



# WFST解码器

洪青阳



# 纲要

9.1 基于动态网络的Viterbi解码

9.2 WFST理论

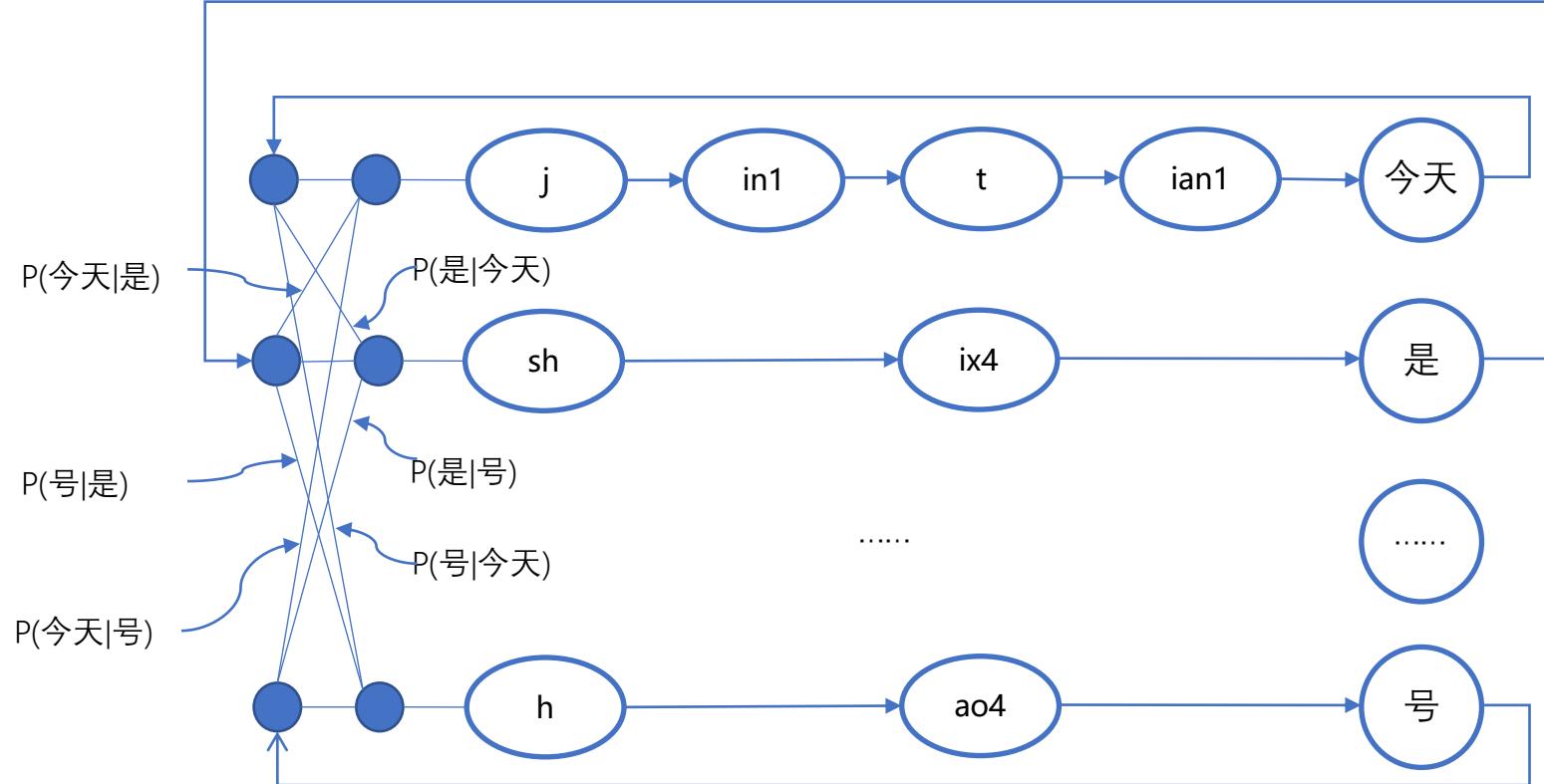
9.3 HCLG构建

9.4 WFST的Viterbi解码

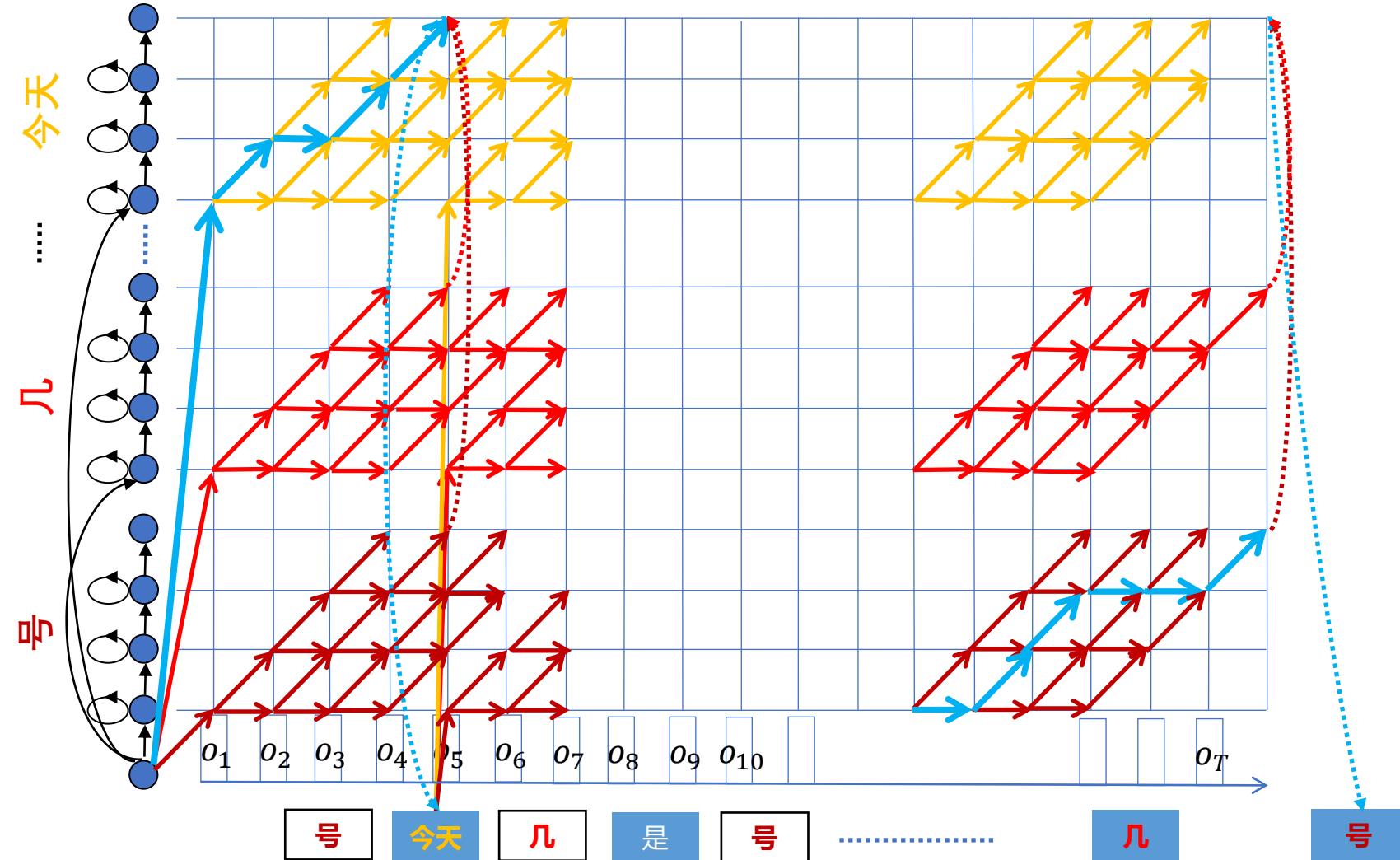
9.5 Lattice解码

9.6 本章小结

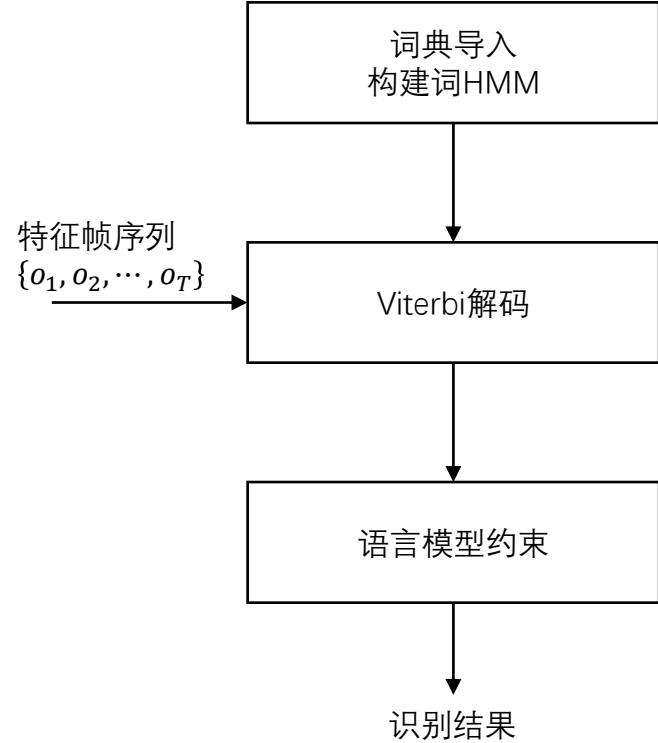
# 9.1 基于动态网络的Viterbi解码



# 9.1 基于动态网络的Viterbi解码

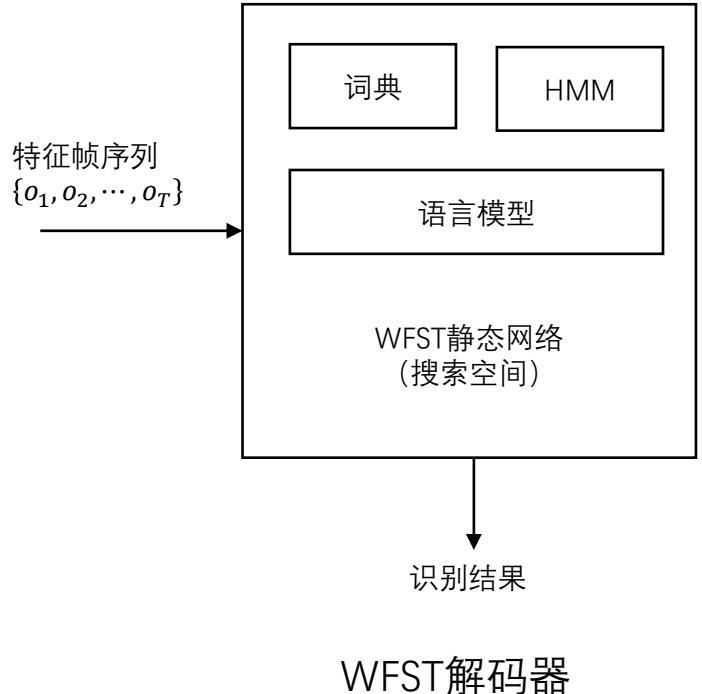


# 9.1 基于动态网络的Viterbi解码

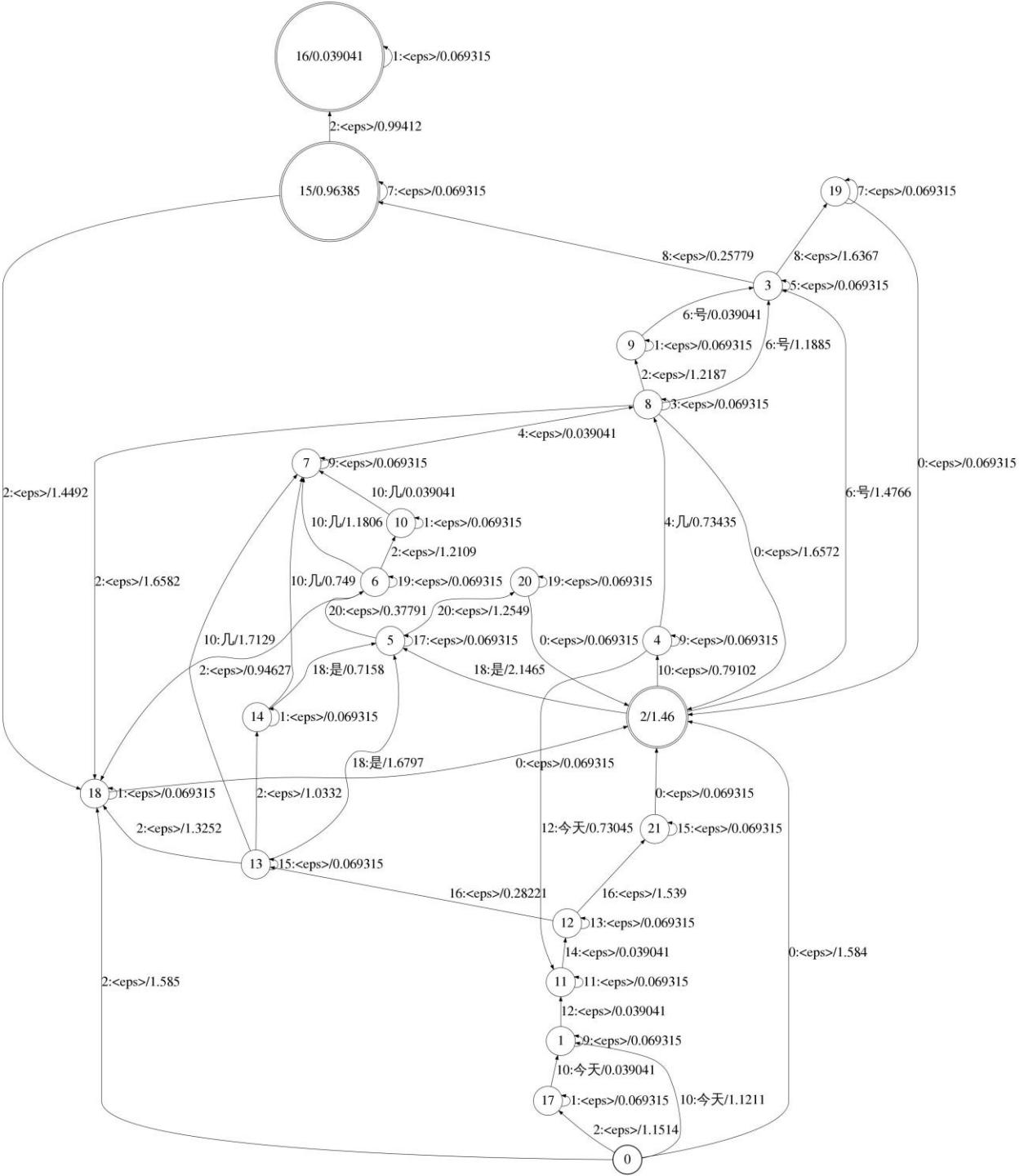


传统解码器

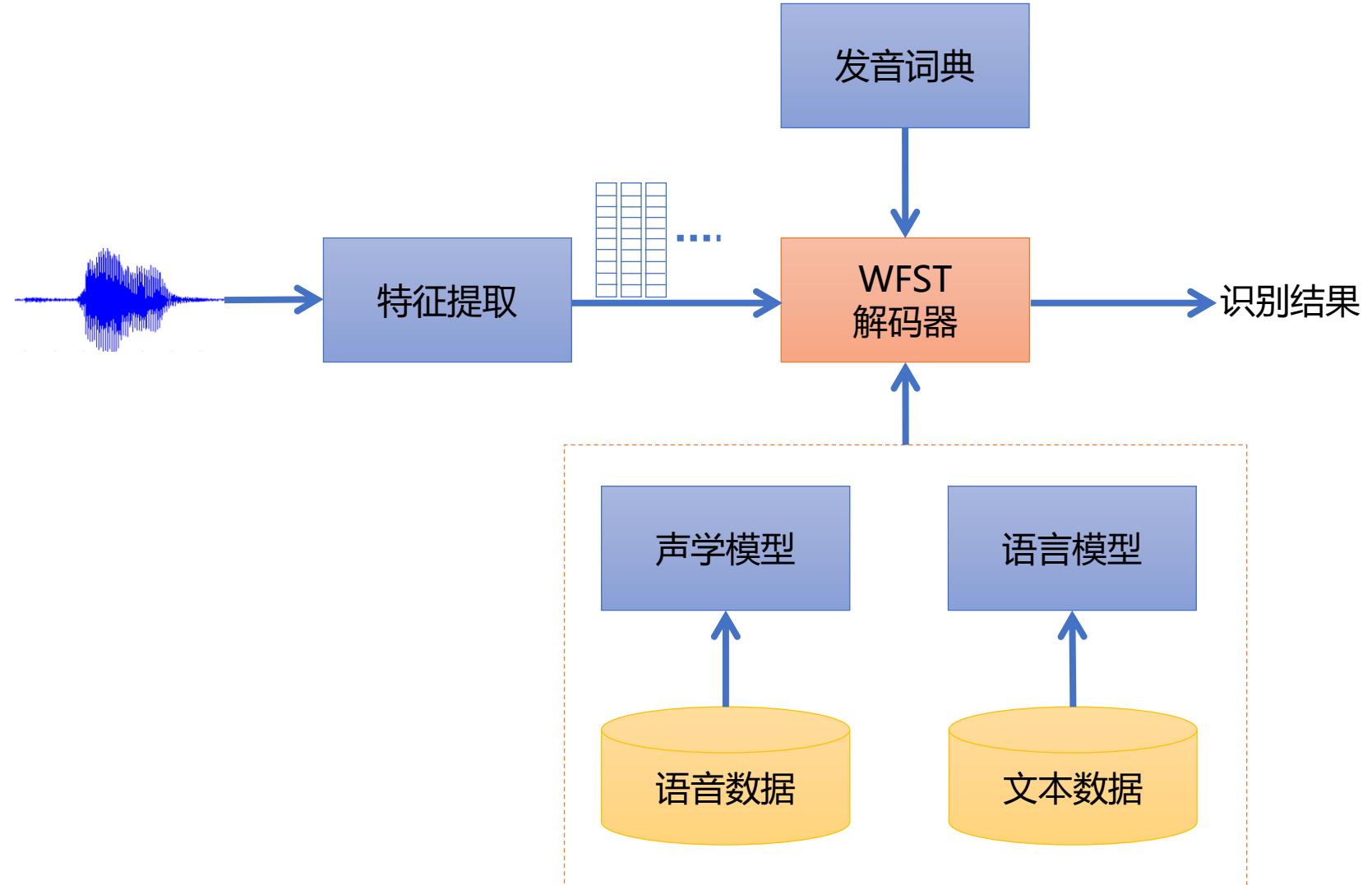
# WFST静态网络



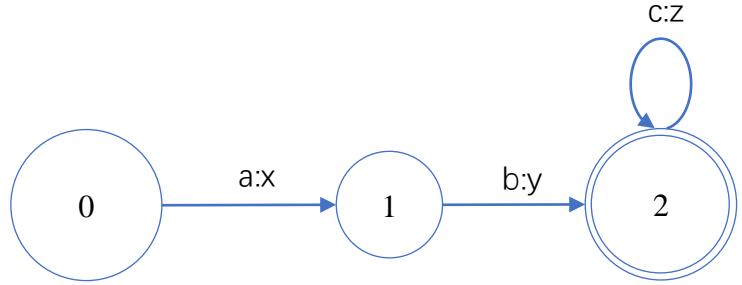
# HCLG



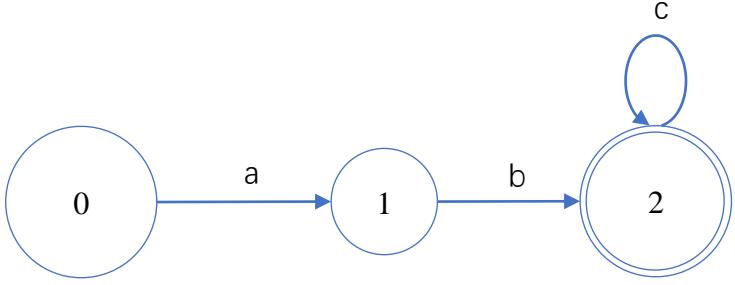
# 基于WFST的语音识别系统



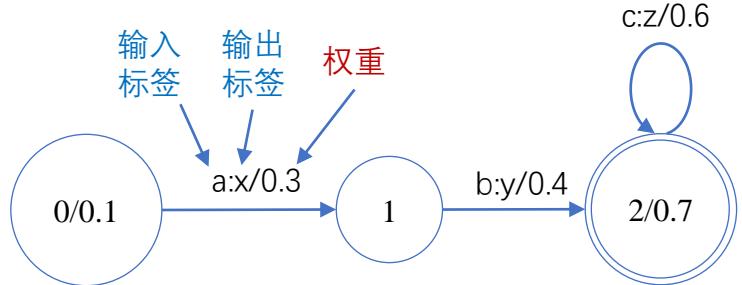
# 9.2 WFST理论



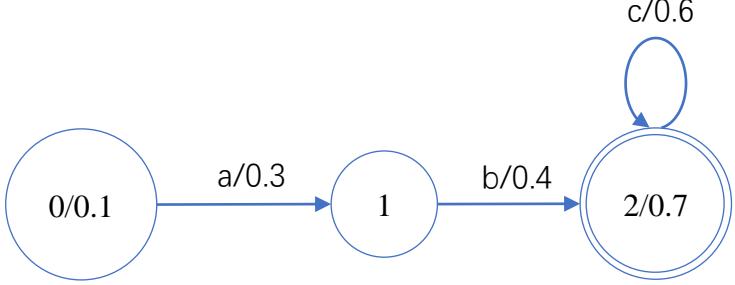
(a) Finite State Transducer (FST)



(b) Finite State Acceptor (FSA)



(c) Weighted Finite State Transducer (WFST)

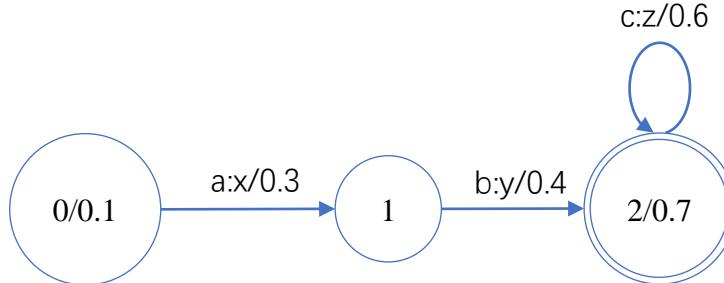


(d) Weighted Finite State Acceptor (WFSA)

# 9.2 WFST理论—WFST八元组

定义在数据集 $K$ 上的WFST用八元组来表示,

$$T = (\Sigma, \Delta, Q, I, F, E, \lambda, \rho)$$



符号	含义	图例
$\Sigma$	有限的输入集	{ $a, b, c$ }
$\Delta$	有限的输出集	{ $x, y, z$ }
$Q$	状态集	{0,1,2}
$I$	初始状态集	{0}
$F$	结束状态集	{2}
$E$	状态转移集	{(0, a, x, 0.3, 1), (1, b, y, 0.4, 2), (2, c, z, 0.6, 2)}
$\lambda$	初始状态权重	0.1
$\rho$	结束状态权重	0.7



## 9.2 WFST理论—WFST半环及操作

半环	集合	$\oplus$	$\otimes$	'0'	'1'
Boolean	{0,1}	$\vee$	$\wedge$	0	1
Probability	$\mathbb{R}_+$	+	$\times$	0	1
Log	$\mathbb{R} \cup \{-\infty, +\infty\}$	$\oplus_{\log}$	+	$+\infty$	0
Tropical	$\mathbb{R} \cup \{-\infty, +\infty\}$	min	+	$+\infty$	0

其中， $\oplus_{\log}$ 定义为 $x \oplus_{\log} y = -\log(e^{-x} + e^{-y})$

WFST是基于半环结构。半环可以用 $(\oplus, \otimes, '0', '1')$ 来表示，其中 $\oplus$ 和 $\otimes$ 是集合的二元操作，分别对应min和+，而'0'和'1'是零元素和幺元素。



## 9.2 WFST理论—WFST半环及操作

对于状态转移 $e \in E$ 来说，可以定义 $p[e]$ 为 $e$ 的起始状态， $n[e]$ 为 $e$ 的到达状态； $i[e]$ 为输入的标签， $o[e]$ 为输出的标签， $w[e]$ 是最终的转移权重。那么，对于连续的状态转移路径来说， $\pi = e_1 \cdots e_k$ ，且

$$\begin{aligned}n[e_{i-1}] &= p[e_i], i = 2, \dots, k \\p[\pi] &= p[e_1], n[\pi] = n[e_k]\end{aligned}$$

对于集合的二元运算，则整条路径权重为

$$w[\pi] = w[e_1] \otimes \cdots \otimes w[e_k]$$

对于多条有限路径 $\pi \in R$ ，则有

$$w[R] = \bigoplus w_{\pi \in R}[\pi]$$

综上，WFST最终的运算( $x$ 输入到 $y$ 输出)为

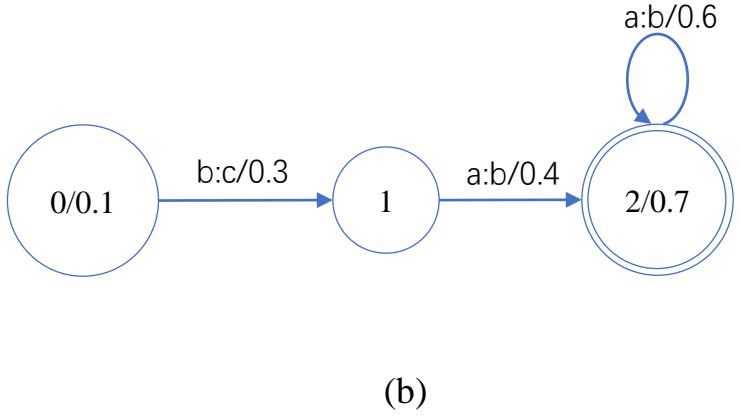
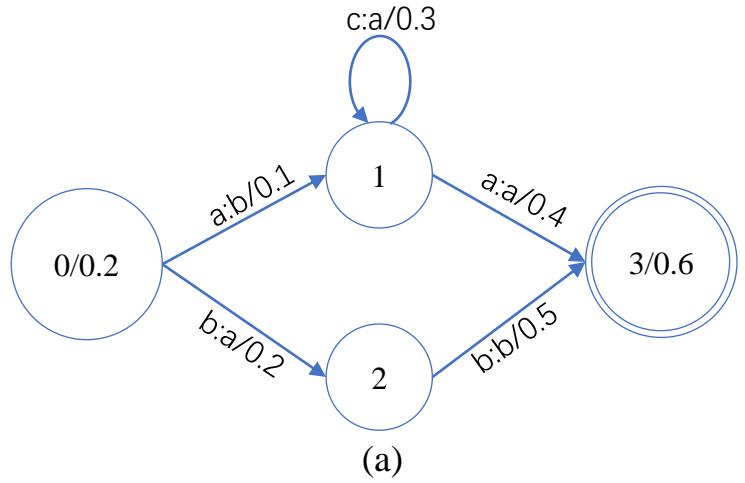
$$T(x, y) = \bigoplus_{x \in P(I, x, y, F)} \lambda[p[\pi]] \otimes w[\pi] \otimes \rho[n[\pi]]$$



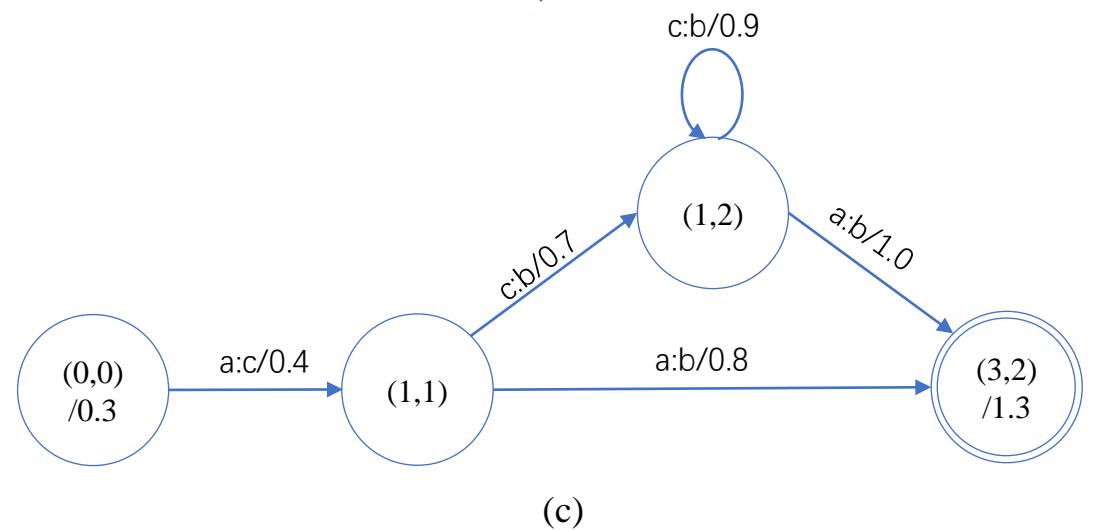
# WFST三大算法

- Composition(合并)
- Determinization(确定化)
- Minimization(最小化)

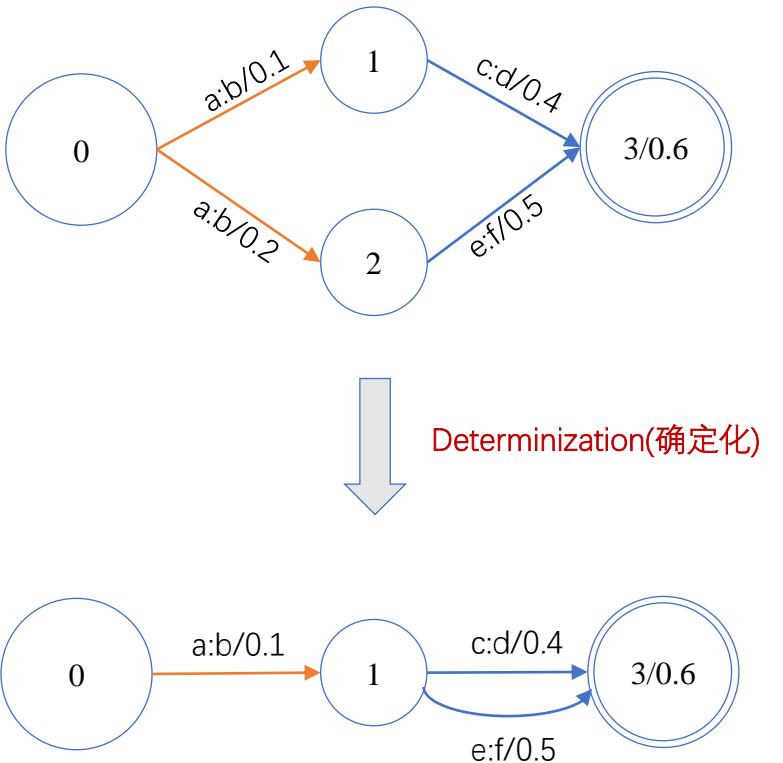
# Composition(合并)



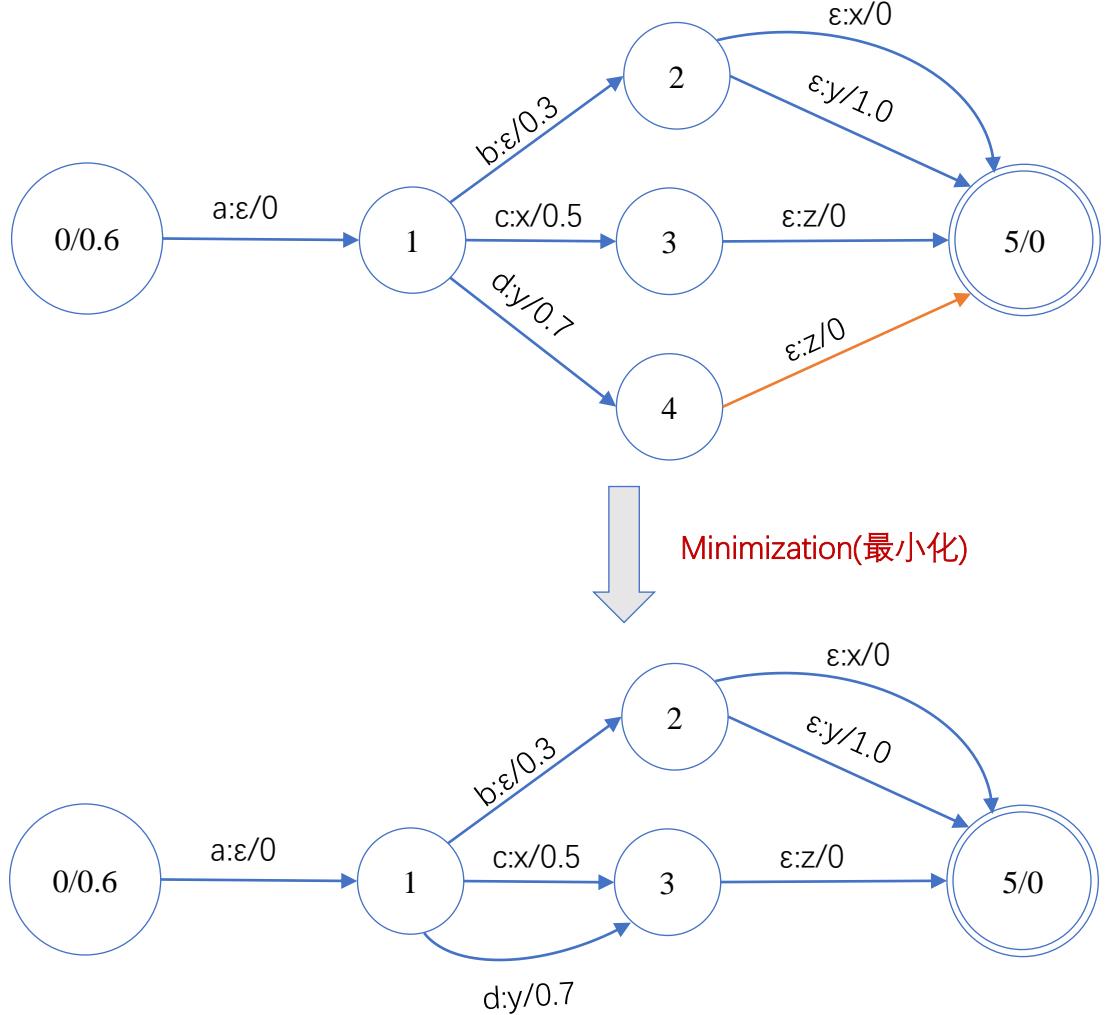
Composition(合并)



# Determinization(确定化)



# Minimization(最小化)





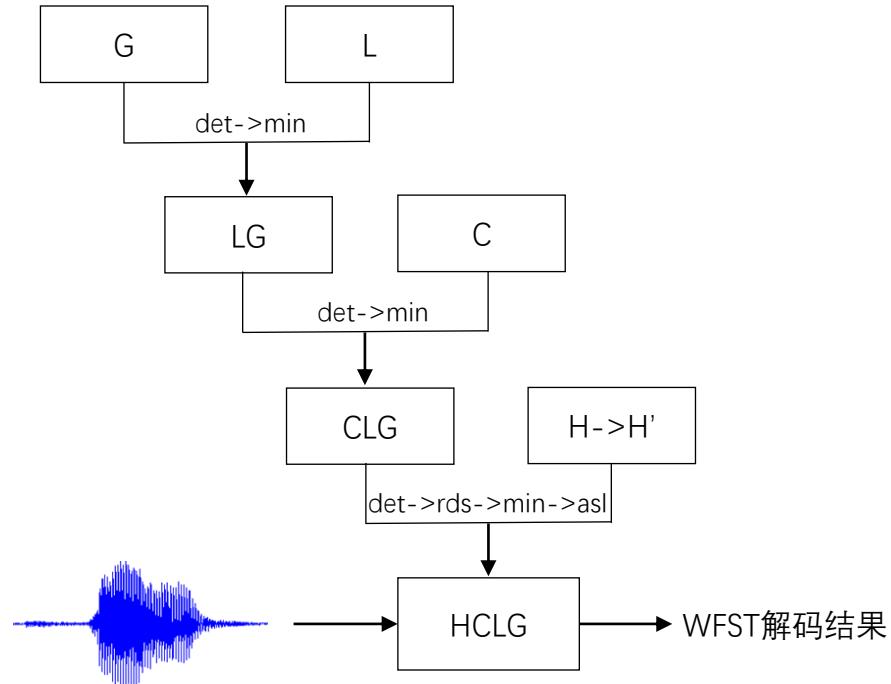
# 9.3 HCLG构建

组成	转换器	输入序列	输出序列
H	HMM	HMM的转移-id	单音子/三音子(triphone)
C	上下文相关	单音子/三音子	单音子(monophone)
L	发音词典	单音子	词(word)
G	语言模型	词	词(word)

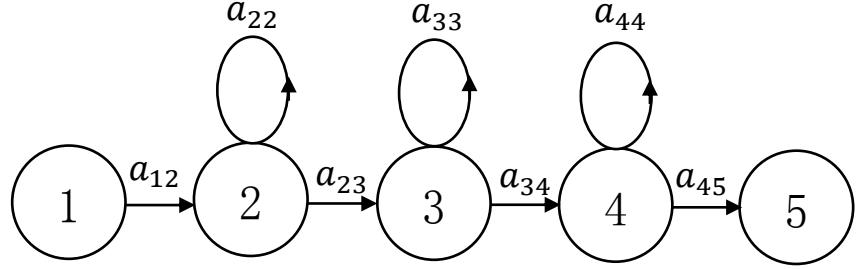
# 9.3 HCLG构建

$$HCLG = \text{asl} \left( \min \left( \text{rds} \left( \text{det} \left( H' \circ \min \left( \text{det} \left( C \circ \left( \min(\text{det}(L \circ G)) \right) \right) \right) \right) \right) \right) \right)$$

其中**asl**代表添加自环, **rds**代表移除歧义符号,  $H'$ 则代表没有自环的HMM, **o**代表composition操作, **det**代表determinization操作, **min**代表minimization操作。



# 9.3.1 H的构建



Transition-state 1: phone = sil hmm-state = 2 pdf = 0

Transition-id = 1 p =  $a_{22}$  [self-loop]

Transition-id = 2 p =  $a_{23}$  [2 -> 3]

Transition-state 2: phone = sil hmm-state = 3 pdf = 1

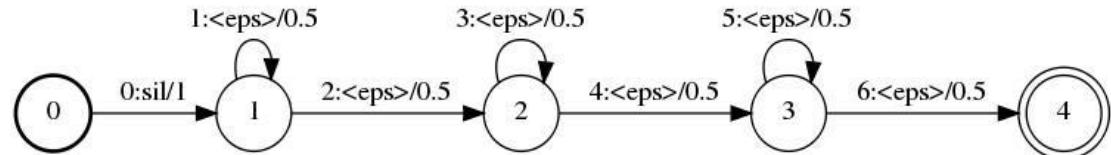
Transition-id = 3 p =  $a_{33}$  [self-loop]

Transition-id = 4 p =  $a_{34}$  [3 -> 4]

Transition-state 3: phone = sil hmm-state = 4 pdf = 2

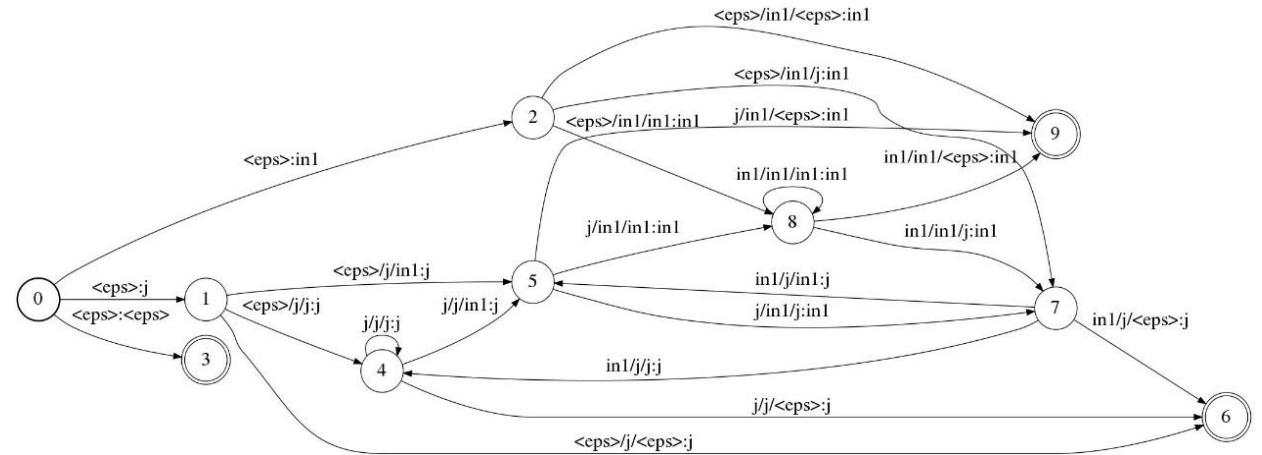
Transition-id = 5 p =  $a_{44}$  [self-loop]

Transition-id = 6 p =  $a_{45}$  [4 -> 5]



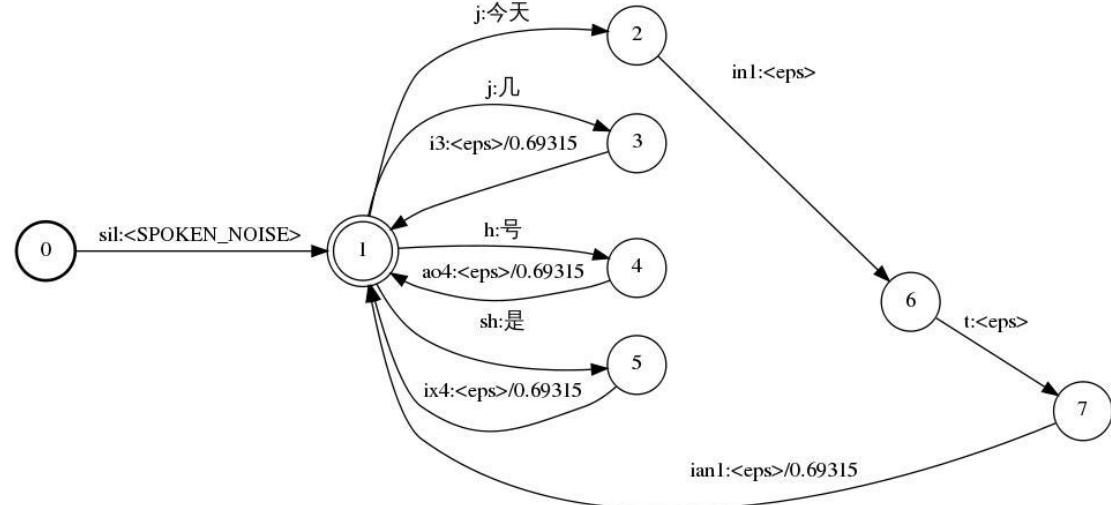
# 9.3.2 C的构建

$\langle \text{eps} \rangle / j/j$	j
$\langle \text{eps} \rangle / j/in1$	j
$\langle \text{eps} \rangle / j/\langle \text{eps} \rangle$	j
$\langle \text{eps} \rangle / in1/j$	in1
$\langle \text{eps} \rangle / in1/in1$	in1
$\langle \text{eps} \rangle / in1/\langle \text{eps} \rangle$	in1
j/j/j	j
j/j/in1	j
j/j/\langle \text{eps} \rangle	j
j/in1/j	in1
j/in1/in1	in1
j/in1/\langle \text{eps} \rangle	in1
in1/j/j	j
in1/j/in1	j
in1/j/\langle \text{eps} \rangle	j
in1/in1/j	in1
in1/in1/in1	in1
in1/in1/\langle \text{eps} \rangle	in1



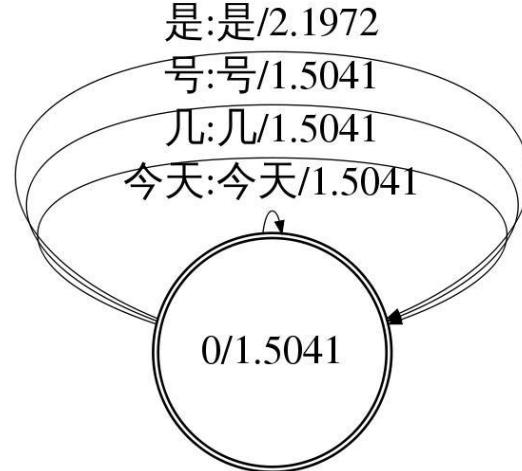
## 9.3.3 L的构建

词	音素
<SPOKEN_NOISE>	sil
今天	j in1 t ian1
是	sh ix4
几	j i3
号	h ao4



## 9.3.4 G的构建

```
\data\  
ngram 1=6  
  
\1-grams:  
-0.6532125      </s>  
-99              <s>  
-0.6532125      今天  
-0.6532125      几  
-0.6532125      号  
-0.9542425      是  
  
\end\
```



$$w = -\ln(10^L)$$

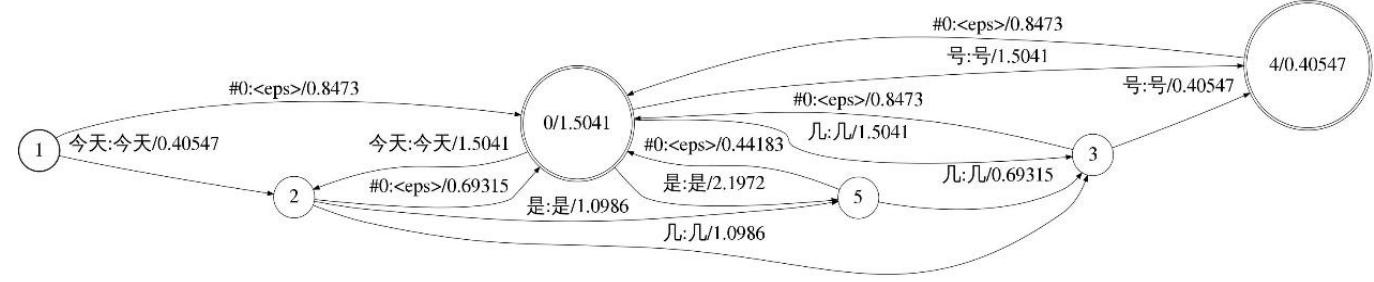
如1-gram里面的“今天”对数概率为 $-0.6532125$ , 则权重为 $-\ln(10^{-(-0.6532125)})$ , 其值为1.5041。

# 9.3.4 G的构建

```
\data\
ngram 1=6
ngram 2=6

\1-grams:
-0.6532125      </s>
-99               <s>      -0.3679768
-0.6532125      今天     -0.30103
-0.6532125      几       -0.3679768
-0.6532125      号       -0.3679768
-0.9542425      是       -0.1918855

\2-grams:
-0.1760913      <s> 今天
-0.4771213      今天 几
-0.4771213      今天 是
-0.1760913      几 号
-0.1760913      号 </s>
-0.30103         是 几
\end\
```

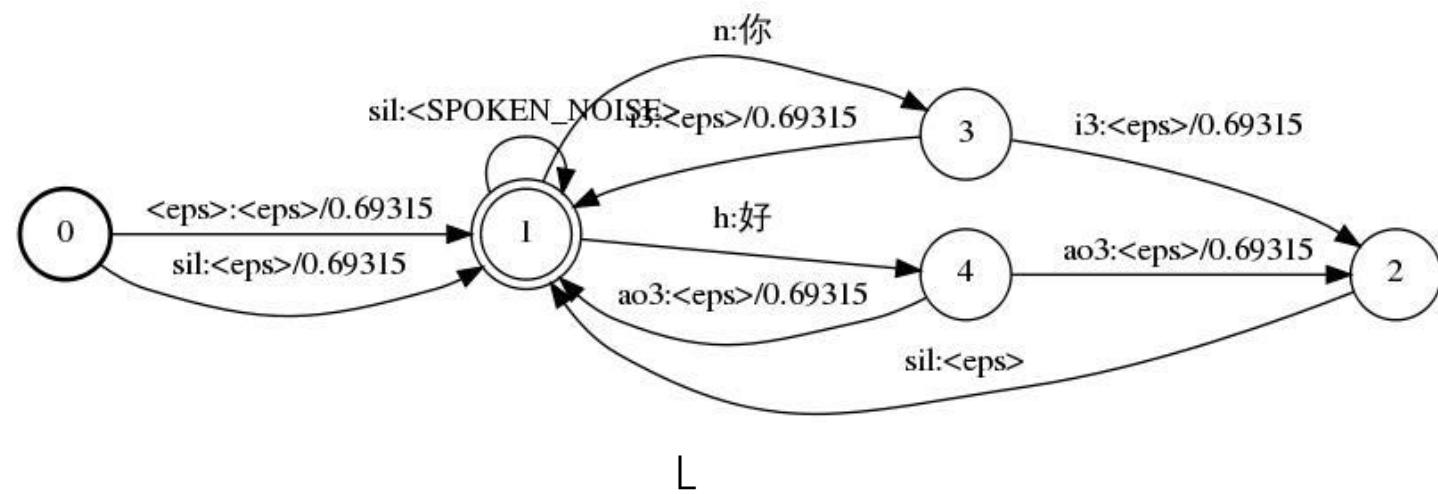
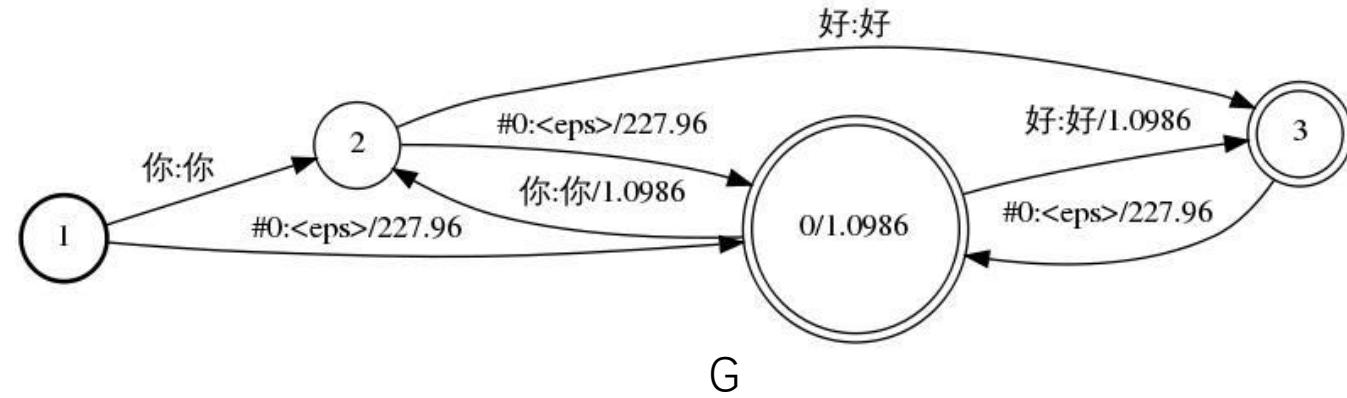


节点1表示句子起始 $<\text{s}>$ ，节点2表示“今天”，从节点1到2的转移弧为“今天：今天 /0.40547”，0.40547由 $-\ln(10^{-0.1760913})$ 计算得到。

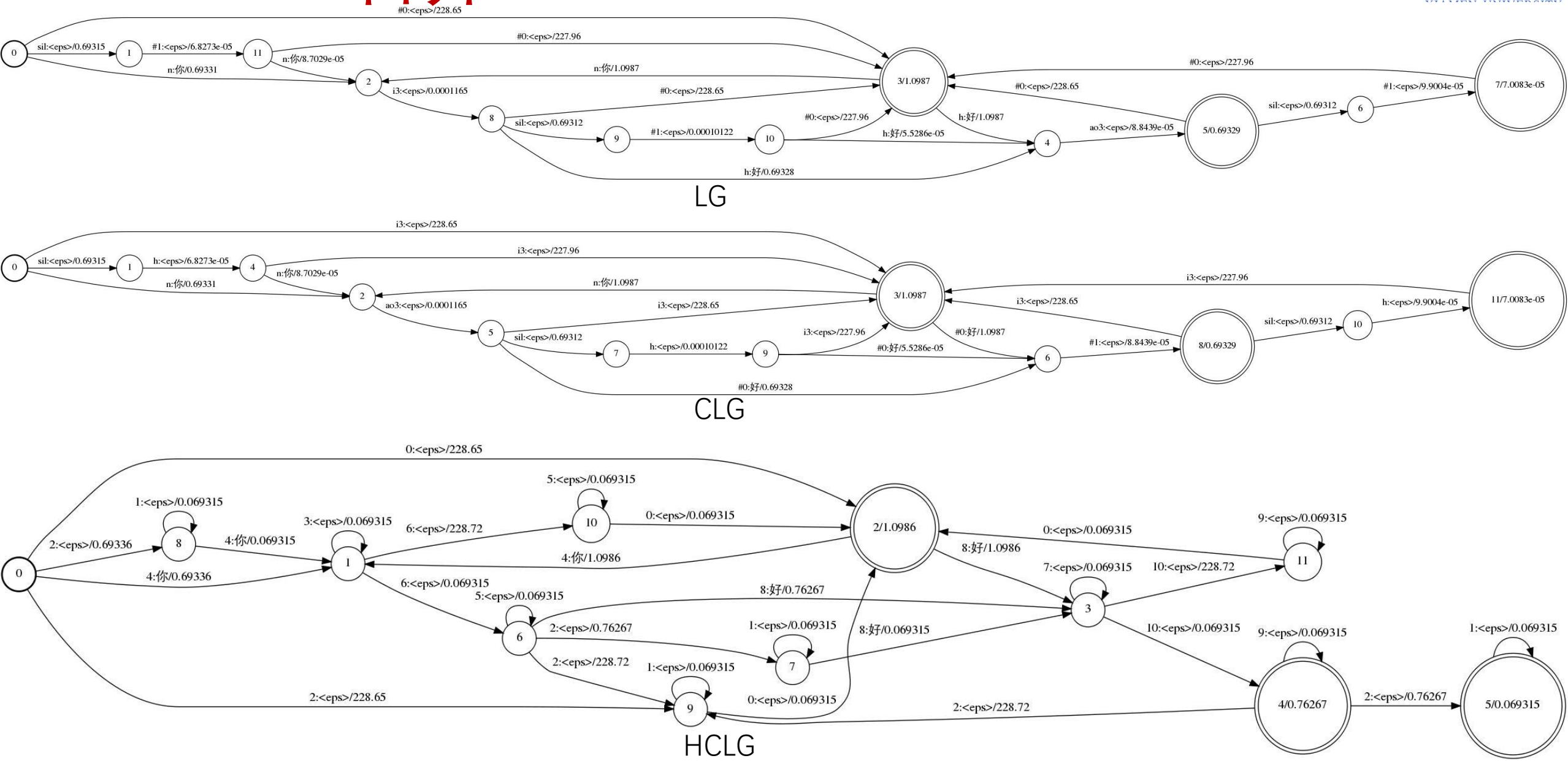
如果 $w_1$ 和 $w_2$ 两个词的组合不存在，则 $w_1$ 和 $w_2$ 之间的条件概率由 $w_1$ 回退概率和 $w_2$ 概率相乘得到。例如，图中节点0表示回退状态，节点1到节点0的转移弧权重为0.8473，对应 $<\text{s}>$ 的回退概率-0.3679768，即0.8473由 $-\ln(10^{-0.3679768})$ 计算得到。

节点2到节点0的转移弧权重为0.69315，对应“今天”的回退概率-0.30103，即0.69315由 $-\ln(10^{-0.30103})$ 计算得到。

## 9.3.5 HCLG合并



# 9.3.5 HCLG合并

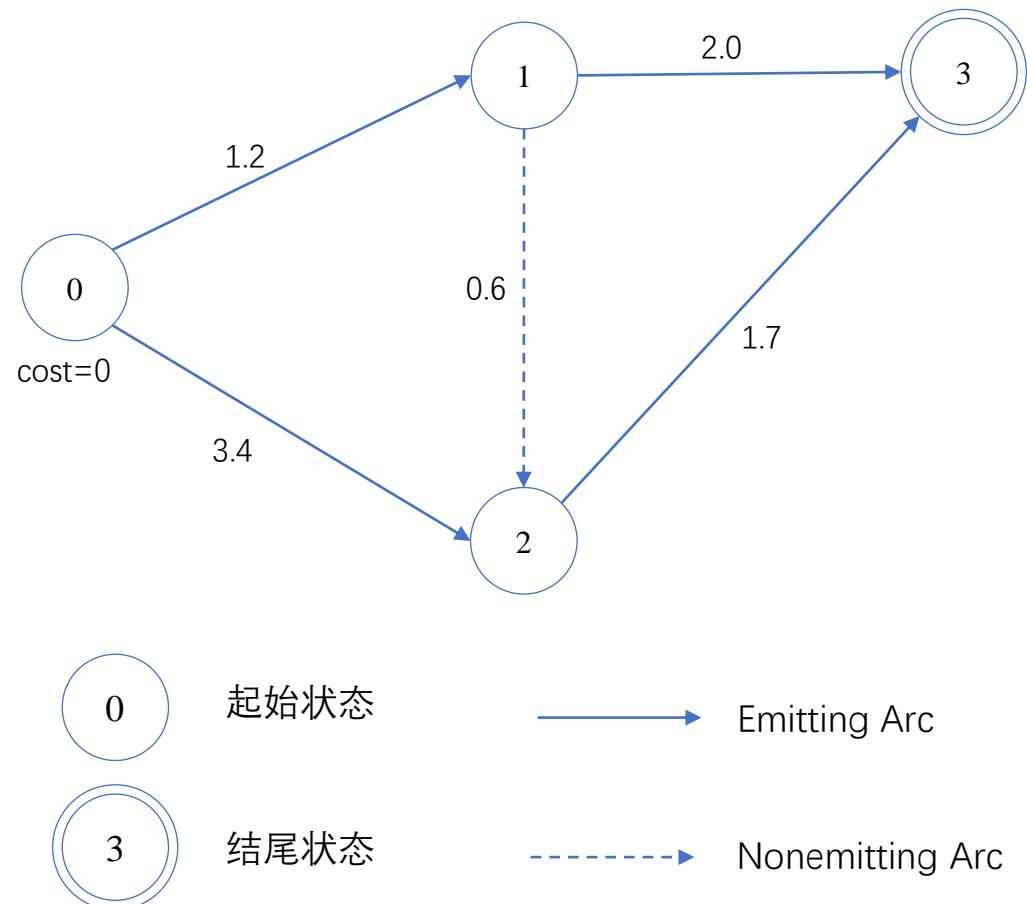


## 9.4 WFST的Viterbi解码

- WFST解码本质上也是Viterbi解码，根据输入的特征序列，进行帧同步对齐，寻求最佳状态序列。
- 注意这里的状态不是HMM状态，而是HCLG的状态节点，所遍历状态节点之间的衔接，可能是产生观察值的转移弧（如状态 $S_1$ 和 $S_2$ 、 $S_2$ 和 $S_3$ 之间的实线），也可能不产生观察值的转移弧（如状态 $S_4$ 和 $S_5$ 之间的虚线）。



# Viterbi算法（帧同步）



```
HashList<StatId, Token*> prev_toks, cur_toks;  
frame = 0;  
while(frame < LastFrame)  
{  
    Swap(prev_toks, cur_toks);  
    double weight_cutoff = ProcessEmitting(frame);  
    ProcessNonemitting(weight_cutoff);  
    frame++;  
}
```



# ProcessEmitting函数

## ProcessEmitting:

```
for all tokens old_tok in prev_toks
{
    u = old_tok.key;
    for all emitting arc (u, v)
    {
        cost = old_tok.cost;
        cost += arc.weight + acoustic_cost(frame);
        Add/Replace new Token for v if cost is lower;
    }
}
```



# ProcessNonemitting函数

## ProcessNonemitting:

```
Add all tokens in cur_toks in a queue q;  
while (!queue_.empty())  
{  
    u = q.pop();  
    for all nonemitting arc (u, v)  
    {  
        cost = old_tok.cost;  
        cost += arc.weight + acoustic_cost(frame);  
        Add/Replace new Token for v if cost is lower;  
        Add new token into q;  
    }  
}
```



# Token定义

```
struct Arc {  
    int ilabel; //转移弧的输入标签  
    int olabel; //转移弧的输出标签  
    float weight; //转移弧上的权重  
    int nextstate; //转移弧连接的下一个状态  
}
```

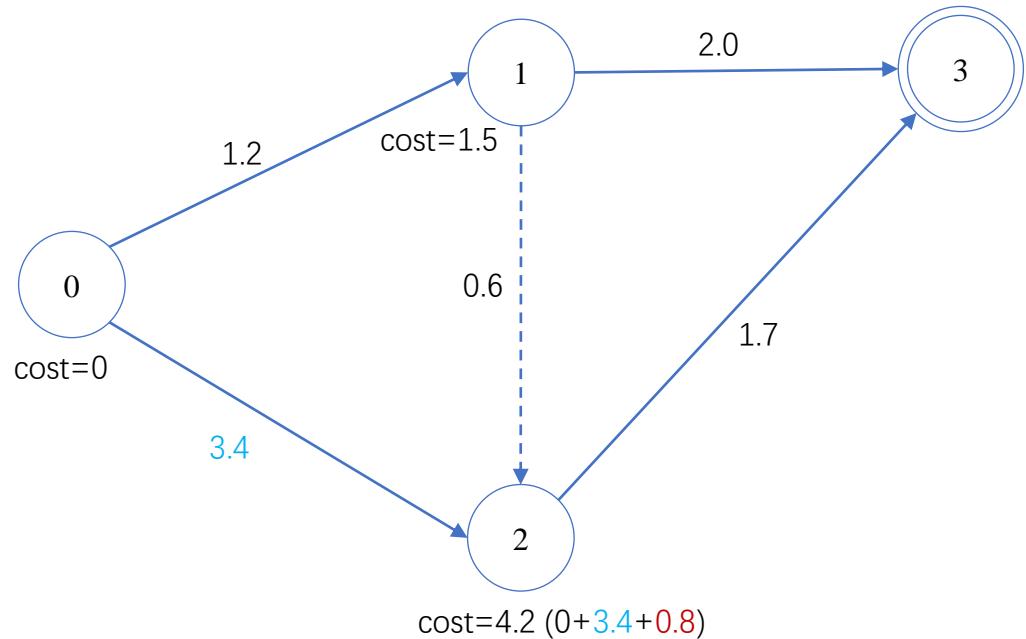
```
class Token {  
    Arc arc; //与Token对应的转移弧  
    Token *prev; //解码路径上一个Token指针  
    int32 ref_count; //后续关联的Token数量  
    double cost; //累计代价  
}
```



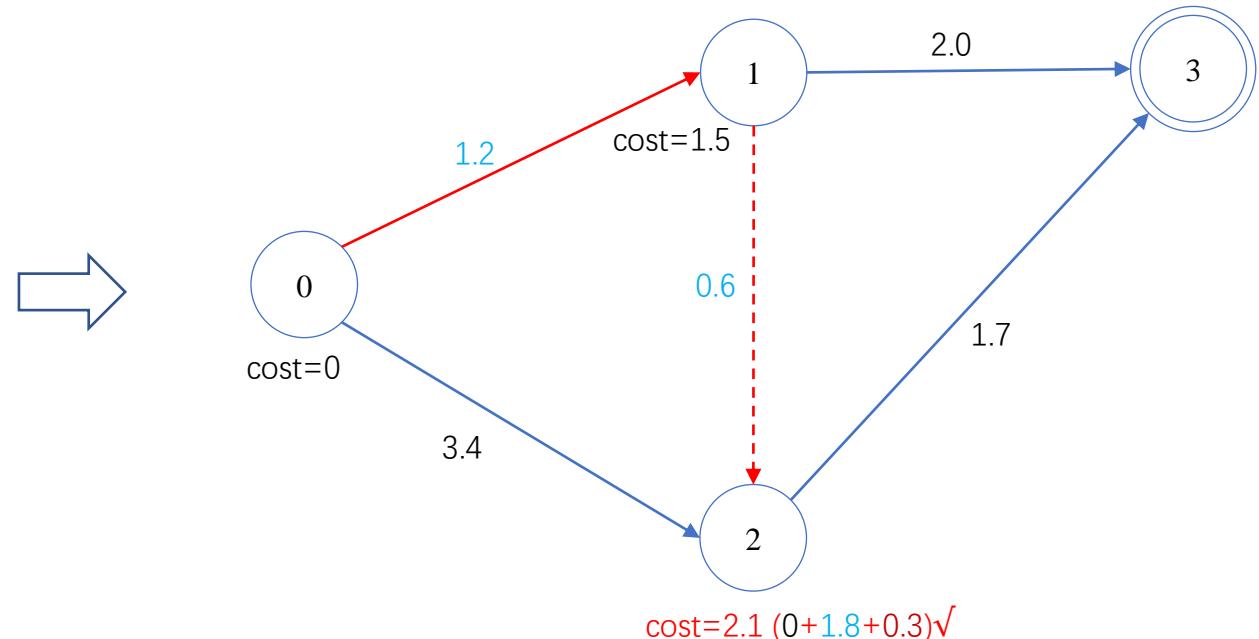
# 基于Token的Viterbi解码

- 类似HMM的解码过程，WFST的Viterbi解码也是逐帧推进，分别计算每帧的声学得分，然后结合转移弧的权重，得到每个时刻扩展路径的累计代价，这些代价用Token的cost保存。
- WFST的Viterbi解码通过对比指向同一个状态的不同路径的由Token（该Token与状态节点关联，如果状态节点还没有Token，则创建一个新的Token）保存的累计代价（cost），选择值更小的并更新Token信息。
- Viterbi算法的每个状态最多只有一个Token，如果有条路径到某个状态，则Token可能存在冲突，根据值更小原则保持或进行替代。

# Viterbi算法的Token替代 (cost有变化)

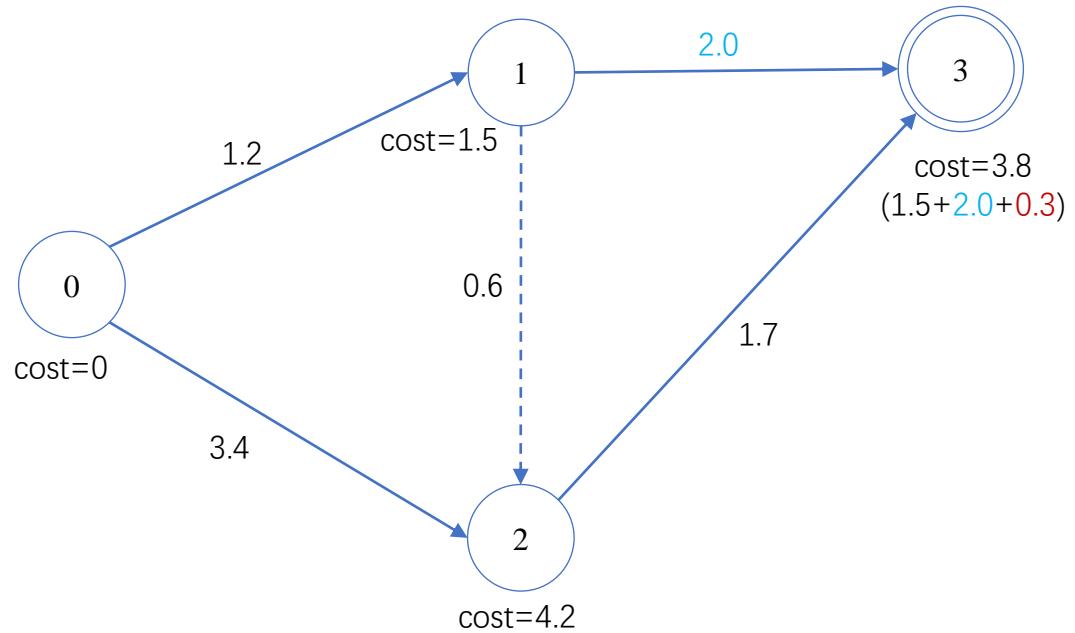


(a) 首次路径: 0->2  
graph\_cost=3.4, acoustic\_cost=0.8 (另外计算)

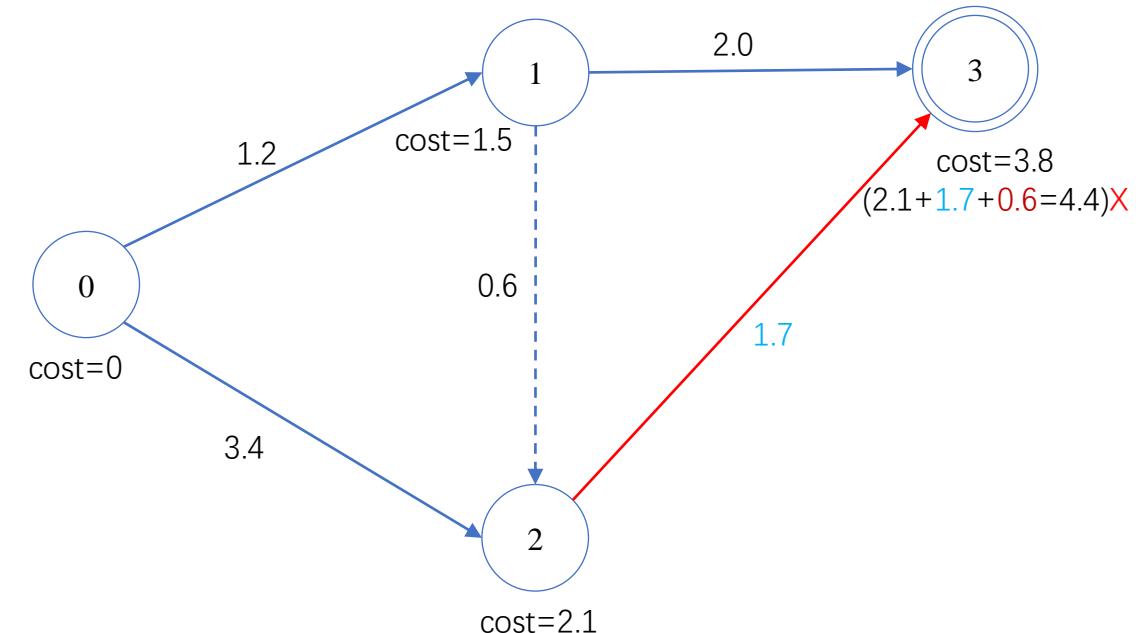


(b) 竞争路径: 0->1->2  
graph\_cost=1.2+0.6, acoustic\_cost=0.3 (0->1)

# Viterbi算法的Token替代 (cost无变化)

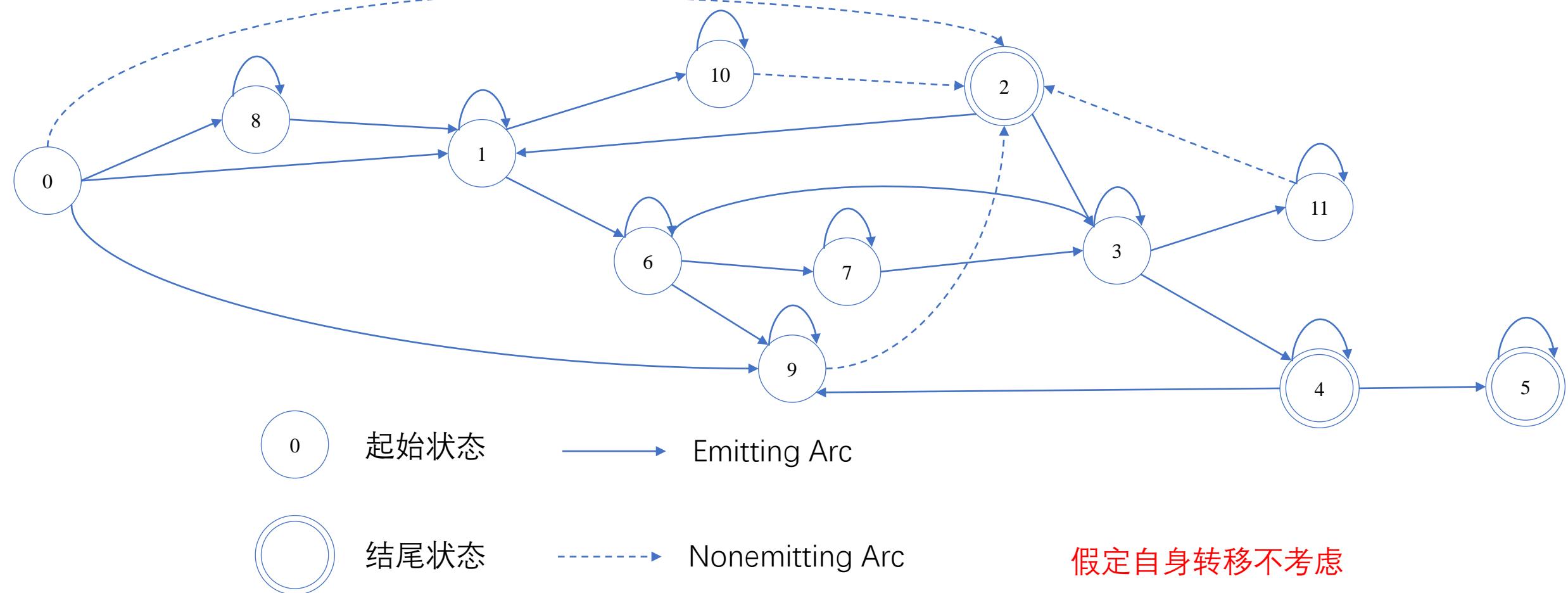


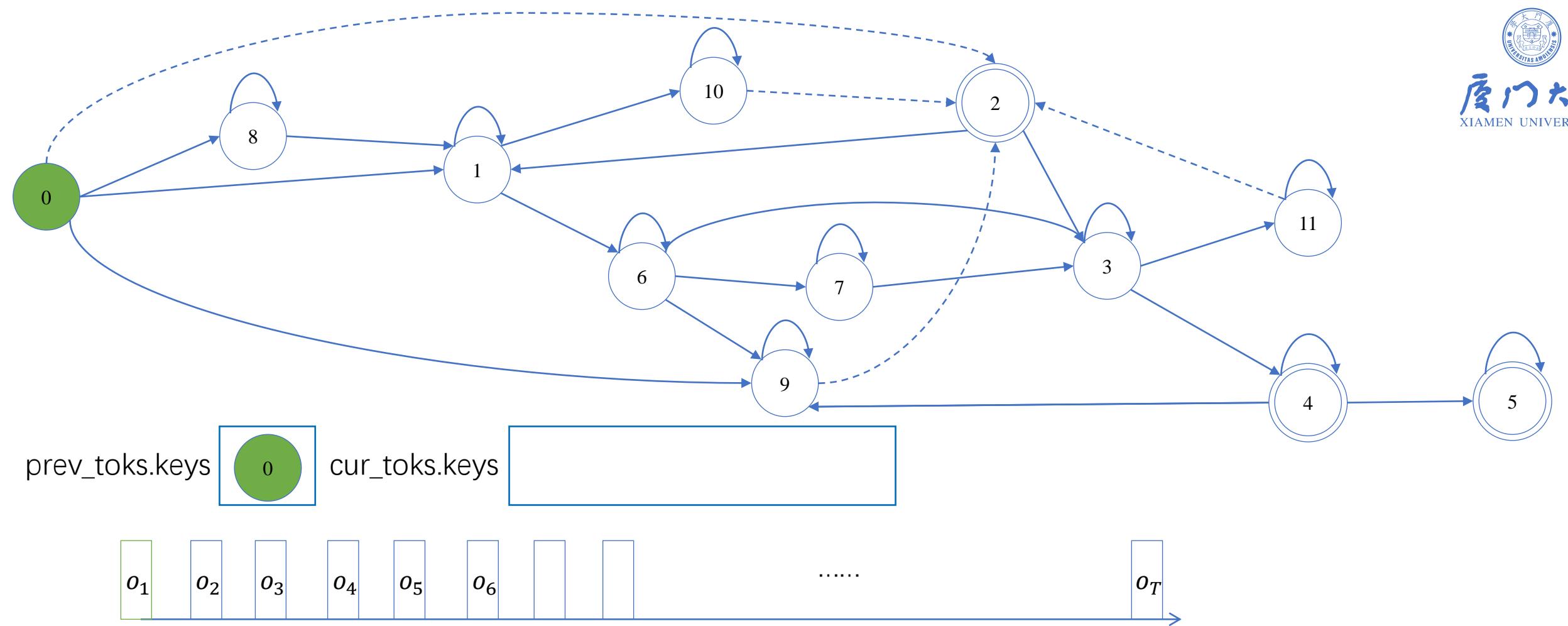
(a) 首次路径: 1->3  
graph\_cost=2.0, acoustic\_cost=0.3



(b) 竞争路径: 2->3  
graph\_cost=1.7, acoustic\_cost=0.6

## HCLG网络

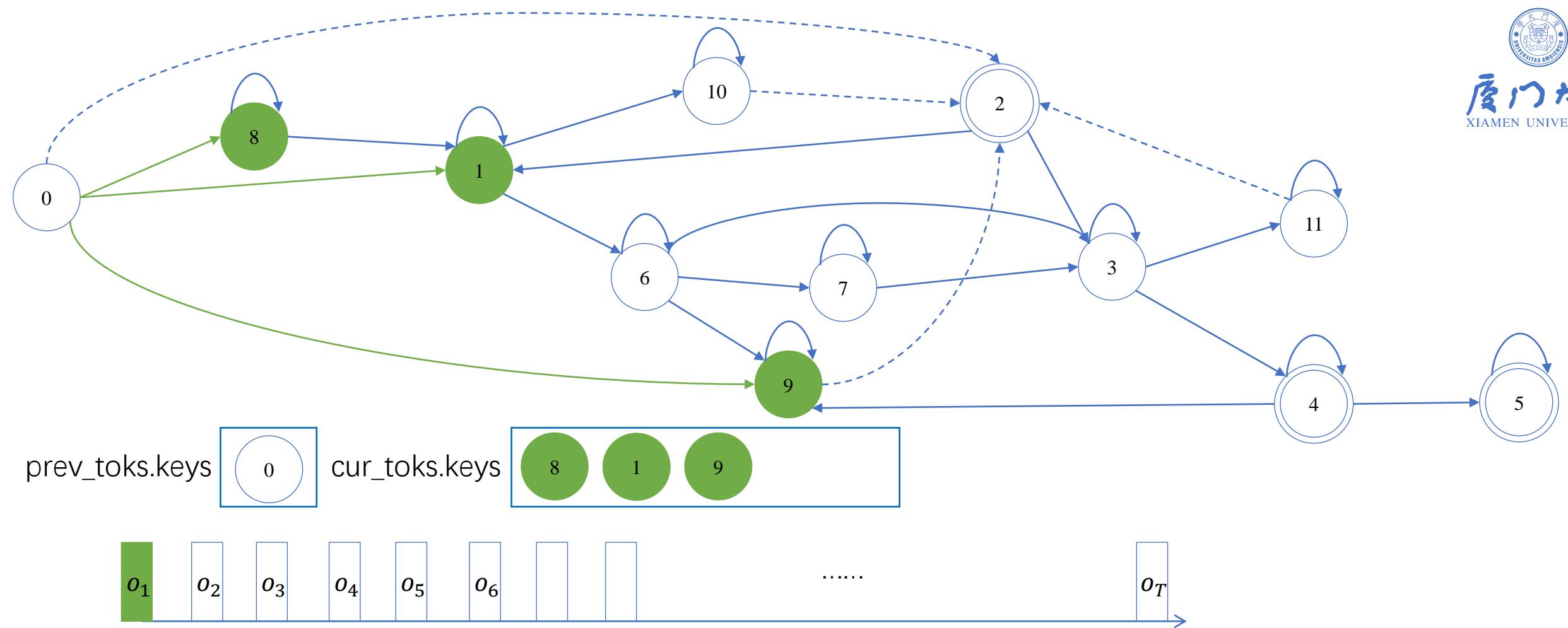




```

while(frame < LastFrame)
{
    Swap(prev_toks, cur_toks);
    double weight_cutoff = ProcessEmitting(frame);
    ProcessNonemitting(weight_cutoff);
    frame++;
}

```

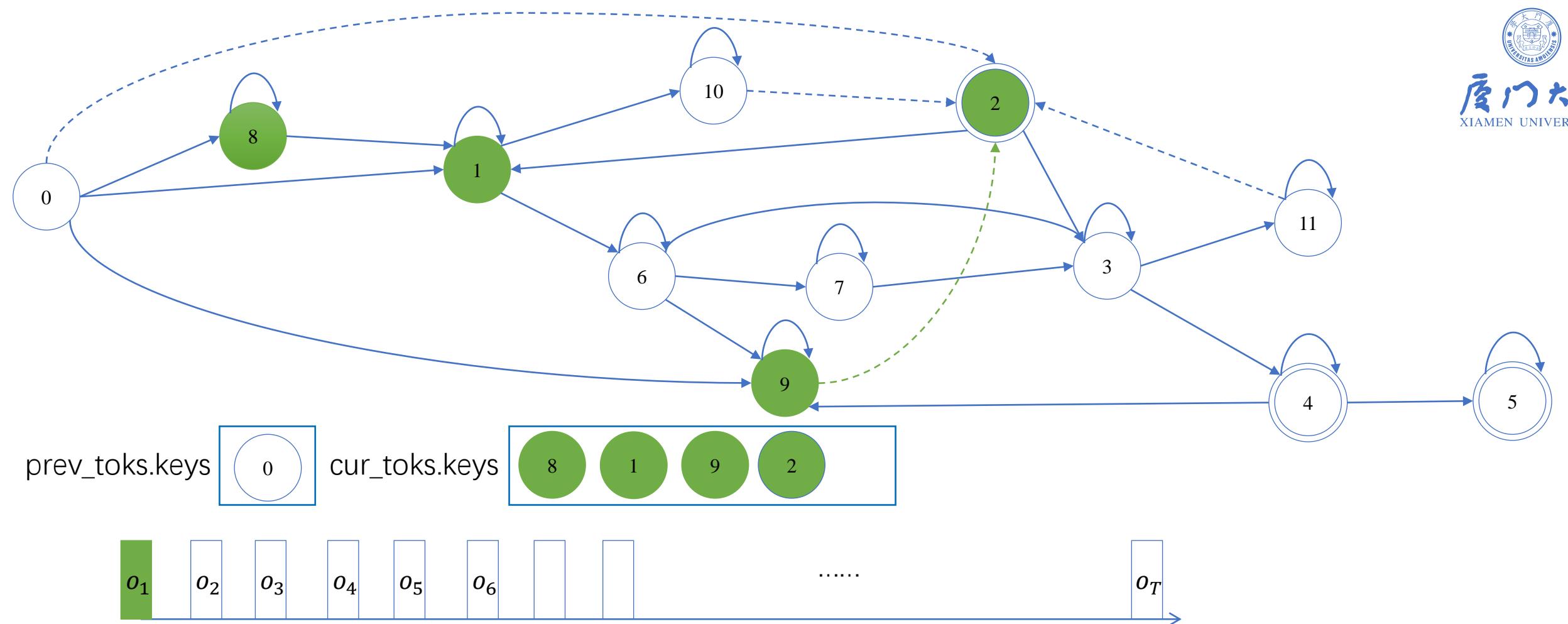


```

while(frame < LastFrame)
{
    Swap(prev_toks, cur_toks);
    double weight_cutoff = ProcessEmitting(frame);
    ProcessNonemitting(weight_cutoff);
    frame++;
}

```

ProcessEmitting:  
for all tokens old\_tok in prev\_toks:  
 u = old\_tok.key;  
**for all emitting arc (u, v):**  
 cost = old\_tok.cost;  
 cost += arc.weight + acoustic\_cost(frame);  
 Add/Replace new Token for v if cost is lower;

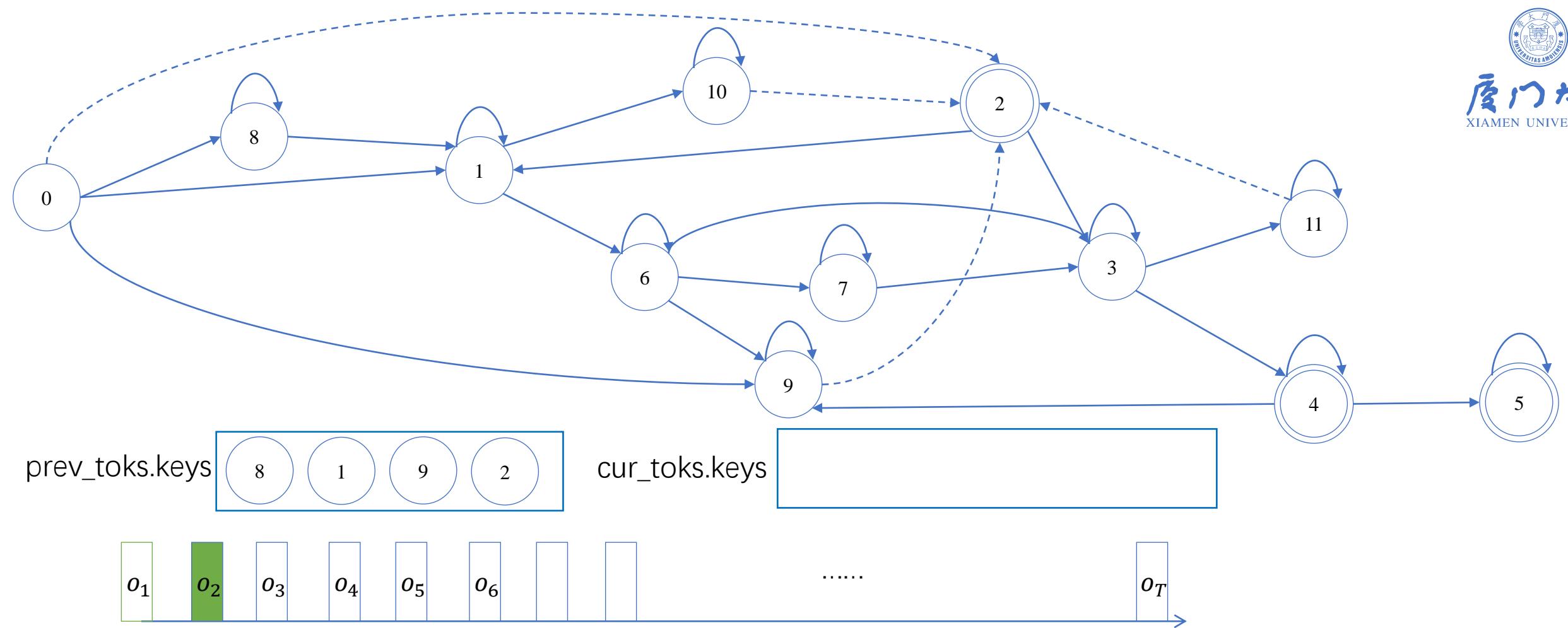


```

while(frame < LastFrame)
{
    Swap(prev_toks, cur_toks);
    double weight_cutoff = ProcessEmitting(frame);
    ProcessNonemitting(weight_cutoff);
    frame++;
}

```

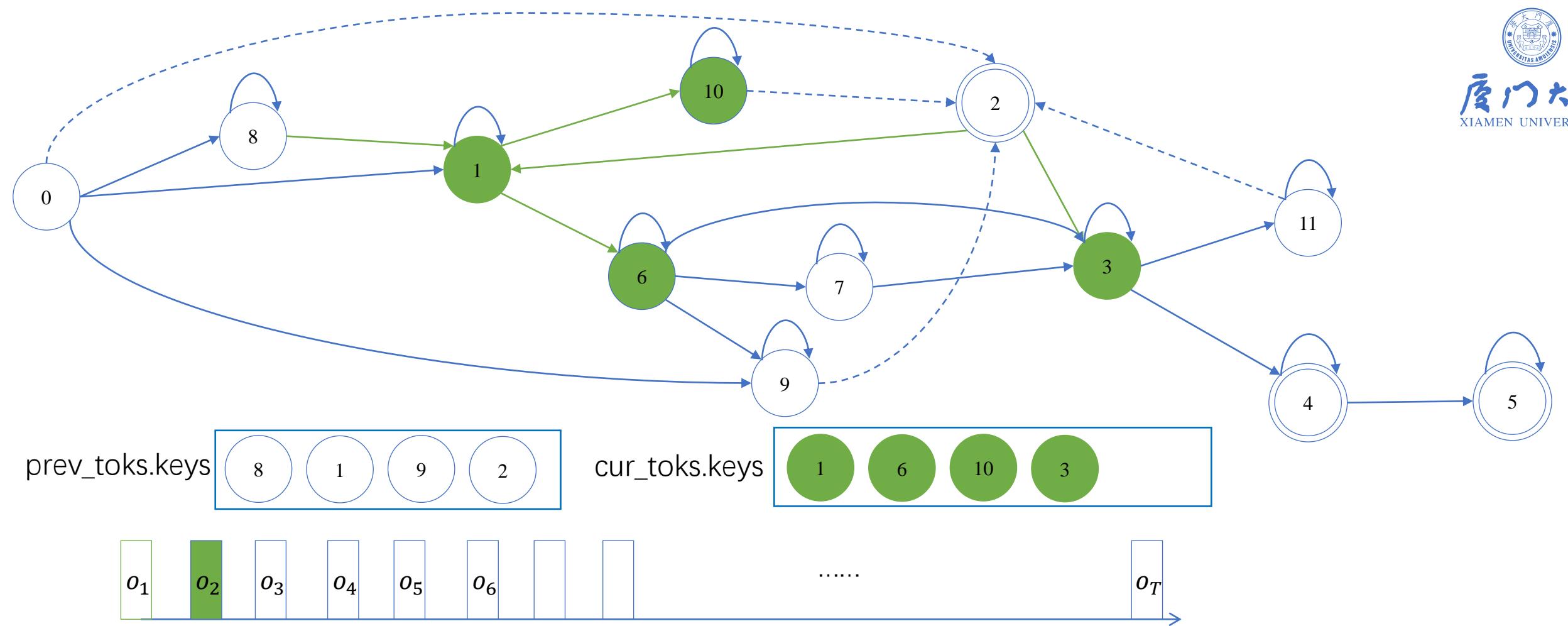
ProcessNonemitting:  
Add all tokens in cur\_toks in a queue q  
while (!queue\_.empty())  
 u = q.pop();  
**for all nonemitting arc (u, v):**  
 cost = old\_tok.cost;  
 cost += arc.weight + acoustic\_cost(frame);  
 Add/Replace new Token for v if cost is lower;  
 Add new token into q;



```

while(frame < LastFrame)
{
    Swap(prev_toks, cur_toks);
    double weight_cutoff = ProcessEmitting(frame);
    ProcessNonemitting(weight_cutoff);
    frame++;
}

```

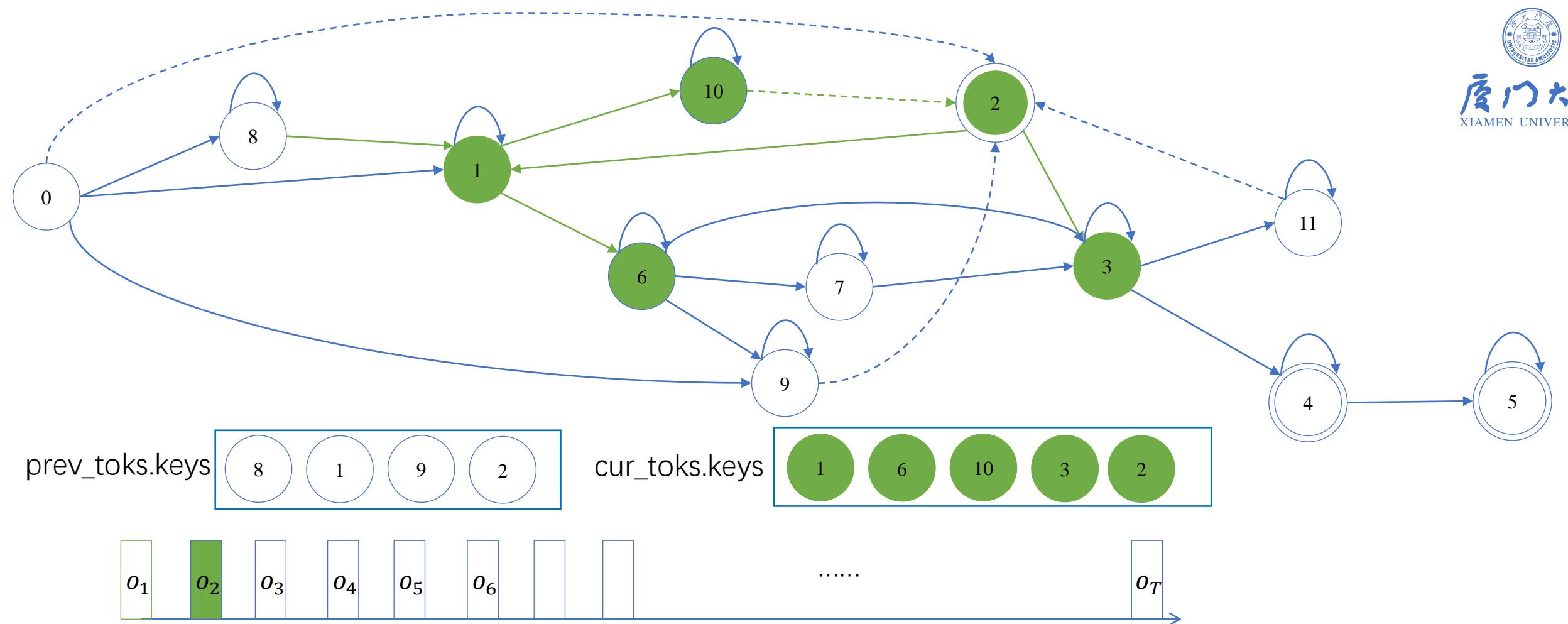


```

while(frame < LastFrame)
{
    Swap(prev_toks, cur_toks);
    double weight_cutoff = ProcessEmitting(frame);
    ProcessNonemitting(weight_cutoff);
    frame++;
}

```

ProcessEmitting:  
for all tokens old\_tok in prev\_toks:  
 u = old\_tok.key;  
 for all emitting arc (u, v):  
 cost = old\_tok.cost;  
 cost += arc.weight + acoustic\_cost(frame);  
 Add/Replace new Token for v if cost is lower;

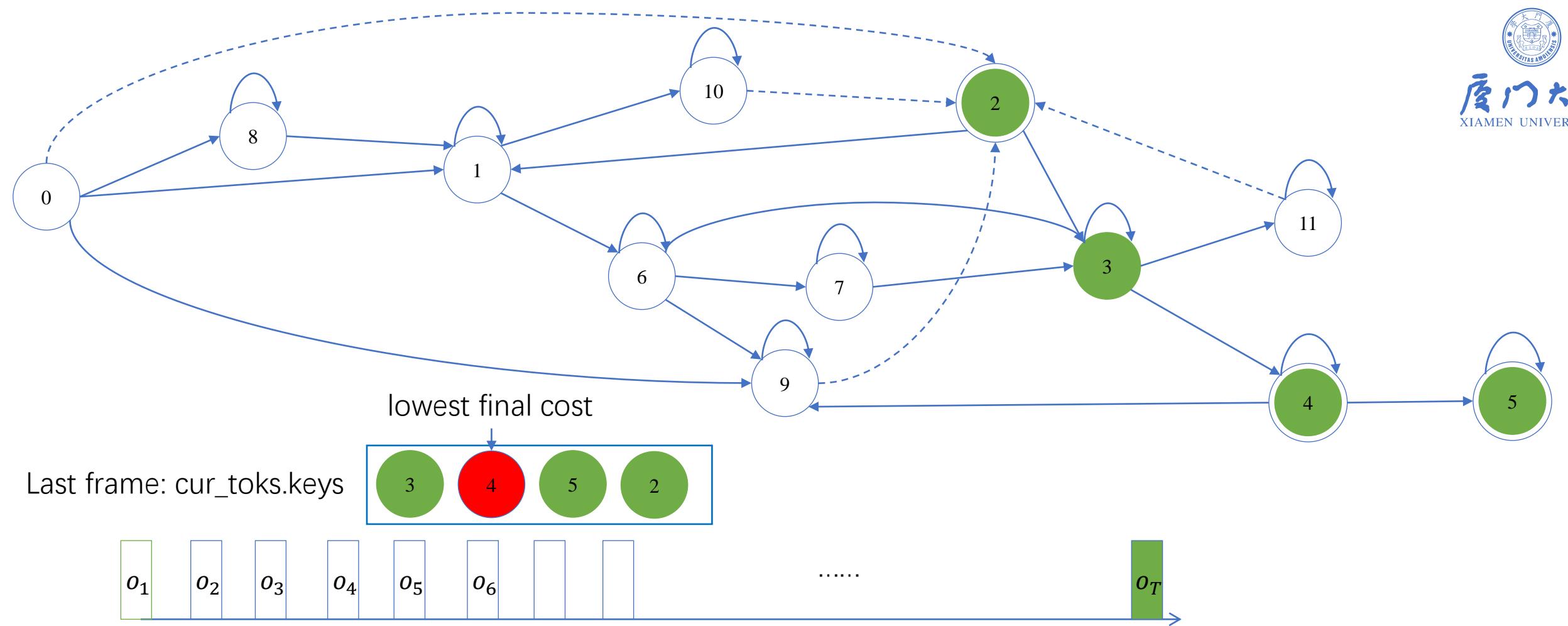


```

while(frame < LastFrame)
{
    Swap(prev_toks, cur_toks);
    double weight_cutoff = ProcessEmitting(frame);
    ProcessNonemitting(weight_cutoff);
    frame++;
}

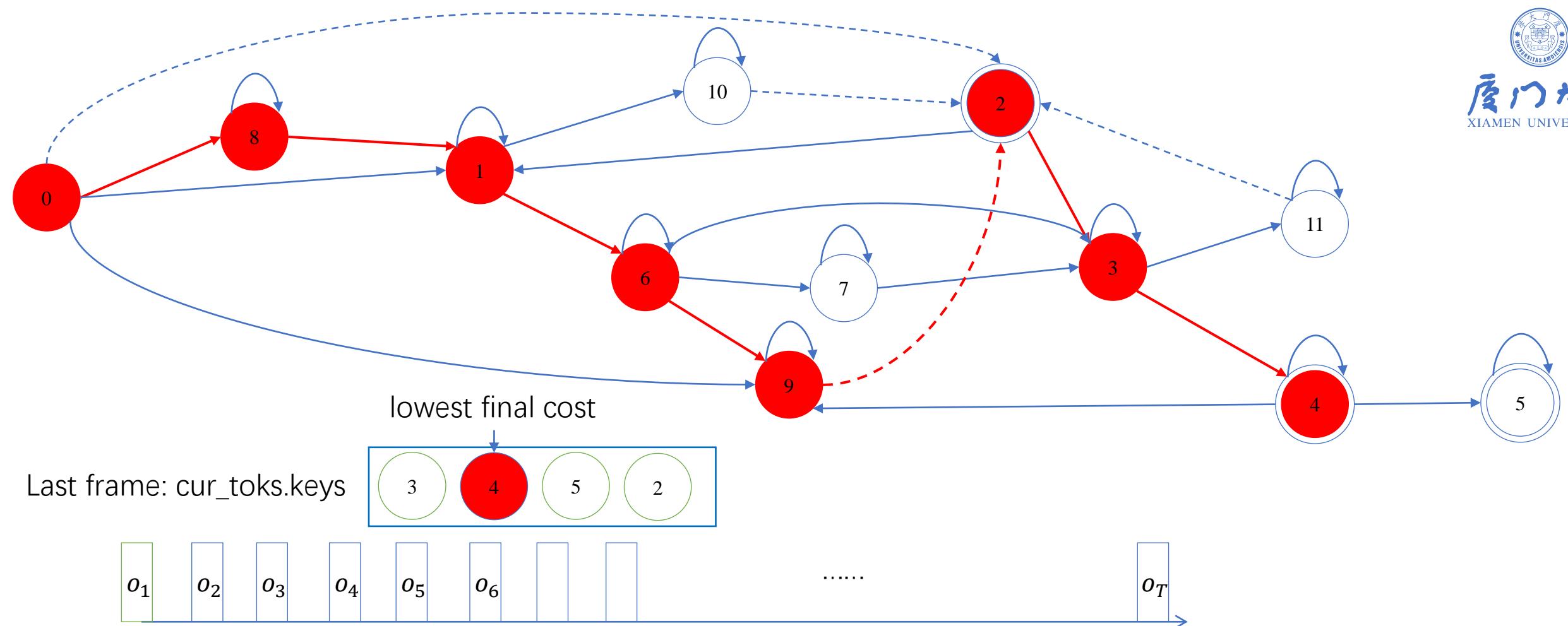
```

ProcessNonemitting:  
Add all tokens in cur\_toks in a queue q  
while (!queue\_.empty())  
 u = q.pop();  
**for all nonemitting arc (u, v):**  
 cost = old\_tok.cost;  
 cost += arc.weight + acoustic\_cost(frame);  
 Add/Replace new Token for v if cost is lower;  
 Add new token into q;



Backtracking:  
token = Token in `cur_toks` with lowest final cost

```
while (token != first_token)
    arcs_reverse.push_back(token.arc);
    token = token.prev;
Reverse(arcs_revserse);
```



Backtracking:  
token = Token in `cur_toks` with lowest final cost

```
while (token != first_token)
    arcs_reverse.push_back(token.arc);
    token = token.prev;
Reverse(arcs_revserse);
```

Best path: 0->8->1->6->9->2->3->4



## 9.5 Lattice解码

- 在语音识别中，经常用Lattice来保存多种候选的识别结果，以便后续进行其它处理（如二次解码）。
- 针对有Lattice的解码，需要保存多条搜索路径，包括中间遍历的路径信息，因此Token之间还需要有**链表信息**，以便跟踪到其可能存在的多个传播来源，即多条回溯路径。

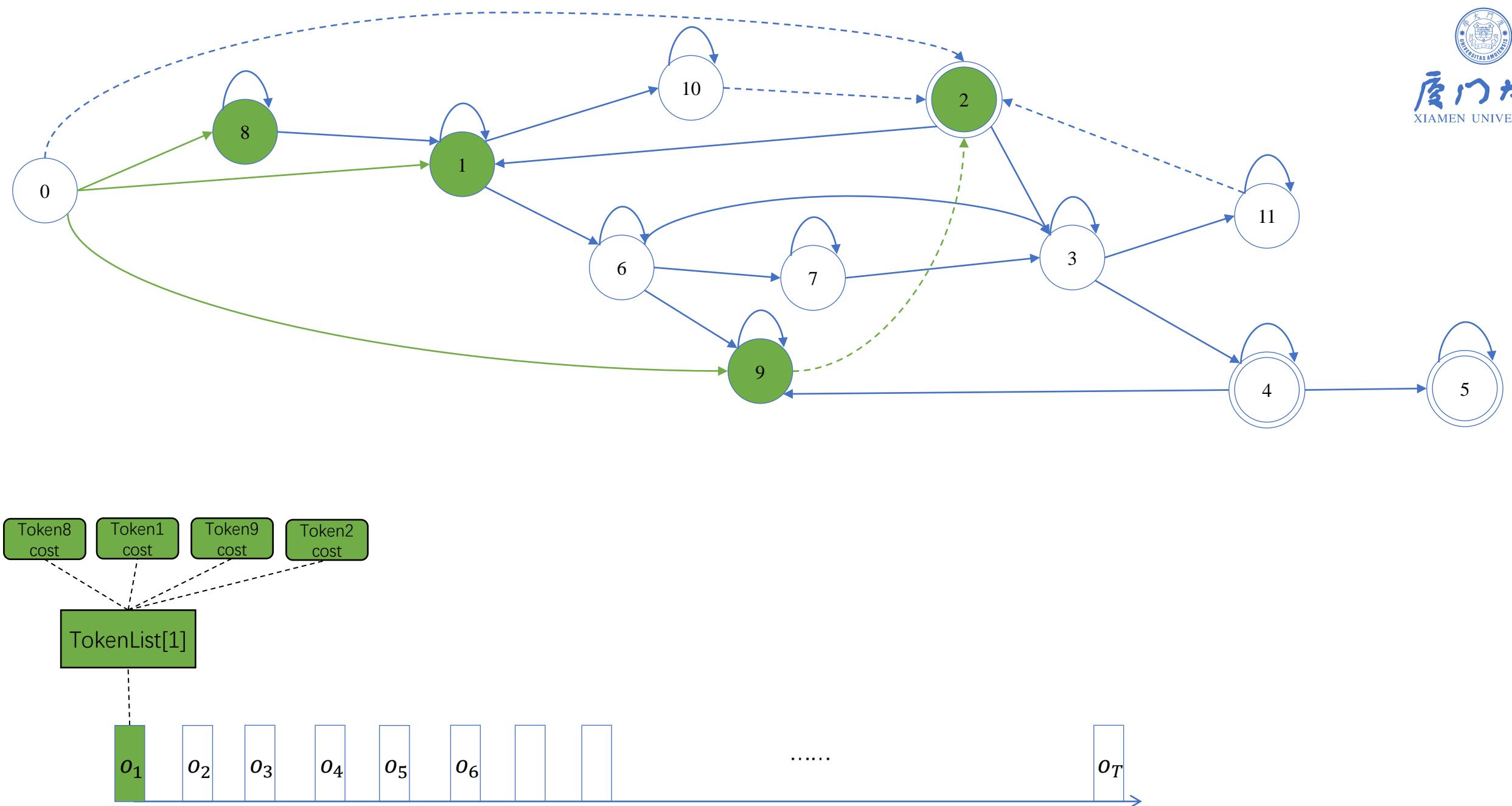


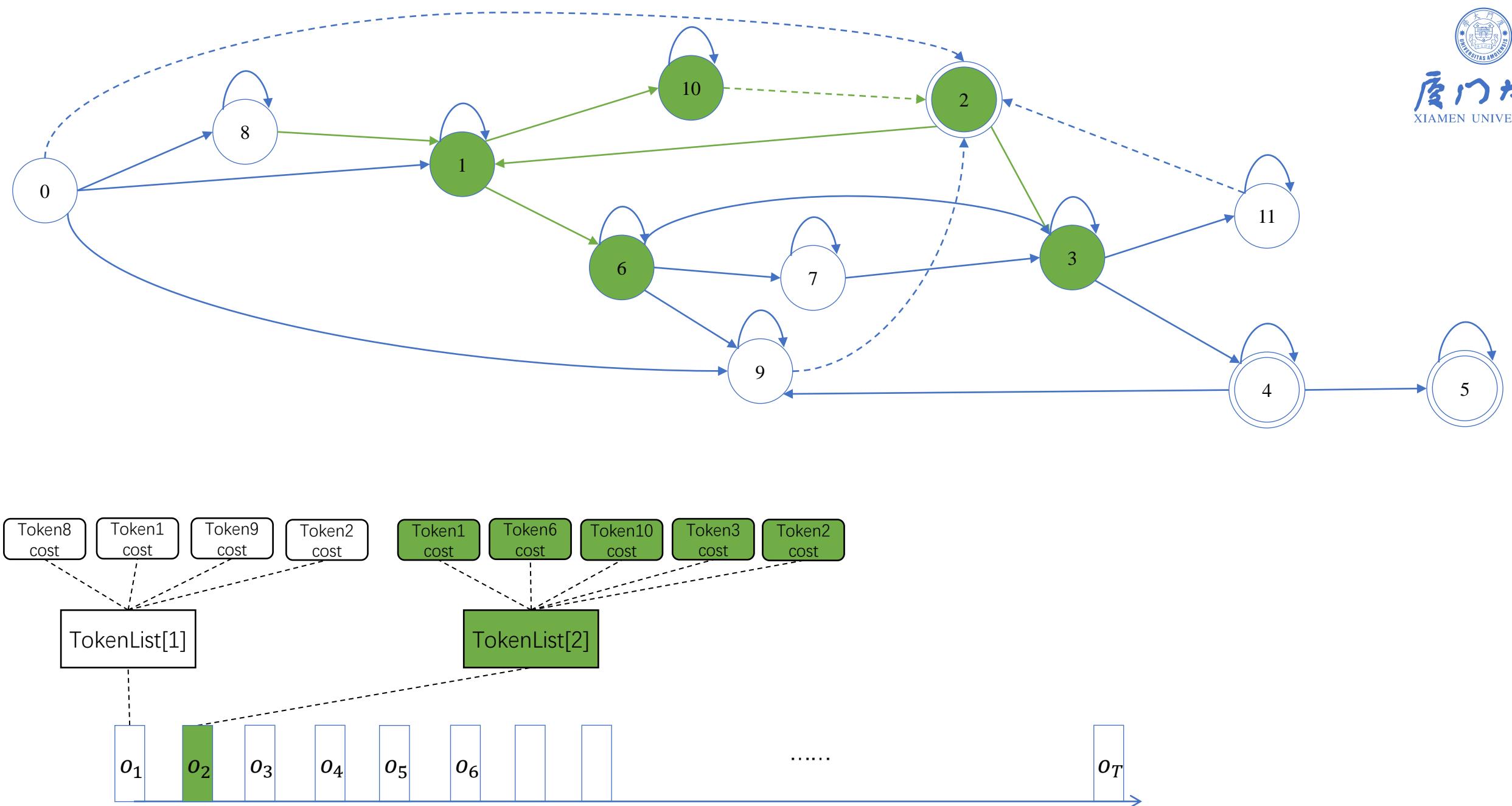
# 数据结构

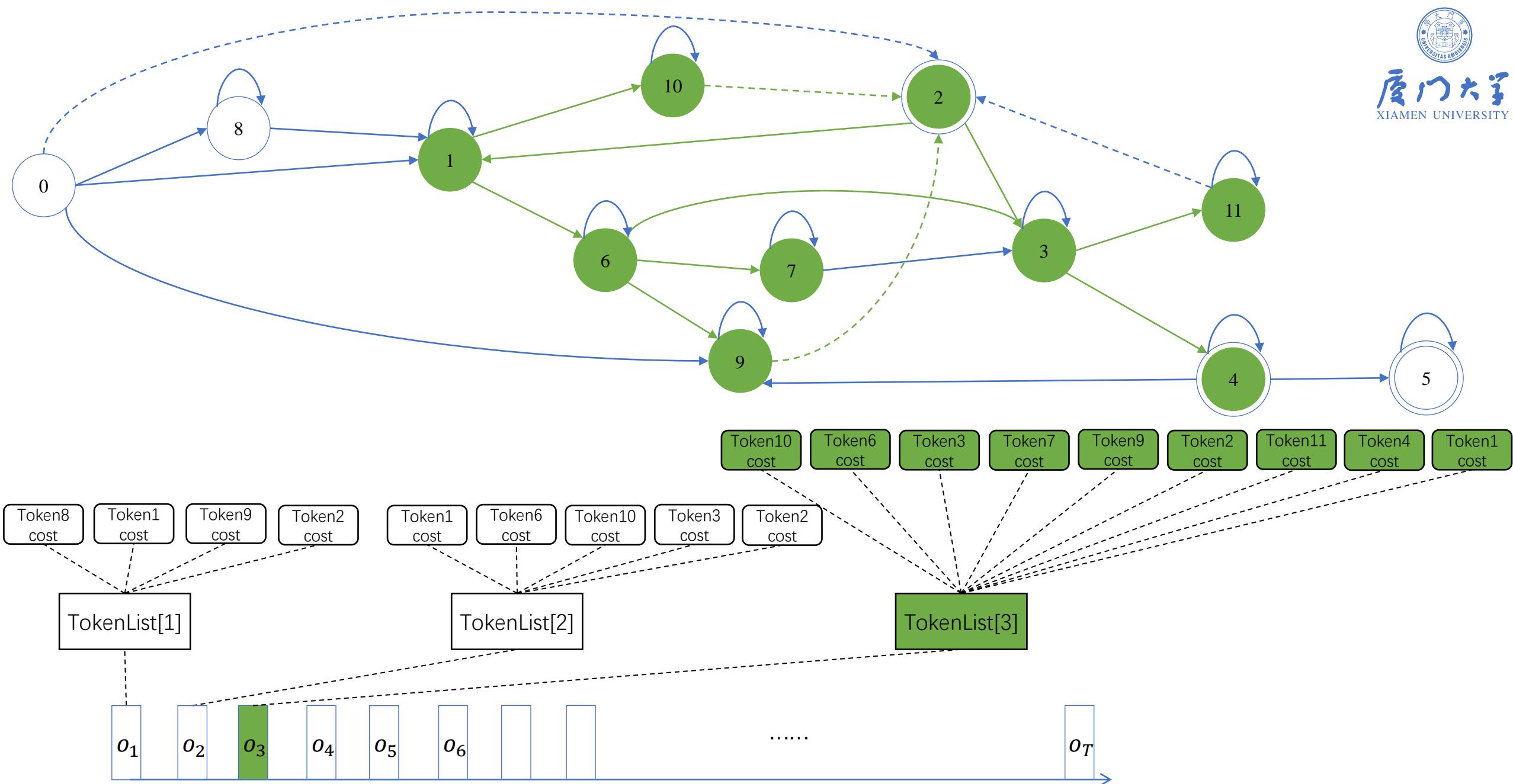
```
struct ForwardLink {  
    Token *next_tok; //链接传到的Token  
    int ilabel; //转移弧的输入标签  
    int olabel; //转移弧的输出标签  
    float graph_cost; //遍历图代价(包含语言模型得分)  
    float acoustic_cost; //声学代价  
    ForwardLink *next; //指向同一时刻的下一个链接  
}
```

```
struct Token {  
    float tot_cost; //累计的最优代价 (包括语言模型和声学代价)  
    float extra_cost; //所有ForwardLink中和最优路径代价差异的最小值  
    ForwardLink *links; //前向链接, 指向新创建的Token, 用于Lattice生成  
    Token *next; //指向同一时刻的下一个Token  
}
```

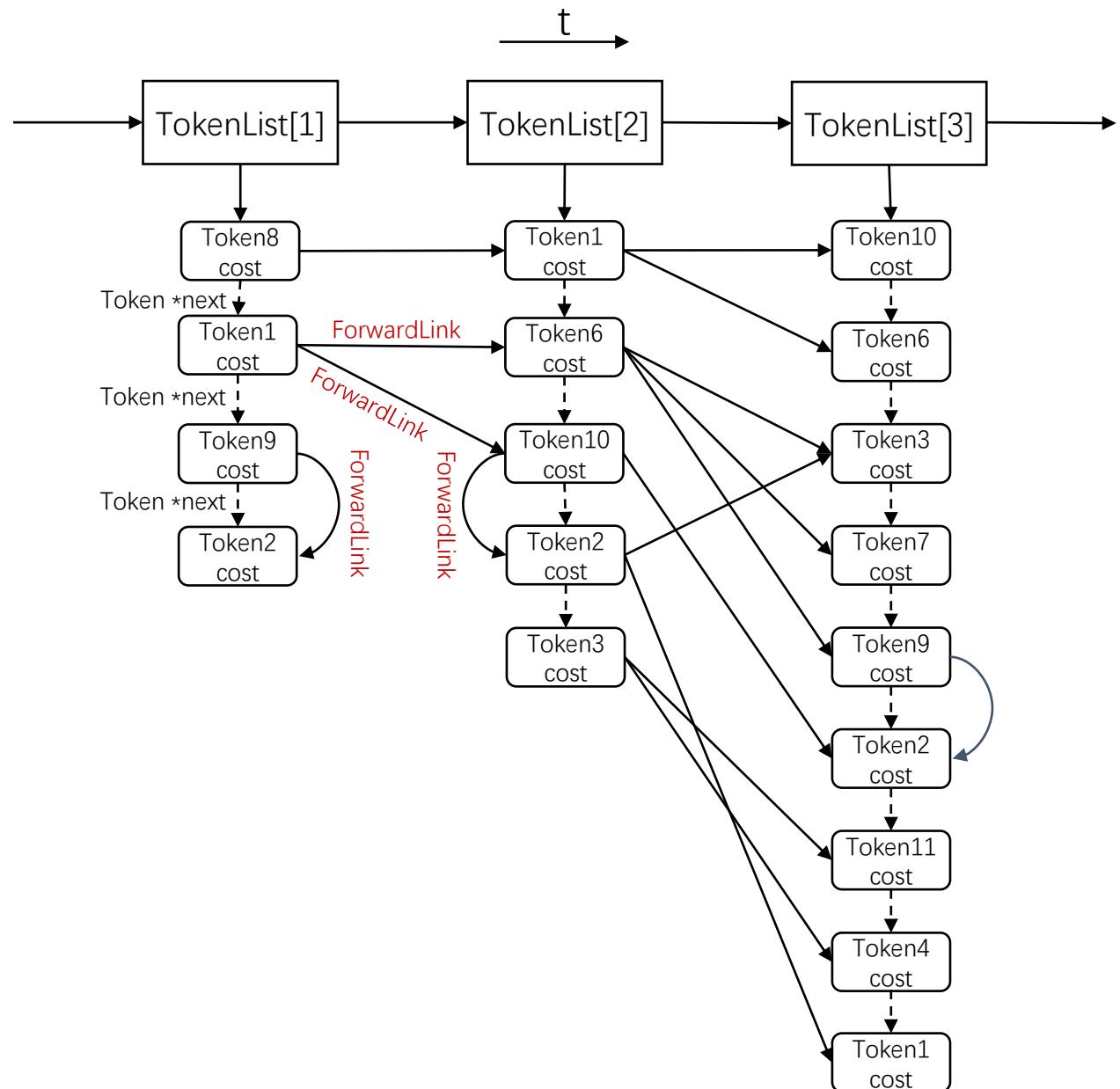
```
struct TokenList {  
    Token *toks; //指向同一时刻的第一个Token  
    bool must_prune_forward_links; //用于剪枝, 默认true  
    bool must_prune_tokens; //用于剪枝, 默认true  
}
```







# Lattice解码



Lattice解码的令牌传播 (Token Passing) 过程

# 剪枝策略

- 在解码过程中，针对当前帧找出最低代价 (best\_cost) 的Token，即最优的Token，根据该Token设定剪枝上限 (cur\_cost)，其值为best\_cost+beam，同时对该Token进行后续一帧的扩张，计算每个转移弧新的cost，结合beam阈值得到后续扩张的剪枝上限 (next\_cost)。
- 第一轮：**对同一帧所有Token做一次剪枝，抑制一批超过cur\_cost 的Token，即这批Token不再扩张，如图9-30所示的Token9。
- 第二轮：**对当前帧除了最优Token之外的其它Token，也对后续的转移弧计算更新的cost，如果超过扩张剪枝上限next\_cost，则该转移弧不再扩张，如图9-31的Token7到Token3的转移弧被剪枝，实际上是Token3不生成。

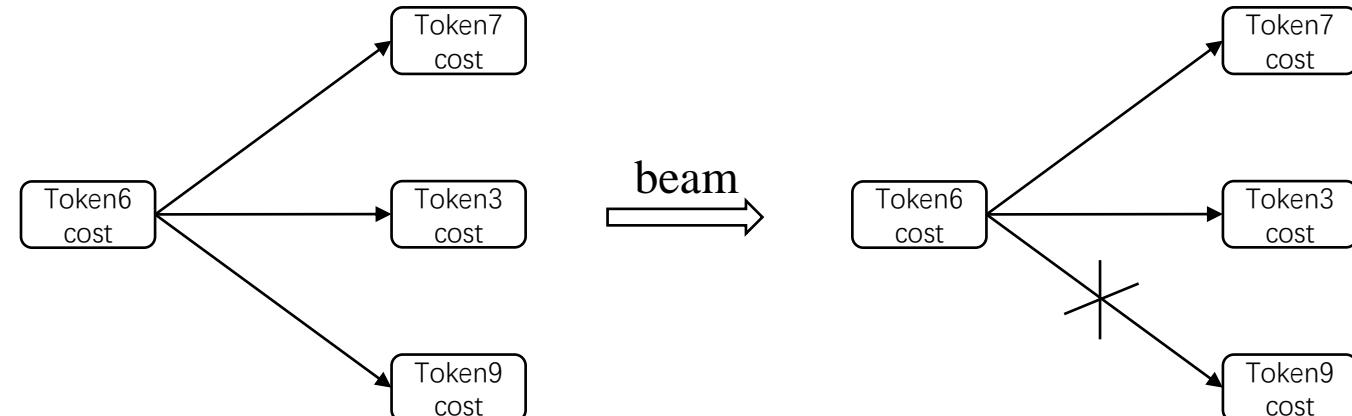


图9-30 剪枝过程（第一轮）

# 剪枝策略—第二轮

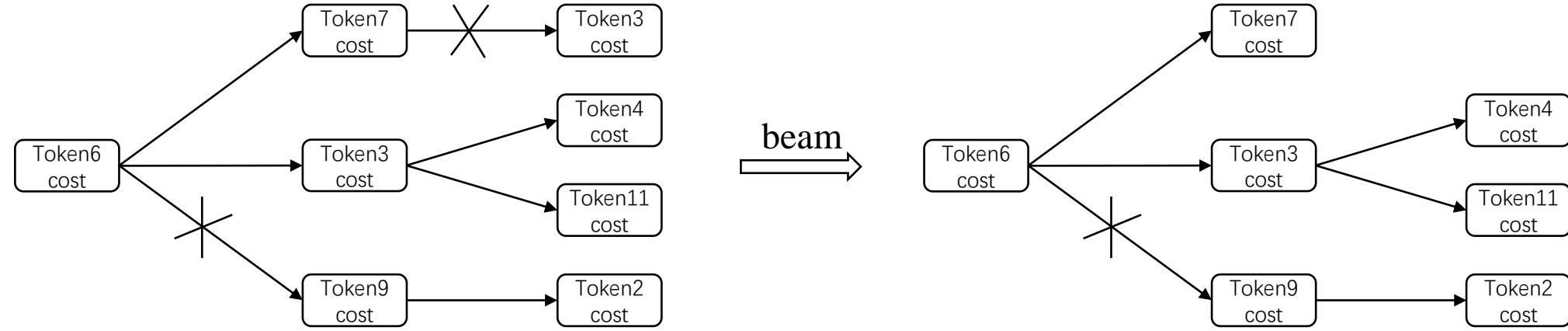


图9-31 剪枝过程（第二轮）

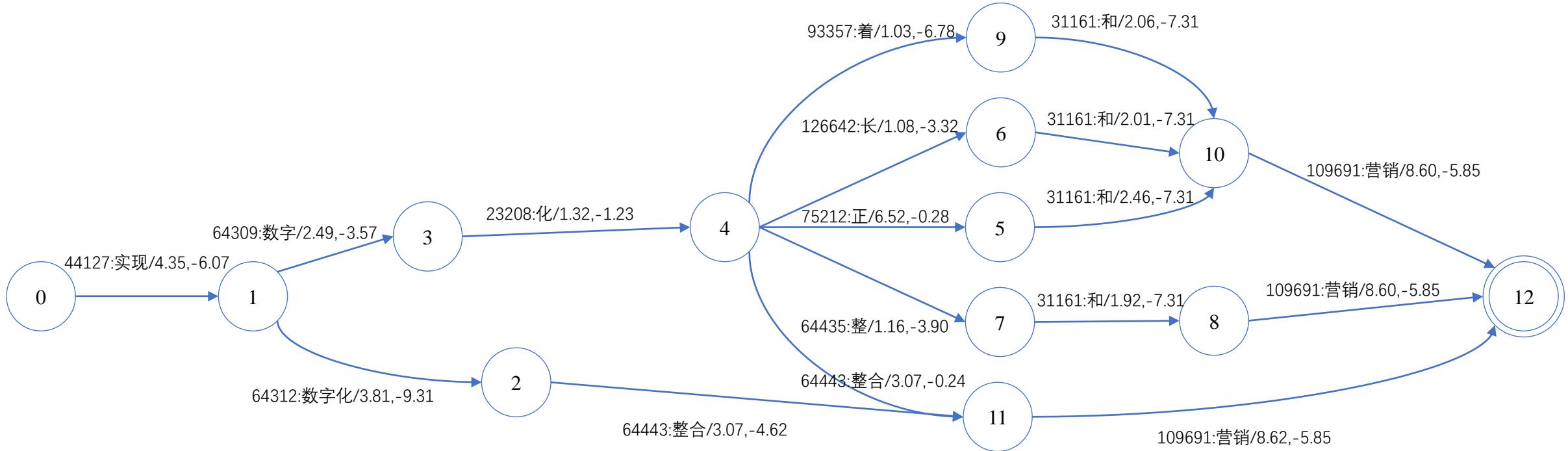


# Lattice (词图)

- 语音识别中，Lattice用来保存多种候选结果，每个节点可对应到具体的时间（帧索引），节点之间的弧包含了候选词信息。HTK用Standard Lattice Format (SLF) 保存Lattice，而Kaldi则用FST形式保存，但也可转化成SLF形式。
- Kaldi Lattice是在解码后，通过每个时刻的TokenList包含的Token和与之关联的ForwardLink遍历生成，并用转移弧Arc保存路径信息。

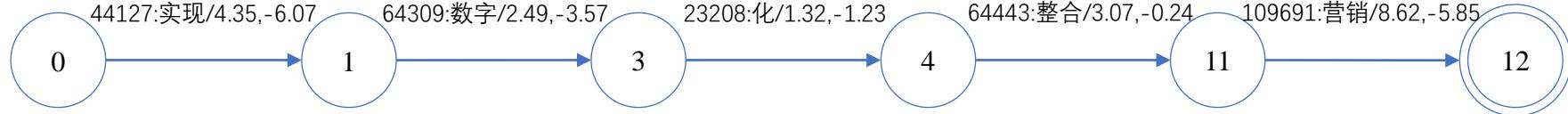
# Lattice (词图)

- Lattice的基础结构可以表示为{input, output, weight}，即Lattice包括输入、输出和权重。Lattice每条弧上的状态输入为transition-id，状态输出为words，其中权重weight包含两个值，即图代价（graph\_cost）和声学代价（acoustic\_cost）。



# 最优路径

- Lattice有多条路径，按最后的cost排序，最小的排在前面，即可从Lattice得到最优路径。注意最优路径也可在解码结束时直接获取，无需经过Lattice。
- 最优路径只包含一个结果，如图所示，即识别结果为“实现数字化整合营销”。



# Lattice & CompactLattice

词图类别	每行格式
Lattice	node-id1,node-id2,transition-id,word, [graph cost, acoustic cost]
CompactLattice	node-id1,node-id2,word, [graph cost, acoustic cost],[transition-ids sequence]

CompactLattice	Lattice
utt-id 0 1 2 101.893,-3842.81, 1 2 3 2.77301,-5.57195, 2 0,0.4_2_1_1_1_1_1_1_1_1_1_1_1_1 _1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1 _1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1 _1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1 _1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1 _1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1 _1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1 _1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1_1	utt-id 0 1 0 2 101.893,-3842.81 1 2 0 3 2.77301,-5.57195 2 3 4 0 ... 148 149 1 0 149 150 6 0 150

同一条语音在CompactLattice和Lattice上的不同表现形式。但是其代表的意义是一样的，比如utt-id的下面第一行，两个Lattice都表示到达word-id为2的图形代价是101.893，声学代价是-3842.81。

更具体地，在这行中，CompactLattice表示结点'0'到结点'1'的弧上，输入和输出标签的id (word-id) 都为2，它的图形代价和声学代价为101.893和-3842.81。而在Lattice中对应的行则表示结点0到结点1（这两个结点是重新编号过的结点）的弧上，输入标签的id (transition-id) 为0，输出标签的id (word-id) 为2，该转移弧上的图形代价和声学代价分别为101.893和-3842.81。



# 9.6本章小结

- 本章分别介绍了基于动态网络的传统Viterbi解码过程和基于WFST静态网络的Viterbi解码过程，对基于WFST的HCLG构建过程做了详细讲解。
- WFST把发音词典、声学模型和语言模型三大组件构成统一的静态网络，因此解码速度非常快。
- 本章还从原理上深入讲解了WFST解码过程中的令牌传播机制，以及对应的剪枝策略，最后还介绍了针对识别结果的Lattice保存形式。