

1 Mining Functional Dependencies

1.1 the closure algorithm

In our implementation we have collected together functional dependencies with the same left side. This style of data organizations allows for the simple storage as dictionary or map of attribute sets where the antecedent becomes the dictionary key and the different consequents with same antecedent make up the value. Both key and value of the dictionary consist of attribute sets.

This style of data organization has made the implementation of the closure algorithm more efficient since the current attribute set needs to be only tested once for the containment of the antecedent in order to acquire all the consequents which have a common left side. In general bundling rules with same left side is more efficient whenever these rules are used together. In cases where individual rules split by each attribute of a right side are needed like in the mincoverage algorithm there was no runtime penalty either although the implementation became a bit more intricate as the rules are split by runtime logic rather than by generating a datastructure to hold split rules. We will show later on when it comes to the implementation of the mincoverage algorithm how this is done.

For now let us consider the implementation of the closure algorithm:

```

01 function closure(attrs, dependencies) :
02   haschanged := true;
03   while haschanged :
04     haschanged := false;
05     foreach li → re ∈ dependencies :
06       if li ⊆ attrs ∧ re ⊄ attrs :
07         attrs := attrs ∪ re;
08         haschanged := true;
09   return attrs;

```

There are a couple of lemmata about the closure algorithm which we will use:

$$\begin{array}{l}
 (i) \quad I \in A_E^+ \stackrel{def}{\Leftrightarrow} I \in \text{closure}(A, E) \Leftrightarrow E \vdash A \rightarrow I \\
 \text{we will prove now: } Q = (AB)_E^+ \Rightarrow E \vdash AB \rightarrow Q \\
 \text{assume that: } \{A \rightarrow C, C \rightarrow D\} \in E \text{ then follows} \\
 \frac{AB \rightarrow ABC, ABC \rightarrow ABCD \text{ (augmentation)}}{AB \rightarrow ABCD \text{ (transitivity)}}
 \end{array}$$

If $li \subseteq attrs$ then both sides of the rule are augmented for all attributes in $attr$. Finally the new attribute set $attrs \cup re$ can be inferred by applying transitivity. After having successfully applied the closure algorithm we may split a singleton right side out of the closure by the rule of decomposition which we will show later on: $I \in A_E^+ \Rightarrow E \vdash A \rightarrow I$.

$$(ii) \quad AJKL \subseteq A_E^+ \Rightarrow (AJKL)_E^+ = A_E^+$$

The given property directly follows out of the closure algorithm: As $AJKL$ can become an intermediate step of calculating A_E^+ the final result will be the same.

$$(iii) \quad G = E \setminus \{M \rightarrow JKL\} \Rightarrow (AJKL)_E^+ = (AJKL)_G^+$$

As the rules we are leaving out do only inference attributes which are already in our base set that will not make a difference for the closure algorithm as $re \subseteq attrs$.

1.2 an algorithm for calculating the minimum coverage

The minimum coverage of a set of functional dependencies is a ruleset that can no more be reduced by leaving out any left or right side attribute of any rule. There may be multiple minimum coverages for a given ruleset which can be gained depending on the order in which the algorithm tests for leaving out individual rules with singleton right side attribute or leaving out left side attributes.

```

01 function mincoverage(dependencies) :
02   foreach  $li \rightarrow re \in dependencies$  :
03      $reddep := dependencies \setminus \{li \rightarrow re\}$ 
04      $othersclosure := closure(li, reddep)$ ;
05      $newre := re \setminus othersclosure$ ;
06      $precond := li \cup re$ 
07     foreach  $r \in newre$  :
08        $precond := precond \setminus \{r\}$ ;
09       if  $r \in closure(precond, reddep)$  :
10          $newre := newre \setminus \{r\}$ ;
11       else :
12          $precond := precond \cup \{r\}$ ;
13     foreach  $l \in li$  :
14        $tryred := li \setminus \{l\}$ ;
15        $redcl := closure(tryred, reddep)$ ;
16       if  $li \subseteq redcl$  :
17          $li := li \setminus \{l\}$ 
18   if  $\exists x : li \rightarrow x \in reddep$  :
19      $dependencies := reddep \setminus \{li \rightarrow x\} \cup \{li \rightarrow x \cup re\}$ ;
20   else :
21      $dependencies := reddep \cup \{li \rightarrow re\}$ ;
22   return dependencies;
```

At first we select a whole rule bundle with common left side and individually different consequents collected to a singleton right side (line 2). We remove the current rule bundle from the dependencies (line 3) and test which right sides can still be inferred from the given left side (line 4). Those right sides which can be inferred also without the current rule bundle are redundant and may be removed (line 5). This is because a rule may be decomposed and re-composed by the attributes of its right side when the left side matches.

$$\text{to show: } \{A \rightarrow IJKL\} \in E \\ F = E \setminus \{A \rightarrow IJKL\} \cup \{A \rightarrow KL\} \wedge IJ \subseteq A_F^+ \Rightarrow E^+ = F^+$$

decomposition:

$$\frac{F^+ \subseteq E^+ : \quad A \rightarrow IJKL, \quad IJKL \rightarrow IJ, \quad IJKL \rightarrow KL \quad (refl.)}{A \rightarrow IJ, \quad A \rightarrow KL \quad (trans.)}$$

composition:

$$\frac{E^+ \subseteq F^+ : \quad \begin{array}{l} IJ \subseteq A_F^+ \Leftrightarrow A \rightarrow IJ, A \rightarrow KL \\ A \rightarrow AIJ, AIJ \rightarrow IJKL \end{array}}{A \rightarrow IJKL} \begin{array}{l} (augm.) \\ (trans.) \end{array}$$

Now that we have already tested for which attributes leaving out the whole rule bundle does not matter we need to test for individual rules with singleton right side attribute whether they could still be left out. We could have done by creating a datastructure of split rules and then we could have tried to leave out any singleton rule of these. However this would have come with a runtime penalty and additional implementation complexity.

The trick about doing this without creating split rules is to include the consequents of individual rules which are considered to be necessary by the current test in the source attribute set of the closure call. We know these attributes would be inferenced any way by our additional split rules so we just include them in the source argument rather than letting the closure call derive them from singleton rules on its own. Arguments in the source attribute set of the closure call will always appear in the output set.

$$\begin{array}{l} G = E \setminus \{A \rightarrow IJKL\}, F = E \setminus \{A \rightarrow I\} \\ \text{to show : } I \in (AJKL)_G^+ \Leftrightarrow I \in A_F^+ \\ G = E \setminus \{A \rightarrow I\} \setminus \{A \rightarrow JKL\} \\ G = F \setminus \{A \rightarrow JKL\} \\ F = G \cup \{A \rightarrow JKL\} \\ \Rightarrow AJKL \subseteq A_F^+ \\ \Rightarrow_{ii} (AJKL)_F^+ = A_F^+ \\ \Rightarrow_{iii} (AJKL)_G^+ = (AJKL)_F^+ \\ \Rightarrow I \in A_F^+ \Leftrightarrow I \in (AJKL)_G^+ \end{array}$$

While F is the set of dependencies we wanna test for G is the set of reduced dependencies we have already calculated in line 3 by removing the whole rule bundle. We now wanna show how to do the same calculation with G rather than F when it comes to remove individual rules. As $A \rightarrow JKL$ is in F adding these attributes to the source argument of the closure call in addition to attribute A does not change its result. Finally we do not need the attributes now added to the source argument as right sides in our rules any more.

The reason why we have checked to remove rules by leaving out the whole rule bundle prior to checking for rules with individual right side is that this is generally the faster way to get rid of redundant attributes. Otherwise the checks with simulated individual right side would suffice.

In lines 13 to 17 we test whether we can leave out individual attributes on the left side of a rule. This is the case if one of the attributes on the left side can be inferenced by the other attributes. Consequently we try to leave out every singleton attribute once. The fact that it is legal to leave out an attribute on the left side implied by others can easily be verified based on the Armstrong axioms:

$$\begin{aligned}
& \{ABC \rightarrow IJKL\} \in E \quad \wedge \quad AB \rightarrow C \in E^+, \\
& F = E \setminus \{ABC \rightarrow IJKL\} \cup \{AB \rightarrow IJKL\} \\
& \Rightarrow E^+ = F^+
\end{aligned}$$

$$\begin{array}{c}
E^+ \subseteq F^+ : \quad AB \rightarrow IJKL \\
\hline
\begin{array}{c}
ABC \rightarrow IJKLC, \quad IJKLC \rightarrow IJKL \quad (aug., refl.) \\
ABC \rightarrow IJKL \quad (trans.)
\end{array}
\end{array}$$

$$\begin{array}{c}
F^+ \subseteq E^+ : \quad ABC \rightarrow IJKL, \quad AB \rightarrow C \\
\hline
\begin{array}{c}
AB \rightarrow ABC \quad (aug.) \\
AB \rightarrow IJKL \quad (trans.)
\end{array}
\end{array}$$

You may still believe that there is a possible flaw in the given implementation to reduce the left side so that it becomes minimal: It tries to reduce the left side for the whole rule bundle only. However leaving out the other rules of the same rule bundle is o.k. since these rules will never execute as long as their left side is not fully covered. However if it is fully covered we have shown that we can already reduce the left side for the whole rule bundle.

The final statements of the algorithm (lines 18 to 21) do contain code in order to replace the previous rule bundle with the newly generated rule containing a reduced left and reduced right side.

As we have effectively tried to leave out any rule split by singleton right side attribute and every attribute of a left side of any rule we have tried through all the given possibilities to reduce the existing ruleset. As we have now successfully completed this task we may not only assume that the returned closure is equivalent to the input closure but also that it is a minimum coverage of the given ruleset.

The actual code for calculating the minimum coverage of a ruleset does also contain some code to reorder the input rules because if there are more than one minimum coverage of functional dependencies we wanna be able to return them. This is less important for calculating keys upon a given ruleset but it may be useful when it comes to normalize the given database schema later on.

1.3 algorithm for finding keys

```

01 function keysTreeAlgorithm(attr, dependencies) :
02   foreach a ∈ attr : attrch[a] := 0;
03   foreach li → re ∈ dependencies :
04     foreach l ∈ li : attrch[l] := attrch[l] bitwise_or 1;
05     foreach r ∈ re : attrch[r] := attrch[r] bitwise_or 2;
06   for i := 0 to 3 : sets[i] := ∅;
07   foreach a ∈ attr :
08     sets[attrch[a]] := sets[attrch[a]] ∪ {a};
09   (independent, left, right, both) := sets[];
10   subkey := left ∪ independent;
11   if closure(subkey, dependencies) == attr : return subkey
12   curlvl := ∅; keys := ∅;
13   foreach m ∈ both :
14     kk = subkey ∪ {m};
15     if closure(kk, dependencies) == attr : keys.add(kk);
16     else : curlvl[kk] := m;
17   while curlvl ≠ ∅ :
18     prevlvl := curlvl; curlvl := ∅;
19     for subkey ↦ maxm ∈ prevlvl :
20       missingattr := ∅;
21       foreach a ∈ both : if a > maxm ∧ a ∉ subkey : missingattr := missingattr ∪ {a};
22       foreach m ∈ missingattr :
23         kk := subkey ∪ {m}; isPartOfKey := false;
24         for key ∈ keys :
25           if key ⊆ kk : isPartOfKey := true; break;
26           if ¬isPartOfKey :
27             if closure(kk, dependencies) == attr : keys.add(kk);
28             else : curlvl[kk] := m;
29   return keys;

```

A key is by definition a minimal set of attributes from which all the other attributes can be deduced. We check this with a closure call to any key candidate. If the closure call succeeds to deduce all other attributes we call the given base set a superkey as we have not yet verified that the given set of attributes is minimal; i.e. it could stay a superkey after removing one attribute.

There can be found key search algorithms in literature which start with the whole set of attributes and reduce them one by one until a key is found. However we have chosen the other way round to assemble keys by adding one attribute after another because this is generally the faster way to find a key. Typically only a few attributes are required to be part of a key as there are many functional dependencies between the different data items of a table row.

At first we split the attributes in four sets (lines 1-9): Those which are contained in none of the functional dependencies, those which do only occur at the left side of a dependency, those only occurring on the right side and finally attributes which appear in an antecedent as well as in the consequent of a rule.

All attributes which can not be inferred from others need to be contained in the key: This comprises the independent and the left-side sets of attributes.

If the resulting set is already a key it is returned since this will then be the only key. Otherwise we go on with adding attributes of the middle set contained on both sides of a rule.

The reason why attributes appearing only on a right side do not need to be considered is that first none of the other attributes can be deduced from them and second that there are other attributes deducing them. The other attributes need to be deducible by the key as all attributes need to be deducible by a key.

The algorithm now starts with the minimum set of attributes that need to be contained in every key (left + independent set). Then in every step one more attribute is added. If the given attribute set ends up to be a superkey in the first step it is also a key because we have already checked that no fewer attributes can serve as a key. In the following step we check whether the newly constructed attribute set is a superset of an existing key. We can do so successfully because all keys comprising one element less than the current set have already been found. If the current set is a superkey and does not contain any key with at least one element less it is a key since we have added attributes one by one. Otherwise the set that results from adding one candidate attribute becomes stored for the next step where new attributes are added subsequently.

Some algorithms found in literature add just any attribute to the new set which is not yet part of the set. However this is a suboptimal way to do it since the same set of attributes can be constructed in multiple ways. The consequence is that one would need to check for duplicate nodes which slowed the needed execution time down. Nonetheless there is a simple way to avoid this.

You need a strong monotonic ordering of the attributes for this like alphabetical comparison is. If you only add attributes which are greater than the greatest attribute in the current set then there is only one way to construct a given attribute set. This is because an attribute that has been chosen to be omitted in step i will never be selected in any subsequent step $i+j$ again. If every step makes a choice about some ignored and one added attribute we arrive at an algorithm that only generates key candidates in ascending attribute order which is unambiguous and therefore avoids duplicates.