Efficient order dependency detection**总结

符号含义

$C_l(X)$：在l层，包含所有的Y,使得$X \rightarrow_< Y$是否有效仍需要验证，且$|X|+|Y|=l$

CS_l：l层的所有候选集的集合

X'：X的最长前缀，例：$X=ABCD$，则$X'=ABC$

τ_X：属性列X的排序分区，将元组根据属性列X的值分割成一些等价类，使得这些等价类的X值相等

τ_X^k：表示X中值第k小的元组

$|\tau_X|$：表示τ_X中等价类数量

> 因为使用τ_X这种方法，所以没有必要去存储整个元组的数据值。只需要存储元组的标识符(行索引)

Table 8 Sorted partitions and data from which they are created

Tid	Weight	Shipping cost
(a) Table with shipped goods		
0	5	14
1	10	22
2	3	10
3	10	25
4	5	14
5	20	40
(b) Corresponding sorted partitions. Tuples are denoted by their identifiers		
$\tau_{weight} = (\{2\}, \{0, 4\}, \{1, 3\}, \{5\})$		
$\tau_{cost} = (\{2\}, \{0, 4\}, \{1\}, \{3\}, \{5\})$		

定义

Definition 4 (*Order-minimality*) An attribute list $\mathbf{X}$ is *minimal*, iff for any disjoint, contiguous sub-lists $\mathbf{V}$ and $\mathbf{W}$ in $\mathbf{X}$ where

1. $\mathbf{W}$ directly precedes $\mathbf{V}$ in $\mathbf{X}$, *or*
2. $\mathbf{W}$ follows (not necessarily directly) after $\mathbf{V}$ in $\mathbf{X}$,

$\mathbf{V} \rightarrow_{\leq} \mathbf{W}$ is not valid.

（不太清楚）

Definition 5 *(Minimality of order dependencies)* The order dependency $\mathbf{X} \rightarrow_{<} \mathbf{Y}$ is minimal, iff

1. $\nexists\, \mathbf{V} \in \text{PREFIXES}(\mathbf{X})$, s.t. $\mathbf{V} \nrightarrow_{<} \mathbf{Y}$, *and*
2. $\nexists\, \mathbf{W} \in \text{PREFIXES}(\mathbf{Y})$, s.t. $\mathbf{X} \rightarrow_{<} \mathbf{W}$ is valid, *and*
3. $\mathbf{X}$ is minimal, *and*
4. $\mathbf{Y}$ is minimal.

引理

Lemma 1 *A functional dependency $\mathcal{X} \rightarrow A$ is valid, iff there is no split among (B, A) for any $B \in \mathcal{X}$.*

Lemma 2 *An order dependency $X \rightarrow_{\leq} Y$ is valid, iff there is neither a split nor a swap among (X, Y).*

Lemma 3 *An order dependency* $\mathbf{X} \to_{<} \mathbf{Y}$ *is valid, iff there is neither a merge nor a swap among* $(\mathbf{X}, \mathbf{Y})$.

Lemma 4 $\mathbf{X} \to_{\leq} \mathbf{Y}$ *is valid* $\Rightarrow \mathcal{X} \to \mathcal{Y}$ *is valid [22].*

Lemma 5

$$\begin{aligned} \mathbf{X} \to_{<} \mathbf{Y} \text{ is valid} &\Rightarrow \mathbf{Y} \to_{\leq} \mathbf{X} \text{ is valid} \\ &\Rightarrow \mathcal{Y} \to \mathcal{X} \text{ is valid} \end{aligned}$$

Lemma 6 *If* $\mathbf{V} \to_{<} \mathbf{W}$ *is valid and* $\mathbf{V}$ *is unique, the following statements are true for any attribute lists* $\mathbf{X}$ *and* $\mathbf{Y}$ *over a relation* $\mathcal{R}$.

$$\mathbf{W} \text{ is unique.} \tag{1}$$

$$\mathbf{W} \to_{<} \mathbf{V} \text{ is valid.} \tag{2}$$

$$\mathbf{VX} \to_{<} \mathbf{WY} \text{ is valid.} \tag{3}$$

$$\mathbf{WY} \to_{<} \mathbf{VX} \text{ is valid.} \tag{4}$$

$$\mathbf{VX} \to_{\leq} \mathbf{WY} \text{ is valid.} \tag{5}$$

$$\mathbf{WY} \to_{\leq} \mathbf{VX} \text{ is valid.} \tag{6}$$

$$\mathcal{V} \to \mathcal{W} \text{ is valid.} \tag{7}$$

$$\mathcal{W} \to \mathcal{V} \text{ is valid.} \tag{8}$$

Lemma 7 *There is a merge among* $(\mathbf{X}, \mathbf{Y}) \iff$ *the relation* $e \subseteq \tau_{\mathbf{X}} \times \tau_{\mathbf{Y}}$ *is not injective, with* $(x, y) \in e$ *if* $\exists t_y \in y$*, s.t.* $t_y \in x$*, where* t_y *is a tuple identifier in* y *and* $x \in \tau_{\mathbf{X}}$, $y \in \tau_{\mathbf{Y}}$.

> 排序分区和merge之的关系，若(X,Y)中存在merge，则元组s,t中存在后等前不等的情况，则在τ_X中s,t位于不同的等价类中，但是 τ_Y 中s,t位于同一个等价类中。
>
> ***引理7的表述不太清楚***（$\tau_X \times \tau_Y$是什么意思，injective是什么意思），e表示一个等价类

定理

Theorem 1 $X \rightarrow_< Y$ *is valid* $\Longleftrightarrow$ $Y \rightarrow_\leq X$ *is valid*

定理1表明，发现所有基于<操作符的n元OD，便可以发现所有基于≤操作符的n元OD。

晶格网络

用晶格网络来表示搜索空间

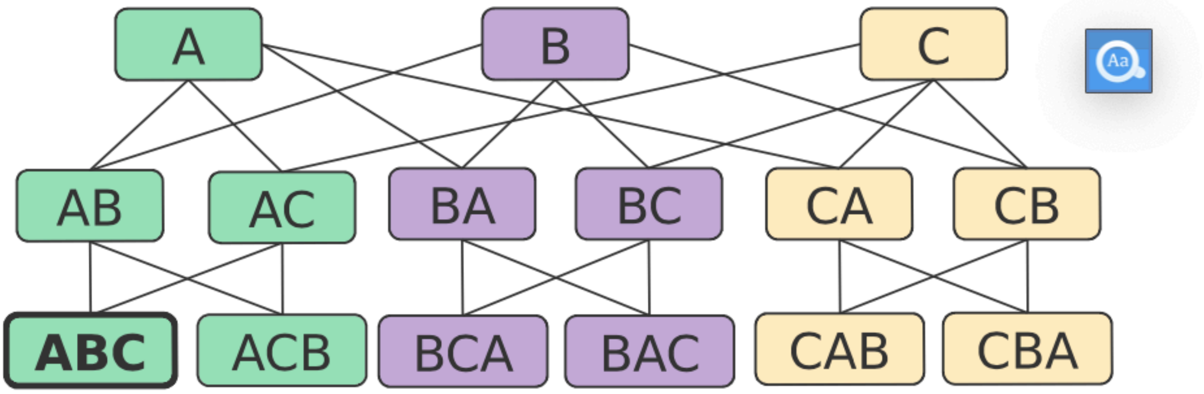


Fig. 1 A candidate lattice created from three attributes. The *highlighted node* ABC generates the ODs $A \rightarrow_< BC$ and $AB \rightarrow_< C$

第K层的节点包含k个属性并且代表了k-1个候选ODs。

---

checkForSwap算法

算法检查τ_X，τ_Y中是否至少存在一个swap

e_X表示τ_X中的第i小的等价类（簇）

Algorithm 1 CHECKFORSWAP$_<$

Input: $\tau_{\mathbf{X}}$, $\tau_{\mathbf{Y}}$

Output: "*swap*" if there is at least one swap among $(\mathbf{X}, \mathbf{Y})$,
"*merge*" if there is no swap, but a merge,
"*valid*" if $\mathbf{X} \rightarrow_< \mathbf{Y}$ is valid

```
next_eX ← next_eY ← true; i ← 1; j ← 1
merge ← false
while i < τ_X, j < τ_Y
    if (next_eX) e_X ← τ_X^i
    if (next_eY) e_Y ← τ_Y^j
    if |e_X| < |e_Y|
        if not e_X ⊆ e_Y
            return "swap"
        else
            merge ← true
            i ← i + 1
            next_eX ← true
            e_Y ← e_Y − e_X
            next_eY ← false
    else
        if not e_Y ⊆ e_X
            return "swap"
        else
            j ← j + 1
            next_eY ← true
            e_X ← e_X − e_Y
            if |e_X| = 0
                i ← i + 1
                next_eX ← true
            else
                next_eX ← false
if (merge) return "merge"
else return "valid"
```

代码解析：

line3: $i < |\tau_X|$...

考虑两种情况：

1. e_X标识符数量小于e_Y
   - 若没有$e_X \subseteq e_Y$，则返回swap
   - 否则$merge = true$，由于$|e_X| < |e_Y|$，e_Y中还会有一个元组没进行比较，所以取下一个e_X，然后把e_Y中没处理过的元组保留，进行下一次比较。
2. $|e_X|$等于或大于$|e_Y|$
   - 若没有$e_Y \subseteq e_X$，则返回swap
   - 否则取下一个e_Y，删除当前e_X中处理过的元组
     - 若$|e_X| = 0$，则取下一个e_X
     - 否则继续处理当前e_X

若$merge = true$成立，则返回merge

否则返回valid

Product of sorted partitions(分区的乘积)

简述思路

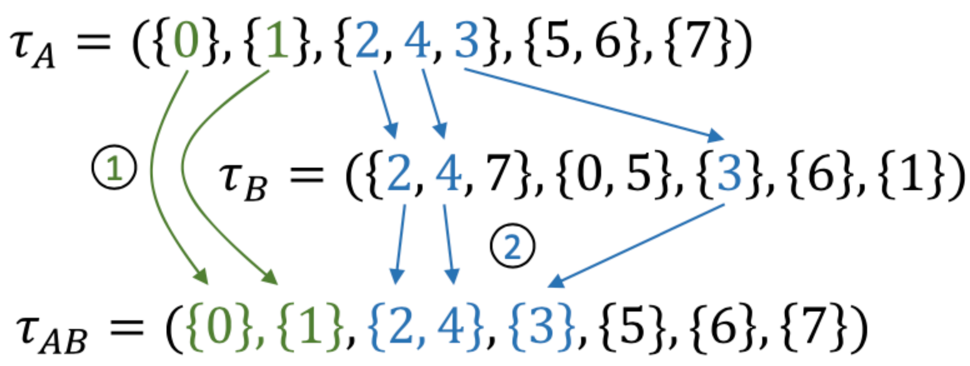


Fig. 2 Sorted partition product of τ_A and τ_B yields τ_{AB}. The product can be implemented efficiently using a *hash-join-like* procedure as shown in Fig. 3

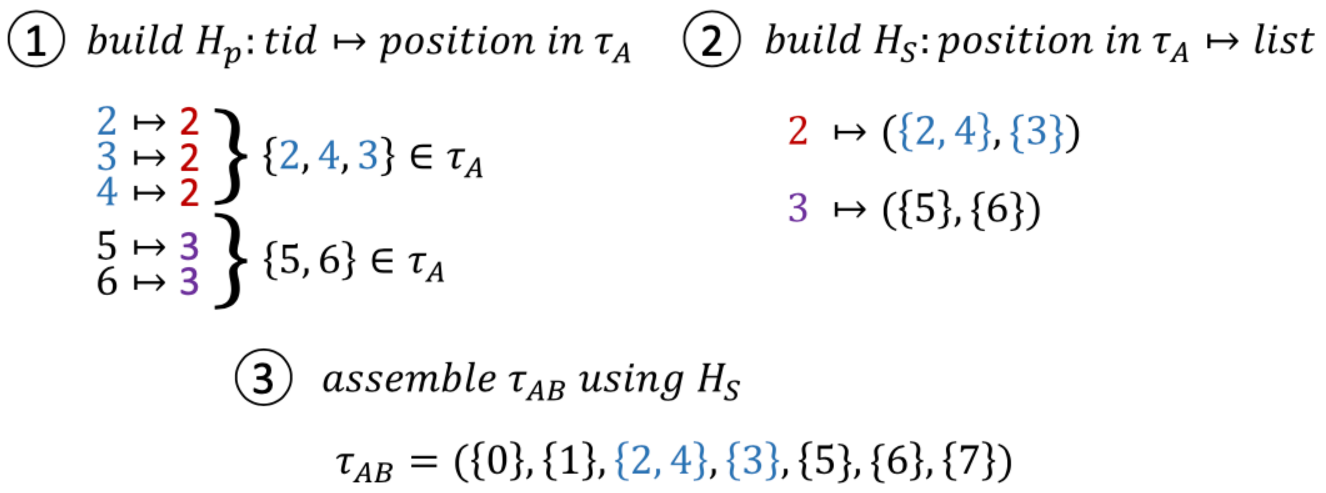


Fig. 3 Workflow of the sorted partition product algorithm on τ_A and τ_B from Fig. 2

> 1. 等价类大小为1时
>   - 直接加入到τ_{AB}
> 2. 等价类大小大于1时
>   - 根据在τ_B中出现的位置分成新的等价类再加入τ_{AB}中

> 使用类似哈希连接的过程实现两个排列分区的乘积
>
> 1. 创建哈希表H_P，将元组标识符t映射到t所在的τ_A中的位置（只保存大小大于1的等价类）
> 2. 遍历τ_B，建立哈希表H_S，将τ_A中大小大于1的等价类的位置映射到τ_{AB}的等价类列表中

Algorithm 2 PARTITIONPRODUCT

Input: τ_A, τ_B, hash table H_p
Output: τ_{AB}

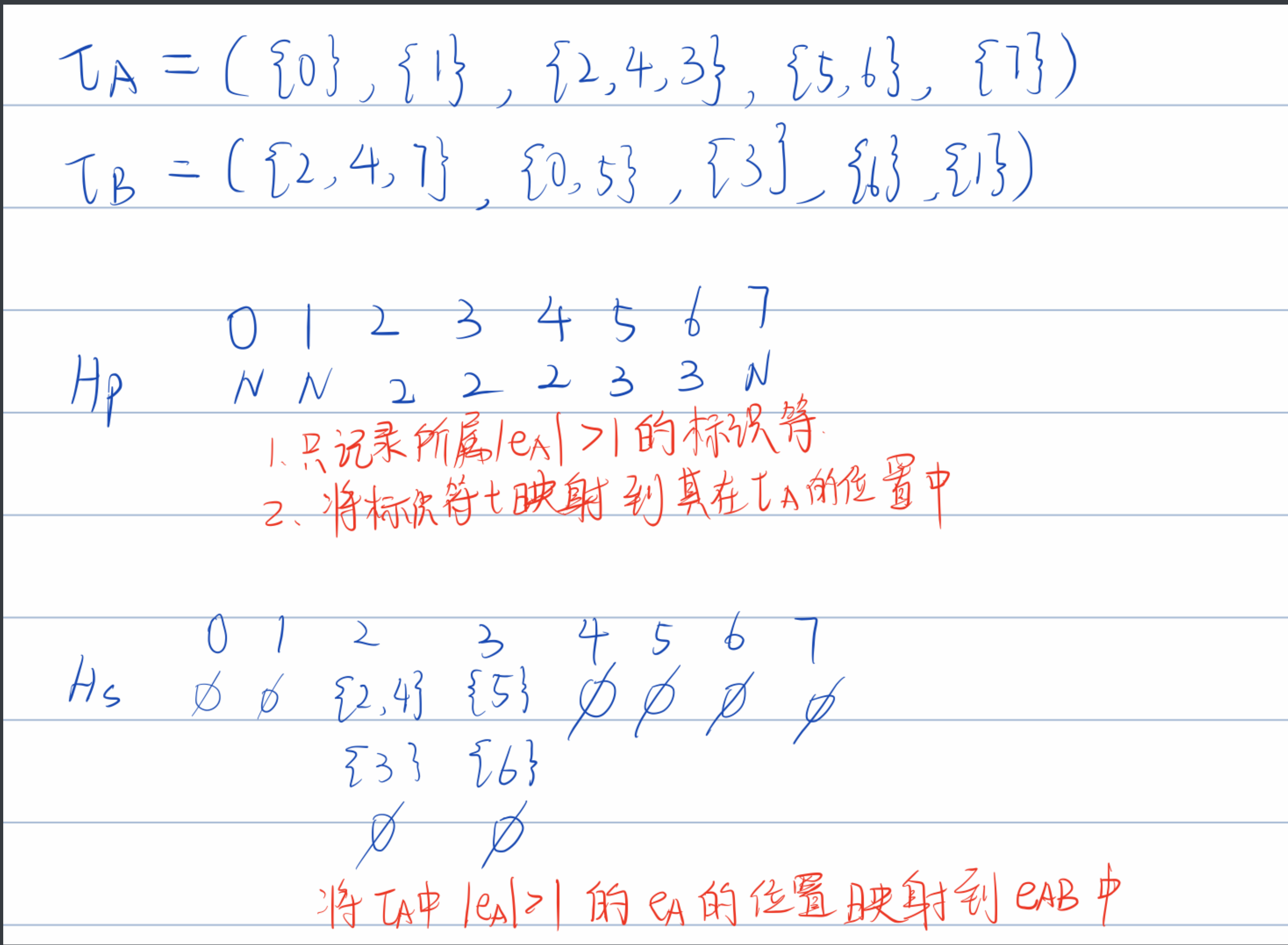
$\tau_{AB} \leftarrow [\,]$
for each $e_B \in \tau_B$, $|\tau_B| > 1$
    $visited \leftarrow \emptyset$
    for each $tid_{e_B} \in e_B$
        if $H_p(tid_{e_B}) = \perp$
            continue
        $pos \leftarrow H_p(tid_{e_B})$
        add pos **to** $visited$
        add tid_{e_B} to last equivalence class in $H_S(pos)$
    for each $pos \in visited$
        append $\emptyset$ **to** $H_S(pos)$
for each index i in τ_A
    if $|\tau_A^i| = 1$
        append e_A **to** τ_{AB}
    else
        for each $e_{AB} \in H_S(i)$, $e_{AB} \neq \emptyset$
            append e_{AB} **to** τ_{AB}

$tide_{eB}$：等价类e_B中的元组标识符

Visited：记录τ_B中的等价类e_B中的元组tid_{eB}(在τ_A中所属e_A大小大于1)，在τ_A中的位置i（$H_p(tid_{eB})$）的集合。

H_p：记录大小大于1的等价类中的元组标识符，$H_p(3) = 2$表示元组标识符3在τ_A的第二个等价类中

H_S：根据逐一遍历τ_B在中的等价类中的元组标识符的结果，记录由τ_A中大小大于1的等价类经过τ_B处理得到的τ_{AB}中的等价类的所有标识符，$H_S(2) = \{\{2,4\},\{3\}\}$表示在$\tau_A$中的等价类$\{2,3,4\}$，由于$\tau_B$在$\tau_{AB}$中分为2个等价类$\{2,4\},\{3\}$。



代码解析：

Line5：如果H_p的tid_{eB}位置为空，表示该元组的等价类大小为1，则继续遍历。

Line7：若不为空，表示等价类大小大于1，先记录下该元组在τ_A中的位置pos。

Line8：将pos加入visited。

Line9：将该元组的标识符加入H_S中pos位置的最后一个等价类中。

Line10~11：遍历完τ_B中一个等价类中所有元组后，在H_S中pos出现过的位置加入一个空集合。来确保τ_A中一个等价类中的元组标识符（H_p中不为空且值相同的位置）能继续添加到H_S中。

Line12~17：在遍历完τ_B后，τ_{AB}中的总体顺序应该与τ_A相同（将表按A排序和将表按AB排序大致相同），再遍历e_A，对τ_A中对等价类e_A处理得到e_{AB}。

- 若$|\tau_A^i| = 1, i.e. |e_A| = 1$，则直接将$e_A$加入$\tau_{AB}$
- 否则将$H_S(i)$中的非空等价类e_{AB}依次加入τ_{AB}

剪枝原则

Pruning rule 1 ***(Invalid under "<")***

$$X \nrightarrow_< Y \Rightarrow XV \nrightarrow_< Y$$

holds for any disjoint attribute lists $\mathbf{X}$, $\mathbf{Y}$, *and* $\mathbf{V}$ *over a schema* $\mathcal{R}$, *where only* $\mathbf{V}$ *may be empty.*

> (A,CD)中存在merge，则(AB,CD)中依然存在merge
>
> (A,CD)中存在swap，则(AB,CD)中依然存在swap

Pruning rule 2 ***(Valid under "<")***

$$X \rightarrow_< Y \text{ is valid} \Rightarrow X \rightarrow_< YW \text{ is valid}$$

holds for any disjoint attribute lists $\mathbf{X}$, $\mathbf{Y}$, *and* $\mathbf{W}$ *over a schema* $\mathcal{R}$, *where only* $\mathbf{W}$ *may be empty.*

> (A,BC)中没有merge或swap，则(A,BCD)中肯定也没有merge或swap

Pruning rule 3 ***(Swap under "<")*** *Given* $X \nrightarrow_< Y$ *is invalidated by a swap,*

$$X \nrightarrow_< Y \Rightarrow XV \nrightarrow_< YW$$

holds for any disjoint attribute lists $\mathbf{X}$, $\mathbf{Y}$, $\mathbf{V}$, *and* $\mathbf{W}$ *over a schema* $\mathcal{R}$, *where only* $\mathbf{V}$ *and* $\mathbf{W}$ *may be empty.*

> (A,C)中存在swap，则(AB,CD)中肯定也存在swap，因为在A，C右侧加入属性并没有改变元组在A,C中原先的顺序

Pruning rule 4 *(Uniqueness under “$<$”) Given* $\mathbf{X}$ *is unique,*

$$\mathbf{X} \rightarrow_{<} \mathbf{Y} \text{ is valid} \Rightarrow \mathbf{XV} \rightarrow_{<} \mathbf{YW} \text{ is valid}$$

holds for any disjoint attribute lists $\mathbf{X}$, $\mathbf{Y}$, $\mathbf{V}$, *and* $\mathbf{W}$ *over a schema* $\mathcal{R}$, *where only* $\mathbf{V}$ *and* $\mathbf{W}$ *may be empty.*

> 由Lemma6得

由于自下而上的生成候选集，晶格网络中l层的节点均是$l+1$层节点的前缀，在生成候选集的时候使用剪枝规则可以有效降低时间复杂度

ORDER算法

代码

Algorithm 3 Algorithm ORDER, which finds all valid ODs $\mathbf{X} \rightarrow_{<} \mathbf{Y}$

Input: Relation $\mathcal{R}$, relational instance r

Output: *valid*, the set of all minimal valid order dependencies that are valid in r

```
l ← 1
valid ← ∅                          ▷ global variable: set of all valid ODs
C_0([ ]) ← ∅; C_1([ ]) ← {A | A ∈ R}
CS_0 ← ∅; CS_1 ← ∅
L_1 ← {A | A ∈ R}
while L_l ≠ ∅
    CS_l ← UPDATECANDIDATESETS(CS_{l−1})
    COMPUTEDEPENDENCIES(L_l, CS_l)
    PRUNE(L_l, CS_l)
    L_{l+1} ← GENERATENEXTLEVEL(L_l)
    l ← l + 1
output valid
```

> 过程：自下而上的遍历整个晶格网络

---

updateCandidateSets()函数

> 由下一层的候选集集合CS_{l-1}生成l层候选集集合CS_l

- 当$|X| = l - 1$时，$C_l(X)$由$C_{l-1}(X^{'})$生成

> 当$B \in C_{l-1}(X^{'})$且B与X不相交，则$B \in C_l(X)$

- 当$1 \leq |X| \leq l - 1$时，$C_l(X)$由逐渐加入$Y \in C_{l-1}(X)$扩展而得到的U得到

> $|U| = |Y| + 1$
>
> $Y \in C_{l-1}(X), E \in R, U = YE$，其中 Y, E, X两两不相交

代码

Algorithm 5 ORDER: UPDATECANDIDATESETS

```
1:  procedure UPDATECANDIDATESETS(CS_{l−1})
2:      CS_l ← ∅
3:      for each C_{l−1}(X) ∈ CS_{l−1}
4:          C_l(X) ← ∅
5:          if |X| ≠ l − 1                                  ▷ extend C_{l−1}(X)
6:              for each Y ∈ C_{l−1}(X)
7:                  if X →_< Y ∈ valid
8:                      continue
9:                  for each U ∈ EXTEND (X, Y)
10:                     if |X| > 1
11:                         X′ ← MAXPREFIX(X)
12:                         if (U ∉ C_{l−1}(X′)) and
                            (∄ Q ∈ PREFIXES(U) s.t.
                                  X′ →_< Q ∈ valid)
13:                             continue
14:                     if U is not minimal
15:                         continue
16:                     add U to C_l(X)
17:         else                    ▷ |X| = l − 1: create new candidate set
18:             if X is minimal
19:                 for B ∈ C_{l−1}(MAXPREFIX(X))
20:                     if X and B are disjoint
21:                         add B to C_l(X)
22:         if C_l(X) ≠ ∅
23:             add C_l(X) to CS_l
24:     return CS_l
```

代码解读:

- 当$|X|! = l - 1$时
    - 若$X \to_< Y \in vaild$，则不用扩展Y，因为$X \to_< YW \in valid$，且YW不是最小,不用加入$C_l(X)$
        - $EXTEND(X, Y)$函数将Y拓展成U，$U = YE, E \in R$，需要X参数的原因是控制X, Y, E均不相交，该U会经过后续一系列的判断后再决定是不是要加入$C_l(X)$
            - **按10到13行的规则进行筛选，但是我没太弄懂**
                - 大致分析分析

- 若$U \notin C_{l-1}(X^{'})$,则$X^{'}V \to UW$的有效性已知，当$W=\emptyset, X^{'}V=X$时，$X \to_< U$有效性已知，不用将U加入$C_l(X)$
  - $U \notin C_{l-1}(X^{'})$有共5种情况
    1. U非最小
    2. U由于***swap***被剪枝
    3. U由于***uniqune***被剪枝
    4. U由于***merge-pruning***被剪枝
    5. $X \to_< T, T \in prefix(U)$（**为什么？**）
  - 1～4种情况不用将U加入候选集
  - 第5种需要。当第5种情况时，必定存在$X^{'} \to Q$，$Q \in prefix(U)$，（**为什么?**）
  - Line12：表示若不是第5种情况，则进行下一次循环，不把U加入候选集
- 最小的U才加入候选集$C_l(X)$

- 当$|X|=l-1$时
  - $C_l(X)$由$C_{l-1}(X^{'})$生成，当$B \in C_{l-1}(X^{'})$且B与X不相交，则$B \in C_l(X)$
- 最后将生成的$C_l(X)$加入到CS_l中，知道该层候选集集合填充完毕

> ***updateCandidateSets***函数生成的下一层的候选集集合CS_l中的每个$C_l(X)$，均是根据$C_l(X)$的准则而生成，还需要对其中的OD进行验证，以及对其中的$C_l(X)$中的Y进行修剪。

---

computeDependencies()函数

> 计算l层的节点X的候选集$C_l(X)$中有效OD,并对其中的Y进行删减

obtainCandidates()函数

前提知识：一个第n层的结点，可以产生$n-1$个候选OD

例：在第三层，结点ABC，可以产生$A \to_< BC, AB \to_< C$ 这2个候选OD

> obtainCandidates()函数将结点node,分为候选OD的左侧LHS和右侧RHS

从$C_l(X)$删除Y的规则

从 $C_l(X)$ 中删除Y的规则：

当从剪枝规则中，我们知道 $XV \to_< YW$ 这个式子的有效性时，才从 $C_l(X)$ 中删除 Y.

3种情况：

1° $X \to_< Y$ 有效，且 X 是 unique

可推 $XV \to_< YW$.

仅V, W可为空

2° $X \not\to_< Y$ (by swap)

可推 $XV \not\to_< YW$. 仅V, W可为空.

3° merge-pruning

规则如下：

X'是X最长前缀，若 $\exists B \in C_l(X')$，且B是从Y开头的list，即 $B = YA$，$|A|=1$.

则 可推 $XV \to_< YW$ 是否有效已知.

推理过程在上面红字部分.

merge-pruning

> 仅由于$merge$导致$X \nrightarrow_< Y$时，$XV \to_< YW$是否有效无法得知，可能有效也可能无效。只能推理出$XV \nrightarrow_< Y$，

文章中原话：

> Instead, assume we know in addition to $A \nrightarrow_< B$ being invalidated only by a *merge* that any $A \to_< BY, (Y \neq \emptyset)$ need not be checked. Then, we need not check any OD of the form $AX \nrightarrow_< BY$

翻译翻译：

> 然而，假如除了知道$A \nrightarrow_{<} B$是由于***merge***造成的，我们还知道任何$A \rightarrow_{<} BY,(Y \neq \emptyset)$ 均不需要检查，于是，我们不需要检查形如$AX \nrightarrow_{<} BY$格式的OD

<u>那么$A \rightarrow_{<} BY,(Y \neq \emptyset)$ 均不需要检查有两种情况</u>

1. $A \rightarrow_{<} BY,(Y \neq \emptyset)$

   $\because A \nrightarrow_{<} B$

   $\therefore AX \nrightarrow B$

   又$\because A \rightarrow_{<} BY$

   <u>不会证了</u>

2. $A \nrightarrow_{<} BY,(Y \neq \emptyset)$

   $\because A \nrightarrow_{<} B,$

   $\therefore AX \nrightarrow_{<} B$

   又 $\because A \nrightarrow_{<} BY$

   $\therefore AV \nrightarrow_{<} BY$

   不需要检测上述格式得证

> If the OD $maxPrefix(X) \rightarrow_{<} Y$was invalidated only by a *merge* (this is knowledge gained in level $l-1$), we know that $X \nrightarrow_{<} Y$(and also,$XV \nrightarrow_{<} Y$ for any V).Then,if we cannot find in $C_l(maxPrefix(X))$any V that starts with Y, we need not examine any $X \rightarrow_{<} YZ$, because it is either invalid or not minimal.

证：

$X^{'} \nrightarrow_{<} Y$已知，$X^{'} \rightarrow_{<} YE(V)$的有效性已知（$\because \nexists V \in C_l(X^{'})$, 且$V = YE$）

> $\because |V| + |X^{'}| = l, |Y| + |X| = l$
>
> $\therefore |E| = 1$

$\therefore X^{'}M \rightarrow_{<} VN$有效性已知（由于$\nexists V \in C_l(X^{'})$）

令$X^{'}M = XA, VN = YEN = YZ(V = YE, Z = EN)$时，上式可以表示为$XA \rightarrow_{<} YZ$,其有效性已知

$\therefore$可当满足上述条件，可以将Y从$C_l(X)$中删除

同时当$A = \emptyset$时，上面的那段英文得证：

- 要么$X \nrightarrow_< YZ$
- 要么$X \rightarrow_< YZ$，但非最小OD。最小OD为$X' \rightarrow V, X' \subseteq X, V \subset YZ$

伪代码：

Algorithm 4 ORDER: COMPUTEDEPENDENCIES

```
procedure COMPUTEDEPENDENCIES(L_l, CS_l)
    for each node ∈ L_l
        for each X, Y ∈ OBTAINCANDIDATES(node)
            if Y ∉ C_l(X)
                continue
            if X →< Y is valid
                add X →< Y to valid
                if X is unique
                    remove Y from C_l(X)
            else if X ↛< Y invalidated by swap
                remove Y from C_l(X)
    ▹ merge-pruning:
    for each C_l(X) ∈ CS_l with |X| > 1
        for each Y ∈ C_l(X)
            X' ← MAXPREFIX(X)
            if X' ↛< Y invalidated only by merge
                if ∄ V ∈ C_l(X') MAXPREFIX(V) = Y
                    remove Y from C_l(X)
```

代码解析：

- 遍历该层所有当结点，以得到所有的X，Y
  - 当$Y \notin C_l(X)$时，不进行检查
  - 当$X \rightarrow_< Y$有效时，将$X \rightarrow_< Y$加入$valid$集合
    - 当X时$unique$时，从$C_l(X)$中删除Y。因为可推$XV \rightarrow_< YW$(仅V，W可为$\emptyset$)必定有效，根据从$C_l(X)$中的删除规则，将Y删除
    - 当由于$swap$导致$X \nrightarrow_< Y$时，从$C_l(X)$中删除Y。因为可推$XV \nrightarrow YW$(仅V,W可为$\emptyset$)，从而删除Y
- 遍历完后，再对整层的结点进行***merge-pruning***(因为merge-pruing过程中需要用到整层的CS_l，比如在进行$Y \in C_l(X)$遍历的时候需要用到C_l（X')中的元素进行判断)。

prune()函数

> 从晶格网络中删除结点的规则：只有确认该结点不会再生成包含还不知道有效性的OD候选集的结点
>
> 结点n的$|n|-1$个候选集全为空，表示以n作为前缀的更大的结点不会产生了，我们则将其从晶格网络中删掉

代码：

Algorithm 6 Function PRUNE

```
procedure PRUNE(L_l, CS_l)
    for each node ∈ L_l
        allEmpty ← false
        for each prefix ∈ PREFIXES (node)
            if C_l(prefix) ≠ ∅
                allEmpty ← false
                break
            else
                allEmpty ← true
        if allEmpty = true
            remove node from L_l
    for each C_l(X) ∈ CS_l
        if C_l(X) = ∅
            remove C_l(X) from CS_l
```

---

generateNextLevel()函数

> 1. 将两个大小为l的结点分成2部分(X, Y),其中每个结点的$|X| = l - 1$,且相同，Y则不同
> 2. 再将这2个结点结合在一起生成大小为$l + 1$的结点
> 3. 为下一层产生空的候选集

prefixBlocks()函数

> 将大小为l的结点分成有着相同的大小为$l - 1$的前缀的结点列表$prefixBlock$，每个$list$中所有节点的$l - 1$前缀相同

代码：

Algorithm 7 ORDER: GENERATENEXTLEVEL

```
function GENERATENEXTLEVEL(L_l)
    L_{l+1} ← Ø
    for each prefixBlock ∈ PREFIXBLOCKS (L_l)
        for each node ∈ prefixBlock
            for each joinNode ∈ prefixBlock
                if node = joinNode
                    continue
                joinedNode ← JOIN (node, joinNode)
                add joinedNode to L_{l+1}
    for each node ∈ L_l
        C_l(node) ← Ø                          ▷ needed in the next level
        add C_l(node) to CS_l
    return L_{l+1}
```