[Introduction to Theory of Computation - YouTube](https://www.youtube.com/watch?v=58N2N7zJGrQ&list=PLBlnK6fEyqRgp46KUv4ZY69yXmpwKOIev&index=1&pp=iAQB)

强烈推荐这个视频教程

# DFA 和 NFA

## 二者区别

DFA就是Determistic Finite Atomata 确定性有限自动机

NFA就是 Non-Determistic Finite Atomata 非确定性有限自动机

DFA的一个输入一定只有一个输出，但是NFA则不是，一个输入可以是多个输出

A drawing of a diagram

AI-generated content may be incorrect.

如图所示，在A state的时候，given 0，可以得到 A 或者 C，given 1，可以得到 B 或者 D。

## NFA的例子

A screenshot of a computer

AI-generated content may be incorrect.

这个例子是创建一个NFA，他所允许的输入是以0作为结尾的字符串。可以看到，他本质上是一个包含多个分支的或逻辑结构，只要其中一个分支通过了，就表示这个输入通过了。像第二个输入 01 ，当输入都结束后，所有分支中并没有一个分支的结尾状态是B，所以这个输入不通过。

A blackboard with text and numbers

AI-generated content may be incorrect.

这里列举了3个NFA，第一个例子中，你可以想象，有无穷多的分支，最后都会有个分支走向NFA所接受的输入。可能比较困惑的是第二个例子，其实A这个初始状态，他这样弄自循环，本质上是个模块，这个模块接受任意的输入。

## NFA 转成DFA

这是这个课程第一个最有意思的地方了。

A screenshot of a computer

AI-generated content may be incorrect.

首先，在NFA中，没有一个dead state，比如A输入1之后去哪，所以DFA给他添加了个dead state，当然，上面的案例我们只是一个品感觉做的DFA，这个方法并不系统。

A blackboard with text and graphics

AI-generated content may be incorrect.

先看NFA的状态转换表（transition），可以看到A可以trans到A或者B，那么我们把A或者B直接合并成一个新的状态AB，那么AB state given 0 就是 A given 0 并上 B given 0，因为B given 0 是空集，因此，AB given 0 就是 {A}，而 { AB | input = 1 } = { A| input = 1 } U {B | input = 1} = { AB }，那么这个合并是否有正确？确实是正确的，{AB}表示的分支就是两个分支，一个A一个B，整个AB 的input就是它内部的不同分支分别给相同的input，结果也是分支，而分支的并集就是AB的结果并集，你结合上面的 NFA分支来看就懂了。

# 状态压缩

## 什么是状态压缩

我们知道，状态机的状态其实并不是说你字符串的某个位置，它其实可以和你当前的功能毫无关系，可以不带任何意义。整个 Atomata 干的事情，只是接受一串input或者拒绝一串input，所有的状态其实只有两类，一类是Final State，也就是可以最终接受的状态，一类是Not-Final State，不可以接受的状态，因此如果两个状态无论怎么输入，最终都是走向Final State或者Not-Final State，那这两个状态我们其实可以认为是一个状态。

这个状态的合并和上面的NFA转DFA不是一个概念，上面是为了把NFA转成DFA，将本来一个输入对应多个输出转成一个输入对应一个输出，它合并的是不同分支的状态，得到最终的状态数量并不一定是最优的。

而我们这里的状态压缩，要压缩的是DFA的状态，DFA是不存在分支状态这个概念的。

举个例子说明

A drawing of a diagram

AI-generated content may be incorrect.

E是我们的Final state，压缩后成这样子

A drawing of a diagram

AI-generated content may be incorrect.

你想象下，当输入1的时候，二者是否等效，当输入0的时候，二者是否又等效，当输入10，01，001，000….等等是否都等效。

回头看这个状态压缩是什么意思？

A screenshot of a computer

AI-generated content may be incorrect.

可以看到无论AC输入什么，他们的输出类别都是一致的。

## 状态压缩算法

### 正常算法

A drawing of a diagram

AI-generated content may be incorrect.

我还是以这个图为例子，当输入input的长度为0的时候，根据类别（是否final state），可以分为两类 {A, B, C, D} 和 { E }，这个是第一次迭代的结果。

当输入input的长度为1的时候，A可能的下一个状态是{B, C}，B可能的下一个状态是 {B, D}，C的是 {B, C}，D的是{B, E}，E不用看了，因为根据我们前面说的定义，他在第一轮迭代中就已经被分到另外一个类了，我们此时只需要检查{A, B, C, D}是否还要再细分类。那么，因为D的可能下个状态中有E，而E是和ABCD不同类的，因此D要再细分出来，这里就有三类了{A, B, C} ，{D}，{ E }，这个是第二次迭代的结果，表示如果输入长度为1，那么从ABC三个状态输入，结果是一样的，而从D，E输入，结果是不一样的，你可能会好奇为什么D和E是不一样的，因为D在input=1的时候既能变成Final state也能变成Not-final state。

好了，重点来了，当input的长度等于2，A可能的状态是什么？我们定义下一个东西，

S( A | input = 2 ) 表示given input length == 2，A输入后得到的所有可能状态的集合。那么S(A | input = 2) = S( S( A | input = 1 ) | input = 1 )，什么意思，比如这里S( A | input = 1 ) = {B, C}，而 S( B| input = 1 ) = {B, D}，S( C| input = 1 ) = {B, C}，因此，

S(A | input = 2) = {B, D} U{B, C} = {B, C, D}。

那么我们到底想干嘛？ 我们想知道的是S(A | input = 2) 和 S(B| input = 2)，S(C| input = 2)，他们的状态集合是否都是同一个类别，你稍微推断下就会发现，S(B| input = 2) = {B, D, E}，很明显，它的可能状态包含了个Final State，那么在input length = 2 的时候，B是没办法和A一样达到一个状态类别的。

可以看到上面是个递归过程，那么我们有没有办法记忆化这个过程呢？有的

因为我们的目的只是为了知道 F( S(A | input = 2) ) 是否等于 F( S(B | input = 2) )，是否等于 F( S(C | input = 2) )，F表示类别的值，那么这个表达式就可以变成

F( S( A | input = 1 ) | input = 1 ) = F( S(B | input = 1) U S(C | input = 1) )

关键就是下面的分配律了

= F( S(B | input = 1) ) U F( S(C | input = 1) )

而 F( S(B | input = 1) ) 和 F( S(C | input = 1) )这些值哪里来？就是从第二次迭代结果来的，F值是否相等，就看是否在同一个组别（所以你根本不用管这个F的值是多少），因此你第三次迭代，只需要看这个A的下一个可能状态集合 和 B的下一个可能状态集合 是否在第二次迭代中的结果中处于同一组，很明显并不是，B的可能状态集合包含了D，D在第二次迭代中是不和 A, C一个组的，因此第三次迭代的结果是 { A, C}, { B }, { D }, { E }，你继续迭代下去的话，会发现不会再变化，说明以后无论input多长，这个组不会再继续分下去了。