can be transformed reversely to $F_2 : \{AB \to C, AB \to D, C \to D\}$.

Minimum boolean expression problem aims to find the equivalent boolean expression with the fewest number of symbols (repeated).

3.10.2 Quine-McCluskey Method

By converting a given set of functional dependencies F, we get an equivalent minimum boolean expression problem for computing mini cover. Quine-McCluskey method [16] is a traditional algorithm to deal with minimum boolean expression problem.

	A	B	C	D	F
0	0	0	0	0	0
1	0	0	0	1	0
2	0	0	1	0	1
3	0	0	1	1	0
4	0	1	0	0	0
5	0	1	0	1	0
6	0	1	1	0	1
7	0	1	1	1	0
8	1	0	0	0	0
9	1	0	0	1	0
10	1	0	1	0	1
11	1	0	1	1	0
12	1	1	0	0	1
13	1	1	0	1	1
14	1	1	1	0	1
15	1	1	1	1	0

Table 3.1: The table of function F_1

Let's take the example in the previous section. We get a boolean expression $b_1 : AB\overline{C} + AB\overline{D} + C\overline{D}$ from the given functional dependency set $F_1 : \{AB \to C, AB \to D, C \to D\}$. First, we write the function as a table iterating all the possible values of attributes and the related desired output. Table 3.1 shows the table of the function F_1. Then, we'll try to iterate pairs of the terms and try to combine the pairs of terms m_i and m_j where m_i and m_j has only one different column (0 or 1). Table 3.2 shows how we combine the pairs of terms and find prime implicants. Then, we will try to find the combination of implicants (with a minimum total number of symbols) to cover all the original terms in Table 3.1. As shown in Table 3.3, the best solution should be choosing $m(2, 6, 10, 14)$ and $m(12, 13)$. So the equivalent minimum boolean expression should be $b_2 = AB\overline{C} + C\overline{D}$.

3.10.3 Integer Programming Problem

The remaining problem in the previous section is how we can find the best combination of implicants with the minimum number of symbols. One simple solution could be using a $O(2^m)$ searching approach (where m is the number of prime implicants generated after Quine-McCluskey method). But here, we have another solution. This problem can be easily converted to an equivalent 01-Integer programming problem.

Let's say we have a matrix M where $C_{i,j} \in 0, 1$ indicates whether the $i-th$ prime implicant contains the $j - th$ original term. And $S_i \in Z^+$ indicates the number of symbols contained in

Number of 1s	Original terms	Size 2 implicants	Size 4 implicants
1	m(2) : 0010	m(2, 6) : 0x10 m(2, 10) : x010	m(2, 6, 10, 14) : xx10
2	m(6) : 0110 m(10) : 1010 m(12) : 1100	m(6, 14) : x110 m(10, 14) : 1x10 m(12, 13) : 110x m(12, 14) : 11x0	
3	m(13) : 1101 m(14) : 1110		

Table 3.2: Find prime implicants

	2	6	10	12	13	14
m(2,6,10,14)	√	√	√			√
m(12, 13)				√	√	
m(12, 14)				√		√

Table 3.3: Prime implicant chart

the $i-th$ prime implicant. Z_i indicates whether we use the $i-th$ prime implicant in the result or not. Then we have:

$$Z_i \in 0, 1 \tag{3.1}$$

$$\forall j, \sum_i Z_i * M_{i,j} \geq 0 \tag{3.2}$$

$$\min_Z \sum_i Z_i * S_i \tag{3.3}$$

So we converted the last step of Quine-McCluskey method into a pure 0-1 integer programming problem. Although 0-1 integer programming problem is still NP-Complete, it is helpful to make it descriptive. And in real engineering, we may use some common integer programming solver library to have better performance. Since the first step, we may need to iterate an exponential-level number of terms and the final integer programming solving may also take exponential time. The overall time complexity of computing mini cover is still exponential level.

3.11 Optimal Cover

Now, let's go into the hardest problem, optimal cover. As introduced in Peng & Xiao [20], with a given mini cover, we can get an equivalent optimal cover by applying the same MINIMIZE approach used in section 3.7. So here we can use a very similar approach to compute optimal cover. Algorithm 10. shows the process of how we compute an optimal cover from a given functional dependency set F under a given universe U. Since the step computing mini cover is NP-Complete, the overall time complexity of computing optimal cover is still NP-Complete.

Algorithm 10: optimal(U, F)

```
G = mini(U, F);
for X → Y ∈ G do
    D[X] = depend(U, G, X);
end
for X_i → Y_i ∈ G do
    for X_j → Y_j ∈ G do
        M[i][j] = X_j ⊆ X_i;
    end
end
for X_i → Y_i ∈ G do
    D' = depend(U, G, X_i);
    for X_j → Y_i ∈ G do
        if i ≠ j ∧ M[i][j] ∧ M[j][i] ∧ X_j ⊆ D' then
            G = G − X_i → Y_i;
            G = G − X_j → Y_j;
            G = G ∪ X_j → Y_iY_j;
        end
    end
end
```

3.12 Implementation

We implemented all the algorithms mentioned in this chapter using C++ (around 1700 lines of code) and the project is fully open-sourced on Github [22]. We used the Standard Template Library (STL) [21] to construct basic data structures, Lohmann's JSON library [10] to deal with the format of our datasets and GNU Linear Programming Kit (GLPK) [14] to solve some integer programming problem for computing optimal covers. We also used Google's GTest [3] during the development process to set up enough test cases validating the correctness of the implementation.

Figure 3.1 shows the project structure of our implementation. We use CMake [15] as the build tool chain. Src folder contains the most source code, ‘fdc.h‘ provides the definitions of data structures and functions. ‘io.cpp‘ contains the IO related function implementations. ‘algorithm.cpp‘ contains the most implementations of cover algorithms. The Quine-McCluskey method and 0-1 integer programming parts are separated into ‘qmc.cpp‘.

Test folder contains test code based on the GTest [3] framework. In algorithm subfolder, we have unit test cases to validate the correctness of each cover algorithm. In io subfolder, we validate the correctness of input-output related functions. And in dataset folder, we have some test case to run the experiment for Chapter 4.

Also, the code is well documented. We use Doxygen [9] to generate documents automatically. The full source code can be found via Github [22] or in the Appendix of this paper.

```
/home/aguang/Repositories/fdc/
▶ build/
▶ dataset/
▶ docs/
▶ lib/
▼ src/
  ▼ fdc/
      algorithm.cpp
      CMakeLists.txt
      fdc.h
      io.cpp
      qmc.cpp
    CMakeLists.txt
    main.cpp
▼ test/
  ▼ algorithm/
      canonical.cpp
      depend.cpp
      equal.cpp
      is_direct.cpp
      l-minimum.cpp
      lr-minimum.cpp
      membership.cpp
      mini.cpp
      minimum.cpp
      optimal.cpp
      qmc.cpp
      redundant.cpp
  ▼ dataset/
      analyse.cpp
  ▼ io/
      json.cpp
      to_str.cpp
    CMakeLists.txt
    main.cpp
    README.md
▶ thesis/
  CMakeLists.txt
  Doxyfile
  README.md
```

Figure 3.1: Project structure of FDC library

3.13 Summary

So in this chapter, we introduced some basic algorithms to find dependent set, validate membership, attributes equivalence and functional dependencies equivalence. Based on these basic approaches, we introduced how to compute non-redundant cover, canonical cover, minimum cover, l-minimum cover and lr-minimum cover. And then we talked about the computation of mini cover. It can be converted to an equivalent minimum boolean expression problem using Delobel-Casey transform. To solve the minimum boolean expression problem, we introduced the Quine-McCluskey method and 0-1 integer programming model. Finally, based on the computation of the mini cover, we introduced the optimal cover algorithm.

Chapter 4

Experiments

In this chapter, we'll show some experimental results based on our c++ implementation and 11 datasets (mostly from real-world). We'll first talk about how we set up the experiment and introduce the characteristics of the datasets. The experiment is divied into two parts: for 3 small datasets, we successfully run all the algorithms; for other 9 big datasets, we only run the other algorithms except mini cover and optimal cover in a limited time. Section 4.2 and 4.3 will show the details of these experimental results. And section 4.4 will show more trade-off between cover size and time consumption for each algorithm on the full dataset.

To better understand our experiment result, we'll show the results of some example on a tiny dataset abalone in section 4.5. Then in section 4.6, we'll analysis these experimental results and try to give some answer to1 our research question.

4.1 Experiment Setup

To evaluate the performance of different optimum cover algorithms, we used 12 different datasets for testing. Table 4.1 shows the characteristics of the 12 different datasets. Fd_reduced is the only synthetic dataset. Other datasets are from real-world, obtained from the UCI Machine Learning Repository [2]. Most of these datasets were also used in some experiment of functional dependency discovery [18].

And our experiment runs on the same Linux laptop (Ubuntu 18.04.1, x86_64, Intel i7-7700 HQ, 2.80GHz).

Dataset	Attributes(Columns)	Rows	FDs	Attributes(Repetitively counted)
abalone	9	4177	54	371
letter	17	20000	61	786
adult	15	48842	68	451
lineitem	16	6001215	901	8755
china_weather	18	262920	2955	26763
fd_reduced	30	250000	3573	100272
hepatitis	20	155	5372	39926
uniprot	30	233	5794	20732
horse	28	368	86583	241536
diabetic	30	1151	97341	1080319
plista	63	1000	111360	527093
flight	109	1000	268262	5823354

Table 4.1: Summary characteristics of various datasets

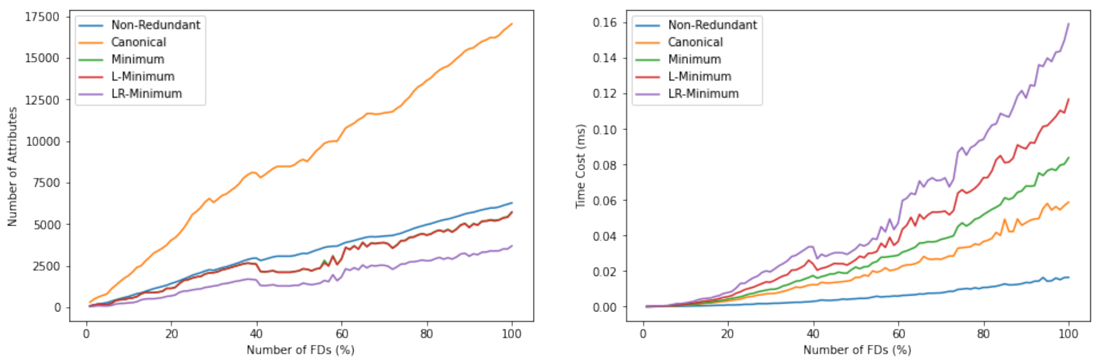
4.2 Experiment for Non-exponential Algorithms

In this section, we only test the non-exponential time complexity algorithms (non-redundant, canonical, minimum, L-minimum and LR-minimum algorithms). For each dataset, we first sorted the functional dependencies by ascending order. Then we tested 100 times for the algorithms. For each time, we only used the first k (k is from 1 to 100) percentage of the functional dependencies as the input. And we recorded the time consumption and the number of attributes in the resulting cover for each algorithm.

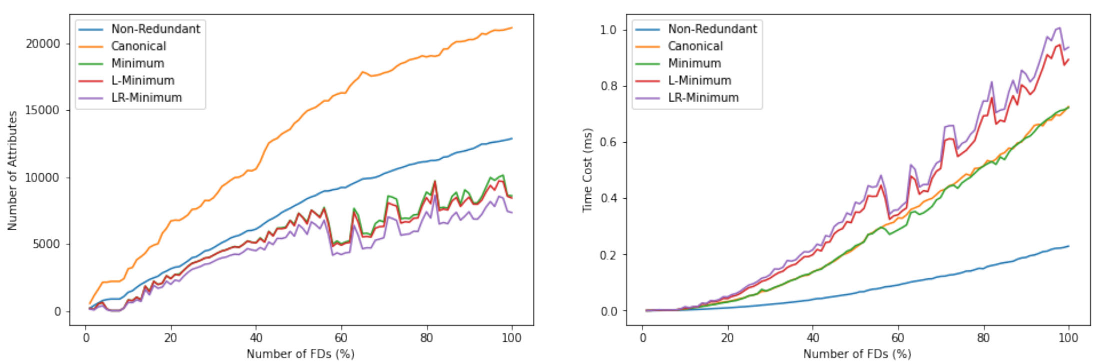
The line charts below show the experiment result for this section. Each pair of line charts indicates the result from one dataset. Both x-axises indicate the percentage of the dataset we used as the input. For the left diagrams, the y-axis indicates the number of attributes (repetitively counted) in the resulting cover for each algorithm. For the right diagrams, the y-axis indicates the time consumption for each algorithm.

Table 4.2-4.3 shows the time consumption and the result cover size for each algorithm and each dataset when using full functional dependencies as the input.

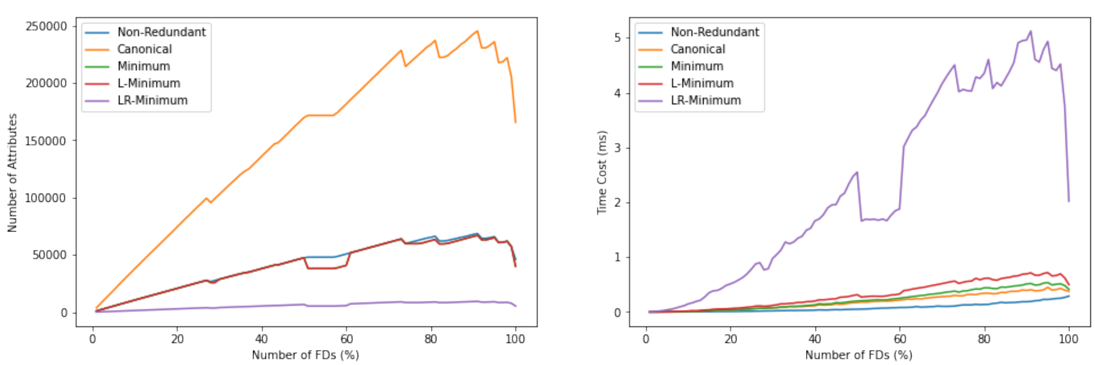
Performance on lineitem.json with 16 attributes and 901 FDs



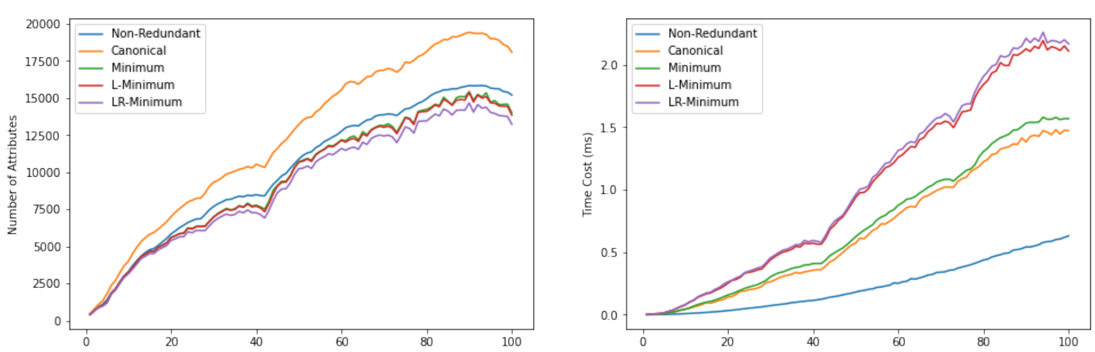
Performance on china_weather.json with 18 attributes and 2955 FDs



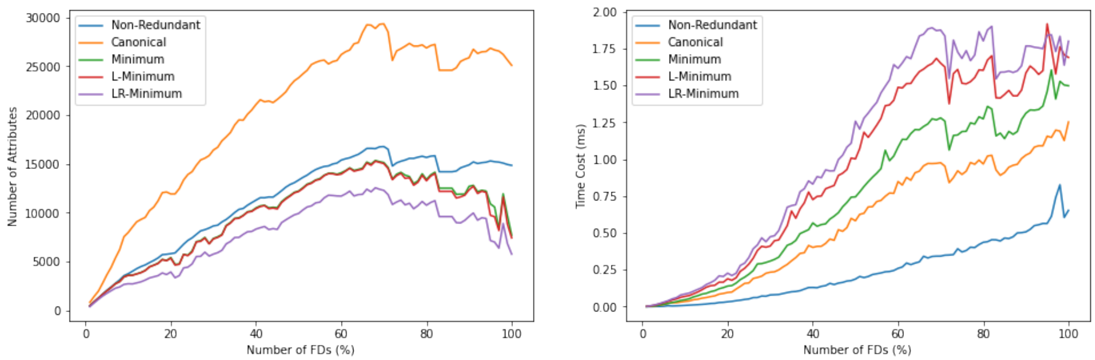
Performance on fd_reduced.json with 30 attributes and 3573 FDs



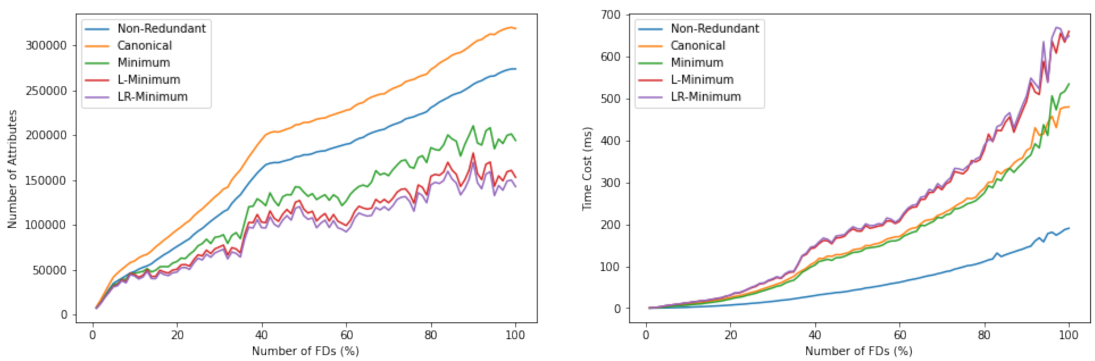
Performance on hepatitis.json with 20 attributes and 5372 FDs



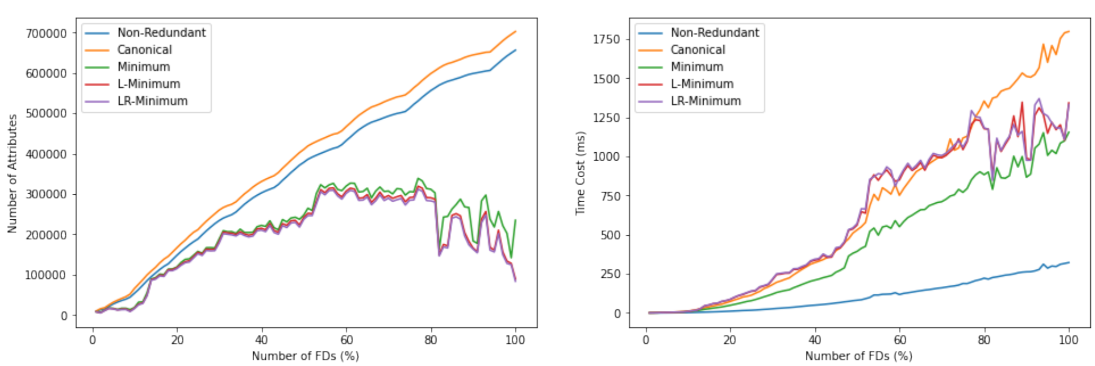
Performance on uniprot_512001r_30c.json with 30 attributes and 5794 FDs



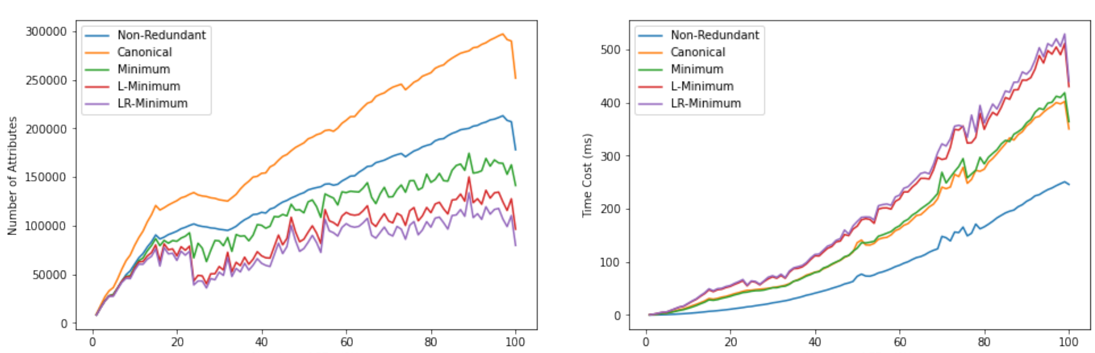
Performance on horse.json with 28 attributes and 86583 FDs

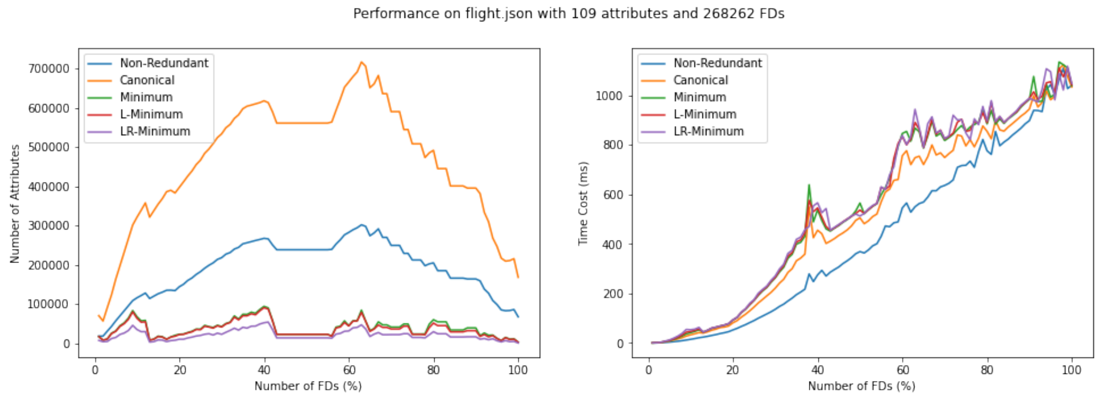


Performance on diabetic.json with 30 attributes and 97341 FDs



Performance on plista.json with 63 attributes and 111360 FDs





Dataset	Algorithm	Time consumption (sec)	Cover size
lineitem	Non-Redundant	0.017	6284
	Canonical	0.059	17049
	Minimum	0.084	5737
	L-Minimum	0.117	5709
	LR-Minimum	0.159	3700
china_weather	Non-Redundant	0.229	12868
	Canonical	0.726	21142
	Minimum	0.722	8602
	L-Minimum	0.893	8435
	LR-Minimum	0.937	7349
fd_reduced	Non-Redundant	0.290	46042
	Canonical	0.376	165594
	Minimum	0.414	39917
	L-Minimum	0.499	39917
	LR-Minimum	0.219	5340
hepatitis	Non-Redundant	0.630	15206
	Canonical	1.470	18093
	Minimum	1.567	14008
	L-Minimum	2.108	13844
	LR-Minimum	2.165	13239
uniprot	Non-Redundant	0.653	14848
	Canonical	1.252	25080
	Minimum	1.498	7701
	L-Minimum	1.690	7424
	LR-Minimum	1.800	5764
horse	Non-Redundant	190.415	273647
	Canonical	479.815	318631
	Minimum	534.537	193863
	L-Minimum	660.160	152868
	LR-Minimum	648.983	142482

Table 4.2: Time consumption and cover size of non-exponential algorithms

Dataset	Algorithm	Time consumption (sec)	Cover size
diabetic	Non-Redundant	321.835	656457
	Canonical	1798.100	702696
	Minimum	1154.360	234914
	L-Minimum	1341.980	89483
	LR-Minimum	1331.660	83712
plista	Non-Redundant	245.758	178077
	Canonical	350.197	251700
	Minimum	363.911	141314
	L-Minimum	429.861	96389
	LR-Minimum	439.981	79671
flight	Non-Redundant	1040.280	67090
	Canonical	1033.250	167867
	Minimum	1038.800	2563
	L-Minimum	1042.440	2541
	LR-Minimum	1047.610	436

Table 4.3: Time consumption and cover size of con-exponential algorithms

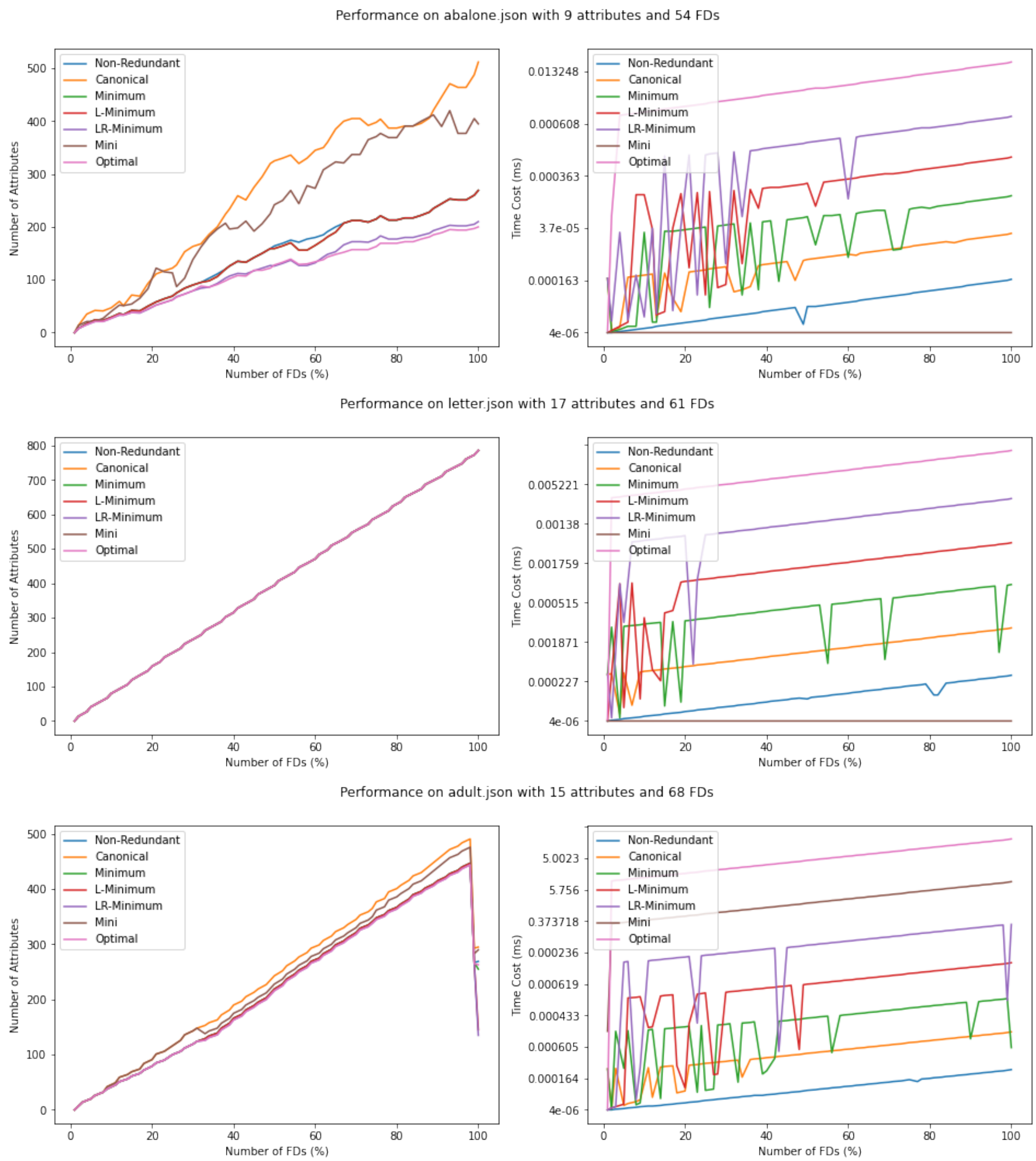
4.3 Experiment for All Algorithms

Since the computation of mini covers and optimal covers is exponential time complexity, we cannot run the mini cover algorithm and the optimal cover algorithm for big datasets in a limited time. So in this section, we tested all the algorithms on three data sets where the size is limited.

And the approach is as same as the previous one. For each dataset, we first sorted the functional dependencies by ascending order. Then we repeated the test for 100 times. For each time, we only used the first k (k from 1 to 100) percentage of functional dependencies as the input and recorded the time consumption and the number of attributes (repetitively counted) in the result cover for each algorithm.

The digrams shown below indicate the experiment result for this section. As same as the previous diagrams, both x-axises represent the percentage of functional dependencies we used for one specific test. The y-axis in left diagrams represents the number of attributes in the result cover. And the y-axis in the right diagrams represents the time consumption.

Table 4.4 shows the time consumption and the result cover size for each algorithm and each dataset when using full data set as the input.


Performance on abalone.json with 9 attributes and 54 FDs
Non-Redundant
Canonical
Minimum
L-Minimum
LR-Minimum
Mini
Optimal
Number of Attributes
Number of FDs (%)
Time Cost (ms)
0.013248
0.000608
0.000363
3.7e-05
0.000163
4e-06
Performance on letter.json with 17 attributes and 61 FDs
Number of Attributes
Number of FDs (%)
Time Cost (ms)
0.005221
0.00138
0.001759
0.000515
0.001871
0.000227
4e-06
Performance on adult.json with 15 attributes and 68 FDs
Number of Attributes
Number of FDs (%)
Time Cost (ms)
5.0023
5.756
0.373718
0.000236
0.000619
0.000433
0.000605
0.000164
4e-06

Dataset	Algorithm	Time consumption (sec)	Cover size
abalone	Non-Redundant	0.001	269
	Canonical	0.001	512
	Minimum	0.001	269
	L-Minimum	0.001	269
	LR-Minimum	0.001	210
	Mini	0.016	395
	Optimal	0.016	209
letter	Non-Redundant	0.001	786
	Canonical	0.003	786
	Minimum	0.001	786
	L-Minimum	0.004	786
	LR-Minimum	0.004	786
	Mini	0.022	786
	Optimal	0.023	786
adult	Non-Redundant	0.001	269
	Canonical	0.001	295
	Minimum	0.001	269
	L-Minimum	0.001	269
	LR-Minimum	0.001	267
	Mini	40.560	290
	Optimal	40.115	167

Table 4.4: Time consumption and cover size of all algorithms

4.4 Trade-off between Cover Size and Time Consumption

To better understand the experiment result, for each dataset and each pairs of algorithms we compute the observed trade-off between the result cover size and the time consumption. Table 4.5 and Table 4.6 illustrate the time consumption ratio between each pairs of algorithms for each dataset. Table 4.7 and Table 4.8 indicate the result cover size ratio between each pairs of algorithms for each dataset.

And Table 4.9 and Table 4.10 show the overall ratio between each pairs of algorithms for each dataset. In terms of over ratio, for a pair of algorithms A and B, the ratio is calculated by:

$$\frac{\text{B's cover size} \times \text{B's time consumption}}{\text{A's cover size} \times \text{A's time consumption}}$$

And Table 4.11 shows the reduction size and time ratio for each algorithm on each full dataset ($\frac{\text{The number of attributes in result cover} - \text{The number of attributes in input cover}}{\text{Running time}}$).

Dataset	Algorithm	NR	C	Min	L-Min	LR-Min
lineitem	NR	1.000	3.471	4.941	6.882	9.353
	Can	0.288	1.000	1.424	1.983	2.695
	Min	0.202	0.702	1.000	1.393	1.893
	L-Min	0.145	0.504	0.718	1.000	1.359
	LR-Min	0.107	0.371	0.528	0.736	1.000
china_weather	NR	1.000	3.170	3.153	3.900	4.092
	Can	0.315	1.000	0.994	1.230	1.291
	Min	0.317	1.006	1.000	1.237	1.298
	L-Min	0.256	0.813	0.809	1.000	1.049
	LR-Min	0.244	0.775	0.771	0.953	1.000
fd_reduced	NR	1.000	1.297	1.428	1.721	0.755
	Can	0.771	1.000	1.101	1.327	0.582
	Min	0.700	0.908	1.000	1.205	0.529
	L-Min	0.581	0.754	0.830	1.000	0.439
	LR-Min	1.324	1.717	1.890	2.279	1.000
hepatitis	NR	1.000	2.333	2.487	3.346	3.437
	Can	0.429	1.000	1.066	1.434	1.473
	Min	0.402	0.938	1.000	1.345	1.382
	L-Min	0.299	0.697	0.743	1.000	1.027
	LR-Min	0.291	0.679	0.724	0.974	1.000
uniprot	NR	1.000	1.917	2.294	2.588	2.757
	Can	0.522	1.000	1.196	1.350	1.438
	Min	0.436	0.836	1.000	1.128	1.202
	L-Min	0.386	0.741	0.886	1.000	1.065
	LR-Min	0.363	0.696	0.832	0.939	1.000
horse	NR	1.000	2.520	2.807	3.467	3.408
	Can	0.397	1.000	1.114	1.376	1.353
	Min	0.356	0.898	1.000	1.235	1.214
	L-Min	0.288	0.727	0.810	1.000	0.983
	LR-Min	0.293	0.739	0.824	1.017	1.000
diabetic	NR	1.000	5.587	3.587	4.170	4.138
	Can	0.179	1.000	0.642	0.746	0.741
	Min	0.279	1.558	1.000	1.163	1.154
	L-Min	0.240	1.340	0.860	1.000	0.992
	LR-Min	0.242	1.350	0.867	1.008	1.000
plista	NR	1.000	1.425	1.481	1.749	1.790
	Can	0.702	1.000	1.039	1.227	1.256
	Min	0.675	0.962	1.000	1.181	1.209
	L-Min	0.572	0.815	0.847	1.000	1.024
	LR-Min	0.559	0.796	0.827	0.977	1.000
flight	NR	1.000	0.993	0.999	1.002	1.007
	Can	1.007	1.000	1.005	1.009	1.014
	Min	1.001	0.995	1.000	1.004	1.008
	L-Min	0.998	0.991	0.997	1.000	1.005
	LR-Min	0.993	0.986	0.992	0.995	1.000

Table 4.5: Time consumption ratio between paris of algorithms (non-exponential)

Dataset	Algorithm	NR	C	Min	L-Min	LR-Min	Mini	Opt
abalone	NR	1.000	1.000	1.000	1.000	1.000	16.000	16.000
	Can	1.000	1.000	1.000	1.000	1.000	16.000	16.000
	Min	1.000	1.000	1.000	1.000	1.000	16.000	16.000
	L-Min	1.000	1.000	1.000	1.000	1.000	16.000	16.000
	LR-Min	1.000	1.000	1.000	1.000	1.000	16.000	16.000
	Mini	0.062	0.062	0.062	0.062	0.062	1.000	1.000
	Opt	0.062	0.062	0.062	0.062	0.062	1.000	1.000
letter	NR	1.000	3.000	1.000	4.000	4.000	22.000	23.000
	Can	0.333	1.000	0.333	1.333	1.333	7.333	7.667
	Min	1.000	3.000	1.000	4.000	4.000	22.000	23.000
	L-Min	0.250	0.750	0.250	1.000	1.000	5.500	5.750
	LR-Min	0.250	0.750	0.250	1.000	1.000	5.500	5.750
	Mini	0.045	0.136	0.045	0.182	0.182	1.000	1.045
	Opt	0.043	0.130	0.043	0.174	0.174	0.957	1.000
adult	NR	1.000	1.000	1.000	1.000	1.000	40560.000	40115.000
	Can	1.000	1.000	1.000	1.000	1.000	40560.000	40115.000
	Min	1.000	1.000	1.000	1.000	1.000	40560.000	40115.000
	L-Min	1.000	1.000	1.000	1.000	1.000	40560.000	40115.000
	LR-Min	1.000	1.000	1.000	1.000	1.000	40560.000	40115.000
	Mini	0.000	0.000	0.000	0.000	0.000	1.000	0.989
	Opt	0.000	0.000	0.000	0.000	0.000	1.011	1.000

Table 4.6: Time consumption ratio between paris of algorithms (all)

Dataset	Algorithm	NR	C	Min	L-Min	LR-Min
lineitem	NR	1.000	2.713	0.913	0.908	0.589
	Can	0.369	1.000	0.337	0.335	0.217
	Min	1.095	2.972	1.000	0.995	0.645
	L-Min	1.101	2.986	1.005	1.000	0.648
	LR-Min	1.698	4.608	1.551	1.543	1.000
china_weather	NR	1.000	1.643	0.668	0.656	0.571
	Can	0.609	1.000	0.407	0.399	0.348
	Min	1.496	2.458	1.000	0.981	0.854
	L-Min	1.526	2.506	1.020	1.000	0.871
	LR-Min	1.751	2.877	1.170	1.148	1.000
fd_reduced	NR	1.000	3.597	0.867	0.867	0.116
	Can	0.278	1.000	0.241	0.241	0.032
	Min	1.153	4.148	1.000	1.000	0.134
	L-Min	1.153	4.148	1.000	1.000	0.134
	LR-Min	8.622	31.010	7.475	7.475	1.000
hepatitis	NR	1.000	1.190	0.921	0.910	0.871
	Can	0.840	1.000	0.774	0.765	0.732
	Min	1.086	1.292	1.000	0.988	0.945
	L-Min	1.098	1.307	1.012	1.000	0.956
	LR-Min	1.149	1.367	1.058	1.046	1.000
uniprot	NR	1.000	1.689	0.519	0.500	0.388
	Can	0.592	1.000	0.307	0.296	0.230
	Min	1.928	3.257	1.000	0.964	0.748
	L-Min	2.000	3.378	1.037	1.000	0.776
	LR-Min	2.576	4.351	1.336	1.288	1.000
horse	NR	1.000	1.164	0.708	0.559	0.521
	Can	0.859	1.000	0.608	0.480	0.447
	Min	1.412	1.644	1.000	0.789	0.735
	L-Min	1.790	2.084	1.268	1.000	0.932
	LR-Min	1.921	2.236	1.361	1.073	1.000
diabetic	NR	1.000	1.070	0.358	0.136	0.128
	Can	0.934	1.000	0.334	0.127	0.119
	Min	2.794	2.991	1.000	0.381	0.356
	L-Min	7.336	7.853	2.625	1.000	0.936
	LR-Min	7.842	8.394	2.806	1.069	1.000
plista	NR	1.000	1.413	0.794	0.541	0.447
	Can	0.707	1.000	0.561	0.383	0.317
	Min	1.260	1.781	1.000	0.682	0.564
	L-Min	1.847	2.611	1.466	1.000	0.827
	LR-Min	2.235	3.159	1.774	1.210	1.000
flight	NR	1.000	2.502	0.038	0.038	0.006
	Can	0.400	1.000	0.015	0.015	0.003
	Min	26.176	65.496	1.000	0.991	0.170
	L-Min	26.403	66.063	1.009	1.000	0.172
	LR-Min	153.876	385.016	5.878	5.828	1.000

Table 4.7: Cover size ratio between paris of algorithms (non-exponential)

Dataset	Algorithm	NR	C	Min	L-Min	LR-Min	Mini	Opt
abalone	NR	1.000	1.903	1.000	1.000	0.781	1.468	0.743
	Can	0.525	1.000	0.525	0.525	0.410	0.771	0.391
	Min	1.000	1.903	1.000	1.000	0.781	1.468	0.743
	L-Min	1.000	1.903	1.000	1.000	0.781	1.468	0.743
	LR-Min	1.281	2.438	1.281	1.281	1.000	1.881	0.952
	Mini	0.681	1.296	0.681	0.681	0.532	1.000	0.506
	Opt	1.345	2.560	1.345	1.345	1.050	1.975	1.000
letter	NR	1.000	1.000	1.000	1.000	1.000	1.000	1.000
	Can	1.000	1.000	1.000	1.000	1.000	1.000	1.000
	Min	1.000	1.000	1.000	1.000	1.000	1.000	1.000
	L-Min	1.000	1.000	1.000	1.000	1.000	1.000	1.000
	LR-Min	1.000	1.000	1.000	1.000	1.000	1.000	1.000
	Mini	1.000	1.000	1.000	1.000	1.000	1.000	1.000
	Opt	1.000	1.000	1.000	1.000	1.000	1.000	1.000
adult	NR	1.000	1.097	0.948	0.539	0.502	1.078	0.483
	Can	0.912	1.000	0.864	0.492	0.458	0.983	0.441
	Min	1.055	1.157	1.000	0.569	0.529	1.137	0.510
	L-Min	1.855	2.034	1.759	1.000	0.931	2.000	0.897
	LR-Min	1.993	2.185	1.889	1.074	1.000	2.148	0.963
	Mini	0.928	1.017	0.879	0.500	0.466	1.000	0.448
	Opt	2.069	2.269	1.962	1.115	1.038	2.231	1.000

Table 4.8: Time consumption ratio between paris of algorithms (all)

Dataset	Algorithm	NR	C	Min	L-Min	LR-Min
lineitem	NR	1.000	9.416	4.511	6.253	5.507
	Can	0.106	1.000	0.479	0.664	0.585
	Min	0.222	2.087	1.000	1.386	1.221
	L-Min	0.160	1.506	0.721	1.000	0.881
	LR-Min	0.182	1.710	0.819	1.135	1.000
china_weather	NR	1.000	5.209	2.108	2.556	2.337
	Can	0.192	1.000	0.405	0.491	0.449
	Min	0.474	2.471	1.000	1.213	1.109
	L-Min	0.391	2.038	0.825	1.000	0.914
	LR-Min	0.428	2.229	0.902	1.094	1.000
fd_reduced	NR	1.000	4.663	1.238	1.492	0.088
	Can	0.214	1.000	0.265	0.320	0.019
	Min	0.808	3.768	1.000	1.205	0.071
	L-Min	0.670	3.126	0.830	1.000	0.059
	LR-Min	11.417	53.241	14.131	17.032	1.000
hepatitis	NR	1.000	2.333	2.487	3.346	3.437
	Can	0.429	1.000	1.066	1.434	1.473
	Min	0.402	0.938	1.000	1.345	1.382
	L-Min	0.299	0.697	0.743	1.000	1.027
	LR-Min	0.291	0.679	0.724	0.974	1.000
uniprot	NR	1.000	3.941	1.290	1.673	1.334
	Can	0.254	1.000	0.327	0.424	0.338
	Min	0.775	3.055	1.000	1.297	1.034
	L-Min	0.598	2.356	0.771	1.000	0.797
	LR-Min	0.750	2.954	0.967	1.254	1.000
horse	NR	1.000	2.945	1.996	1.944	1.781
	Can	0.340	1.000	0.678	0.660	0.605
	Min	0.501	1.475	1.000	0.974	0.892
	L-Min	0.514	1.515	1.027	1.000	0.916
	LR-Min	0.561	1.653	1.121	1.091	1.000
diabetic	NR	1.000	5.981	1.284	0.568	0.528
	Can	0.167	1.000	0.215	0.095	0.088
	Min	0.779	4.659	1.000	0.443	0.411
	L-Min	1.759	10.522	2.258	1.000	0.928
	LR-Min	1.895	11.334	2.433	1.077	1.000
plista	NR	1.000	2.014	1.175	0.947	0.801
	Can	0.497	1.000	0.583	0.470	0.398
	Min	0.851	1.714	1.000	0.806	0.682
	L-Min	1.056	2.127	1.241	1.000	0.846
	LR-Min	1.248	2.515	1.467	1.182	1.000
flight	NR	1.000	2.485	0.038	0.038	0.007
	Can	0.402	1.000	0.015	0.015	0.003
	Min	26.214	65.146	1.000	0.995	0.172
	L-Min	26.348	65.481	1.005	1.000	0.172
	LR-Min	152.799	379.738	5.829	5.799	1.000

Table 4.9: Overall ratio between paris of algorithms (non-exponential)

Dataset	Algorithm	NR	C	Min	L-Min	LR-Min	Mini	Opt
abalone	NR	1.000	1.903	1.000	1.000	0.781	23.494	11.896
	Can	0.525	1.000	0.525	0.525	0.410	12.344	6.250
	Min	1.000	1.903	1.000	1.000	0.781	23.494	11.896
	L-Min	1.000	1.903	1.000	1.000	0.781	23.494	11.896
	LR-Min	1.281	2.438	1.281	1.281	1.000	30.095	15.238
	Mini	0.043	0.081	0.043	0.043	0.033	1.000	0.506
	Opt	0.084	0.160	0.084	0.084	0.066	1.975	1.000
letter	NR	1.000	3.000	1.000	4.000	4.000	22.000	23.000
	Can	0.333	1.000	0.333	1.333	1.333	7.333	7.667
	Min	1.000	3.000	1.000	4.000	4.000	22.000	23.000
	L-Min	0.250	0.750	0.250	1.000	1.000	5.500	5.750
	LR-Min	0.250	0.750	0.250	1.000	1.000	5.500	5.750
	Mini	0.045	0.136	0.045	0.182	0.182	1.000	1.045
	Opt	0.043	0.130	0.043	0.174	0.174	0.957	1.000
adult	NR	1.000	1.097	0.948	0.539	0.502	43726.394	19386.431
	Can	0.912	1.000	0.864	0.492	0.458	39872.542	17677.797
	Min	1.055	1.157	1.000	0.569	0.529	46127.059	20450.784
	L-Min	1.855	2.034	1.759	1.000	0.931	81120.000	35965.172
	LR-Min	1.993	2.185	1.889	1.074	1.000	87128.889	38629.259
	Mini	0.000	0.000	0.000	0.000	0.000	1.000	0.443
	Opt	0.000	0.000	0.000	0.000	0.000	2.256	1.000

Table 4.10: Overall ratio between paris of algorithms (all)

Dataset	NR	C	Min	L-Min	LR-Min	Mini	Opt
abalone	102000.0	-141000.0	102000.0	102000.0	161000.0	-1500.0	10125.0
letter	0.0	0.0	0.0	0.0	0.0	0.0	0.0
adult	182000.0	156000.0	182000.0	182000.0	184000.0	3.969	7.079
lineitem	145352.941	-140576.271	35928.571	26034.188	31792.452		
china_weather	60676.855	7742.424	25153.739	20524.076	20719.316		
fd_reduced	187000.0	-173728.723	145785.024	120951.903	433479.452		
hepatitis	24952.38	8729.931	10796.426	8103.415	8169.515		
uniprot	9010.719	-3472.843	8698.931	7874.556	8315.555		
horse	-168.636	-160.676	89.185	134.312	152.629		
diabetic	1317.016	210.012	732.358	738.338	748.394		
plista	1420.161	786.394	1060.091	1001.961	1016.912		
flight	5533.379	5473.493	5603.379	5583.835	5558.287		

Table 4.11: Reduction of size per-second of each algorithm

4.5 Qualitative Analysis

To better understand the experiment result, here we use the abalone dataset as an example to show the results of different covers. This could give more sense about what is the difference these covers try to achieve?

4.5.1 Original Cover of Abalone Dataset

The abalone dataset contains 9 attributes and 54 functional dependencies. There are attributes repetitively counted. The below shows the original cover.

```
(0, 1, 2, 3, 5, 7) -> (4, 6)
(0, 1, 2, 3, 5, 8) -> (4, 6, 7)
(0, 1, 2, 3, 6, 7) -> (4)
(0, 1, 2, 6, 7) -> (5)
(0, 1, 3, 4, 7) -> (8)
(0, 1, 3, 5, 6) -> (4, 8)
(0, 1, 3, 5, 7) -> (8)
(0, 1, 4, 5) -> (2, 3, 6, 7)
(0, 1, 5, 6) -> (2, 7)
(0, 1, 6, 7, 8) -> (2, 3, 4, 5)
(0, 2, 3, 5, 6) -> (4, 7, 8)
(0, 2, 3, 6, 7) -> (8)
(0, 3, 4, 6) -> (2)
(0, 3, 4, 7, 8) -> (1)
(0, 4, 6, 8) -> (1, 2, 3, 5, 7)
(0, 5, 6, 8) -> (2, 7)
(1, 2, 3, 4) -> (0)
(1, 2, 3, 4, 8) -> (5, 6, 7)
(1, 2, 3, 5, 7, 8) -> (4, 6)
(1, 2, 3, 6, 8) -> (0, 4, 5, 7)
(1, 2, 4, 5) -> (3, 7)
(1, 2, 4, 7) -> (0, 3, 5, 6, 8)
(1, 2, 5, 7, 8) -> (0)
(1, 2, 6, 7, 8) -> (0, 3, 4, 5)
(1, 3, 4, 5) -> (0, 2, 6, 7)
(1, 3, 4, 6) -> (0, 2, 5, 7, 8)
(1, 3, 4, 8) -> (0)
(1, 3, 5, 6) -> (2)
(1, 3, 5, 7, 8) -> (0)
```

```
(1, 4, 5) -> (8)
(1, 4, 5, 6) -> (3, 7)
(1, 4, 6, 8) -> (0, 2, 3, 5, 7)
(1, 4, 7, 8) -> (0)
(1, 5, 6, 7) -> (0, 2)
(1, 5, 6, 8) -> (0, 2, 3, 4, 7)
(2, 3, 4, 5) -> (1, 7, 8)
(2, 3, 4, 7) -> (0)
(2, 3, 4, 7, 8) -> (1, 5, 6)
(2, 3, 5, 6) -> (1)
(2, 3, 5, 7, 8) -> (0)
(2, 3, 6, 7, 8) -> (0)
(2, 4, 5) -> (0, 6)
(2, 4, 5, 8) -> (1, 3, 7)
(2, 4, 6) -> (0)
(2, 4, 6, 8) -> (1, 3, 5, 7)
(2, 5, 6, 7) -> (0)
(2, 5, 6, 8) -> (0, 7)
(3, 4, 5, 6) -> (1, 7, 8)
(3, 5, 6, 7) -> (0, 1, 2, 4, 8)
(3, 5, 6, 8) -> (0, 1, 2, 4, 7)
(4, 5, 6) -> (0, 2)
(4, 5, 6, 8) -> (1, 3, 7)
(4, 5, 7) -> (0, 1, 2, 3, 6, 8)
(4, 6, 7) -> (0, 1, 2, 3, 5, 8)
```

4.5.2 Non-Redundant Cover of Abalone Dataset

By removing redundant functional dependencies, we got a non-redundant cover where there are 269 attributes and 40 functional dependencies. We can see that the non-redundant cover algorithm only removes redundant functional dependencies. All the functional dependencies in the result must originally exist in the input cover.

```
(0, 1, 2, 3, 5, 8) -> (4, 6, 7)
(0, 1, 2, 6, 7) -> (5)
(0, 1, 3, 4, 7) -> (8)
(0, 1, 3, 5, 7) -> (8)
(0, 1, 4, 5) -> (2, 3, 6, 7)
(0, 1, 5, 6) -> (2, 7)
(0, 1, 6, 7, 8) -> (2, 3, 4, 5)
```

```
(0, 2, 3, 6, 7) -> (8)
(0, 3, 4, 6) -> (2)
(0, 3, 4, 7, 8) -> (1)
(0, 4, 6, 8) -> (1, 2, 3, 5, 7)
(0, 5, 6, 8) -> (2, 7)
(1, 2, 3, 4) -> (0)
(1, 2, 3, 4, 8) -> (5, 6, 7)
(1, 2, 3, 6, 8) -> (0, 4, 5, 7)
(1, 2, 4, 7) -> (0, 3, 5, 6, 8)
(1, 2, 5, 7, 8) -> (0)
(1, 2, 6, 7, 8) -> (0, 3, 4, 5)
(1, 3, 4, 6) -> (0, 2, 5, 7, 8)
(1, 3, 4, 8) -> (0)
(1, 3, 5, 6) -> (2)
(1, 3, 5, 7, 8) -> (0)
(1, 4, 5) -> (8)
(1, 4, 6, 8) -> (0, 2, 3, 5, 7)
(1, 4, 7, 8) -> (0)
(1, 5, 6, 7) -> (0, 2)
(1, 5, 6, 8) -> (0, 2, 3, 4, 7)
(2, 3, 4, 7) -> (0)
(2, 3, 5, 6) -> (1)
(2, 3, 5, 7, 8) -> (0)
(2, 3, 6, 7, 8) -> (0)
(2, 4, 5) -> (0, 6)
(2, 4, 6) -> (0)
(2, 5, 6, 7) -> (0)
(2, 5, 6, 8) -> (0, 7)
(3, 5, 6, 7) -> (0, 1, 2, 4, 8)
(3, 5, 6, 8) -> (0, 1, 2, 4, 7)
(4, 5, 6) -> (0, 2)
(4, 5, 7) -> (0, 1, 2, 3, 6, 8)
(4, 6, 7) -> (0, 1, 2, 3, 5, 8)
```

4.5.3 Canonical Cover of Abalone Dataset

To compute canonical cover, we first split the functional dependencies where the right side has multiple attributes into multiple functional dependencies when the right side is single. Then we remove the redundant covers. We can see the right side of canonical cover is some kind of simplified. But considering the cover size, the result has 512 attributes and 100 functional dependencies. It is definitely not good to be the smallest cover.

```
(0, 1, 2, 3, 5, 8) -> (4)
(0, 1, 2, 3, 5, 8) -> (6)
(0, 1, 2, 3, 5, 8) -> (7)
(0, 1, 2, 6, 7) -> (5)
(0, 1, 3, 4, 7) -> (8)
(0, 1, 3, 5, 7) -> (8)
(0, 1, 4, 5) -> (2)
(0, 1, 4, 5) -> (3)
(0, 1, 4, 5) -> (6)
(0, 1, 4, 5) -> (7)
(0, 1, 5, 6) -> (2)
(0, 1, 5, 6) -> (7)
(0, 1, 6, 7, 8) -> (2)
(0, 1, 6, 7, 8) -> (3)
(0, 1, 6, 7, 8) -> (4)
(0, 1, 6, 7, 8) -> (5)
(0, 2, 3, 6, 7) -> (8)
(0, 3, 4, 6) -> (2)
(0, 3, 4, 7, 8) -> (1)
(0, 4, 6, 8) -> (1)
(0, 4, 6, 8) -> (2)
(0, 4, 6, 8) -> (3)
(0, 4, 6, 8) -> (5)
(0, 4, 6, 8) -> (7)
(0, 5, 6, 8) -> (2)
(0, 5, 6, 8) -> (7)
(1, 2, 3, 4) -> (0)
(1, 2, 3, 4, 8) -> (5)
(1, 2, 3, 4, 8) -> (6)
(1, 2, 3, 4, 8) -> (7)
(1, 2, 3, 6, 8) -> (0)
(1, 2, 3, 6, 8) -> (4)
(1, 2, 3, 6, 8) -> (5)
(1, 2, 3, 6, 8) -> (7)
(1, 2, 4, 7) -> (0)
(1, 2, 4, 7) -> (3)
(1, 2, 4, 7) -> (5)
(1, 2, 4, 7) -> (6)
```

```
(1, 2, 4, 7) -> (8)
(1, 2, 5, 7, 8) -> (0)
(1, 2, 6, 7, 8) -> (0)
(1, 2, 6, 7, 8) -> (3)
(1, 2, 6, 7, 8) -> (4)
(1, 2, 6, 7, 8) -> (5)
(1, 3, 4, 6) -> (0)
(1, 3, 4, 6) -> (2)
(1, 3, 4, 6) -> (5)
(1, 3, 4, 6) -> (7)
(1, 3, 4, 6) -> (8)
(1, 3, 4, 8) -> (0)
(1, 3, 5, 6) -> (2)
(1, 3, 5, 7, 8) -> (0)
(1, 4, 5) -> (8)
(1, 4, 6, 8) -> (0)
(1, 4, 6, 8) -> (2)
(1, 4, 6, 8) -> (3)
(1, 4, 6, 8) -> (5)
(1, 4, 6, 8) -> (7)
(1, 4, 7, 8) -> (0)
(1, 5, 6, 7) -> (0)
(1, 5, 6, 7) -> (2)
(1, 5, 6, 8) -> (0)
(1, 5, 6, 8) -> (2)
(1, 5, 6, 8) -> (3)
(1, 5, 6, 8) -> (4)
(1, 5, 6, 8) -> (7)
(2, 3, 4, 7) -> (0)
(2, 3, 5, 6) -> (1)
(2, 3, 5, 7, 8) -> (0)
(2, 3, 6, 7, 8) -> (0)
(2, 4, 5) -> (0)
(2, 4, 5) -> (6)
(2, 4, 6) -> (0)
(2, 5, 6, 7) -> (0)
(2, 5, 6, 8) -> (0)
(2, 5, 6, 8) -> (7)
(3, 5, 6, 7) -> (0)
```

```
(3, 5, 6, 7) -> (1)
(3, 5, 6, 7) -> (2)
(3, 5, 6, 7) -> (4)
(3, 5, 6, 7) -> (8)
(3, 5, 6, 8) -> (0)
(3, 5, 6, 8) -> (1)
(3, 5, 6, 8) -> (2)
(3, 5, 6, 8) -> (4)
(3, 5, 6, 8) -> (7)
(4, 5, 6) -> (0)
(4, 5, 6) -> (2)
(4, 5, 7) -> (0)
(4, 5, 7) -> (1)
(4, 5, 7) -> (2)
(4, 5, 7) -> (3)
(4, 5, 7) -> (6)
(4, 5, 7) -> (8)
(4, 6, 7) -> (0)
(4, 6, 7) -> (1)
(4, 6, 7) -> (2)
(4, 6, 7) -> (3)
(4, 6, 7) -> (5)
(4, 6, 7) -> (8)
```

4.5.4 Minimum Cover of Abalone Dataset

Minimum cover tries to achive the smallest number of functional dependencies. Here is the result of minimum cover from the abalone dataset. It has 269 attributes and only 40 functional dependencies.

```
(0, 1, 2, 3, 5, 8) -> (4, 6, 7)
(0, 1, 2, 6, 7) -> (5)
(0, 1, 3, 4, 7) -> (8)
(0, 1, 3, 5, 7) -> (8)
(0, 1, 4, 5) -> (2, 3, 6, 7)
(0, 1, 5, 6) -> (2, 7)
(0, 1, 6, 7, 8) -> (2, 3, 4, 5)
(0, 2, 3, 6, 7) -> (8)
(0, 3, 4, 6) -> (2)
(0, 3, 4, 7, 8) -> (1)
```

```
(0, 4, 6, 8) -> (1, 2, 3, 5, 7)
(0, 5, 6, 8) -> (2, 7)
(1, 2, 3, 4) -> (0)
(1, 2, 3, 4, 8) -> (5, 6, 7)
(1, 2, 3, 6, 8) -> (0, 4, 5, 7)
(1, 2, 4, 7) -> (0, 3, 5, 6, 8)
(1, 2, 5, 7, 8) -> (0)
(1, 2, 6, 7, 8) -> (0, 3, 4, 5)
(1, 3, 4, 6) -> (0, 2, 5, 7, 8)
(1, 3, 4, 8) -> (0)
(1, 3, 5, 6) -> (2)
(1, 3, 5, 7, 8) -> (0)
(1, 4, 5) -> (8)
(1, 4, 6, 8) -> (0, 2, 3, 5, 7)
(1, 4, 7, 8) -> (0)
(1, 5, 6, 7) -> (0, 2)
(1, 5, 6, 8) -> (0, 2, 3, 4, 7)
(2, 3, 4, 7) -> (0)
(2, 3, 5, 6) -> (1)
(2, 3, 5, 7, 8) -> (0)
(2, 3, 6, 7, 8) -> (0)
(2, 4, 5) -> (0, 6)
(2, 4, 6) -> (0)
(2, 5, 6, 7) -> (0)
(2, 5, 6, 8) -> (0, 7)
(3, 5, 6, 7) -> (0, 1, 2, 4, 8)
(3, 5, 6, 8) -> (0, 1, 2, 4, 7)
(4, 5, 6) -> (0, 2)
(4, 5, 7) -> (0, 1, 2, 3, 6, 8)
(4, 6, 7) -> (0, 1, 2, 3, 5, 8)
```

4.5.5 L-Minimum Cover of Abalone Dataset

The L-Minimum cover is based on minimum cover. It has the smallest number of functional dependencies and tries to reduce the left side attributes. But for our example, the left side cannot be reduced any more. So the result has 269 attributes and 40 functional dependencies as same as the previous one.

```
(0, 1, 2, 3, 5, 8) -> (4, 6, 7)
(0, 1, 2, 6, 7) -> (5)
```

```
(0, 1, 3, 4, 7) -> (8)
(0, 1, 3, 5, 7) -> (8)
(0, 1, 4, 5) -> (2, 3, 6, 7)
(0, 1, 5, 6) -> (2, 7)
(0, 1, 6, 7, 8) -> (2, 3, 4, 5)
(0, 2, 3, 6, 7) -> (8)
(0, 3, 4, 6) -> (2)
(0, 3, 4, 7, 8) -> (1)
(0, 4, 6, 8) -> (1, 2, 3, 5, 7)
(0, 5, 6, 8) -> (2, 7)
(1, 2, 3, 4) -> (0)
(1, 2, 3, 4, 8) -> (5, 6, 7)
(1, 2, 3, 6, 8) -> (0, 4, 5, 7)
(1, 2, 4, 7) -> (0, 3, 5, 6, 8)
(1, 2, 5, 7, 8) -> (0)
(1, 2, 6, 7, 8) -> (0, 3, 4, 5)
(1, 3, 4, 6) -> (0, 2, 5, 7, 8)
(1, 3, 4, 8) -> (0)
(1, 3, 5, 6) -> (2)
(1, 3, 5, 7, 8) -> (0)
(1, 4, 5) -> (8)
(1, 4, 6, 8) -> (0, 2, 3, 5, 7)
(1, 4, 7, 8) -> (0)
(1, 5, 6, 7) -> (0, 2)
(1, 5, 6, 8) -> (0, 2, 3, 4, 7)
(2, 3, 4, 7) -> (0)
(2, 3, 5, 6) -> (1)
(2, 3, 5, 7, 8) -> (0)
(2, 3, 6, 7, 8) -> (0)
(2, 4, 5) -> (0, 6)
(2, 4, 6) -> (0)
(2, 5, 6, 7) -> (0)
(2, 5, 6, 8) -> (0, 7)
(3, 5, 6, 7) -> (0, 1, 2, 4, 8)
(3, 5, 6, 8) -> (0, 1, 2, 4, 7)
(4, 5, 6) -> (0, 2)
(4, 5, 7) -> (0, 1, 2, 3, 6, 8)
(4, 6, 7) -> (0, 1, 2, 3, 5, 8)
```

4.5.6 LR-Minimum Cover of Abalone Dataset

The LR-Minimum cover is based on L-minimum cover. It has the minimum number of functional dependencies and also tries to reduce the right side attributes based on the result of L-minimum cover. For our example, LR-minimum cover has 210 attributes (much smaller than L-minimum) and 40 functional dependencies.

```
(0, 1, 2, 3, 5, 8) -> (6)
(0, 1, 2, 6, 7) -> (5)
(0, 1, 3, 4, 7) -> (8)
(0, 1, 3, 5, 7) -> (8)
(0, 1, 4, 5) -> (6)
(0, 1, 5, 6) -> (7)
(0, 1, 6, 7, 8) -> (4)
(0, 2, 3, 6, 7) -> (8)
(0, 3, 4, 6) -> (2)
(0, 3, 4, 7, 8) -> (1)
(0, 4, 6, 8) -> (5)
(0, 5, 6, 8) -> (2)
(1, 2, 3, 4) -> (0)
(1, 2, 3, 4, 8) -> (6)
(1, 2, 3, 6, 8) -> (4)
(1, 2, 4, 7) -> (3)
(1, 2, 5, 7, 8) -> (0)
(1, 2, 6, 7, 8) -> (3)
(1, 3, 4, 6) -> (5)
(1, 3, 4, 8) -> (0)
(1, 3, 5, 6) -> (2)
(1, 3, 5, 7, 8) -> (0)
(1, 4, 5) -> (8)
(1, 4, 6, 8) -> (2)
(1, 4, 7, 8) -> (0)
(1, 5, 6, 7) -> (2)
(1, 5, 6, 8) -> (2)
(2, 3, 4, 7) -> (0)
(2, 3, 5, 6) -> (1)
(2, 3, 5, 7, 8) -> (0)
(2, 3, 6, 7, 8) -> (0)
(2, 4, 5) -> (6)
(2, 4, 6) -> (0)
```

```
(2, 5, 6, 7) -> (0)
(2, 5, 6, 8) -> (7)
(3, 5, 6, 7) -> (1)
(3, 5, 6, 8) -> (1)
(4, 5, 6) -> (2)
(4, 5, 7) -> (1)
(4, 6, 7) -> (2, 1)
```

4.5.7 Mini Cover of ABalone Dataset

Mini cover has the same constraint of single right side attribute. Under this constraint, it tries to achieve the smallest number of attributes. For our example, mini cover has 75 functional dependencies and 395 attributes, which is much smaller than the canonical cover.

```
(0, 1, 2, 3, 5, 7) -> (4)
(0, 1, 2, 3, 5, 8) -> (4)
(0, 1, 2, 3, 5, 8) -> (6)
(0, 1, 2, 6, 7) -> (5)
(0, 1, 3, 4, 7) -> (8)
(0, 1, 3, 5, 6) -> (4)
(0, 1, 3, 5, 7) -> (8)
(0, 1, 4, 5) -> (3)
(0, 1, 4, 5) -> (6)
(0, 1, 4, 5) -> (7)
(0, 1, 5, 6) -> (7)
(0, 1, 6, 7, 8) -> (2)
(0, 1, 6, 7, 8) -> (4)
(0, 2, 3, 5, 6) -> (7)
(0, 2, 3, 6, 7) -> (8)
(0, 3, 4, 6) -> (2)
(0, 3, 4, 7, 8) -> (1)
(0, 4, 6, 8) -> (5)
(0, 4, 6, 8) -> (7)
(0, 5, 6, 8) -> (2)
(1, 2, 3, 4) -> (0)
(1, 2, 3, 4, 8) -> (5)
(1, 2, 3, 4, 8) -> (6)
(1, 2, 3, 5, 7, 8) -> (4)
(1, 2, 3, 6, 8) -> (4)
(1, 2, 3, 6, 8) -> (5)
```

```
(1, 2, 4, 7) -> (3)
(1, 2, 4, 7) -> (6)
(1, 2, 4, 7) -> (8)
(1, 2, 5, 7, 8) -> (0)
(1, 2, 6, 7, 8) -> (0)
(1, 2, 6, 7, 8) -> (3)
(1, 2, 6, 7, 8) -> (4)
(1, 3, 4, 5) -> (2)
(1, 3, 4, 6) -> (0)
(1, 3, 4, 6) -> (7)
(1, 3, 4, 6) -> (8)
(1, 3, 4, 8) -> (0)
(1, 3, 5, 6) -> (2)
(1, 3, 5, 7, 8) -> (0)
(1, 4, 5) -> (8)
(1, 4, 6, 8) -> (0)
(1, 4, 6, 8) -> (3)
(1, 4, 7, 8) -> (0)
(1, 5, 6, 7) -> (2)
(1, 5, 6, 8) -> (0)
(1, 5, 6, 8) -> (2)
(1, 5, 6, 8) -> (7)
(2, 3, 4, 5) -> (1)
(2, 3, 4, 5) -> (7)
(2, 3, 4, 7) -> (0)
(2, 3, 4, 7, 8) -> (1)
(2, 3, 4, 7, 8) -> (5)
(2, 3, 5, 6) -> (1)
(2, 3, 5, 7, 8) -> (0)
(2, 3, 6, 7, 8) -> (0)
(2, 4, 5) -> (0)
(2, 4, 5) -> (6)
(2, 4, 6) -> (0)
(2, 4, 6, 8) -> (1)
(2, 5, 6, 7) -> (0)
(2, 5, 6, 8) -> (7)
(3, 5, 6, 7) -> (1)
(3, 5, 6, 7) -> (4)
(3, 5, 6, 8) -> (0)
```

```
(3, 5, 6, 8) -> (1)
(3, 5, 6, 8) -> (2)
(4, 5, 6) -> (0)
(4, 5, 6) -> (2)
(4, 5, 6, 8) -> (1)
(4, 5, 6, 8) -> (3)
(4, 5, 7) -> (1)
(4, 5, 7) -> (6)
(4, 6, 7) -> (2)
(4, 6, 7) -> (5)
```

4.5.8 Optimal Cover of Abalone Dataset

And finally, optimal cover tries to achieve the possible smallest number of attributes. Compared with LR-minimum cover, the optimal cover of our example has 209 attributes and also 40 functional dependencies. And the below shows the result of optimal cover.

```
(0, 1, 2, 3, 5, 8) -> (4)
(0, 1, 2, 6, 7) -> (5)
(0, 1, 3, 4, 7) -> (8)
(0, 1, 3, 5, 7) -> (8)
(0, 1, 4, 5) -> (6)
(0, 1, 5, 6) -> (7)
(0, 1, 6, 7, 8) -> (4)
(0, 2, 3, 6, 7) -> (8)
(0, 3, 4, 6) -> (2)
(0, 3, 4, 7, 8) -> (1)
(0, 4, 6, 8) -> (7)
(0, 5, 6, 8) -> (2)
(1, 2, 3, 4) -> (0)
(1, 2, 3, 4, 8) -> (5)
(1, 2, 3, 6, 8) -> (4)
(1, 2, 4, 7) -> (6)
(1, 2, 5, 7, 8) -> (0)
(1, 2, 6, 7, 8) -> (0)
(1, 3, 4, 6) -> (7)
(1, 3, 4, 8) -> (0)
(1, 3, 5, 6) -> (2)
(1, 3, 5, 7, 8) -> (0)
(1, 4, 5) -> (8)
```

```
(1, 4, 6, 8) -> (3)
(1, 4, 7, 8) -> (0)
(1, 5, 6, 7) -> (2)
(1, 5, 6, 8) -> (2)
(2, 3, 4, 7) -> (0)
(2, 3, 5, 6) -> (1)
(2, 3, 5, 7, 8) -> (0)
(2, 3, 6, 7, 8) -> (0)
(2, 4, 5) -> (6)
(2, 4, 6) -> (0)
(2, 5, 6, 7) -> (0)
(2, 5, 6, 8) -> (7)
(3, 5, 6, 7) -> (1)
(3, 5, 6, 8) -> (1)
(4, 5, 6) -> (2)
(4, 5, 7) -> (1)
(4, 6, 7) -> (5)
```

4.6 Analysis

From the experiment result shown in sections 4.2, 4.3 and 4.4, we can find that canonical covers always have the largest number of attributes. Because in the canonical cover algorithm, we have to split each functional dependency with multiple attributes in the right side into multiple functional dependencies with a single attribute in the right side. Then, the number of attributes in non-redundant covers are dramatically reduced than canonical covers. And minimum covers have fewer attributes in the result than non-redundant attributes since it has the fewest number of functional dependencies. L-minimum covers have fewer attributes than minimum covers. LR-minimum covers have fewer attributes than L-minimum. The number of attributes in mini covers is between the number in canonical covers and the number in non-redundant covers since it takes some global optimal strategy but the right sides are all single attribute. And optimal covers have the fewest number of attributes for all data sets. Table 4.1 shows the overall comparison of the number of attributes for different kinds of covers.

And considering the time consumption of different cover algorithms, mini cover algorithm and optimal cover algorithm take massive time. And since their time consumption are both exponential growing. They can only be used for the small size of data sets in a limited time. The optimal cover takes one more step to do minimization, so it takes more time than mini cover calculation. For other non-exponential algorithms, LR-minimum cover algorithm takes more time than L-minimum cover. The L-minimum cover algorithm takes more time than the minimum cover algorithm. The canonical cover algorithm takes more time than the minimum cover algorithm. And non-redundant cover algorithm takes least time than all the others. Table 4.2 summarises the overall comparision about the time consumption for different kinds of covers.

Number of attributes	Cover
Big	Canonical
	Mini
	Non-redundant
	Minimum
	L-minimum
	LR-minimum
Small	Optimal

Figure 4.1: Number of Attributes in Different Covers

Time consumption	Cover
Unacceptable	Optimal
	Mini
A little big	LR-minimum
	L-Minimum
	Minimum
	Canonical
Small	Non-redundant

Figure 4.2: Time consumption of Different Covers

Based on the experiment result, optimal cover achieves the best possible cover size, but it may be too expensive to compute. Except mini cover and optimal cover, LR-minimum cover shows the biggest reduction of cover size. And considering the time consumption, LR-minimum cover may not spend much more time other algorithms.

4.7 Summary

So in this chapter, we introduced how to run the experiment and show the characteristics of the datasets that we used. Since mini cover and optimal cover computation are quite expensive, we only successfully run these two algorithms on three small datasets. But for other algorithms, we run them on all 11 datasets. Based on our analysis, we suggest lr-minimum cover as the best choice since the result cover size is good enough and the time consumption is not expensive as the exponential algorithms like mini cover algorithm and optimal cover algorithm.

Chapter 5

Conclusion

Functional dependencies are essential for database schema design and other data processing tasks. Functional dependencies can be represented in many equivalent but different ways. They can be known as covers. In this article, we reviewed many different types of optimum covers. They are non-redundant, canonical, minimum, L-minimum, LR-minimum, mini and optimal covers. We fully implemented all the cover algorithms and tested the implementations using many datasets from the real world. We evaluated their performance by the reduction of the attribute size and time consumption.

The optimal cover algorithm has the best performance on the reduction of input size and the time consumption is not acceptable for many big data sets. The mini cover algorithm also takes exponential growing time so it is not pretty suitable for big input data. Other algorithms can be extended for big data sets. Their time consumption all grows at the same level. The LR-minimum cover algorithm may have the biggest constant but also perform better on the reduction. So based on our experiment, LR-minimum seems to be more suitable for real world datasets.

So considering the output sizes of covers, optimal cover achieves the best possible size. But it might be too expensive for real-world datasets. Overall, LR-minimum cover could be the most ideal choice since the reduction of attributes is good enough and the time consumption is not much bigger than other algorithms.

Chapter 6

Future Work

To some extent, most of the algorithms described in this article are following the same way of thinking: based on what kind of rules, transform the current cover to an equivalent but smaller cover. What if we rethink the problem in another pattern? For example, we may probably try to grow a functional dependency set G. For each time, we make sure that $G^+ \subset F^+$ and try to add some new functional dependency f into G. Obviously, $f \notin G^+$. If we can find an effective way to enumerate potential f for current G and we can find an approoate heuristic function to estimate the potential final size grown from current G. Maybe we can find some more effective algorithm to solve the optimal cover algorithm.

And speaking to the performance, is it possible that we can use some parallel computing techniques to speed up our current algorithms? It may also be a good direction to go further.

References

[1] Rakesh Agarwal, Ramakrishnan Srikant, et al. Fast algorithms for mining association rules. In *Proc. of the 20th VLDB Conference*, pages 487–499, 1994.

[2] Arthur Asuncion and David Newman. Uci machine learning repository, 2007.

[3] E Barca, E Bruno, DE Bruno, and G Passarella. Gtest: a software tool for graphical assessment of empirical distributions gaussianity. *Environmental monitoring and assessment*, 188(3):138, 2016.

[4] Catriel Beeri and Philip A Bernstein. Computational problems related to the design of normal form relational schemas. *ACM Transactions on Database Systems (TODS)*, 4(1):30–59, 1979.

[5] Edgar Codd. F. 1970. a relational model of data for large shared data banks. *Communications of the Association for Computing Machinery 13 (6), 377*, 87.

[6] Claude Delobel and Richard G. Casey. Decomposition of a data base and the theory of boolean switching functions. *IBM Journal of Research and Development*, 17(5):374–386, 1973.

[7] Yka Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. Tane: An efficient algorithm for discovering functional and approximate dependencies. *The computer journal*, 42(2):100–111, 1999.

[8] Henning Koehler. Finding faithful boyce-codd normal form decompositions. In *International Conference on Algorithmic Applications in Management*, pages 102–113. Springer, 2006.

[9] Robert S Laramee. Bobs concise introduction to doxygen. Technical report, Technical report, The Visual and Interactive Computing Group, Computer , 2011.

[10] Niels Lohmann. Fdc - functional dependencies' covers, 2020.

[11] Stéphane Lopes, Jean-Marc Petit, and Lotfi Lakhal. Efficient discovery of functional dependencies and armstrong relations. In *International Conference on Extending Database Technology*, pages 350–364. Springer, 2000.

[12] David Maier. Minimum covers in relational database model. *Journal of the ACM (JACM)*, 27(4):664–674, 1980.

[13] David Maier. *The theory of relational databases*, volume 11. Computer science press Rockville, 1983.

[14] Andrew Makhorin. Glpk (gnu linear programming kit). *http://www. gnu. org/s/glpk/glpk. html*, 2008.

[15] Ken Martin and Bill Hoffman. *Mastering CMake: a cross-platform build system*. Kitware, 2010.

[16] Edward J McCluskey. Minimization of boolean functions. *The Bell System Technical Journal*, 35(6):1417–1444, 1956.

[17] Noel Novelli and Rosine Cicchetti. Fun: An efficient algorithm for mining functional and embedded dependencies. In *International Conference on Database Theory*, pages 189–203. Springer, 2001.

[18] Thorsten Papenbrock, Jens Ehrlich, Jannik Marten, Tommy Neubert, Jan-Peer Rudolph, Martin Schönberg, Jakob Zwiener, and Felix Naumann. Functional dependency discovery: An experimental evaluation of seven algorithms. *Proceedings of the VLDB Endowment*, 8(10):1082–1093, 2015.

[19] J Paredaens. *About functional dependencies in a data base structure and their coverings*. 1977.

[20] Xiaoning Peng and Zhijun Xiao. Optimal covers in the relational database model. *Acta Informatica*, 53(5):459–468, 2016.

[21] Phillip James Plauger, Meng Lee, David Musser, and Alexander A Stepanov. *C++ standard template library*. Prentice Hall PTR, 2000.

[22] Guangrui Wang. Fdc - functional dependencies' covers, 2020.

[23] Catharine Wyss, Chris Giannella, and Edward Robertson. Fastfds: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances extended abstract. In *International Conference on Data Warehousing and Knowledge Discovery*, pages 101–110. Springer, 2001.

[24] Hong Yao and Howard J Hamilton. Mining functional dependencies from data. *Data Mining and Knowledge Discovery*, 16(2):197–219, 2008.

Appendix A

Source Code of FDC Library

Listing A.1: Header file

```
#include <iostream>
#include <vector>

#ifndef __fdc_inc__
#define __fdc_inc__

#define FDC_VERSION_MAJOR @FDC_VERSION_MAJOR @
#define FDC_VERSION_MINOR @FDC_VERSION_MINOR @
#define FDC_VERSION_PATCH @FDC_VERSION_PATCH @

/**
 * @brief A cross-platform library for calculating the covers of functional
 * dependencies.
 */
namespace fdc {

/*! \brief Attribute. */
typedef int attr;

/*! \brief A set of attributes. */
typedef std::vector<attr> attrs;

/*! \brief Functional dependency. */
typedef std::pair<attrs, attrs> fd;
```

```
/*! \brief A set of functional dependencies. */
typedef std::vector<fd> fds;

/*! \brief A boolean expressions. */
typedef std::vector<char> bool_expr;

/*! \brief A collection of boolean expressions. */
typedef std::vector<bool_expr> bool_exprs;

/*! \brief Input/output functions implemented in `FDC`.
 *
 * This module contains the input and output functions implemented in `FDC`.
 *
 * @defgroup IO
 *
 * @{
 */

/*! \brief Convert an attribute `x` to a string.
 *
 * @param x: a given attribute
 */
std::string to_str(const attr &x);

/*! \brief Convert a set of attributes `X` to a string.
 *
 * @param X: a given set of attributes.
 */
std::string to_str(const attrs &X);

/*! \brief Convert a set of functional depdency `f` to a string.
 *
 * @param f: a given functional dependency.
 */
std::string to_str(const fd &f);

/*! \brief Convert a set of functional dependencies `F` to a string.
 *
 * @param F: a given set of functional dependencies.
```

```
 */

std::string to_str(const fds &F);

/*! \brief Convert a json string `input` into a set of functional
 * dependencies `F`.
 *
 * The json string `input` is expected to be under the structure below:
 *
 * {
 *   "R": Integer,
 *   fds": [
 *     {
 *       "lhs": [ Integer ],
 *       "rhs": [ Integer ]
 *     }
 *   ]
 * }
 *
 * @param input: A json string.
 * @param N: The total number of attributes for output.
 * @param F: A set of functional dependencies for output.
 */
void from_json(const std::string input, int &N, fds &F);

/*! \brief Convert a input stream `input` into a set of functional
 * dependencies `F`.
 *
 * The input stream `input` is expected to be under the structure below:
 *
 * {
 *   "R": Integer,
 *   fds": [
 *     {
 *       "lhs": [ Integer ],
 *       "rhs": [ Integer ]
 *     }
 *   ]
 * }
```

```
 *
 * @param input: A input stream.
 * @param N: The total number of attributes for output.
 * @param F: A set of functional dependencies for output.
 */
void from_json(std::istream &input, int &N, fds &F);

/*! \brief Convert a set of functional dependencies `F` to json string and
 * write it into output.
 *
 * The output stream `output` is expected to be written under the structure
 * below:
 *
 * {
 *   "R": Integer,
 *   fds": [
 *     {
 *       "lhs": [ Integer ],
 *       "rhs": [ Integer ]
 *     }
 *   ]
 * }
 *
 * @param output: An output stream.
 * @param N: The total number of attributes for output.
 * @param F: A set of functional dependencies.
 */
void to_json(std::ostream &output, const int &N, const fds &F);

/** @} */

/*! \brief The algorithms implemented in `FDC`.
 *
 * This module contains the algorithms implemented in `FDC`.
 *
 * \dot
 *   digraph algorithsm {
 *
 *     node[shape=rect];
```

```
*
*       depend[label = "Depend"]
*       is_membership[label = "Membership"]
*       equal_attrs[label = "Attributes Equivalence"]
*       equal_fds[label = "Functional Dependencies Equivalence"]
*       is_redundant[label = "Redundant Determination"]
*       non_redundant[label = "Redundant Cover"]
*       is_direct[label = "Direct Determination"]
*       minimum[label = "Minimum Cover"]
*       l_minimum[label = "L-Minimum Cover"]
*       lr_minimum[label = "LR-Minimum Cover"]
*       mini[label = "Mini Cover"]
*
*       is_membership -> depend
*       equal_attrs -> is_membership
*       equal_fds -> is_membership
*       is_redundant -> is_membership
*       non_redundant -> is_membership
*       is_direct -> non_redundant
*       minimum -> non_redundant
*       l_minimum -> minimum
*       lr_minimum -> l_minimum
*       optimal -> mini
*    }
* \enddot
*
* Brief relations between different kinds of covers:
*
* \dot
*    digraph covers {
*
*       rankdir=RL;
*       node[shape=rect];
*
*       canonical[label = "Canonical"]
*       non_redundant[label = "Non-Redundant"]
*       minimum[label = "Minimum"]
*       l_minimum[label = "L-Minimum"]
*       lr_minimum[label = "LR-Minimum"]
```

```
 *       optimal[label = "Optimal"]
 *       mini[label = "Mini"]
 *
 *       canonical -> non_redundant
 *       minimum -> non_redundant
 *       mini -> canonical
 *       l_minimum -> minimum
 *       lr_minimum -> l_minimum
 *       optimal -> lr_minimum
 *       optimal -> mini
 *     }
 * \enddot
 *
 * @defgroup algorithms
 *
 * @{
 */

/*! \brief Dependent calculation.
 *
 * Given a set of functional dependencies \f$ F \f$ and a set of attributes
 * \f$ X \f$, calculate the set of attributes \f$ \{ y \mid X \to y \in F^+ \}
 * \f$.
 *
 * Time complexity: \f$ O(|F|) \f$
 *
 * See also: Algorithm 2. A linear-time membership algorithm in
 *    [Beeri and Bernstein (1979, p.
 * 46)](https://dl.acm.org/doi/10.1145/320064.320066)
 *
 * @param N: The total number of attributes.
 * @param F: A set of functional dependencies.
 * @param X: A set of attributes.
 * @param D: The output of the depend result.
 */
void depend(const int N, const fds &F, const std::vector<int> X, bool D[]);

/*! \brief Membership determination.
 *
```

```
 * Given a set of functional dependencies \f$ F \f$ and a functional
 * dependency \f$ f \f$ , determine if \f$ f \in F^{+} \f$.
 *
 * Time complexity: \f$ O(|F|) \f$
 *
 * See also: Algorithm 2. A linear-time membership algorithm in
 *   [Beeri and Bernstein (1979, p.
 * 46)](https://dl.acm.org/doi/10.1145/320064.320066)
 *
 * @param N: The total number of attributes.
 * @param F: A set of functional dependencies.
 * @param f: A functional dependency.
 */
bool is_membership(const int N, const fds &F, const fd &f);

/*! \brief Sets of attributes equivalence determination.
 *
 * Given two sets of attributes \f$ X \f$ and \f$ Y \f$ and a set of
 * functional dependencies \f$ F \f$.
 *
 * Determine if:
 *
 *   * \f$ X \to Y \in F^+ \f$
 *   * \f$ Y \to X \in F^+ \f$
 *
 * @param N: The number of attributes.
 * @param F: A set of functional dependencies.
 * @param X: A set of attributes.
 * @param Y: A set of attributes.
 */
bool equal(const int N, const fds &F, const attrs &X, const attrs &Y);

/*! \brief Functional dependencies equivalence determination.
 *
 * Given two sets of functional dependencies.
 * Determine if \f$ F^+ = G^+ \f$.
 *
 * @param N: The number of attributes.
 * @param F: A set of functional dependencies.
```

```
 * @param G: A set of functional dependencies.
 */
bool equal(const int N, const fds &F, const fds &G);

/*! \brief Redundant determination.
 *
 * Check if a set of functional dependencies \f$ F \f$ is redundant.
 *
 * If there is a \f$ \ f \in F \f$, where \f$ (F - \{f\})^+ = F^+ \f$, then
 * we say \f$ F \f$ is redundant.
 *
 * Time complexity: \f$ O(|F|^2) \f$.
 *
 * See also: 5.2 Redundancy Tests in
 *   [Beeri and Bernstein (1979, p.
 * 47)](https://dl.acm.org/doi/10.1145/320064.320066)
 *
 * @param N: The number of attributes.
 * @param F: A set of functional dependencies.
 */
bool is_redundant(const int N, const fds &F);

/*! \brief Redundant cover calculation.
 *
 * Given a set of functional dependencies \f$ F \f$, calculate a non-redundant
 * cover \f$ G \f$ of \f$ F \f$, where:
 *
 *   * \f$ G^+ = F^+ \f$
 *   * \f$ \forall H \subset G: H^{+} \neq F^+ \f$
 *
 * Time complexity: \f$ O(|F|^2) \f$.
 *
 * See also: 5.2 Redundancy Tests in
 *   [Beeri and Bernstein (1979, p.
 * 47)](https://dl.acm.org/doi/10.1145/320064.320066)
 *
 * @param N: The number of attributes.
 * @param F: A set of functional dependencies.
 */
```

```
fds non_redundant(const int N, const fds &F);

/*! \brief Canonical determination.
 *
 * Given a set of functional dependencies \f$ F \f$, determine if \f$ F \f$
 * is canonical.
 *
 * The definition of canonical:
 *
 *   * \f$ F \f$ is non-redundant.
 *   * For every \f$ X \to Y \in F \f$, \f$ |Y| = 1\f$ and there is no such
 *     a \f$ X^{'} \subset X \f$ where \f$ X^{'} \to Y \in F \f$.
 *
 * Time complexity: \f$ O(|F|^2) \f$.
 *
 * See also: 5.2 Redundancy Tests in
 *   [Beeri and Bernstein (1979, p.
 * 47)](https://dl.acm.org/doi/10.1145/320064.320066)
 *
 * @param N: The number of attributes.
 * @param F: A set of cuntional dependencies.
 */
bool is_canonical(const int N, const fds &F);

/*! \brief Canonical calculation.
 *
 * Given a set of functional dependencies \f$ F \f$, calculate
 * \f$ G^+ = F^+ \f$ where
 *
 *   * \f$ G \f$ is non-redundant.
 *   * For every \f$ X \to Y \in G \f$, \f$ |Y| = 1 \f$ and there is no such
 *     a \f$ X^{'} \subset X \f$ where \f$ X^{'} \to Y \in G \f$.
 *
 * Time complexity: \f$ O(|F|^2) \f$.
 *
 * See also: 5.2 Redundancy Tests in
 *   [Beeri and Bernstein (1979, p.
 * 47)](https://dl.acm.org/doi/10.1145/320064.320066)
 *
```

```
 * @param N: The number of attributes.
 * @param F: A set of cuntional dependencies.
 */
fds canonical(const int N, const fds &F);

/*! \brief Direct determination.
 *
 * Given a set of functional dependencies \f$ F \f$ and a functional
 * dependency \f$ f:X \to Y \f$, determine if \f$ X \f$ directly determines
 * \f$ Y \f$.
 *
 * Time complexity: \f$ O(|F|^2) \f$.
 *
 * See also: Direct determination in
 *   [Maier(1979, p. 335)](https://dl.acm.org/doi/10.1145/800135.804425)
 *
 * @param N: The number of attributes.
 * @param F: A set of functional dependencies.
 * @param f: A functional dependency.
 */
bool is_direct(const int N, const fds &F, const fd &f);

/*! \brief Minimum cover calculation.
 *
 * Given a set of functional dependencies \f$ F \f$, calculate a minimum
 * cover \f$ G \f$, where:
 *
 *   * \f$ G^+ = F^+ \f$.
 *   * \f$ \forall H^+ = F^+, |H| \geq |G| \f$.
 *
 * Time complexity: \f$ O(|F|^2) \f$.
 *
 * See also: Theorem 3. in
 *   [Maier(1979, p. 335)](https://dl.acm.org/doi/10.1145/800135.804425)
 *
 * @param N: The number of attributes.
 * @param F: A set of functional dependencies.
 */
fds minimum(const int N, const fds &F);
```

```
/*! \brief Minimum determination.
 *
 * Given a set of functional dependencies \f$ F \f$, determine if:
 *
 *   * \f$ \forall G^+ = F^+, |G| \geq |F| \f$.
 *
 * Time complexity: \f$ O(|F|^2) \f$.
 *
 * See also: The definition of minimum in
 *   [Maier(1979, p. 331)](https://dl.acm.org/doi/10.1145/800135.804425)
 *
 * @param N: The number of attributes.
 * @param F: A set of functional dependencies.
 */
bool is_minimum(const int N, const fds &F);

/*! \brief L-minimum determination.
 *
 * Given a set of functional dependencies \f$ F \f$, determine if:
 *
 *   * \f$ \forall G^+ = F^+, |G| \geq |F| \f$.
 *   * For every \f$ X \to Y \in F \f$, there is no such
 *     a \f$ X^{'} \subset X \f$ where \f$ X^{'} \to Y \in F \f$.
 *
 * Time complexity: \f$ O(|F|^2) \f$.
 *
 * See also: The definition of L-minimum in
 *   [Maier(1979, p. 331)](https://dl.acm.org/doi/10.1145/800135.804425)
 *
 * @param N: The number of attributes.
 * @param F: A set of functional dependencies.
 */
bool is_lminimum(const int N, const fds &F);

/*! \brief L-minimum calculation.
 *
 * Given a set of functional dependencies \f$ F \f$, calculate \f$ G \f$
 * where:
```

```
 *
 *    * \f$ G^+ = F^+ \f$
 *    * \f$ \forall H^+ = G^+, |H| \geq |G| \f$.
 *    * For every \f$ X \to Y \in G \f$, there is no such
 *      a \f$ X^{'} \subset X \f$ where \f$ X^{'} \to Y \in G \f$.
 *
 * Time complexity: \f$ O(|F|^2) \f$.
 *
 * See also: Corollary 2. in
 *    [Maier(1979, p. 335)](https://dl.acm.org/doi/10.1145/800135.804425)
 *
 * @param N: The number of attributes.
 * @param F: A set of functional dependencies.
 */
fds lminimum(const int N, const fds &F);

/*! \brief LR-minimum determination.
 *
 * Given a set of functional dependencies \f$ F \f$, determine if:
 *
 *    * \f$ \forall G^+ = F^+, |G| \geq |F| \f$.
 *    * For every \f$ X \to Y \in F \f$, there is no such
 *      a \f$ X^{'} \subset X \f$ where \f$ X^{'} \to Y \in F \f$.
 *    * For every \f$ X \to Y \in F \f$, there is no such
 *      a \f$ Y^{'} \subset Y \f$ where
 *      \f$ (F - \{X \to Y\} + \{X \to Y^{'}\})^+ = F^+ \f$.
 *
 * Time complexity: \f$ O(|F|^2) \f$.
 *
 * See also: The definition of LR-minimum in
 *    [Maier(1979, p. 331)](https://dl.acm.org/doi/10.1145/800135.804425)
 *
 * @param N: The number of attributes.
 * @param F: A set of functional dependencies.
 */
bool is_lrminimum(const int N, const fds &F);

/*! \brief LR-minimum calculation.
 *
```

```
 * Given a set of functional dependencies \f$ F \f$, calculate \f$ G \f$
 * where:
 *
 *    * \f$ G^+ = F^+ \f$
 *    * \f$ \forall H^+ = F^+, |H| \geq |G| \f$.
 *    * For every \f$ X \to Y \in G \f$, there is no such
 *      a \f$ X^{'} \subset X \f$ where \f$ X^{'} \to Y \in G \f$.
 *    * For every \f$ X \to Y \in G \f$, there is no such
 *      a \f$ Y^{'} \subset Y \f$ where
 *      \f$ (G - \{X \to Y\} + \{X \to Y^{'}\})^+ = G^+ \f$.
 *
 * Time complexity: \f$ O(|F|^2) \f$.
 *
 * See also: Corollary 2. in
 *   [Maier(1979, p. 335)](https://dl.acm.org/doi/10.1145/800135.804425)
 *
 * @param N: The number of attributes.
 * @param F: A set of functional dependencies.
 */
fds lrminimum(const int N, const fds &F);

/*! \brief Quine-McCluskey algorithm.
 *
 * Compute minimum boolean expression.
 *
 * Time complexity: \f$ O(2^N) \f$.
 *
 * @param exprs: 2d-vector of chars ('0', '1', '-') indicating the boolean
 * expressions.
 */
bool_exprs qmc(bool_exprs exprs);

/*! \brief Mini determination.
 *
 * Given a set of functional dependencies \f$ F \f$, determine if:
 *
 *    * For every \f$ X \to Y \in F \f$, \f$ | Y | = 1 \f$.
 *    * \f$ F \f$ has fewest FDs.
 *    * With the previous constraint, \f$ F \f$ has fewest attributes.
```

```
 *
 * Time complexity: \f$ O(2^N) \f$.
 *
 * See also: The definition of Mini in [Peng & Xiao (2015, p.
 * 461)](https://doi.org/10.1007/s00236-015-0247-9).
 *
 * @param N: The number of attributes.
 * @param F: A set of functional dependencies.
 */
bool is_mini(const int N, const fds &F);

/*! \brief Mini calculation.
 *
 * Given a set of functional dependencies \f$ F \f$, calculate \f$ G \f$
 * where:
 *
 *   * For every \f$ X \to Y \in G \f$, \f$ | Y | = 1 \f$.
 *   * \f$ G \f$ has fewest FDs.
 *   * With the previous constraint, \f$ G \f$ has fewest attributes.
 *
 * Time complexity: \f$ O(2^N) \f$.
 *
 * See also: The definition of Mini in [Peng & Xiao (2015, p.
 * 461)](https://doi.org/10.1007/s00236-015-0247-9).
 *
 * @param N: The number of attributes.
 * @param F: A set of functional dependencies.
 */
fds mini(const int N, const fds &F);

/*! \brief Optimal determination.
 *
 * Given a set of functional dependencies \f$ F \f$, determine if:
 *
 *   * \f$ F \f$ has fewest FDs.
 *
 * Time complexity: \f$ O(2^N) \f$.
 *
 * See also: Optimize algorithm in [Peng & Xiao (2015, p.
```

```
 * 467)](https://doi.org/10.1007/s00236-015-0247-9).
 *
 * @param N: The number of attributes.
 * @param F: A set of functional dependencies.
 */
bool is_optimal(const int N, const fds &F);

/*! \brief Optimal calculate.
 *
 * Given a set of functional dependencies \f$ F \f$, calculate \f$ G \f$:
 *
 *    * \f$ G \f$ has fewest FDs.
 *
 * Time complexity: \f$ O(2^N) \f$.
 *
 * See also: Optimize algorithm in [Peng & Xiao (2015, p.
 * 467)](https://doi.org/10.1007/s00236-015-0247-9).
 *
 * @param N: The number of attributes.
 * @param F: A set of functional dependencies.
 */
fds optimal(const int N, const fds &F);

/** @} */
} // namespace fdc
#endif
```

Listing A.2: IO related functions

```
#include "fdc.h"
#include "json/json.hpp"

namespace fdc {

using namespace std;

using json = nlohmann::json;

string to_str(const attr &x) { return to_string(x); }

string to_str(const attrs &X) {
```

```
  bool first = true;

  string str = "(";

  for (auto x : X) {

    if (!first) {

      str += ", ";
    }

    str += to_str(x);

    first = false;
  }

  str += ")";

  return str;
}

string to_str(const fd &f) {

  return to_str(f.first) + " -> " + to_str(f.second);
}

string to_str(const fds &F) {

  bool first = true;

  string str = "FDS {";

  for (auto f : F) {

    str += "\n  " + to_str(f);

    first = false;
  }
```

```
  if (!first) {
    str += "\n";
  }
  str += "}";
  return str;
}

void from_json(const json input, int &N, fds &F) {
  F.clear();
  N = input["R"].get<int>();
  for (auto f : input["fds"]) {
    auto X = attrs();
    for (auto x : f["lhs"]) {
      X.push_back(attr(x.get<int>()));
    }
    auto Y = attrs();
    for (auto y : f["rhs"]) {
      Y.push_back(attr(y.get<int>()));
    }
    F.push_back(fd(X, Y));
  }
}

void from_json(const string input, int &N, fds &F) {
```

```
  from_json(json::parse(input), N, F);
}

void from_json(istream &input, int &N, fds &F) {

  from_json(json::parse(input), N, F);
}

void to_json(ostream &output, const int &N, const fds &F) {

  json j;

  j["R"] = N;

  auto fds = vector<map<string, vector<int>>>();

  for (auto &f : F) {

    auto lhs = vector<int>();

    for (attr x : f.first) {

      lhs.push_back(x);
    }

    auto rhs = vector<int>();

    for (attr y : f.second) {

      rhs.push_back(y);
    }

    auto fd = map<string, vector<int>>();

    fd["lhs"] = lhs;
    fd["rhs"] = rhs;

    fds.push_back(fd);
  }
```

```
  j["fds"] = fds;

  output << j.dump(2) << endl;
}
} // namespace fdc
```

Listing A.3: Algorithms related functions

```
#include <cstring>
#include <queue>
#include <set>
#include <vector>

#include "fdc.h"

namespace fdc {

using namespace std;

const fd FD_EMPTY = fd(attrs({}), attrs({}));

void depend(const int N, const fds &F, const attrs X, bool D[]) {

  vector<int> attrlist[N];

  int counter[F.size()];

  for (int i = 0; i < F.size(); i++) {

    const fd &f = F[i];

    counter[i] = f.first.size();

    for (const int &x : f.first) {

      attrlist[x].push_back(i);
    }
  }

  memset(D, 0x00, sizeof(bool) * N);
```

```
  queue<int> que;

  for (const int &x : X) {

    if (!D[x]) {

      D[x] = true;

      que.push(x);
    }
  }

  for (; que.size() > 0; que.pop()) {

    int x = que.front();

    for (const int &i : attrlist[x]) {

      if (--(counter[i]) == 0) {

        for (const int &y : F[i].second) {

          if (!D[y]) {

            D[y] = true;

            que.push(y);
          }
        }
      }
    }
  }
}

bool is_membership(const int N, const fds &F, const fd &f) {

  bool D[N];
```

```
  depend(N, F, f.first, D);

  for (const int &y : f.second) {

    if (!D[y]) {

      return false;
    }
  }

  return true;
}

bool equal(const int N, const fds &F, const attrs &X, const attrs &Y) {

  return is_membership(N, F, fd(X, Y)) && is_membership(N, F, fd(Y, X));
}

bool equal(const int N, const fds &F, const fds &G) {

  for (const fd &f : F) {

    if (!is_membership(N, G, f)) {

      return false;
    }
  }

  for (const fd &f : G) {

    if (!is_membership(N, F, f)) {

      return false;
    }
  }

  return true;
}
```

```
bool is_redundant(const int N, const fds &F) {

  fds G = fds(F);

  for (int i = 0; i < G.size(); i++) {

    const fd f = G[i];

    // Assigning G[i] to $\empty \to \empty$ is equivalent to removing G[i],
    // but assigning operating takes less time.
    G[i] = FD_EMPTY;

    if (is_membership(N, G, f)) {

      return true;
    }

    // Recovery G[i].
    G[i] = f;
  }

  return false;
}

fds non_redundant(const int N, const fds &F) {

  fds G = fds(F);

  for (int i = 0; i < G.size();) {

    const fd f = G[i];

    // Assigning G[i] to $\empty \to \empty$ is equivalent to removing G[i],
    // but assigning operating takes less time.
    G[i] = FD_EMPTY;

    if (is_membership(N, G, f)) {

      // Since the erasing operation of vector is linear to the number of
```

```
      // elements between the erasing position and the end of the vector,
      // swapping G[i] to the end first can reduce time cost efficiently.
      G[i] = G[G.size() - 1];
      G.erase(G.end() - 1);

    } else {

      G[i++] = f;
    }
  }

  return G;
}

bool is_canonical(const int N, const fds &F) {

  for (const fd &f : F) {

    if (f.second.size() > 1) {

      return false;
    }
  }

  if (is_redundant(N, F)) {

    return false;
  }

  for (const fd &f : F) {

    if (f.first.size() > 1) {

      fd f2 = fd(f);
      attrs &X = f2.first;

      for (int i = 0; i < X.size(); i++) {

        const attr x = X[i];
```

```
        X[i] = X[X.size() - 1];
        X.erase(X.end() - 1);

        if (is_membership(N, F, f2)) {

          return false;
        }

        X.push_back(X[i]);
        X[i] = x;
      }
    }
  }

  return true;
}

fds canonical(const int N, const fds &F) {

  fds G = non_redundant(N, F);

  for (fd &f : G) {

    if (f.first.size() > 1) {

      fd f2 = fd(f);
      attrs &X = f2.first;

      for (int i = 0; i < X.size();) {

        const attr x = X[i];

        X[i] = X[X.size() - 1];
        X.erase(X.end() - 1);

        if (!is_membership(N, G, f2)) {

          X.push_back(X[i]);
```

```
        X[i++] = x;
      }
    }

    if (X.size() < f.first.size()) {

      f = f2;
    }
  }
}

fds H = fds();

for (const fd &f : G) {
  for (const attr &y : f.second) {

    H.push_back(fd(f.first, attrs({y})));
  }
}

return H;
}

bool is_direct(const int N, const fds &F, const fd &f) {

  // 0. Check if \f$ X \to Y \in F^+ \f$.
  if (!is_membership(N, F, f)) {

    return false;
  }

  const attrs &X = f.first;

  // 1. Find a non-redundant cover for `F`.
  fds G = non_redundant(N, F);

  // 2.1 Calculate X^+.
  bool D[N];
```

```
depend(N, G, X, D);

// 2.2 Determine ef(X).
bool EFX[G.size()];

for (int i = 0; i < G.size(); i++) {

  const attrs &Y = G[i].first;

  bool contained = true;

  for (const int &y : Y) {

    if (!D[y]) {

      contained = false;

      break;
    }
  }

  if (contained && is_membership(N, G, fd(Y, X))) {

    EFX[i] = true;

  } else {

    EFX[i] = false;
  }
}

// 3. Check if X \to Y \in (F - EF(X))^+.

fds H = fds();

for (int i = 0; i < G.size(); i++) {

  if (!EFX[i]) {
```

```
      H.push_back(G[i]);
    }
  }

  return is_membership(N, H, f);
}

fds minimum(const int N, const fds &F) {

  // 1. Find a non-redundant cover for `F`.
  fds G = non_redundant(N, F);

  // 2. Find all equivalence classes for `G`.

  // 2.1 Calculate $ X^+ for each X \to Y \in G $.
  bool **D = new bool *[G.size()];

  for (int i = 0; i < G.size(); i++) {

    D[i] = new bool[N];
  }

  for (int i = 0; i < G.size(); i++) {

    depend(N, G, G[i].first, &(D[i][0]));
  }

  // 2.2 Calculate equivalence classes.
  // M[i][j] indicates X_i <-> X_j.
  bool **M = new bool *[G.size()];

  for (int i = 0; i < G.size(); i++) {

    M[i] = new bool[G.size()];
  }

  for (int i = 0; i < G.size(); i++) {
    for (int j = 0; j < G.size(); j++) {
```

```
      M[i][j] = true;

      if (i != j) {

        for (const int &x : G[j].first) {

          if (!D[i][x]) {

            M[i][j] = false;

            break;
          }
        }
      }
    }
  }

  // 3. Replacing process.
  for (int i = 0; i < G.size(); i++) {

    // Since G[i] has already been merged into another functional dependency,
    // ignore G[i].
    if (G[i].first.size() == 0)
      continue;

    // G[i] is Y -> Y'

    // H is $ F - EF(X) $.
    fds H = fds();

    for (int j = 0; j < G.size(); j++) {

      if (!(M[i][j] && M[j][i])) {

        H.push_back(G[j]);
      }
    }

    // D2 is $ Y^+ $ under $ H = F - EF(X) $.
```

```
bool D2[N];

depend(N, H, G[i].first, D2);

for (int j = 0; j < G.size(); j++) {
  if (j != i && G[j].first.size() > 0 && M[i][j] && M[j][i]) {

    bool direct = true;

    for (const int &x : G[j].first) {

      if (!D2[x]) {

        direct = false;

        break;
      }
    }

    if (direct) {

      // Let's say:
      //    G[i]: Y1 \to Y2
      //    G[j]: Z1 \to Z2
      //
      // Since Y1 <-> Z1 and Y1 directly determine Z2, we can replace
      // G[i], G[j] with Z1 \to Y2Z2.

      set<int> Z2;

      copy(G[i].second.begin(), G[i].second.end(), inserter(Z2, Z2.end()

      copy(G[j].second.begin(), G[j].second.end(), inserter(Z2, Z2.end()

      G[i] = FD_EMPTY;
      G[j].second = attrs(Z2.begin(), Z2.end());

      break;
    }
```

```
      }
    }
  }

  // 4. Since we use two dynamic array D, M, clean up here.
  for (int i = 0; i < G.size(); i++) {

    delete[] D[i];
    delete[] M[i];
  }

  delete[] D;
  delete[] M;

  // 5. Filter the functional dependencies which are already been assigned
  // to FD_EMPTY.
  fds J = fds();

  for (const fd &f : G) {

    if (f.first.size() != 0 && f.second.size() != 0) {

      J.push_back(f);
    }
  }

  return J;
}

bool is_minimum(const int N, const fds &F) {

  return minimum(N, F).size() == F.size();
}

bool is_lminimum(const int N, const fds &F) {

  if (!is_minimum(N, F)) {

    return false;
```

```
  }

  for (const fd &f : F) {

    if (f.first.size() > 1) {

      fd f2 = fd(f);
      attrs &X = f2.first;

      for (int i = 0; i < X.size(); i++) {

        const attr x = X[i];

        X[i] = X[X.size() - 1];
        X.erase(X.end() - 1);

        if (is_membership(N, F, f2)) {

          return false;
        }

        X.push_back(X[i]);
        X[i] = x;
      }
    }
  }

  return true;
}

fds lminimum(const int N, const fds &F) {

  fds G = minimum(N, F);

  for (fd &f : G) {

    if (f.first.size() > 1) {

      fd f2 = fd(f);
```

```
      attrs &X = f2.first;

      for (int i = 0; i < X.size(); i) {

        const attr x = X[i];

        X[i] = X[X.size() - 1];
        X.erase(X.end() - 1);

        if (!is_membership(N, G, f2)) {

          X.push_back(X[i]);
          X[i++] = x;
        }
      }

      if (X.size() < f.first.size()) {

        f = f2;
      }
    }
  }

  return G;
}

bool is_lrminimum(const int N, const fds &F) {

  if (!is_lminimum(N, F)) {

    return false;
  }

  fds G = fds(F);

  for (fd &f : G) {
    if (f.second.size() > 1) {

      fd f2 = fd(f);
```

```
      attrs &Y = f.second;

      for (int i = 0; i < Y.size(); i++) {

        const attr y = Y[i];

        Y[i] = Y[Y.size() - 1];
        Y.erase(Y.end() - 1);

        if (is_membership(N, G, f2)) {

          return false;
        }

        Y.push_back(Y[i]);
        Y[i] = y;
      }
    }
  }

  return true;
}

fds lrminimum(const int N, const fds &F) {

  fds G = lminimum(N, F);

  for (fd &f : G) {
    if (f.second.size() > 1) {

      fd f2 = fd(f);
      attrs &Y = f.second;

      for (int i = 0; i < Y.size();) {

        const attr y = Y[i];

        Y[i] = Y[Y.size() - 1];
        Y.erase(Y.end() - 1);
```

```
        if (!is_membership(N, G, f2)) {

          Y.push_back(Y[i]);
          Y[i++] = y;
        }
      }
    }
  }

  return G;
}

bool is_mini(const int N, const fds &F) {

  fds G = mini(N, F);

  for (auto f : F) {
    if (f.second.size() != 1) {
      return false;
    }
  }

  if (G.size() != F.size()) {
    return false;
  }

  int cnt_g = 0;

  for (auto g : G) {
    cnt_g += g.first.size() + g.second.size();
  }

  int cnt_f = 0;

  for (auto f : F) {
    cnt_f += f.first.size() + f.second.size();
  }
```

```
  return cnt_f == cnt_g;
}

fds mini(const int N, const fds &F) {

  // 0. Split fds to make sure the right side of each fd is 1.
  fds G;

  for (auto f : F) {
    for (auto r : f.second) {
      G.push_back(fd(f.first, attrs({r})));
    }
  }

  bool_exprs expr_input;

  fprintf(stderr, " - Start generation of boolean expression.\n");

  // 1. Generate minimum boolean expressions.
  queue<bool_expr> expr_queue;
  expr_queue.push(bool_expr());

  while (expr_queue.size() > 0) {
    auto top = expr_queue.front();

    expr_queue.pop();

    if (top.size() == N) {
      expr_input.push_back(top);
    } else {
      for (char c = '0'; c <= '1'; c++) {

        bool_expr new_expr = bool_expr(top);
        new_expr.push_back(c);

        bool matched = false;

        for (auto f : G) {
          matched = true;
```

```
          for (auto id : f.first) {
            if (id < new_expr.size() && new_expr[id] != '1') {
              matched = false;
              break;
            }
          }

          for (auto id : f.second) {
            if (id < new_expr.size() && new_expr[id] != '0') {
              matched = false;
              break;
            }
          }

          if (matched) {
            break;
          }
        }

        if (matched) {
          expr_queue.push(new_expr);
        }
      }
    }
  }

  fprintf(stderr, " - Start Quine-McCluskey method. \n");

  // 2. Quine-McCluskey method.
  bool_exprs expr_output = qmc(expr_input);

  fprintf(stderr, " - Start generating the result.\n");

  // 3. Generate functional dependencies.
  fds H;

  for (auto expr : expr_output) {
```

```
    attrs l;

    for (int i = 0; i < N; i++) {
      if (expr[i] == '1') {
        l.push_back(i);
      }
    }

    for (int i = 0; i < N; i++) {
      if (expr[i] == '0') {
        H.push_back(fd(l, attrs({i})));
      }
    }
  }

  return H;
}

bool is_optimal(const int N, const fds &F) {

  fds G = optimal(N, F);

  int cnt_g = 0;

  for (auto g : G) {
    cnt_g += g.first.size() + g.second.size();
  }

  int cnt_f = 0;

  for (auto f : F) {
    cnt_f += f.first.size() + f.second.size();
  }

  return cnt_f == cnt_g;
}

fds optimal(const int N, const fds &F) {
```

```
  // Optimize algorithm is as same as minimize algorithm. The only difference
  // is that optimize algorithm requires the input must be a mini-cover.

  return minimum(N, mini(N, F));
}

} // namespace fdc
```

Listing A.4: Quine-Cluskey related functions

```
#include <map>
#include <queue>
#include <set>

#include "fdc.h"
#include "glpk/glpk.h"

namespace fdc {

using namespace std;

typedef pair<set<int>, bool_expr> qmc_combined_expr;

// Remove redundant expressions.
bool_exprs qmc_redundant_filter(bool_exprs exprs) {

  set<bool_expr> expr_set;

  for (auto expr : exprs) {
    expr_set.insert(expr);
  }

  return bool_exprs(expr_set.begin(), expr_set.end());
}

// Check if a and b has only one different element which are '0' and '1'.
bool qmc_match_expr(bool_expr a, bool_expr b, int &pos) {

  pos = -1;

  for (int i = 0; i < a.size(); i++) {
```

```
      if (a[i] != b[i]) {
        if (a[i] == '0' && b[i] == '1' && pos == -1) {
          pos = i;
        } else {
          return false;
        }
      }
    }

    return pos != -1;
}

int qmc_count_zero(bool_expr x) {

    int count = 0;

    for (auto c : x) {
      if (c == '0') {
        count++;
      }
    }

    return count;
}

int qmc_count_dash(bool_expr x) {

    int count = 0;

    for (auto c : x) {
      if (c == '-') {
        count++;
      }
    }

    return count;
}

int qmc_count_attributes(bool_expr x) {
```

```
  int count = 0;

  for (auto c : x) {
    if (c == '0' || c == '1') {
      count++;
    }
  }

  return count;
}

bool_expr expr_except(bool_expr expr, int pos) {

  bool_expr new_expr;

  for (int i = 0; i < expr.size(); i++) {
    if (i != pos) {
      new_expr.push_back(expr[i]);
    }
  }

  return new_expr;
}

bool expr_cover(bool_expr a, bool_expr b) {

  for (int i = 0; i < a.size(); i++) {
    if (a[i] != b[i] && a[i] != '-') {
      return false;
    }
  }

  return true;
}

// Try to combine expressions.
vector<qmc_combined_expr> qmc_combine(int N, bool_exprs exprs) {
```

```
set<bool_expr> reduced_set;
set<bool_expr> buckets[N + 1][N + 1];

for (int id = 0; id < exprs.size(); id++) {
  buckets[qmc_count_dash(exprs[id])][qmc_count_zero(exprs[id])].insert(
      exprs[id]);
}

fprintf(stderr, "Shape: %d x %ld\n", N, exprs.size());

for (int num_dash = 0; num_dash < N; num_dash++) {
  for (int num_zero = 0; num_zero < N; num_zero++) {

    fprintf(stderr, "Look %d, %d => %ld x %ld\n", num_dash, num_zero,
            buckets[num_dash][num_zero + 1].size(),
            buckets[num_dash][num_zero + 1].size());

    for (int pos = 0; pos < N; pos++) {

      // Prepare for matching.
      map<bool_expr, bool_expr> match_map;

      for (auto a : buckets[num_dash][num_zero + 1]) {
        if (a[pos] == '0') {
          match_map[expr_except(a, pos)] = a;
        }
      }

      for (auto b : buckets[num_dash][num_zero]) {
        if (b[pos] == '1') {
          bool_expr sign = expr_except(b, pos);

          if (match_map.find(sign) != match_map.end()) {

            auto a = match_map[sign];

            bool_expr new_expr = bool_expr(a);
            new_expr[pos] = '-';
```

```
              buckets[num_dash + 1][num_zero].insert(new_expr);

              reduced_set.insert(a);
              reduced_set.insert(b);
            }
          }
        }
      }
    }
  }

  vector<qmc_combined_expr> result;

  for (int num_dash = 0; num_dash < N; num_dash++) {
    for (int num_zero = 0; num_zero < N; num_zero++) {
      for (auto a : buckets[num_dash][num_zero]) {
        if (reduced_set.find(a) == reduced_set.end()) {

          set<int> ids;

          for (int i = 0; i < exprs.size(); i++) {
            if (expr_cover(a, exprs[i])) {
              ids.insert(i);
            }
          }

          result.push_back(qmc_combined_expr(ids, a));
        }
      }
    }
  }

  return result;
}

bool_exprs qmc_search(bool_exprs exprs,
                      vector<qmc_combined_expr> combined_exprs) {

  set<int> refs[exprs.size()];
```

```
glp_prob *lp = glp_create_prob();

glp_set_prob_name(lp, "qmc");

glp_set_obj_dir(lp, GLP_MIN);

// Rows.
glp_add_rows(lp, exprs.size());

for (int i = 1; i <= exprs.size(); i++) {
  glp_set_row_bnds(lp, i, GLP_LO, 1, 0);
}

// Columns.
glp_add_cols(lp, combined_exprs.size());

for (int i = 1; i <= combined_exprs.size(); i++) {
  glp_set_col_kind(lp, i, GLP_IV);
  glp_set_col_bnds(lp, i, GLP_LO, 0, 0);
  glp_set_col_bnds(lp, i, GLP_UP, 0, 1);
  glp_set_obj_coef(lp, i, qmc_count_attributes(combined_exprs[i - 1].secon
}

// Matrix.
int matrix_size = 0;

for (auto expr : combined_exprs) {
  matrix_size += expr.first.size();
}

int ia[matrix_size + 1], ja[matrix_size + 1];
double ar[matrix_size + 1];

for (int i = 0, j = 1; i < combined_exprs.size(); i++) {
  for (auto id : combined_exprs[i].first) {
    ia[j] = id + 1;
    ja[j] = i + 1;
    ar[j] = 1;
```

```
      j++;
    }
  }

  glp_load_matrix(lp, matrix_size, ia, ja, ar);

  // Solve the described 01-IP problem.
  glp_simplex(lp, NULL);

  bool_exprs result;

  for (int i = 1; i <= combined_exprs.size(); i++) {
    if (glp_get_col_prim(lp, i) > 0.5) {
      result.push_back(combined_exprs[i - 1].second);
    }
  }

  return result;
}

bool_exprs qmc(bool_exprs exprs) {

  if (exprs.size() == 0) {
    return exprs;
  }

  int N = exprs[0].size();

  // 1. Remove redundant expressions.
  exprs = qmc_redundant_filter(exprs);

  fprintf(stderr, " - Start QMC-Combine.\n");

  // 2. Try to combine and reduce expressions.
  auto combined_exprs = qmc_combine(N, exprs);

  fprintf(stderr, " - Start QMC-Searching.\n");

  // 3. Search for the best selection of combined expressions.
```

```
  return qmc_search(exprs, combined_exprs);
}

} // namespace fdc
```