后缀自动机的理解与运用 & 树形数据结构例题选讲

福建省福州第一中学 吴作同 (E.SPACE)

关于内容

- 本次讲课内容较为基础，dalao们可以选择自觉（自行睡觉）
- 估计今天的内容会比之后的所有内容都要简单吧
- 第一部分是后缀自动机
- 这是一个关于后缀自动机的入门级教程，讲得比较详细
- 为了让直接阅读的读者能够读懂，我把要讲的内容几乎都写到了PPT上
- 当然这一部分也有例题
- 第二部分是树形数据结构
- 这部分我会讲几道比较简单的例题
- 我做过的题很少，所以例题不会很多，但是都比较典型
- 下午由于只有3小时，所以题目绝对比上午讲的内容简单

什么是后缀自动机

- 后缀自动机就是能接受某个串的所有后缀/子串的自动机
- 你说什么是自动机？
- 你可以联想一下AC自动机
- 如果不知道什么是AC自动机的话你可以认为自动机是一张带有根和转移边的有向图，每条转移边上有一个字符，一个点出发的所有转移边上的字符必须两两不相同
- 这个说法不是很严谨，但是在这里够用了
- 还有一个问题，什么是接受？

什么是后缀自动机

- 考虑自动机上一条由根出发的由节点和转移边构成的有向路径，把路径的转移边上的字符按顺序拼接起来可以得到一个字符串，假设我们称它为S
- 那么我们就说这条路径的终点（这是一个节点）接受字符串S
- 反过来，如果一个自动机上存在一条由根出发的有向路径使得转移边上的字符按顺序拼接起来可以得到S，那么我们就说S被这个自动机接受
- 比如这个自动机

- 它接受空串，"a"，"ab"，"c"，"cb"
- 其中空串被根接受，"a"，"c"被1号点接受，"ab"，"cb"被2号点接受

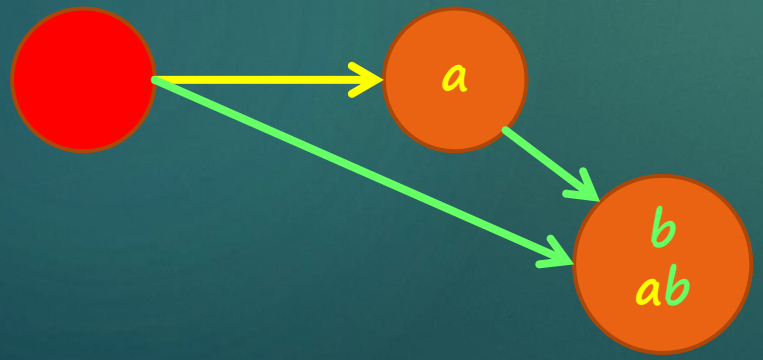
构造——简单的想法

- 我们的目的是构造一个节点数尽量少且能接受字符串的所有子串的自动机
- 先从一些简单的情况开始考虑
- 比如字符串“abc”
- 考虑到一个字符串的子串的子串也是它的子串，所以我们可以在“a”的基础上构造出“ab”的后缀自动机，然后在此基础上构造出“abc”的后缀自动机
- “a”的后缀自动机很显然

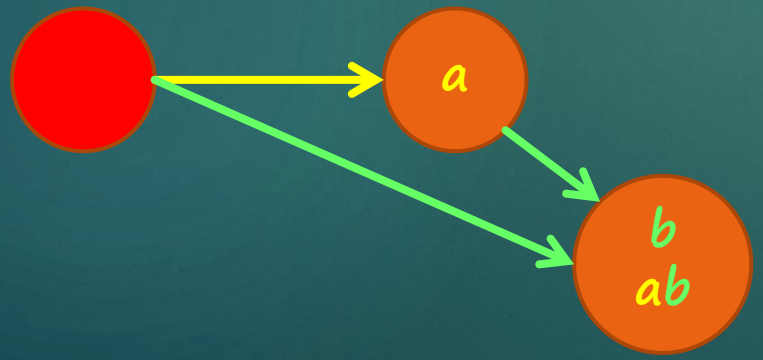
- （用红色节点代表根，每个节点上写着它接受的所有字符串）

构造——简单的想法

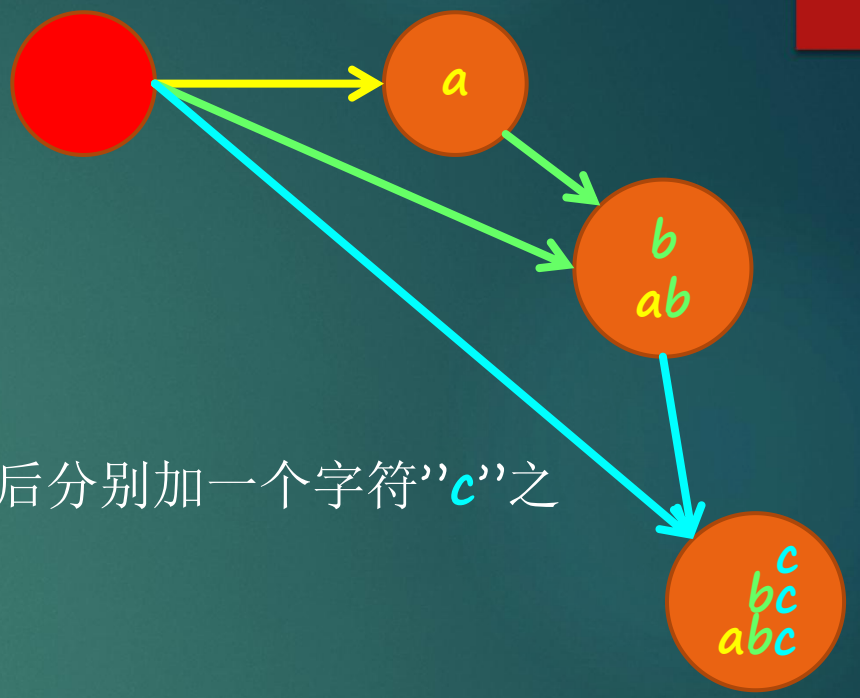
- 接着是''ab''的后缀自动机
- 比起''a''的所有子串，''ab''的子串中多了''b''和''ab''，这两个串是由''a''的所有后缀（空串和''a''）分别加上字符''b''得到的
- 于是我们只需要新建一个节点，把代表空串的根节点和代表''a''的节点分别向新点连一条''b''的转移边即可



构造——简单的想法



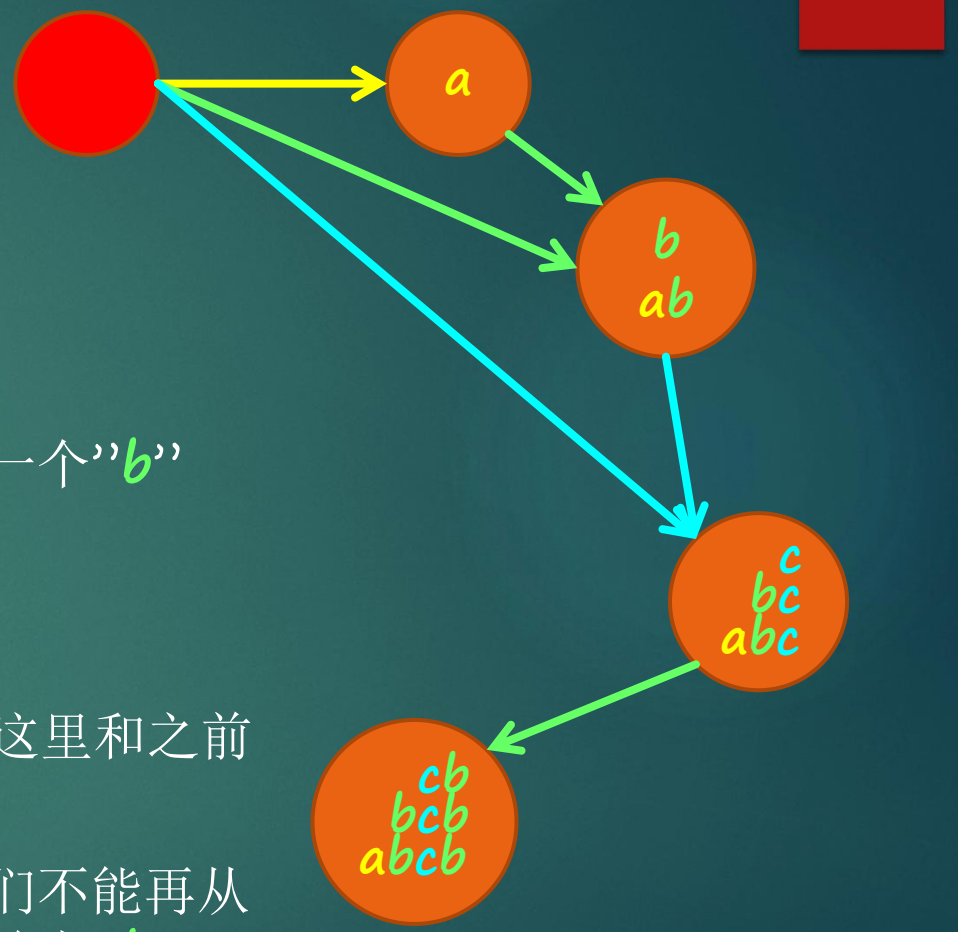
构造——简单的想法



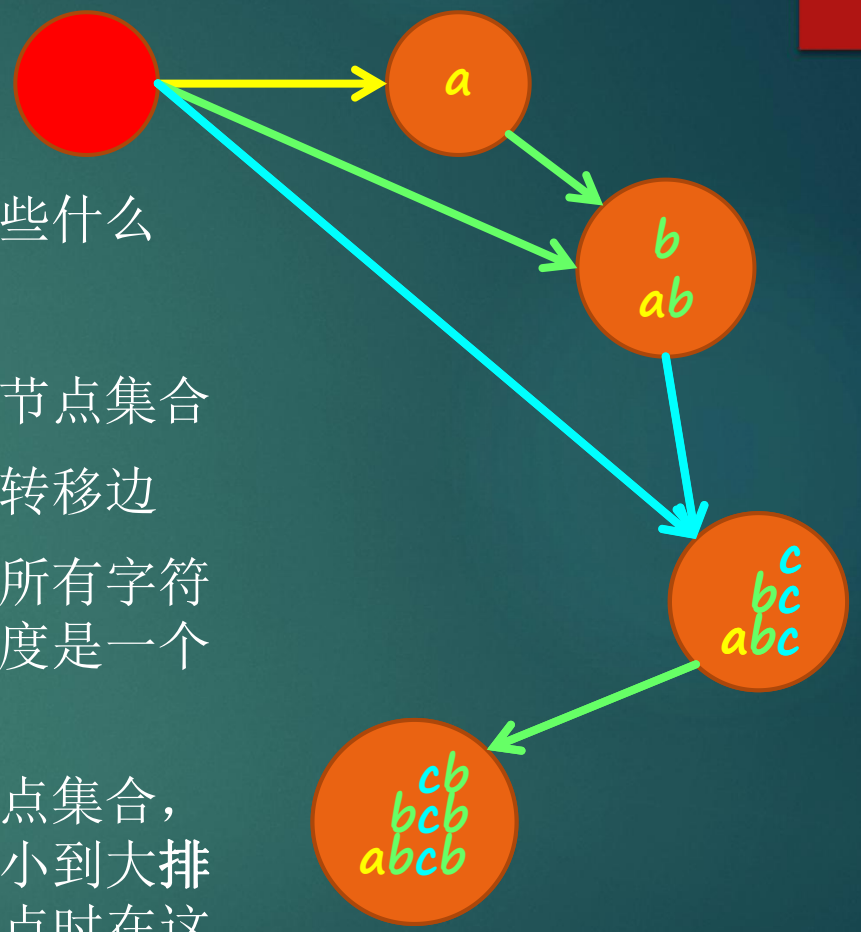
- 然后加上字母“c”
- 这时多了3个子串：“c”,“bc”,“abc”
- 它们是原串的3个后缀（空串,“b”,“ab”）之后分别加一个字符“c”之后得到的串
- 其中“b”和“ab”被这个节点接受
- 我们新建一个节点，连一条“c”的转移边，让新点接受字符串“abc”和“bc”
- 空串被根节点接受
- 我们从根节点向新点连一条“c”的转移边，让新点接受字符串“c”
- 这样“abc”的后缀自动机就建好了

构造——更进一步

- 如果情况更复杂一些呢？
- 比如，在后面再加一个“b”？
- 我们要在空串，“c”，“bc”，“abc”之后分别再加一个“b”
- 其中“c”，“bc”，“abc”被这个节点接受
- 我们新建一个点，连一条“b”的转移边
- 自然，它就接受了“cb”，“bcb”和“abcb”，到这里和之前都一样
- 但是，根节点已经有了一条“b”的转移边，我们不能再从根向新点连一条“b”的转移边，让新点接受字符串“b”
- 那么，怎么办呢？



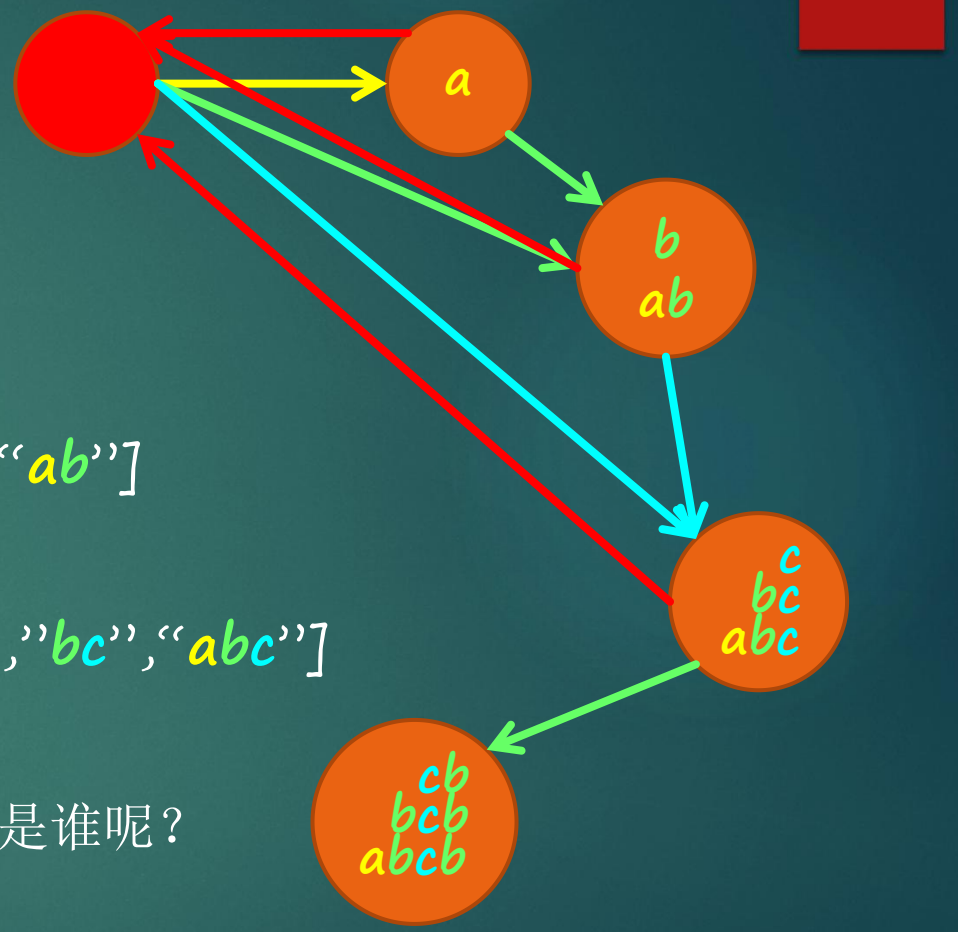
构造——更进一步



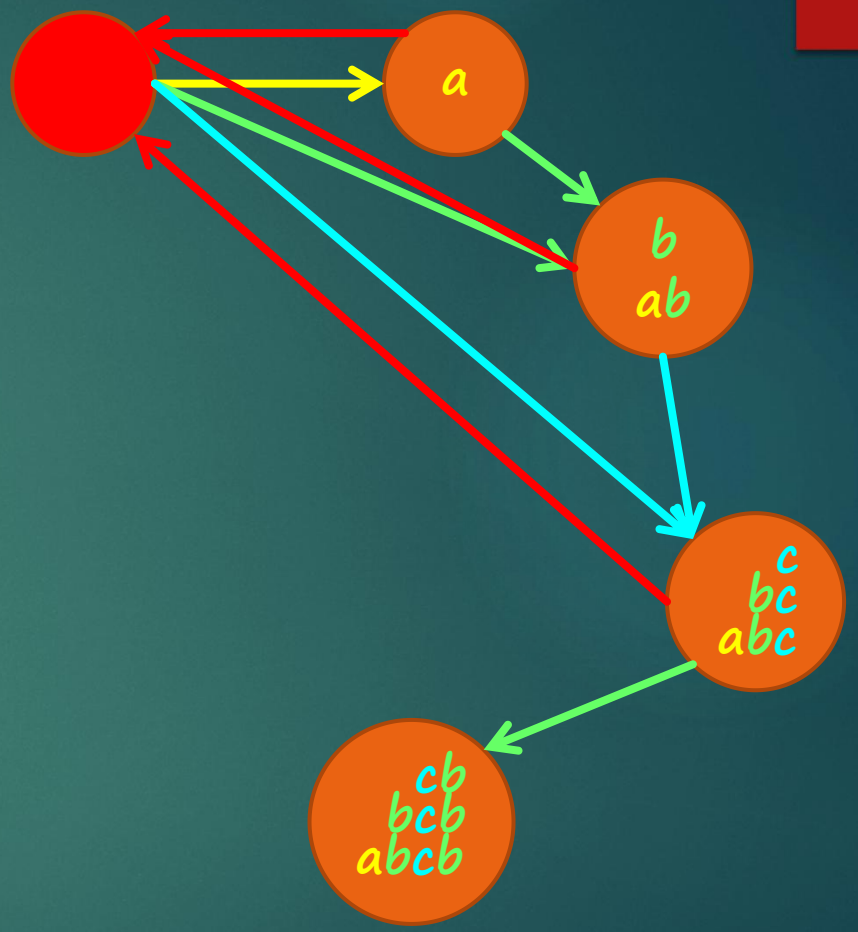
- 回想一下，之前我们添加新字符的时候都做了些什么
- 我们先新建了一个点
- 然后找到了一个不重不漏接受原串所有后缀的节点集合
- 分别给集合中的每个点向新点连一条新字符的转移边
- 通过观察可以发现，目前我们每个节点接受的所有字符串都是其中最长的字符串的某个后缀，并且长度是一个（以该字符串长度为右端点的）连续的区间
- 那么对于一个不重不漏接受原串所有后缀的节点集合，我们可以把这些节点按照接受的字符串长度从小到大**排序**，然后记一个非根节点的父亲为新建这个节点时在这个排序中该节点的前一个节点（根节点没有父亲）
- 以及根据定义可以发现一个节点的父亲接受的最长的字符串是该点接受的最短的字符串去掉首字母后的字符串

构造——更进一步

- 当字符串为”a”的时候，排序为[空串],[“a”]
- 所以[“a”]的父亲是[空串]，也就是根
- 当字符串为”ab”的时候，排序为[空串],[“b”,“ab”]
- 所以[“b”]的父亲是[空串]
- 当字符串为”abc”的时候，排序为[空串],[“c”,”bc”,“abc”]
- 所以[“c”]的父亲还是[空串]
- 那么，这个新点[“cb”,”bcb”,”abcb”]的父亲是谁呢？

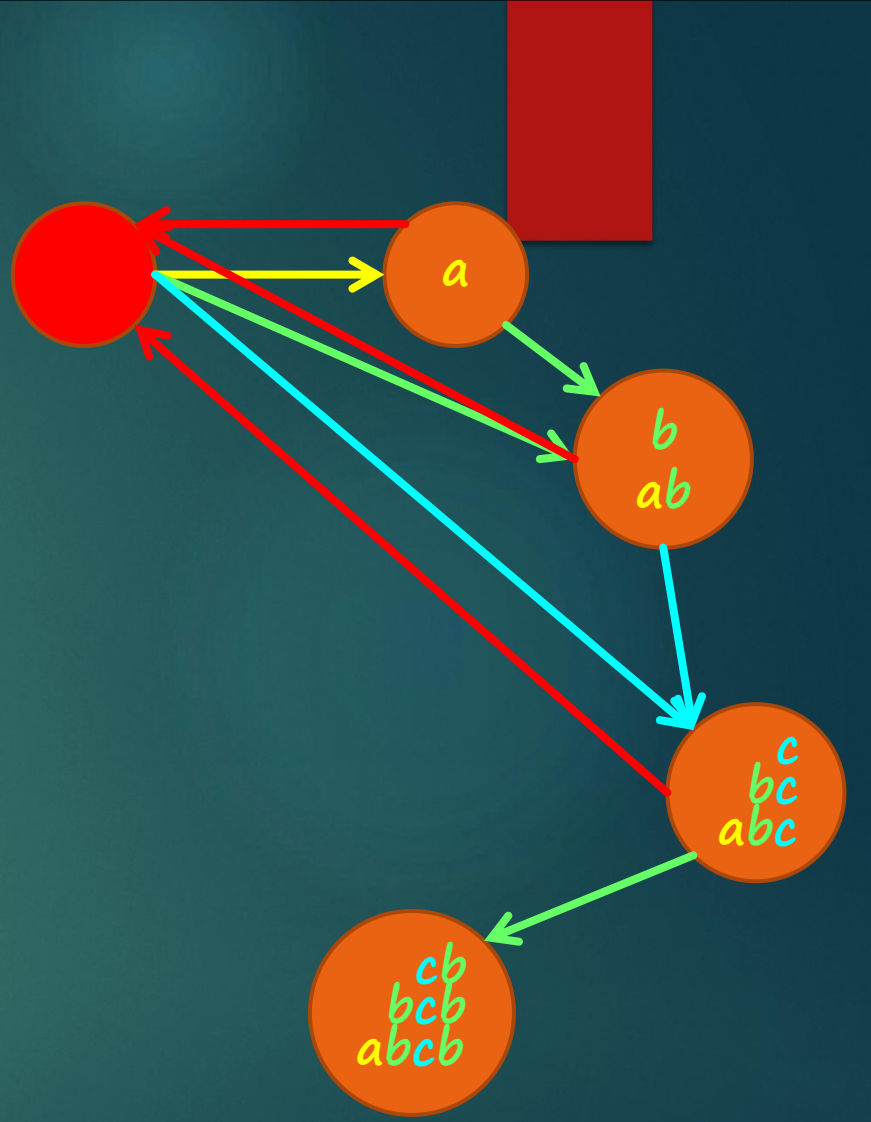


构造——更进一步



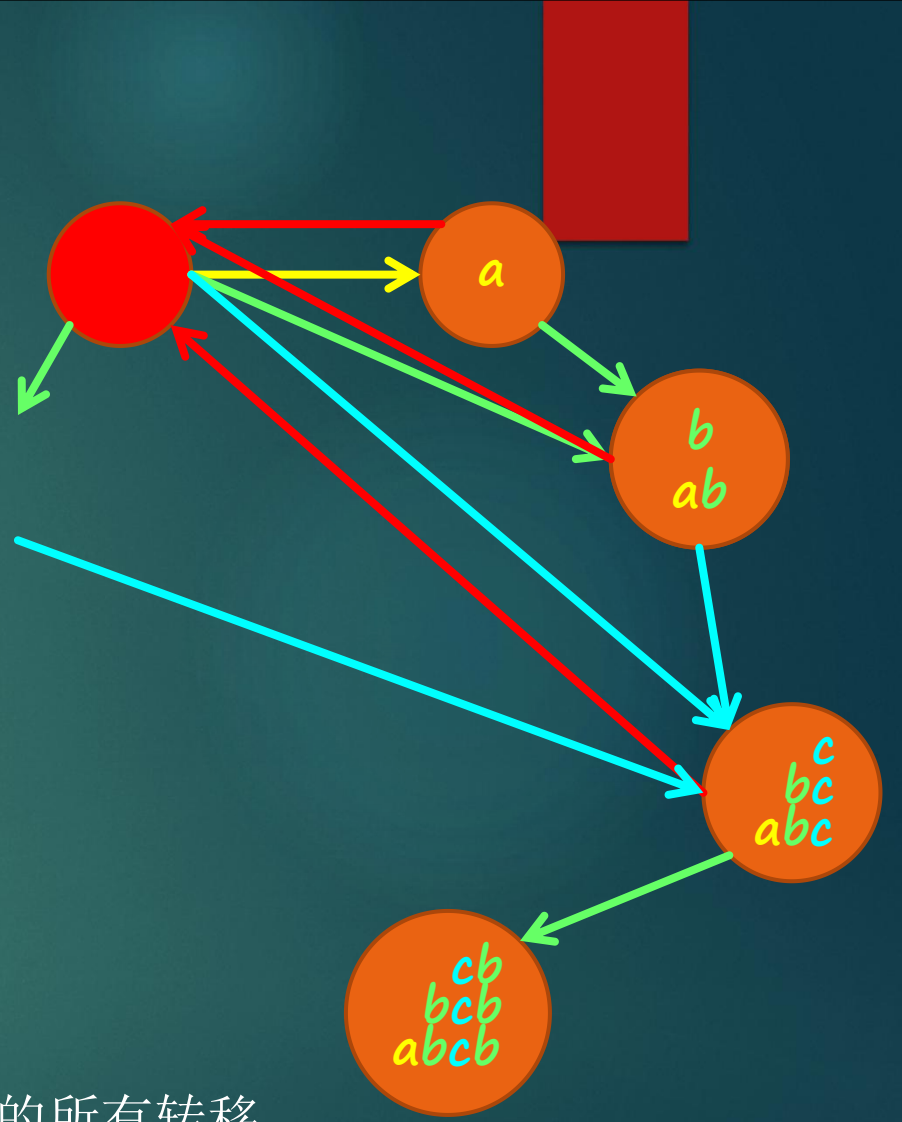
构造——更进一步

- 现在字符串是''abcb''，我们已经有了[空串]和["cb","bcb","abcb"]
- 显然我们必须要一个节点["b"]在这两个节点之间，这样我们才有一个不重不漏接受原串所有后缀的节点集合
- 然而，同一个字符串不能被两个点同时接受（为什么呢）
- 所以，我们必须找到原来接受''b''的节点进行处理
- 有一种很直接的处理方法，就是把它拆开
- 原来它是["b","ab"]，现在我们把它拆成两个节点，一个是["b"]，一个是["ab"]，这样我们就有节点["b"]了
- 那我们要怎么拆呢？



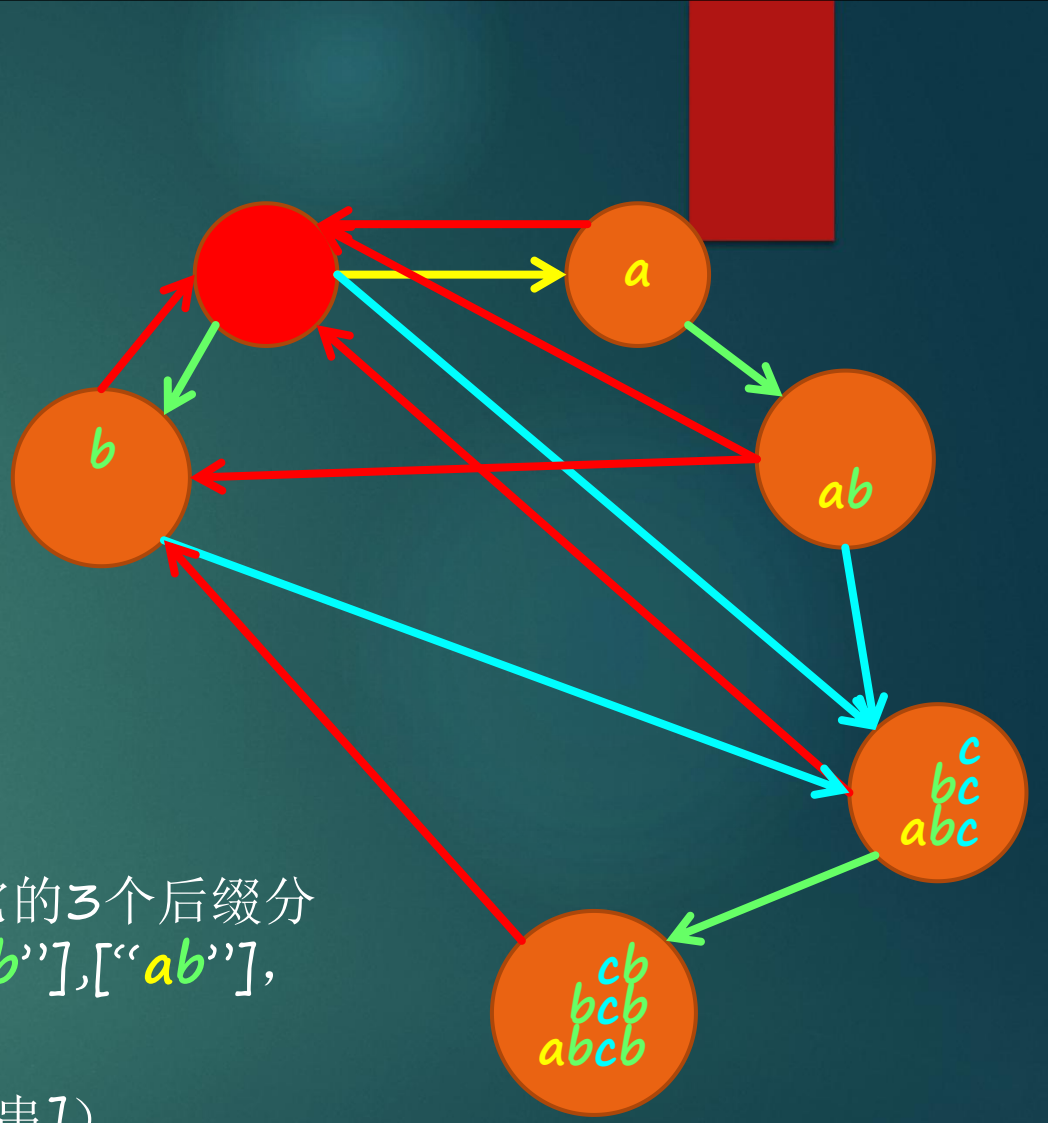
构造——更进一步

- 首先我们新建一个节点，并试图让它接受“b”
- 原来的节点能够接受“b”是因为根节点向它有一条“b”的转移边
- 现在把这条转移边改成指向新点，就可以只让新点接受“b”了
- 但是慢着！
- 因为“b”的接受节点的变化使得后面两个节点分别不再接受“bc”和“bcb”了！
- 解决方法很简单，由于出现问题的原因是原来从“b”出发的所有转移边都不见了，所以我们只要把从原来节点出发的所有转移边全部复制一遍到新的节点上就好



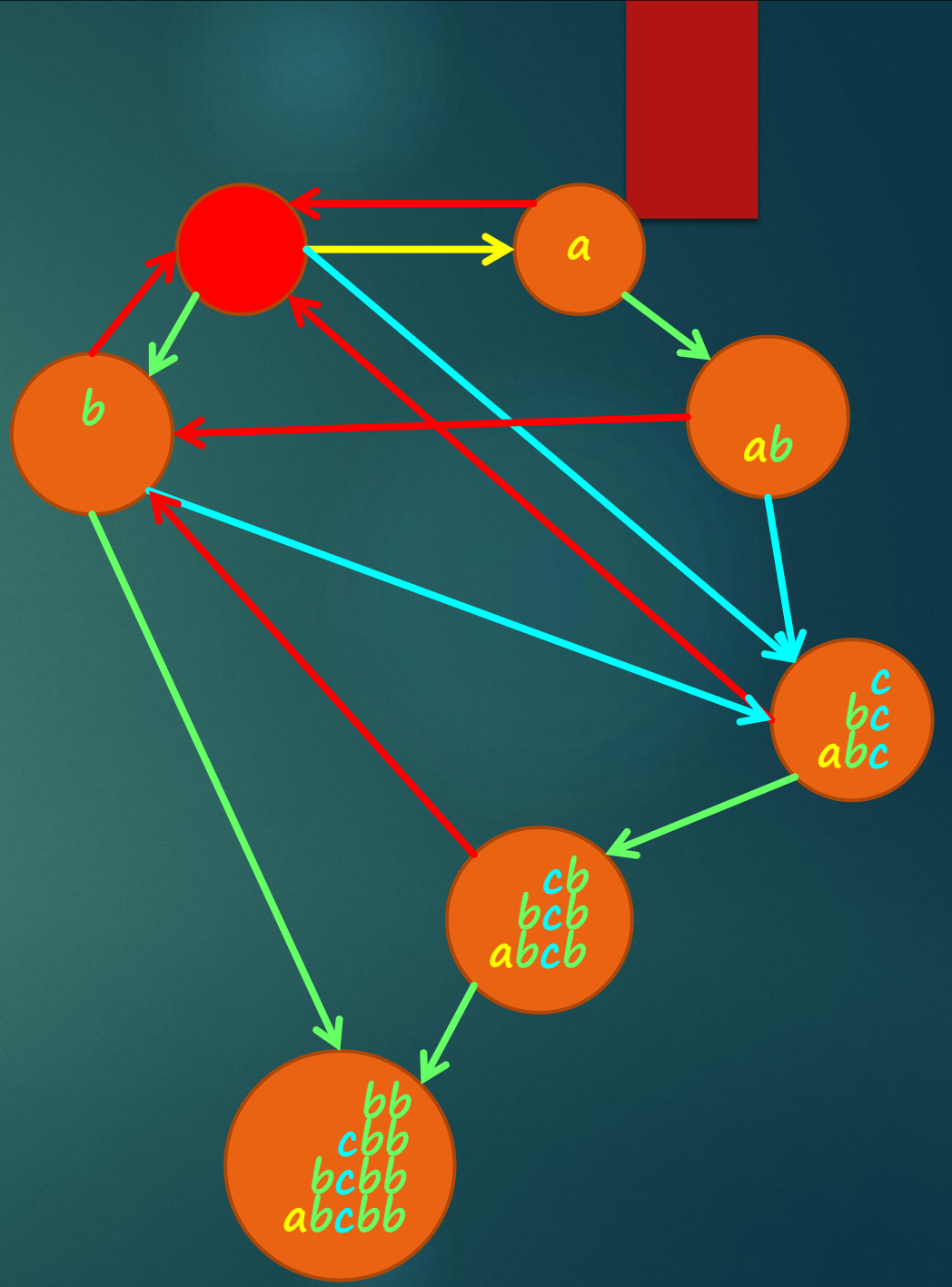
构造——更进一步

- 现在可以确定新建节点的父亲了
- 排序为[空串],[“b”],[“cb”,”bcb”,”abcb”]
- 根据定义，我们可以确定[“b”]的父亲是[空串]，[“cb”,”bcb”,”abcb”]的父亲是[“b”]
- 但是，[“ab”]的父亲怎么办？
- 我们知道，新建这个节点的时候字符串是”ab”，它的3个后缀分别是空串,”b”,”ab”，对应节点的排序是[空串],[“b”],[“ab”]，所以这个节点的父亲应该是[“b”]
- （我们注意到在这个排序中[“b”]的父亲仍然是[空串]）
- 现在这个就是”abcb”的后缀自动机



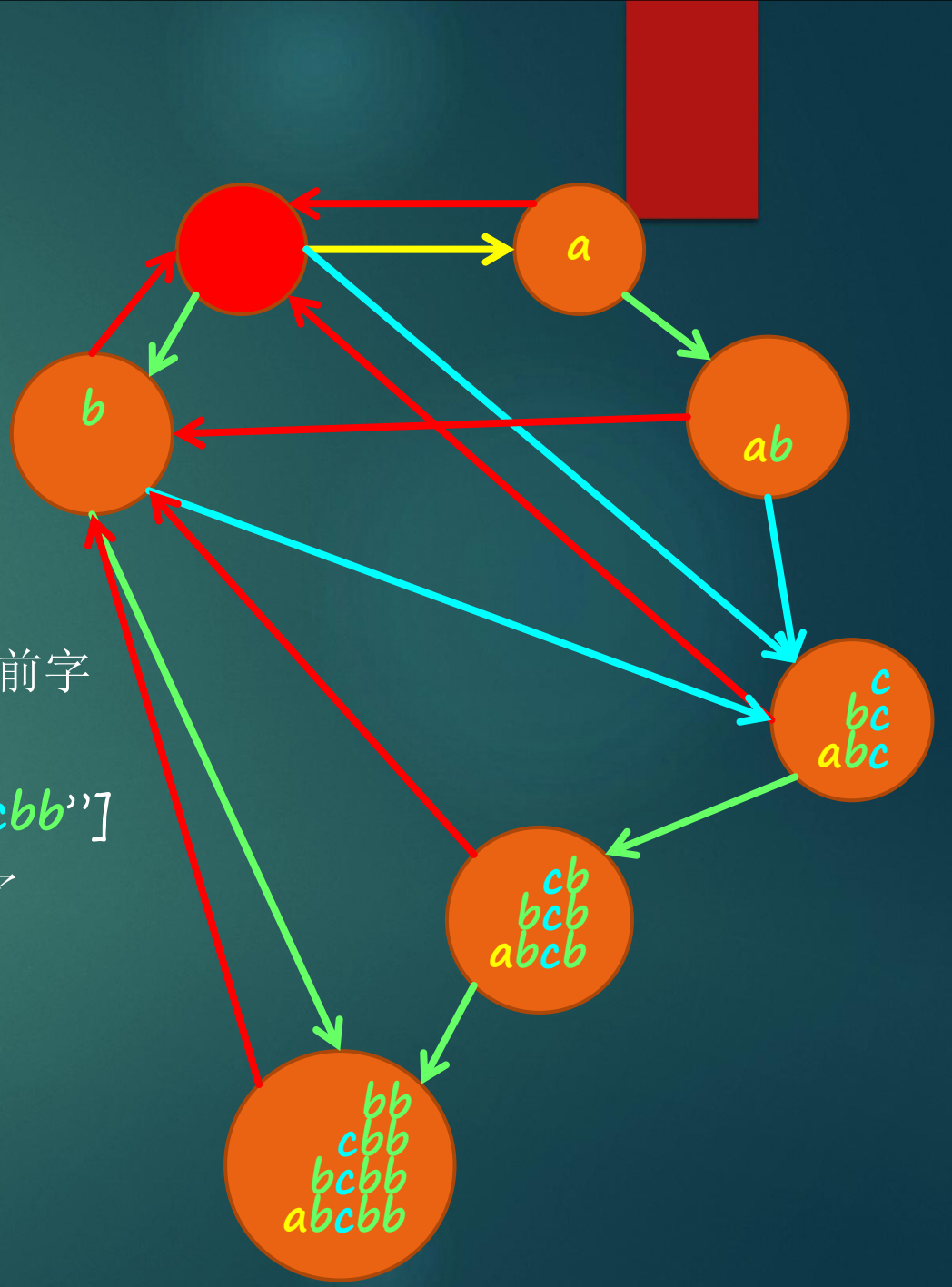
构造——更进一步

- 我们试试往后再加一个“b”会发生什么事情
- 上一个排序是[空串],[“b”],[“cb”,”bcb”,”abcb”]
- 所以首先我们新建一个节点，从”abcb”所在的节点连出一条”b”的转移边
- 我们知道排序中上一个元素就是这个节点的父亲
- 同样连出一条”b”的转移边
- 接着是根节点
- 我们发现根节点已经有”b”的转移边了，也就是说，字符串”b”已经出现过了



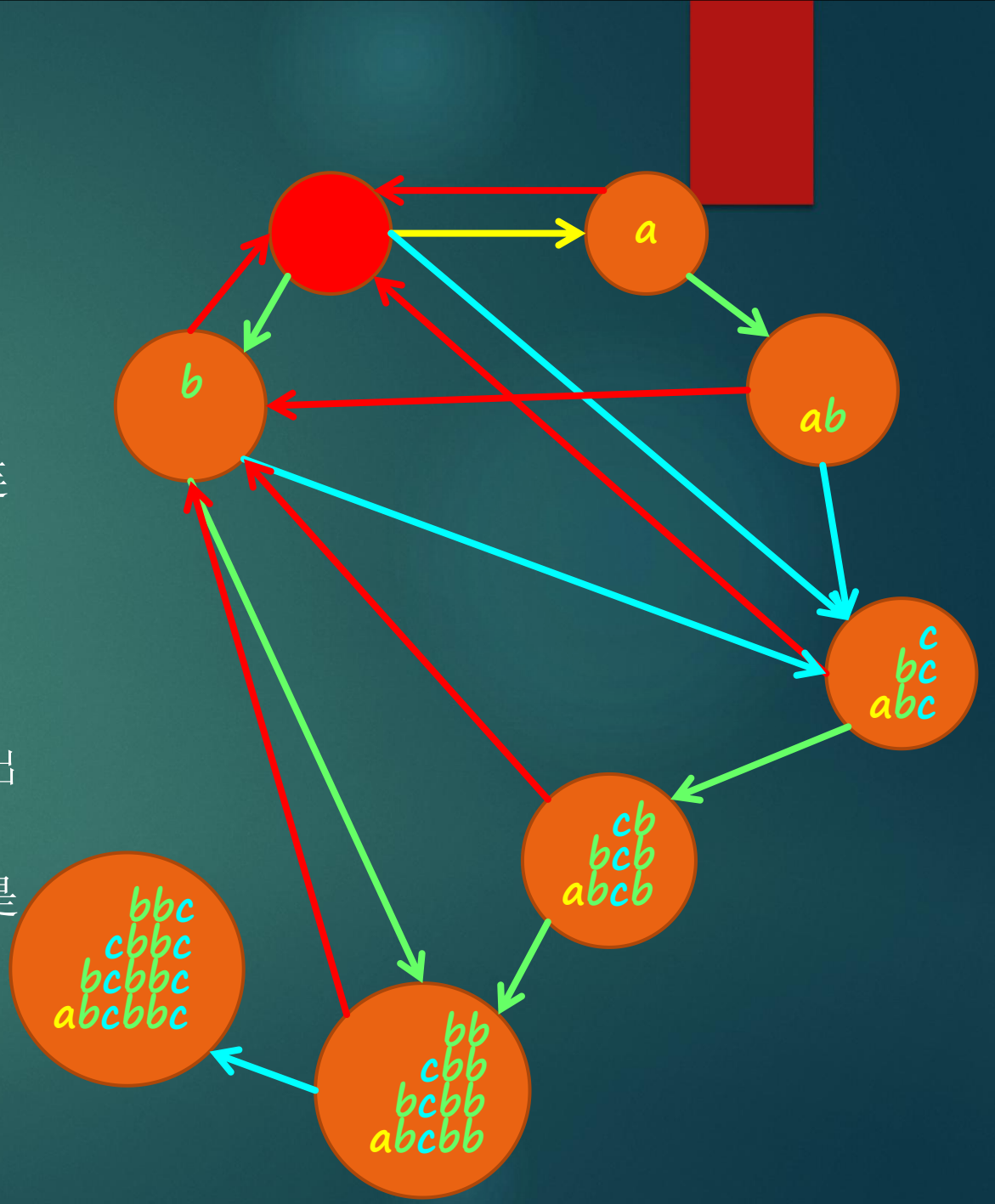
构造——更进一步

- 按照原来的做法，我们要先拆节点
- 但是，我们现在恰好找到了一个不重不漏地接受当前字符串的所有后缀的节点集合
- 且排序为[空串],[“b”],[“bb”,”cbb”,”bcbb”,”abcbb”]
- 于是我们直接将新建节点的父亲设为[“b”]就可以了



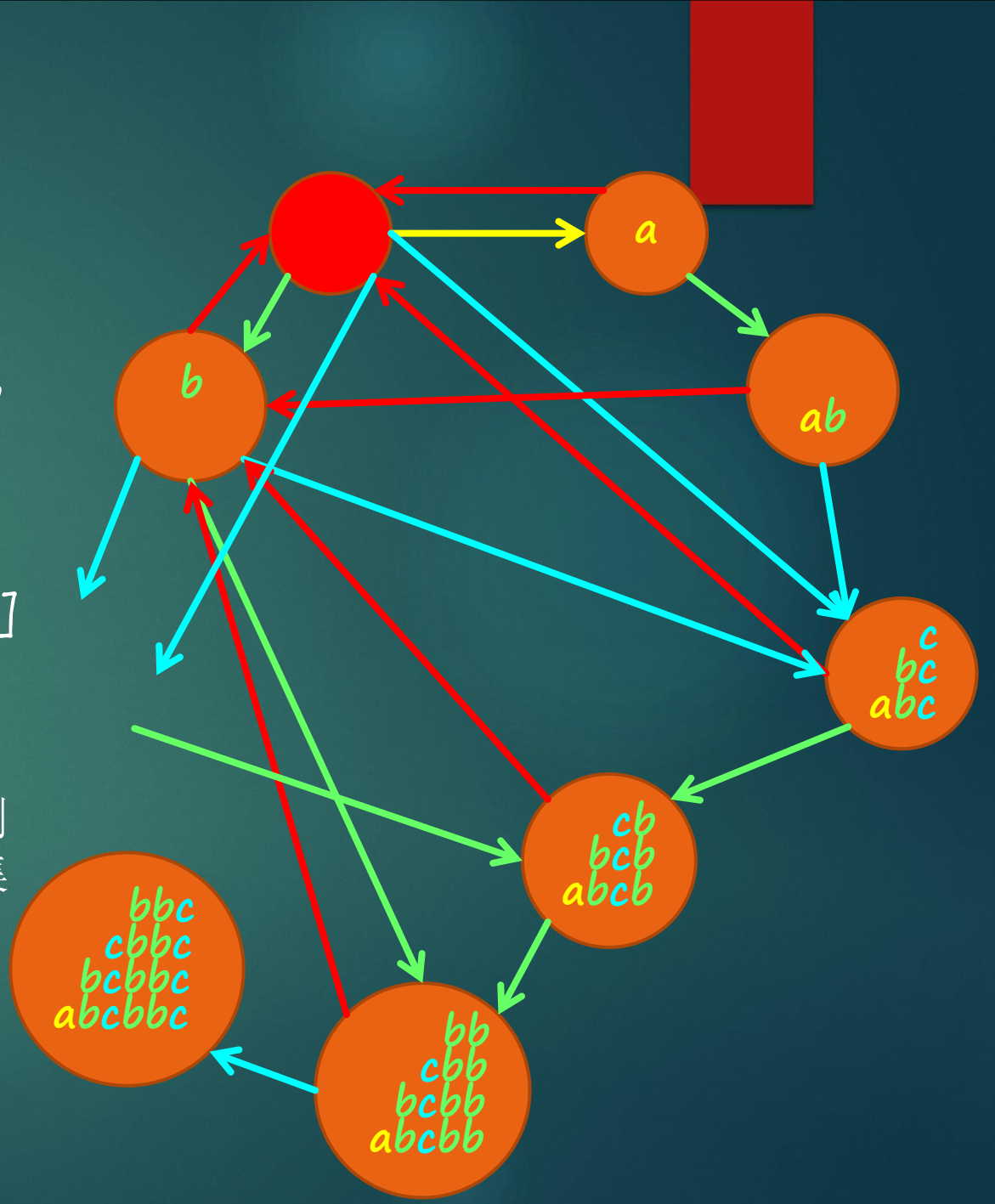
构造——更进一步

- 如果我们再加一个“c”呢？
- 同样地，新建一个节点，从接受原串的节点连一条“c”的转移边
- 接着访问排序的上一个节点
- 我们发现它已经有一条“c”的转移边了
- 也就是说，字符串“bc”以及它的所有后缀都出现过了
- 我们发现，对于这个节点，前面的“c”,“bc”是我们需要的，后面的“abc”是我们不需要的
- 于是我们考虑把[“c”,“bc”,“abc”]拆成[“c”,“bc”]和[“abc”]两个部分



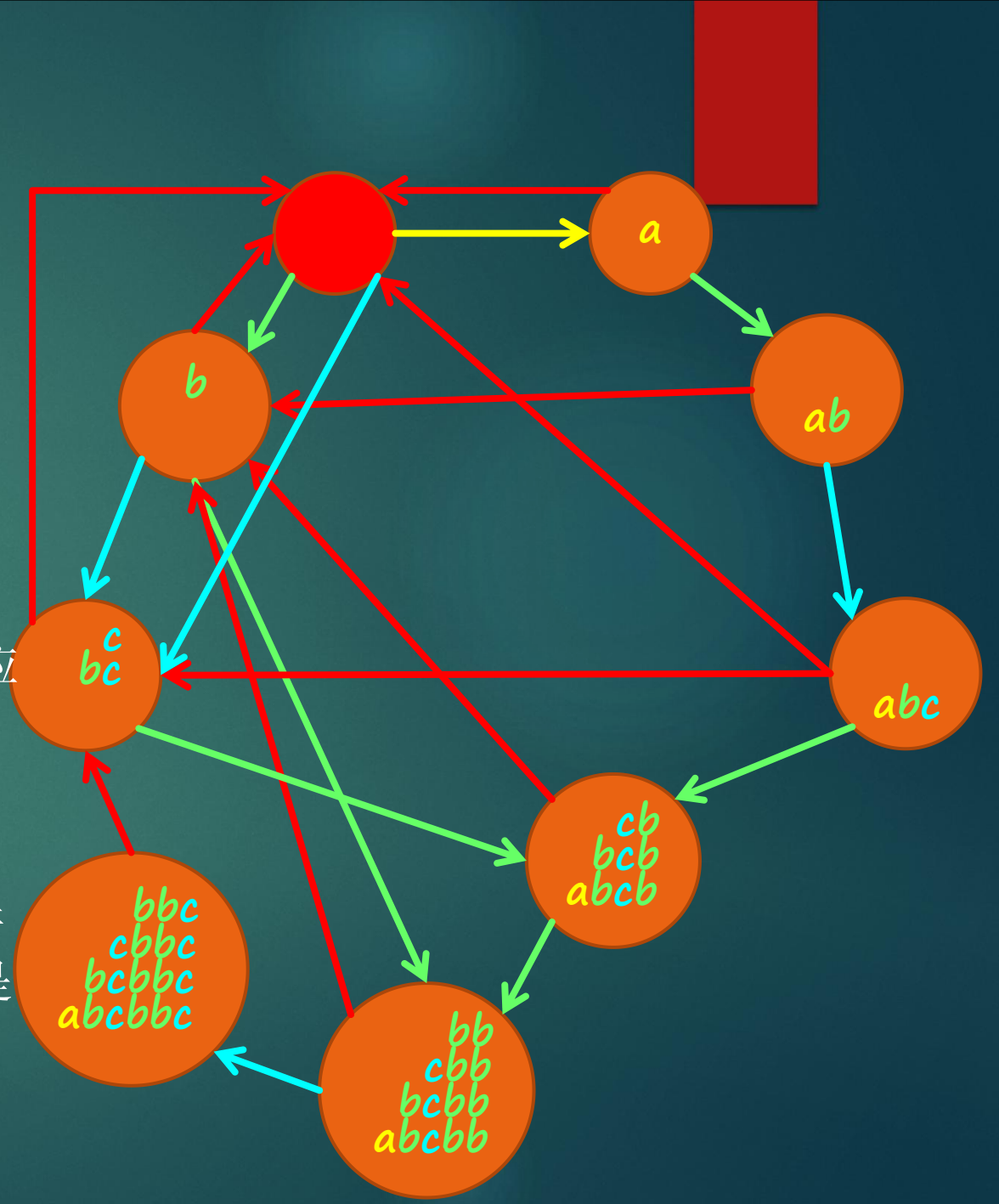
构造——更进一步

- 我们新建一个节点，试图让它接受“c”和“bc”
- 我们知道，原节点能接受“bc”是因为节点[“b”]向原节点有一条“c”的转移边
- 我们知道，原节点能接受“c”是因为节点[“b”]的父亲向原节点有一条“c”的转移边
- 于是我们把这两条边全部改成指向新节点
- 然后仍需要把从原节点出发的所有转移边复制到新节点上才能保证其它节点接受的字符串集合不变
- 现在我们成功地把[“c”,“bc”]拆了出来



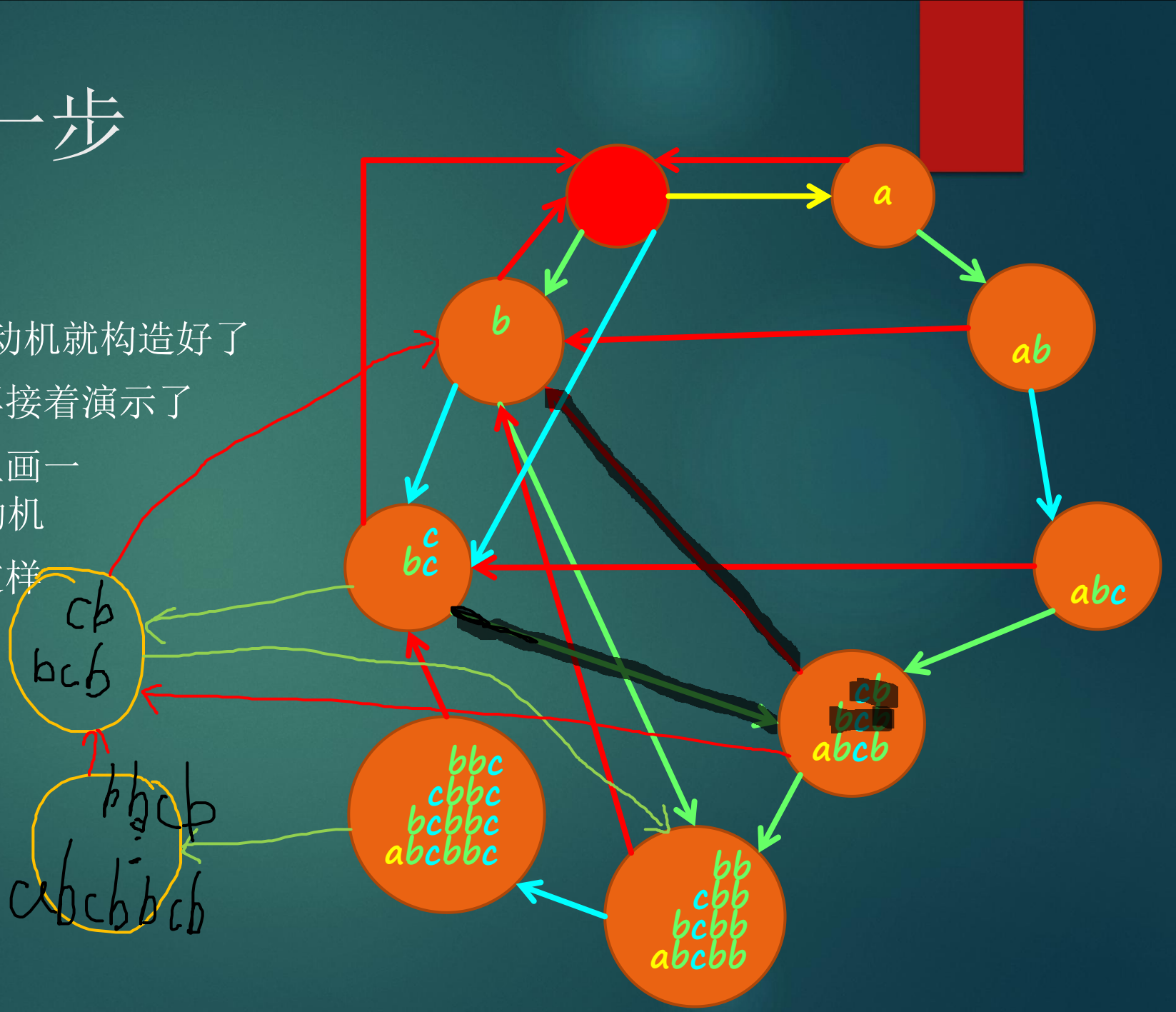
构造——更进一步

- 然后要确定新建节点的父亲
- 现在的排序是[空串],[“c”,”bc”],[“bbc”,”cbbc”,”bcbbc”,”abcbbc”]
- 所以[“bbc”,”cbbc”,”bcbbc”,”abcbbc”]的父亲应该是[“c”,”bc”]，[“c”,”bc”]的父亲应该是根
- 还有[“abc”]是被拆出来的节点，它的父亲也会改变
- 新建这个节点的时候字符串是”abc”，它的所有后缀对应的节点在现在的自动机上的排序是[空串],[“c”,”bc”],[“abc”]
- 所以它的父亲应该是[“c”,”bc”]



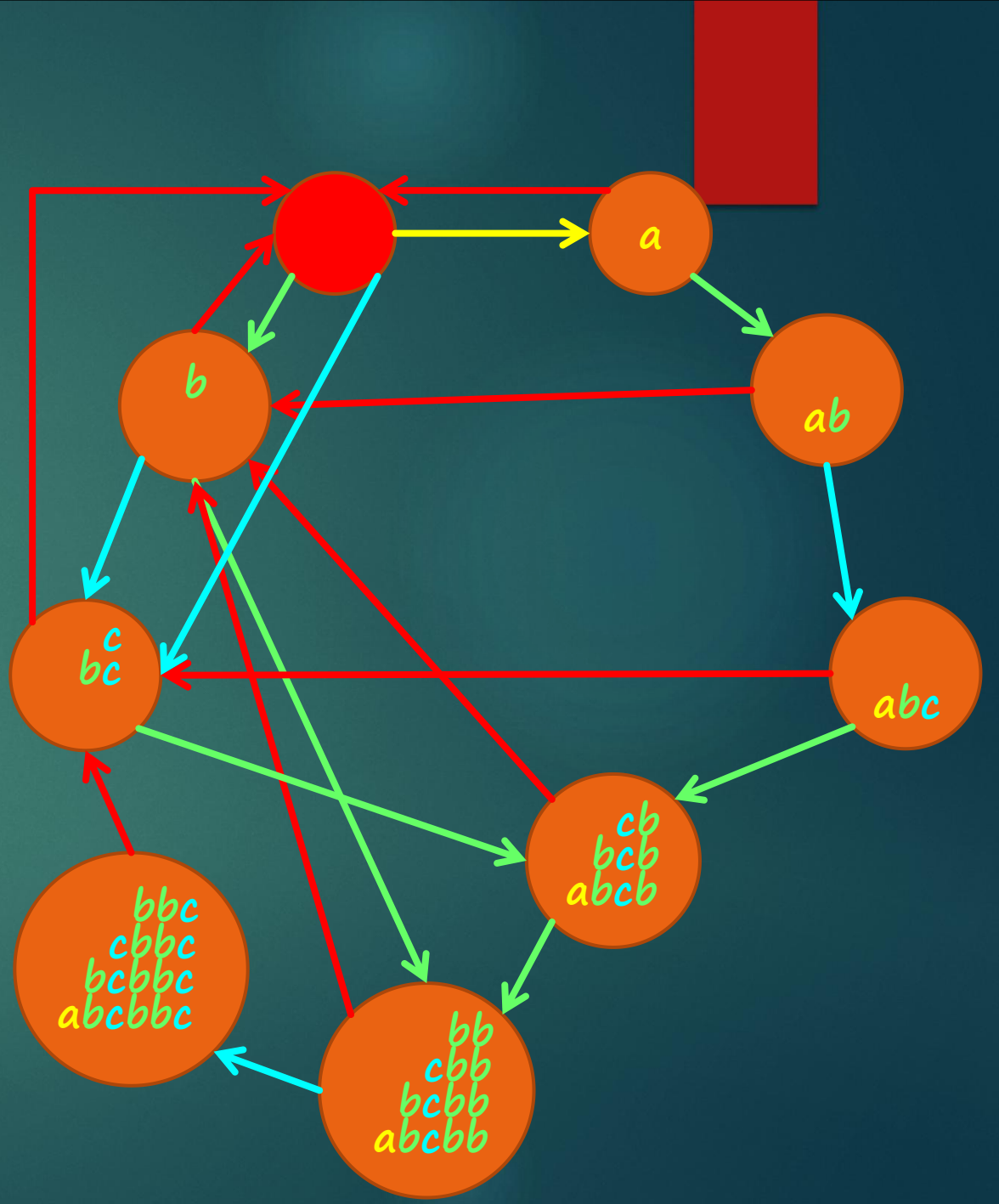
构造——更进一步

- 现在''abcbbc''的后缀自动机就构造好了
- 由于幻灯片太小，就不再接着演示了
- 你可以试着在这个基础上画一下''abcbbcb''的后缀自动机
- 当然我也画了，大概长这样
- （画面略(te)丑请无视）



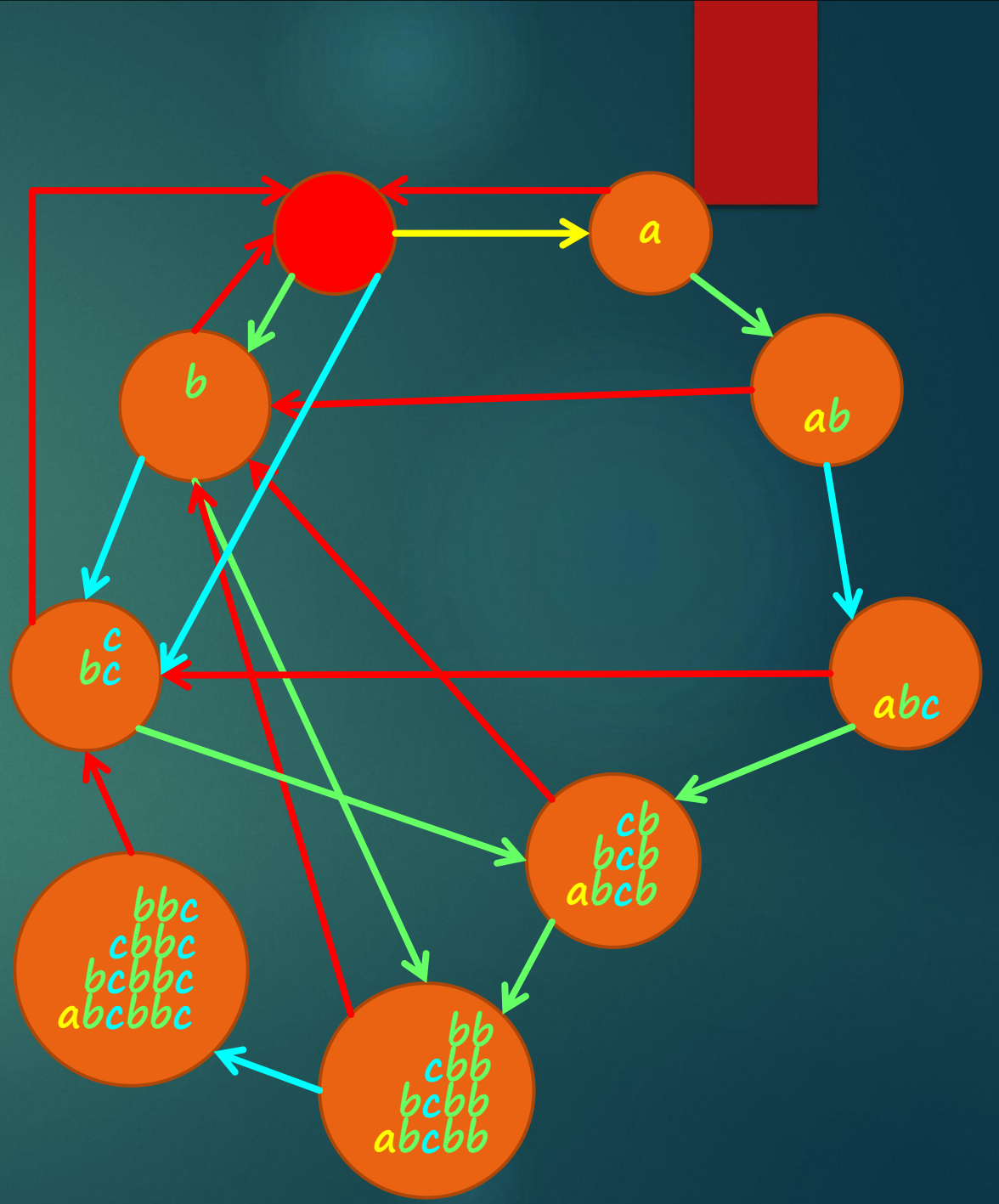
构造——实现

- 想必你们会自己手动构造后缀自动机了吧
- 那我们现在来讲讲代码实现
- 首先，对于一个节点，我们肯定要把从这个点出发的每种转移边到达的节点记下来，以及把它的父亲记下来
- 接着，我们需要记录节点上的字符串的信息
- 但是，如果像图中一样把节点接受的所有字符串都记下来，那肯定太浪费了
- 所以我们要记录什么呢？



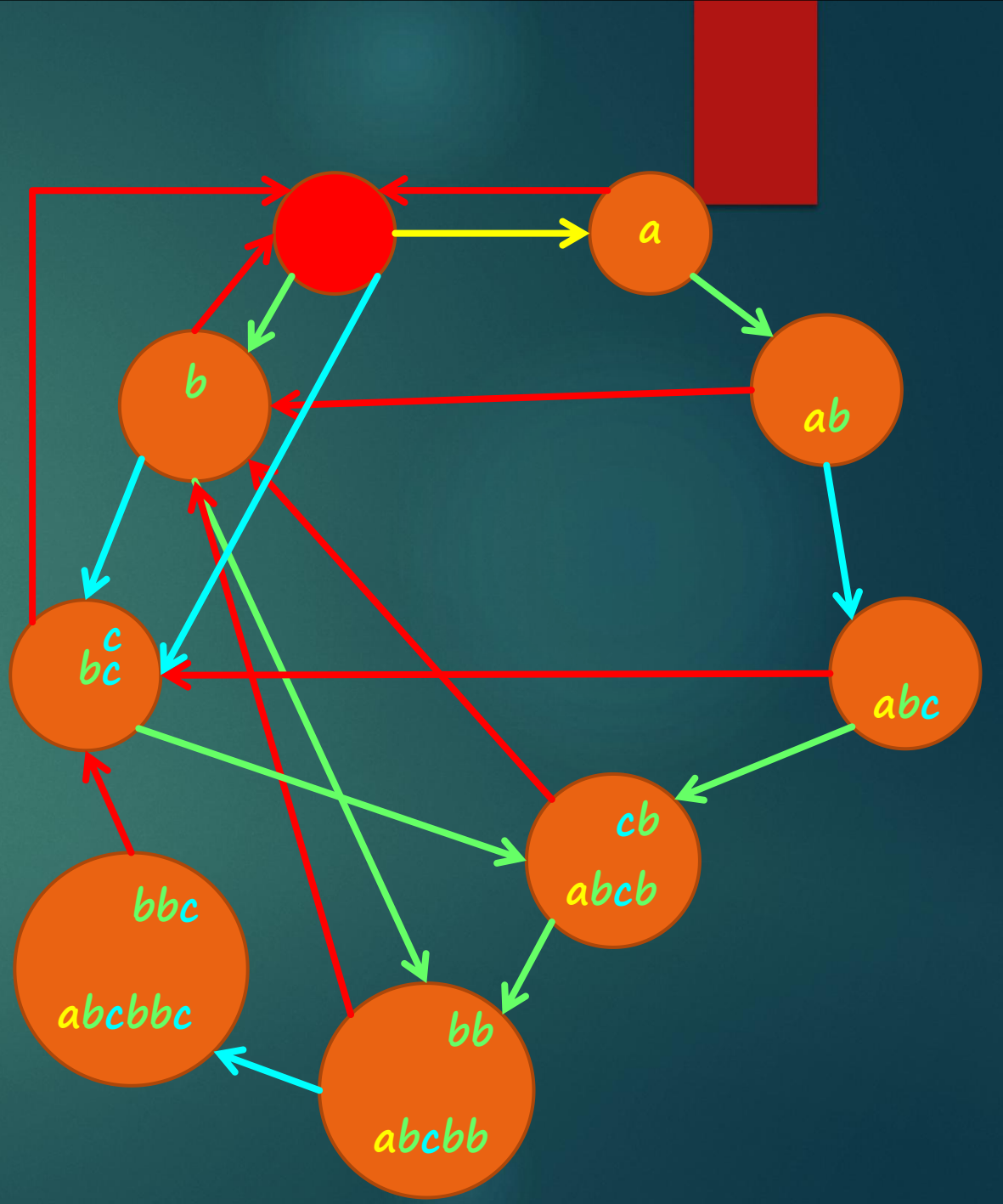
构造——实现

- 首先考虑到之前提到的一个性质，每个节点接受的所有字符串都是其中最长的字符串的某个后缀，并且长度是一个（以该字符串长度为右端点的）连续的区间
- 于是我们可以考虑只记每个节点中最长的字符串以及最短的字符串
- 接着，考虑在一个排序中，显然上一个节点接受的最长字符串的长度比当前节点接受的最短字符串的长度小1
- 所以根据父亲的定义，一个节点接受的最短的字符串的长度就是其父亲接受的最长字符串的长度加1



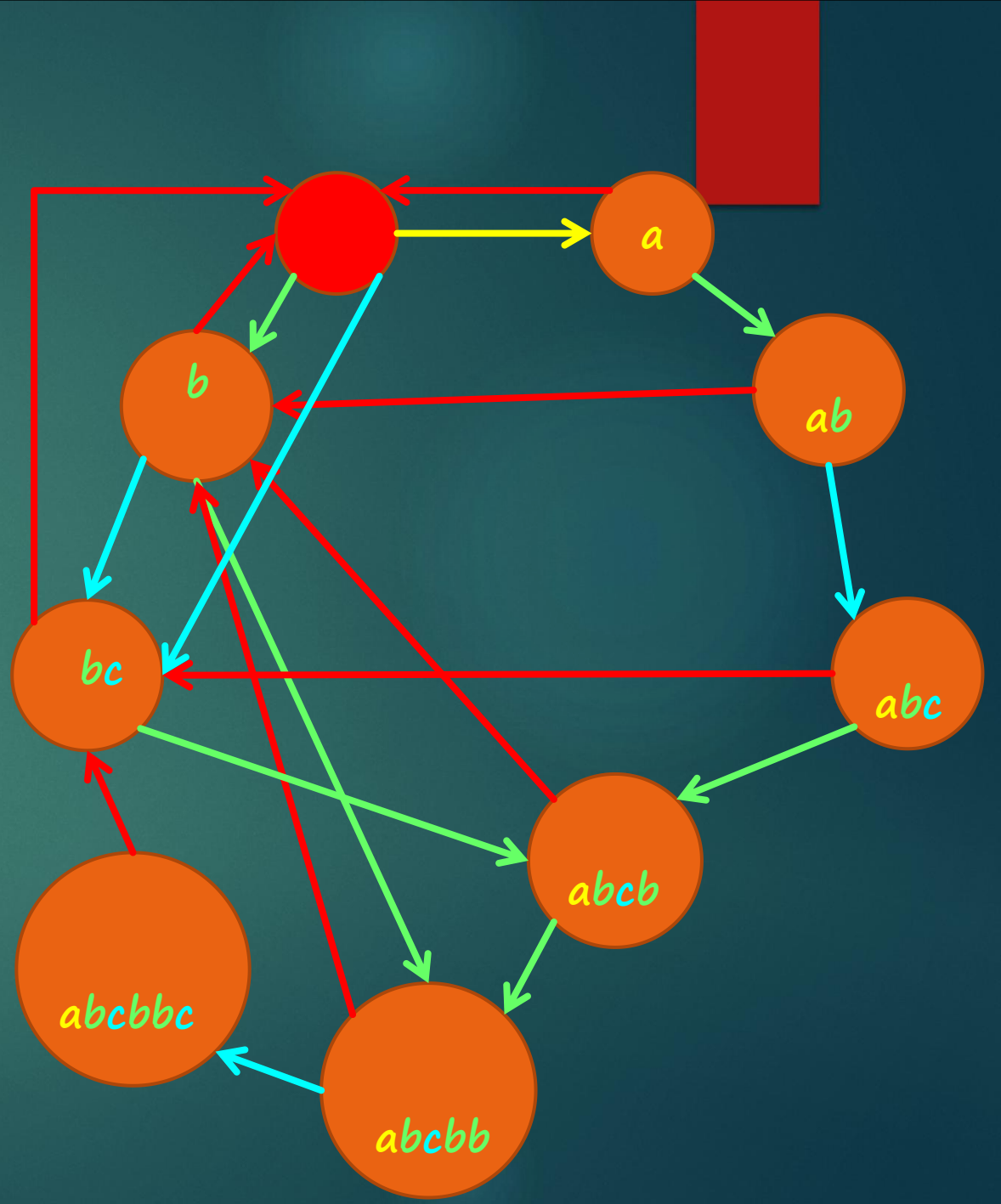
构造——实现

- 又因为最短的字符串的内容可以通过其长度以及最长的字符串的内容得到
- 所以我们只要记每个点接受的最长的字符串就可以了
- 最后，考虑到每个点接受的字符串都是原串（也就是你要对着它建后缀自动机的字符串）的子串
- 所以我们可以进一步简化我们记录的信息，对每个点只记录它接受的最长的字符串的长度和在原串中的位置
- （由于这一步画在图上太不直观我就不加动画演示了）



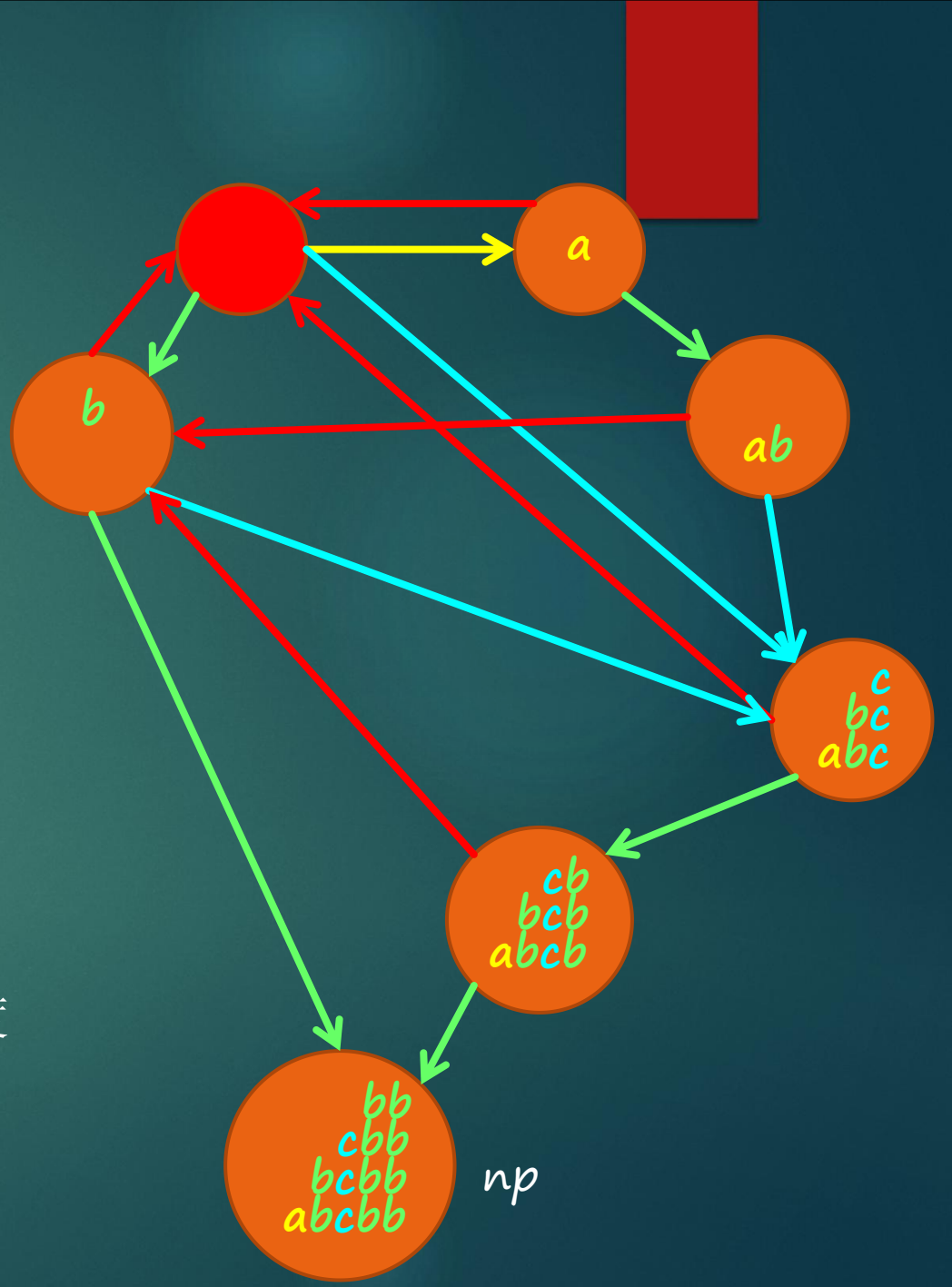
构造——实现

- 我们总结一下，对于每个节点一共要记的信息有：
- 1.指向父亲节点的指针
- 2.指向从该节点出发的每种转移边的到达节点的指针
- 3.该节点能接受的最长的字符串的长度以及它的位置（位置具体怎么记之后再说）
- 接下来为了让演示更直观，我仍然会把每个节点接受的所有字符串显示在节点上
- 之后有些页面我会把这个自动机放在画面上，是为了在讲一些性质的时候你们可以快速找到例子



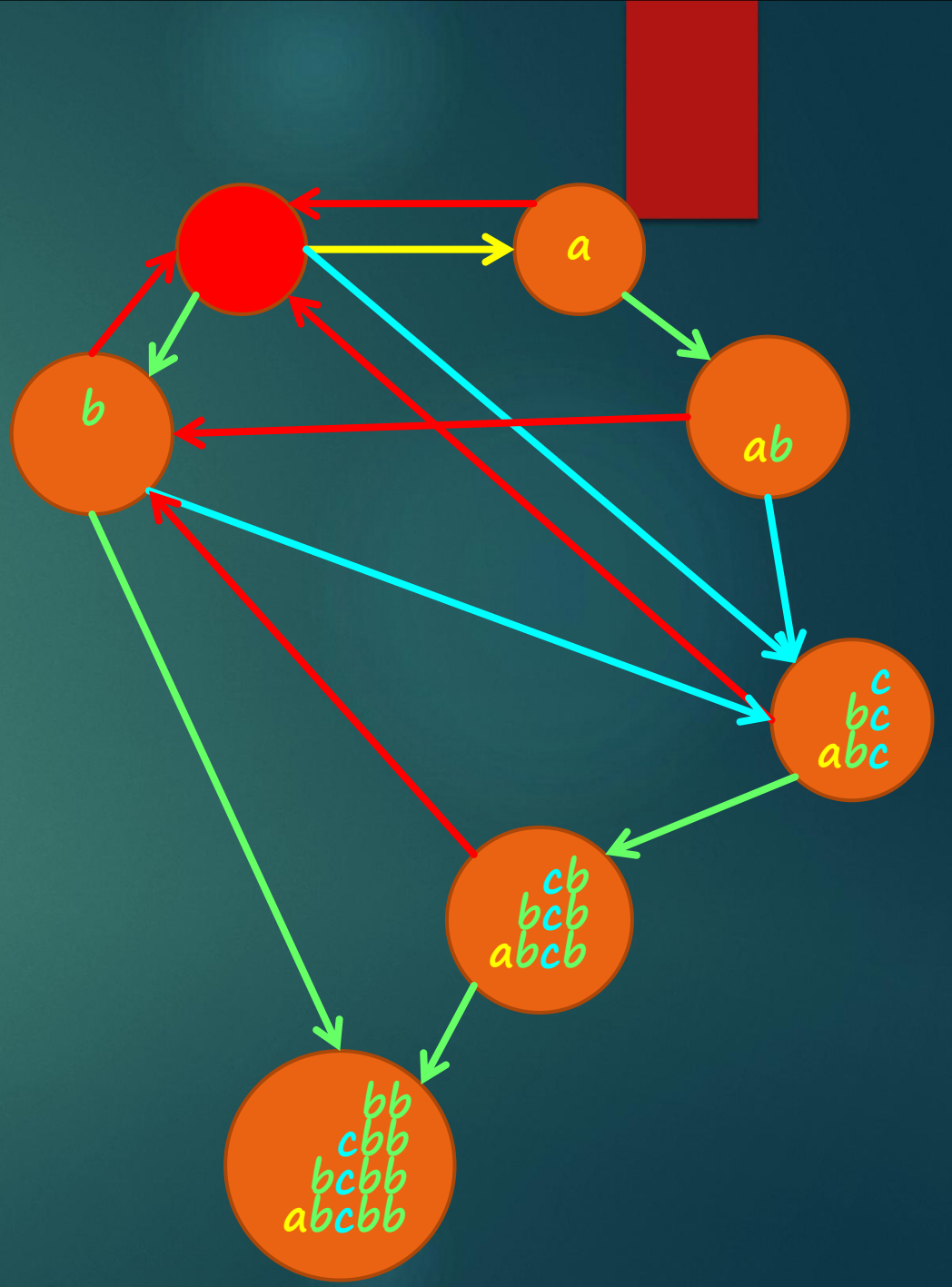
构造——实现

- 开始我们只有一个根节点
- 我们的构造思路是，从前往后逐个加入字符，在前一个后缀自动机的基础上构造后一个后缀自动机
- 在加一个字符时，我们要先找到之前接受原串的那个节点
- 这里我们可以开一个全局指针*last*来维护
- 这样，从*last*开始，顺着父亲边爬上去，我们就能按从后往前的顺序找到上一个排序中的所有节点
- 我们新建一个节点*np*，把它接受的最大字符串长度设为当前字符串长度，在爬的过程中往新节点加一条关于新字符的转移边，直到发现有一个节点已经有了该字母的转移边或者直到加完根的出边为止



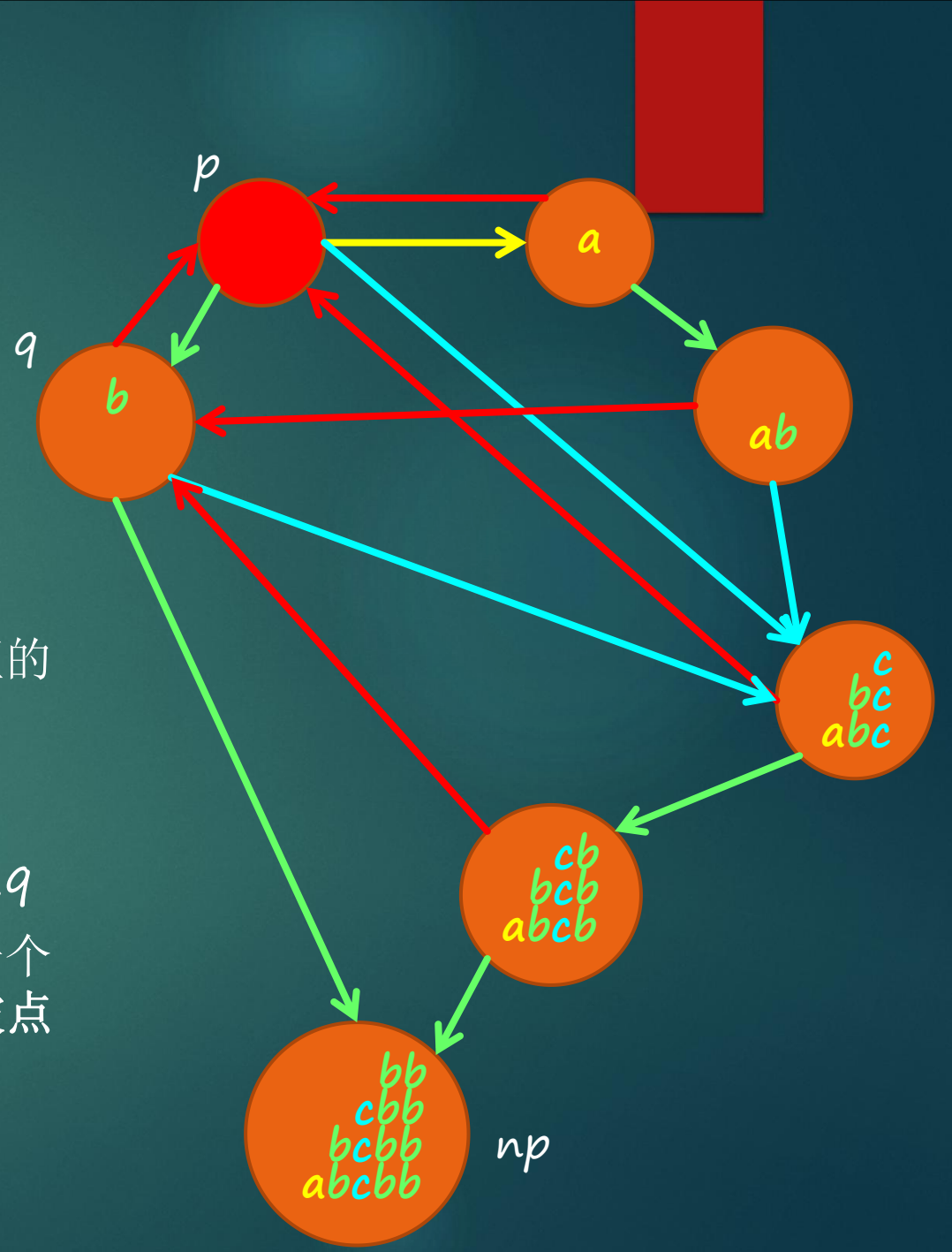
构造——实现

- 这样我们就加完了所有新出现的子串
- 证明？
- 首先所有新出现的子串一定是新串的后缀，否则它会在原串中出现
- 其次如果新串的一个后缀在原串中出现，那么长度比它短的所有后缀也都在原串中出现，因为这些串是那个后缀的子串
- 由于原串的后缀自动机接受了原串的所有子串，所以我们顺着排序爬上去找到第一条已有的转移边指向的节点一定接受了最长的新串的后缀满足它在原串中出现，以及比它长的后缀所在的节点已经全部指向了新点



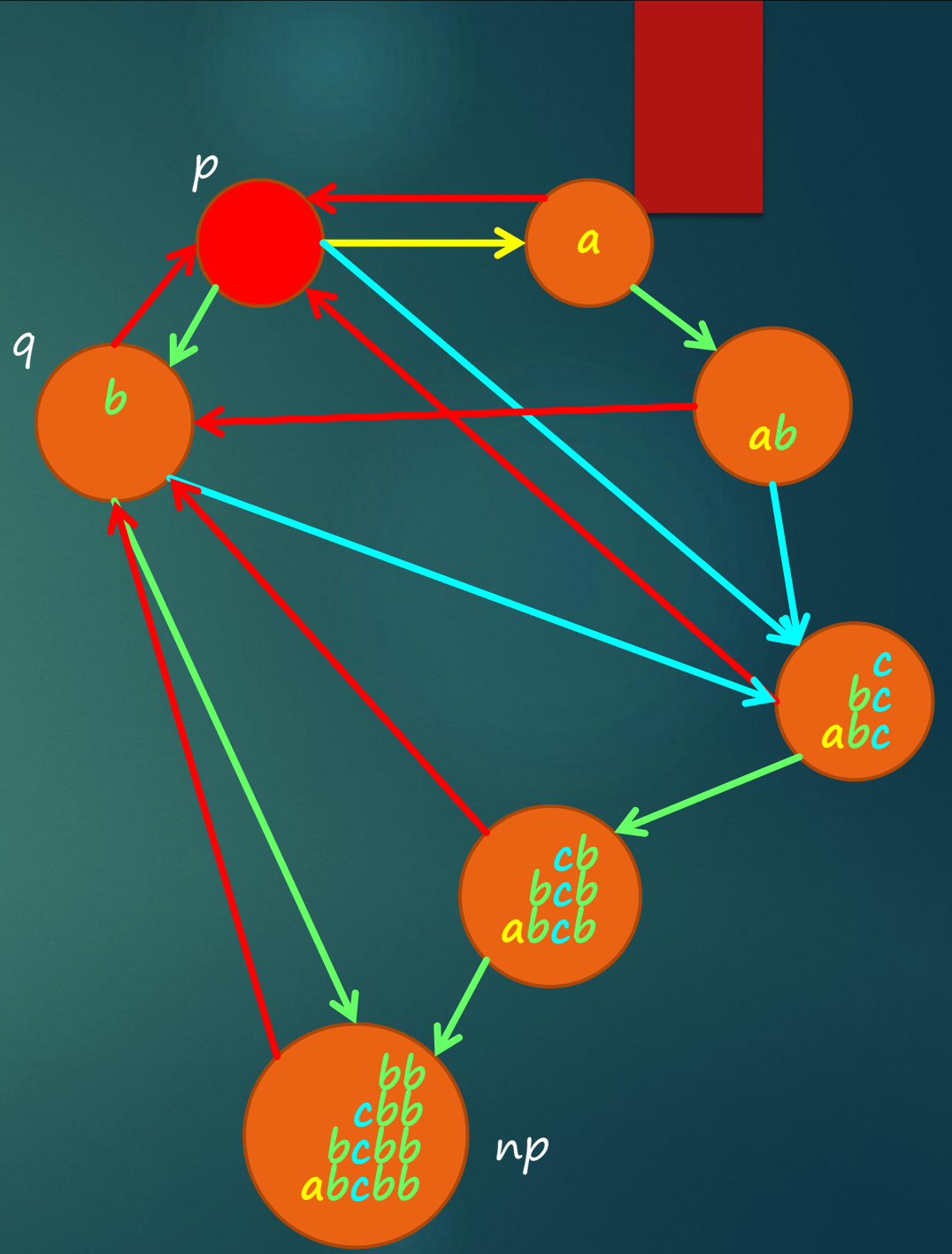
构造——实现

- 如果是遇到了一个点已经有了新字符的出边
- 我们把这个点叫做点p
- 那么我们先算一下np能接受的最短字符串的长度
- 它等于点p在我们爬上来的链上的儿子接受的最短的字符串长度加1
- 也就是点p接受的最长字符串长度加2
- 接着我们把点p关于新字符的转移边指向的点叫点q
- 我们可以知道点p接受的长度最大的串之后加上一个新字符所构成的串（在这里是空串+"b"="b"）**被点q接受**（因为点p关于新字符的转移边指向了点q）并且这个串是最长的出现在原串中的**新串后缀**



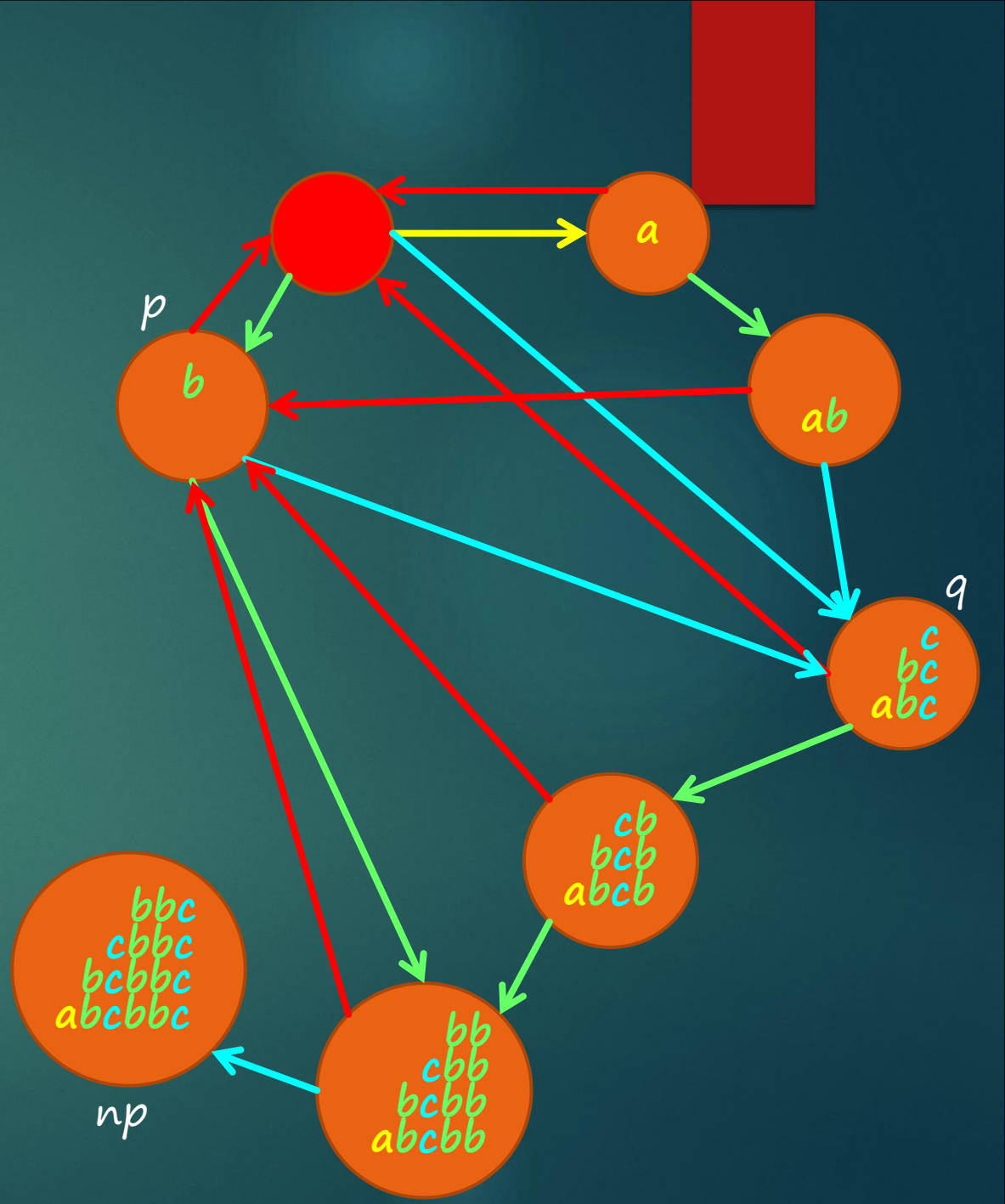
构造——实现

- 那么现在可能有两种情况
- 第一种是像这个图上的一样，正好找到了一个不重不漏地接受新串的所有后缀的节点集合
- 此时满足np接受的最短字符串长度等于点q接受的最长字符串长度加1，也就是说，点q接受的最长字符串的长度是点p接受的最长字符串的长度加1
- 遇到这种情况我们只需要直接把新节点的父亲设为点q即可
- 当然，记得把last更新为np



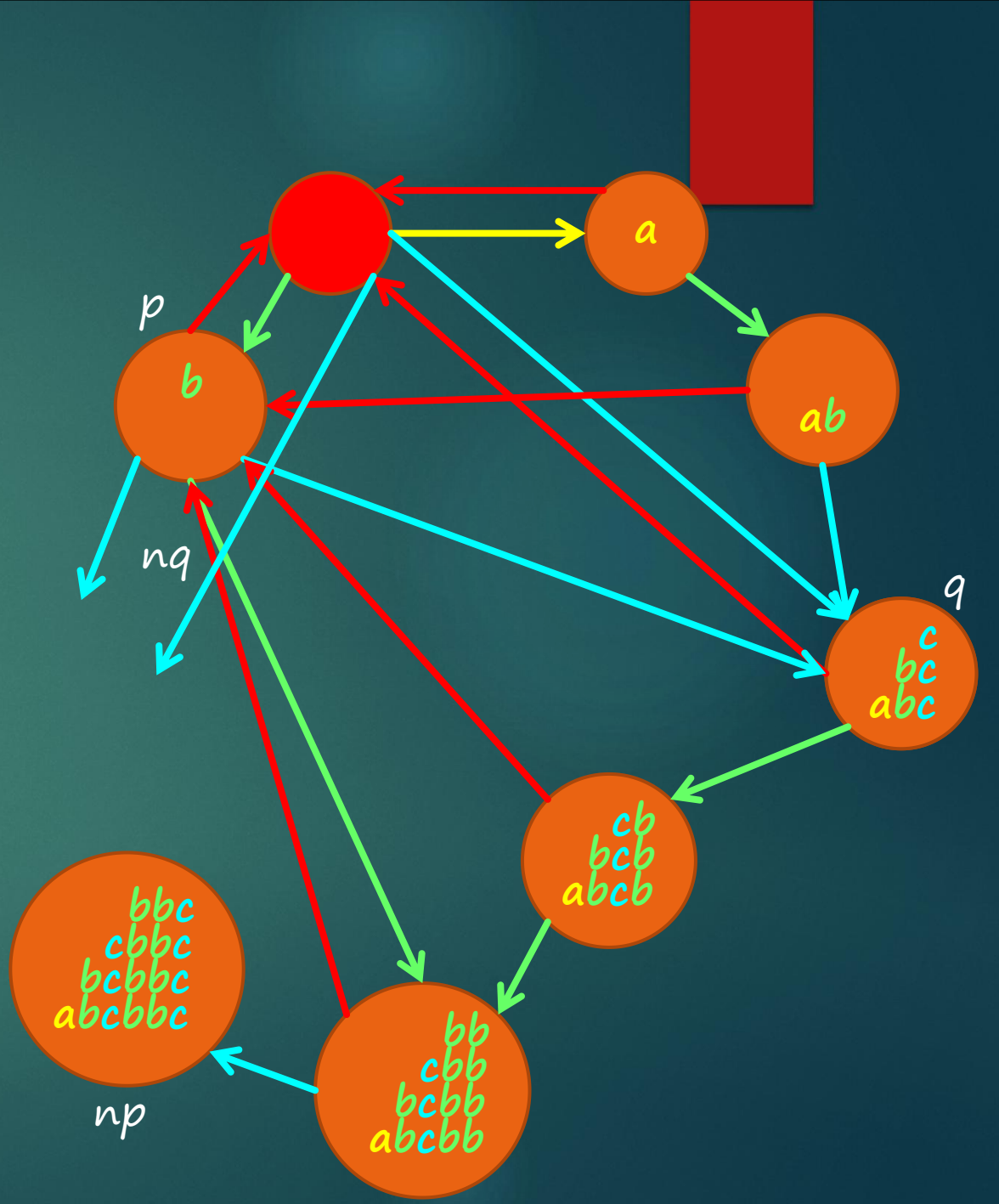
构造——实现

- 第二种情况是，点p接受的最长字符串长度加1不等于点q接受的最长字符串长度
- 在这个例子中，这个是点p
- 这个是点q
- 我们知道点q接受了最长的出现在原串中的新串后缀
- 但此时意味着点q还接受了一些比最长的出现在原串中的新串后缀还长的字符串
- 于是我们就要考虑把点q拆开
- 具体的拆法是这样的：



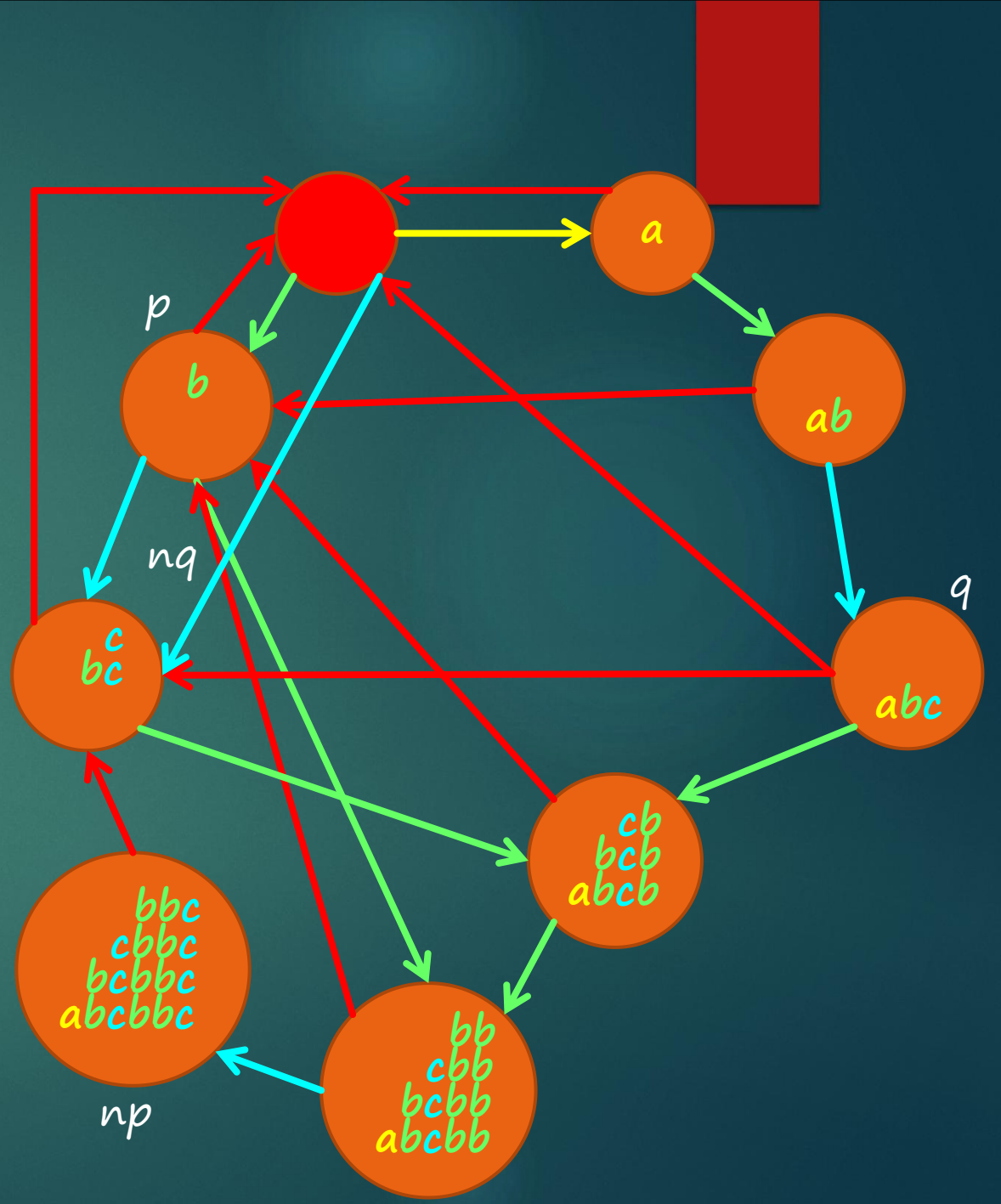
构造——实现

- 首先新建一个节点nq，准备用来接受我们需要的串
- 我们知道点q能够接受最长的出现在原串中的新串后缀（本图中是''bc''）是因为点p向它连了一条关于新字符的转移边
- 以及该串的后缀去掉最后一个字符后一定是点p接受的最长的串的一个后缀（在本图中''bc''和''c''去掉最后一个字符''c''后都是''b''的后缀）
- 所以为了让nq接受你需要的串，你只需要访问p到其祖先的一条链，将指向q的关于新字符的转移边改为指向nq即可（因为顺着父亲链往上爬可以访问到一个串的所有后缀）



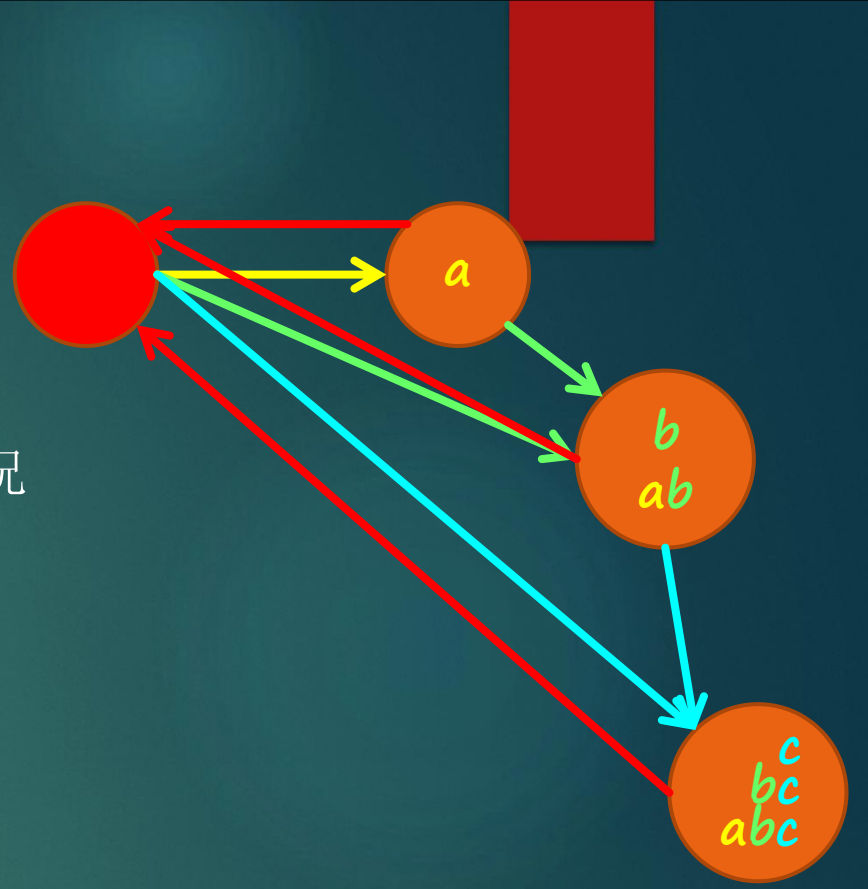
构造——实现

- 注意到一个点接受的字符串的长度是连续的一段区间，所以在访问p的祖先的时候，一旦遇到一个没有连向q的点，就可以跳出循环
- 接着为了使其它节点接受的字符串不变，你需要把从q出发的所有转移边复制到nq上
- 注意要将nq接受的最大字符串长度设为p接受的最大字符串长度加1
- 最后是确定节点的父亲
- nq的父亲显然要设为原来q的父亲（这一步可以在复制的时候完成）
- np和q的父亲显然要设为nq



构造——实现

- 之前只讨论了遇到一个已经有新字符的转移边的节点的情况
- 如果没有遇到，那么就会加到根节点的出边
- 那么说明新节点接受的最短字符串长度为1
- 那么在排序中，它的前一个节点显然是根节点[空串]
- 所以此时只需直接把新节点的父亲设为根节点
- 当然记得把last更新为np（上一种情况也一样，因为页面不够放了就没写了）



构造——实现

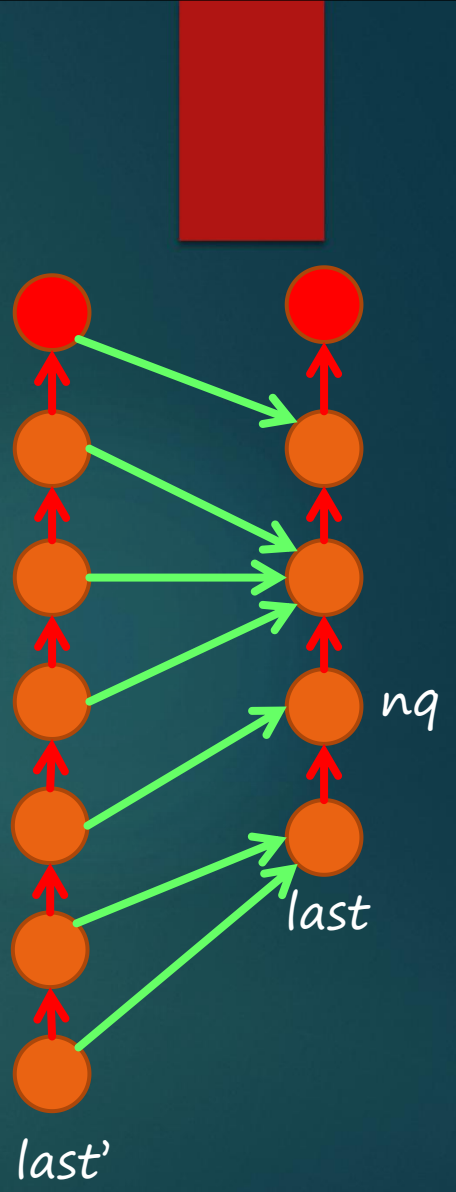
- 现在后缀自动机的构造过程就讲完了
- 但是我们还有一个东西迟迟没有提到
- 那就是关于如何记录每个点接受的最长的字符串的位置
- 首先刚才的步骤告诉我们构造后缀自动机的时候并不必要每次都记录这个位置，因为构造过程是不需要用到这个量的
- 然后我们可以发现无论是新建*np*还是*nq*的时候它接受的最长的字符串一定是当前字符串（新串）的一个后缀（事实上*np*接受的最长的字符串是新串本身）
- 所以我们可以在新建节点的时候记录下当前字符串的长度（或者说是新建这个节点的时刻），这就是该节点接受的最长的字符串在原串中的一个右端点所在的位置，我们把它叫做这个节点的*at*
- 我们可以把一个节点接受的最长的字符串的长度叫做这个节点的*max*

构造——时间复杂度

- 用上述方法构造后缀自动机的时间复杂度是O(串长×字符集大小)
- ~~（如果字符集很大你可以用map/主席树来存边，时间复杂度是O(串长log字符集大小)）~~
- 证明如下：
- 设串长为n
- 首先节点的个数是O(n)的，因为每增加一个字符最多只会新建两个节点np和nq
- （这意味着要注意开两倍数组！）
- 接着从last开始跳父亲这一步是均摊O(n)的，原因如下：
- 因为每次加字符时last一定会连出一条指向np的转移边，所以加字符之后last的父亲的max不会超过加字符之前last的父亲的max加1
- 所以last的父亲的max的总增加量是O(n)的，自然last的父亲的max的总减少量也是O(n)的，又因为每跳一步都会使last的父亲的max减少，所以结论得证

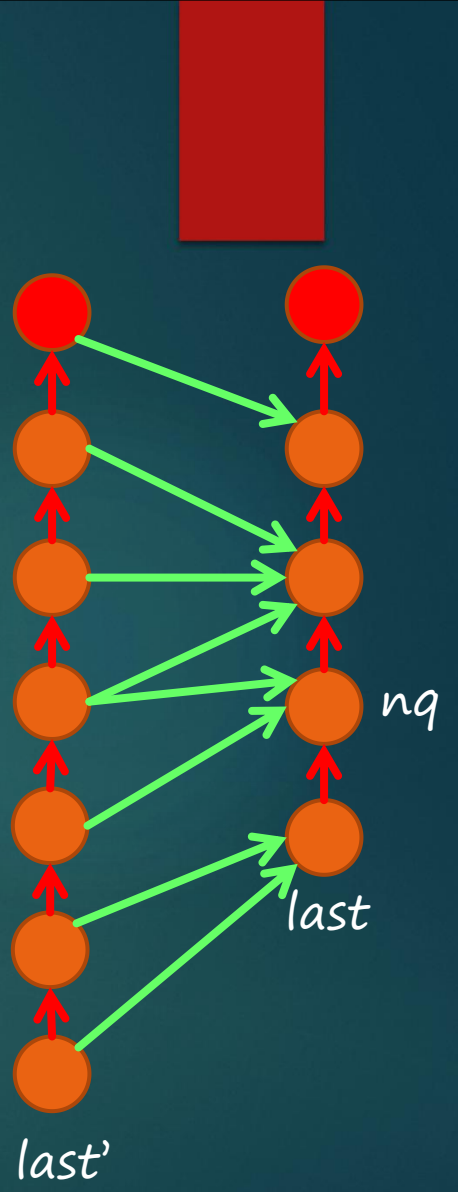
构造——时间复杂度

- 接着，复制节点(q→nq)的时间复杂度是O(n×字符集大小)
- 最后，跳p的父亲把指向q的转移边改为指向nq也是O(n)的，原因如下：
- 考虑加字符前的last'的所有祖先和加字符后的last的所有祖先的关系
- 根据父亲的定义以及节点接受的字符串的性质，可以证明last'的每个祖先都向last的一个非根祖先连了一条转移边
- 并且对于last的每个非根祖先，对应到last'上的连边节点集合是last'祖先链上的一段非空区间
- 并且last'深度越大的祖先会连向last深度越大的祖先（即画在右边的图上不会出现两条交叉的边）
- 跳p的父亲的次数等于连向nq的转移边的条数
- 如果存在nq的话，那么nq是last的父亲



构造——时间复杂度

- 因为多一条指向nq的边，nq的父亲的max就会减少（last的一个非根祖先的max等于连向它的离根最远的last'的祖先的max加1）
- 所以我们考虑last的父亲的父亲的max（若不存在则视为0）的变化
- 因为对于last的每个非根祖先，对应到last'上的连边节点集合是last'祖先链上的一段**非空**区间
- 所以连向last的父亲的父亲的离根最远的last'的祖先一定不是last'的父亲或last'
- 所以last的父亲的父亲的max最多是last'的父亲的父亲的max加1
- 所以last的父亲的父亲的max的总增加量是O(n)的
- 所以last的父亲的父亲的max的总减少量也是O(n)的
- 所以跳p的父亲的总次数也是O(n)的



性质——状态

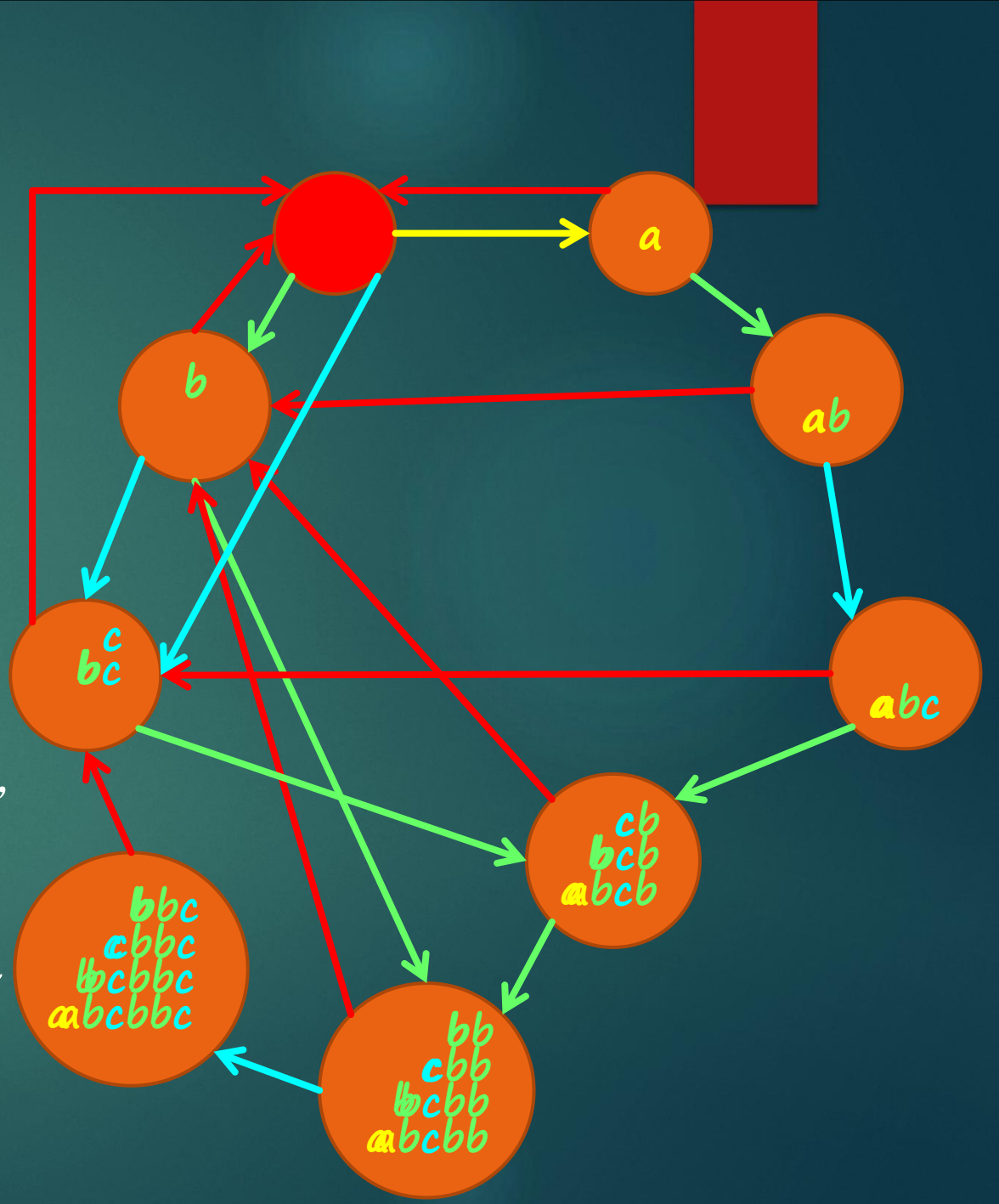
- 接下来说一说后缀自动机的一些性质
- 首先就是状态，也就是说，我们要如何在后缀自动机上表示原串的一个子串
- 后缀自动机不像*trie*和*AC*自动机那样一个节点只用来接受一个字符串
- 所以直接用字符串所在节点是不够的
- 由于一个节点接受的字符串长度互不相同
- 所以我们可以在状态中多记录一个长度来确定当前匹配的字符串
- 所以我们称一个状态为一个（当前节点，长度）的*pair*
- ~~（这和自动机对状态的定义略有不同）~~
- 由于一个字符串不同子串的个数是*O(n^2)*的，所以状态的总数是*O(n^2)*的
- 为了方便，我们把一个状态对应的字符串和这个状态不加以区分

性质——树

- 每个节点的父亲的max（如果存在的话）都比自己的max小（根据父亲的定义得到）
- max不超过0的节点只有根节点
- 除了根节点没有父亲，每个节点都只有一个父亲
- 所以后缀自动机的所有节点和所有父亲边构成了一棵树
- 后缀自动机的根就是树的根
- 每个节点的父亲就是它在树上的父亲
- 根据父亲的定义，一个状态去掉首字母后所在节点要么是原来的所在节点，要么是原来所在节点的父亲

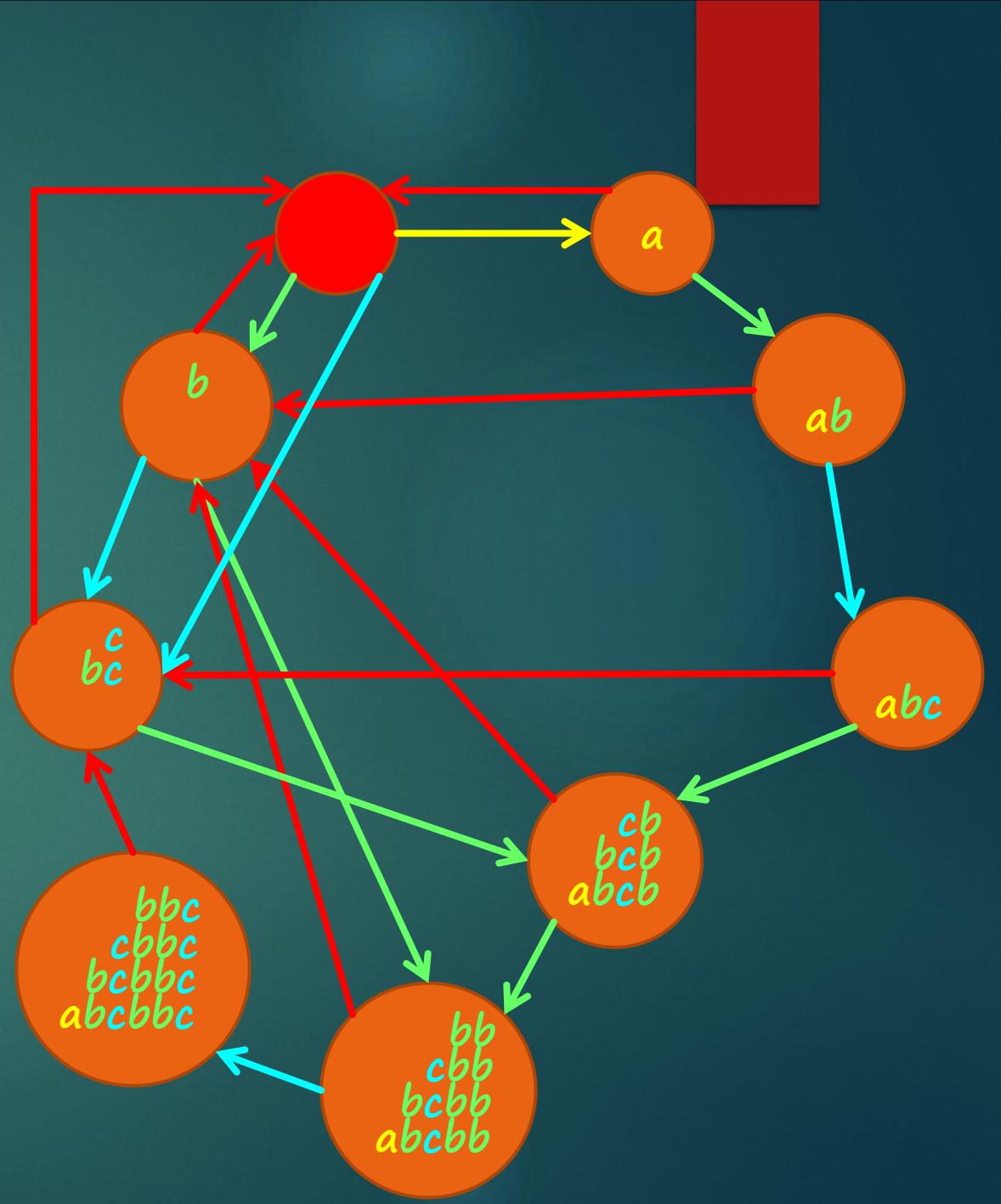
性质——树

- 根据父亲的定义，一个状态去掉首字母后所在节点要么仍是原来的节点，要么会移动到原来所在节点的父亲
- 反过来，一个状态在前面加上一个字符后，若仍是原串的子串，那么所在节点要么仍是原来的节点，要么会移动到原来所在节点的一个儿子
- 那么我们把每个节点上的所有串reverse一下，原先的在前面加字符就变成了reverse之后的在后面加字符
- 于是这些节点和父亲边就构成了一个类似trie树的结构，并且它能接受反串的所有子串
- 所以这棵树同时是反串（图中是"cbbcba"）的后缀树！



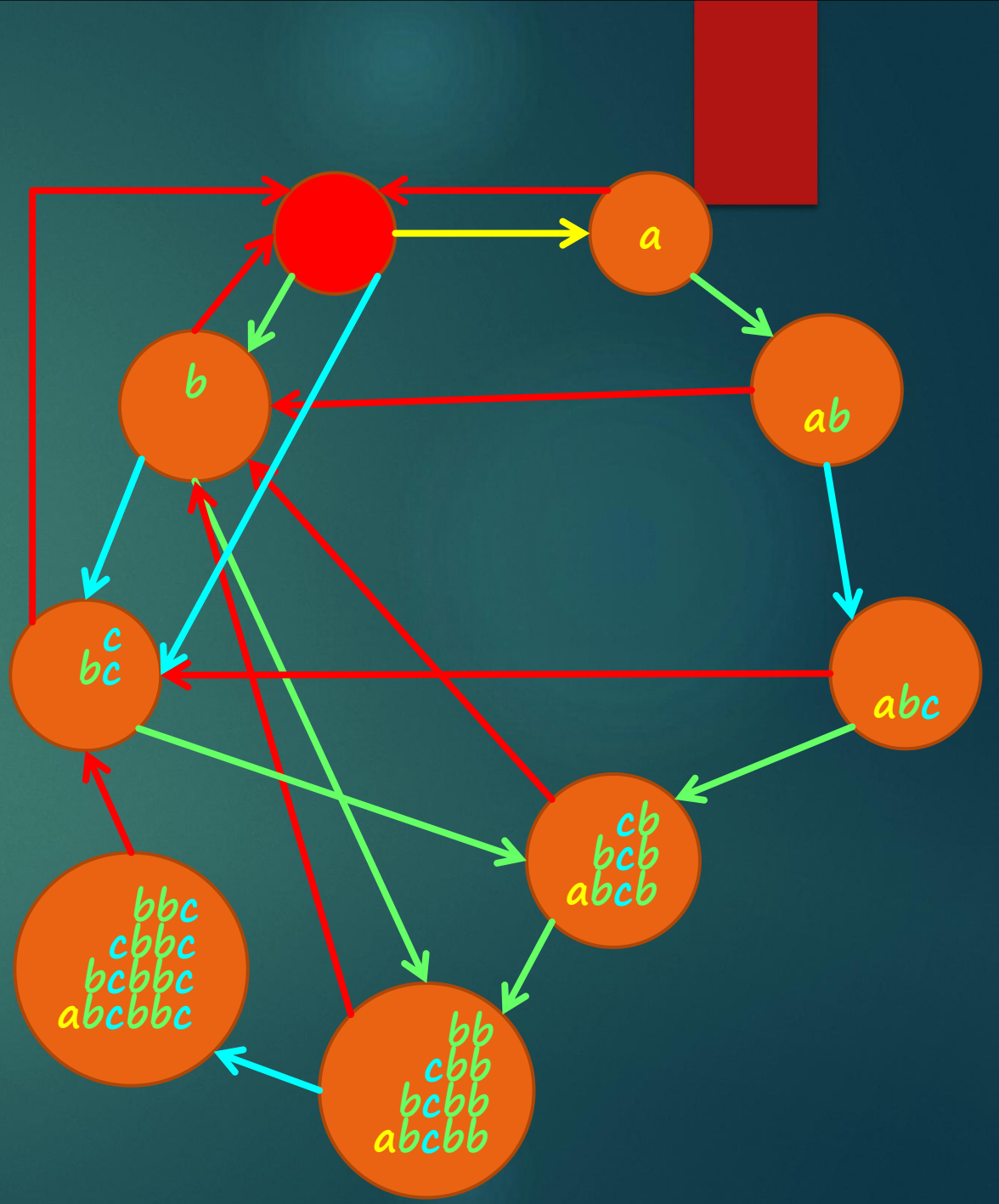
性质——位置

- 对于一个状态，我们知道它所在节点的at是它在原串中出现的一个位置（的右端点）
- 那么如果我们想要知道一个字符串在原串中出现的所有位置呢？
- 我们把新建时为np的节点（也是接受了原串的某个非空前缀的节点）叫做np类节点
- 我们把新建时为nq的节点叫做nq类节点
- 注意根既不是np类节点也不是nq类节点
- np类节点一共有n个，其at的值分别为1~n
- 其中at值为i的np类节点接受了长度为i的前缀



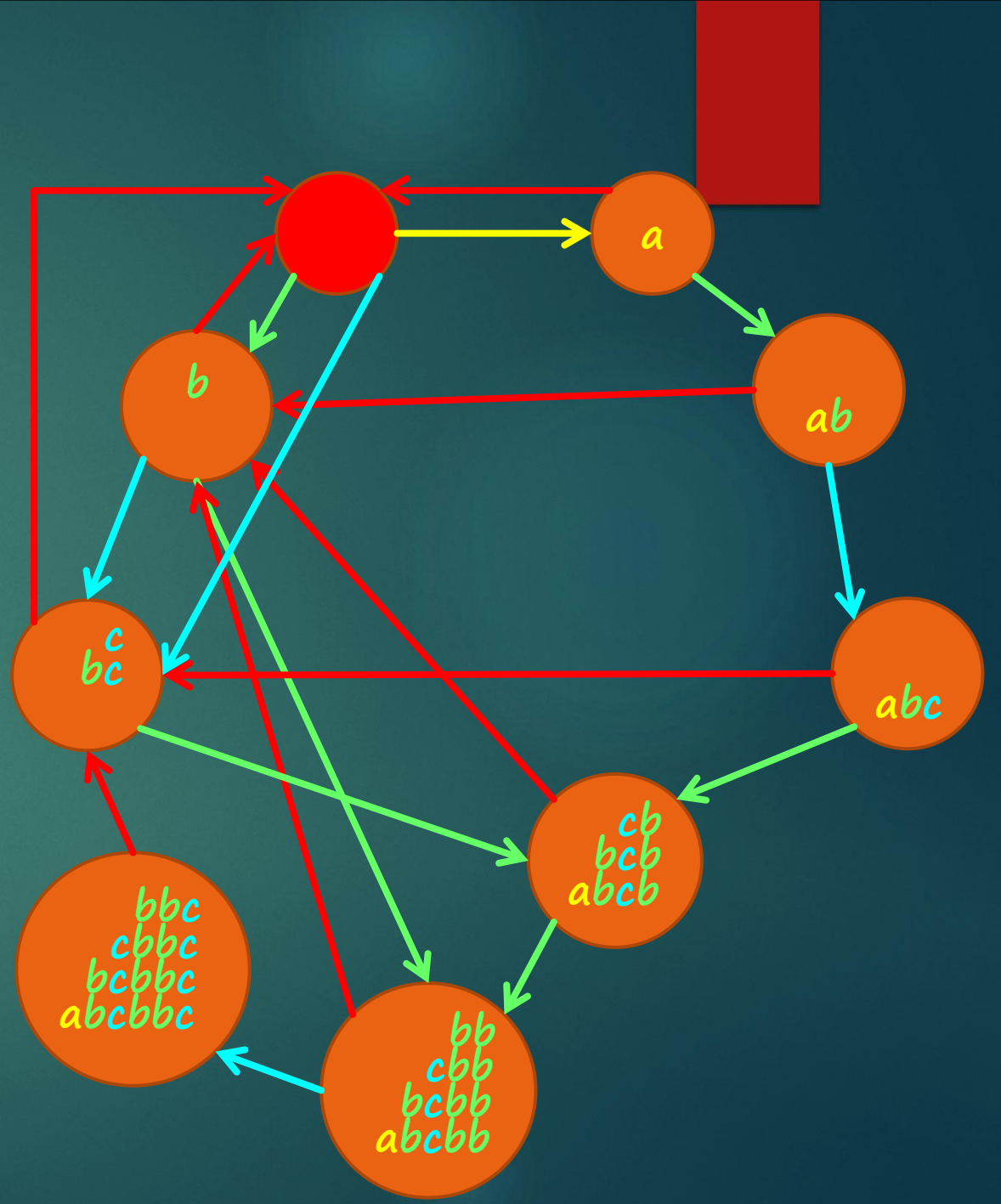
性质——位置

- 我们知道，一个状态A是另一个状态B的一个后缀，当且仅当A的所在节点是B的所在节点的一个祖先（祖先包括节点本身）
- 那么当我们想知道一个状态在原串中的所有位置的时候，我们可以先枚举右端点r，然后进行判断
- r是该状态在原串中一个出现位置的右端点当且仅当该状态所在节点是原串的长度为r的前缀所在节点的祖先
- 于是我们可以发现，一个状态在原串中出现的位置的右端点的集合等于该状态所在节点的子树中所有np类节点的at值的集合



性质——位置

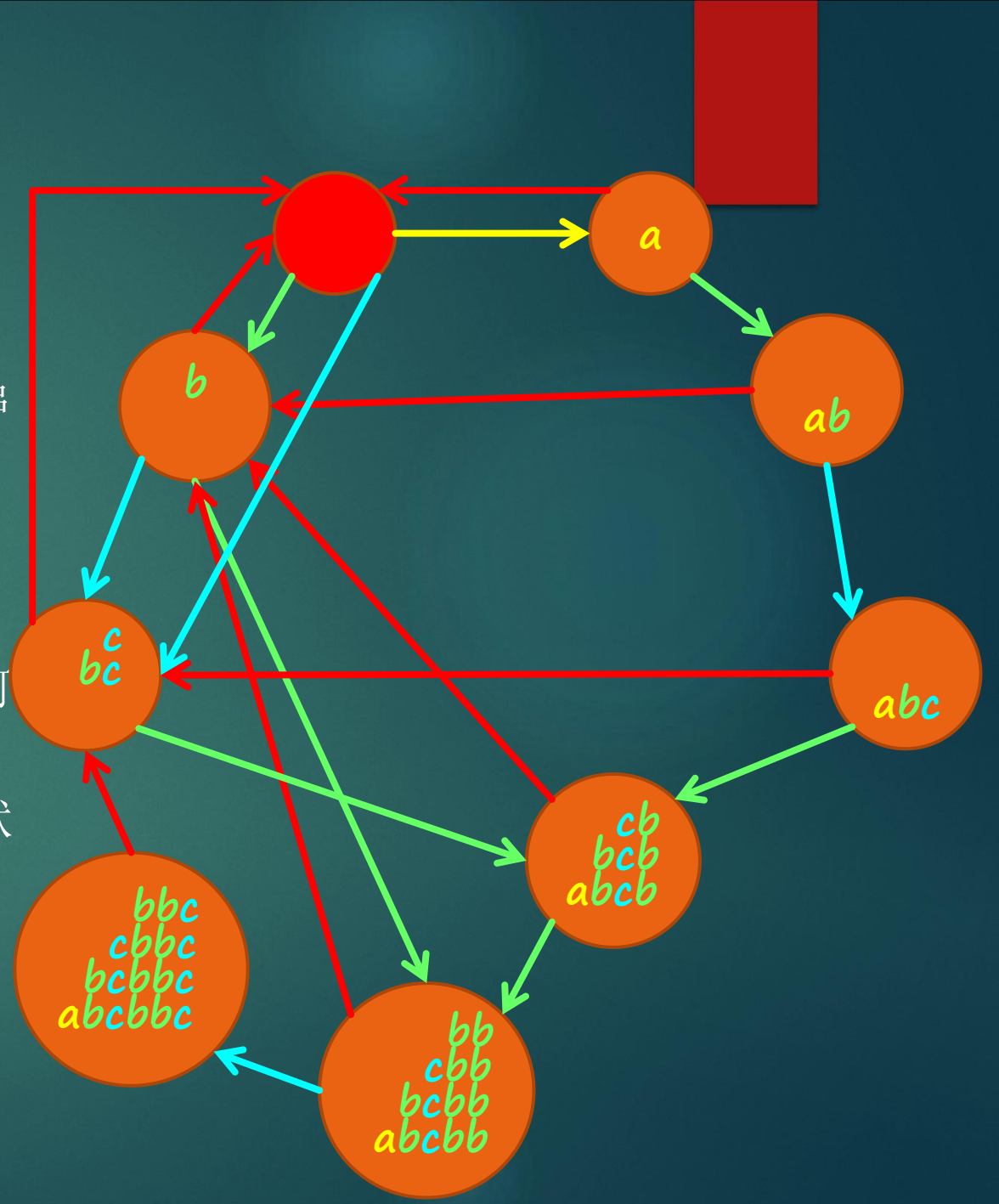
- 注意到这个集合只和状态所在节点有关，与状态的字符串长度无关
- 所以我们可以对每个节点定义它出现在原串中的位置的右端点的集合，把它叫做这个节点的*Right*集合
- 同样一个节点的*Right*集合等于该节点的子树中所有*np*类节点的*at*值的集合
- 于是我们又有一个推论，就是一个状态在原串中的出现次数，等于该状态所在节点的*Right*集合的大小
- 特别地，我们不讨论空串的出现次数
- 于是我们就有了这样一句话



- 有点可惜，原句是错的（我已经标出来了）
- 因
- 有
- 构
- 当

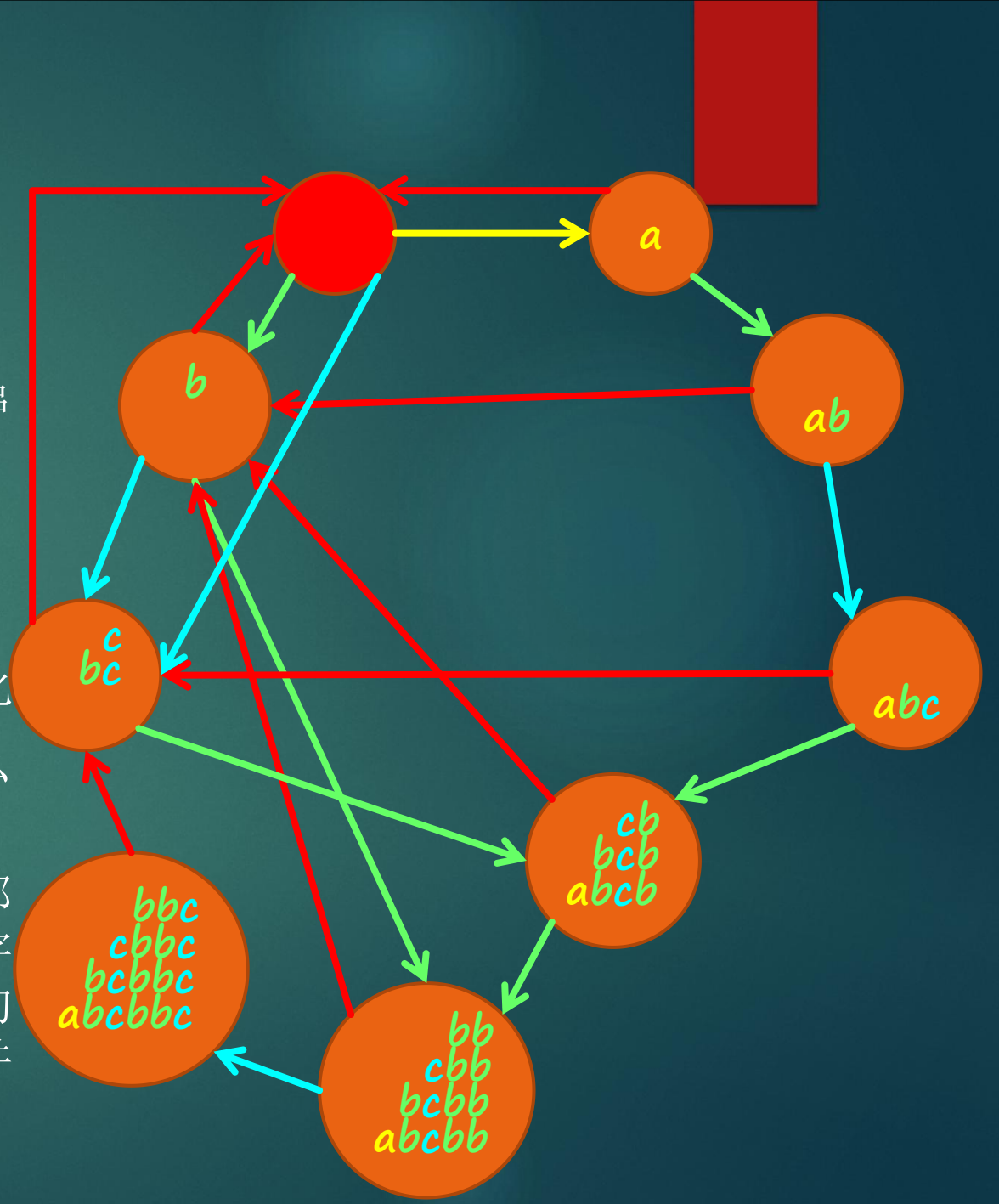
性质——匹配

- 自动机还有一个很重要的功能就是匹配字符串
- 比如用字符串T在S的KMP上跑匹配可以对T的每个前缀求出其最长的是S的前缀的后缀
- 那么S的后缀自动机呢？
- 由于KMP的一个状态是S的一个前缀，所以可以匹配到最长的是S的前缀的后缀
- 那么我们类比一下，由于后缀自动机的一个状态是S的一个子串，那我们是不是可以对T的每个前缀求出最长的是S的子串的后缀呢？
- 答案是可以的
- 我们考虑用一种类似双指针扫描法的方法



性质——匹配

- 自动机还有一个很重要的功能就是匹配字符串
- 比如用字符串T在S的KMP上跑匹配可以对T的每个前缀求出其最长的是S的前缀的后缀
- 那么S的后缀自动机呢？
- 考虑一个状态之后加一个字符会发生什么变化
- 如果该状态所在节点有这种字符的出边，那么可以直接移动该状态所在节点并且让长度加1
- 如果该状态所在节点没有这种字符的出边，那么我们考虑缩短这个状态，删去该状态的首字母。显然只有当所在节点变化时才有本质性的变化，所以我们可以跳到所在节点的父亲，并把长度修改为父亲的max



性质——匹配

- 比如对于T的第i个前缀，匹配的状态为x
- 设c为T的第$i+1$个字符
- 如果xc是S的子串，那么显然T的第$i+1$个前缀最长的（是S的子串的）后缀是xc
- 这同时意味着x所在节点有关于c的转移边
- 于是你只需要将x沿着这条边走一步，再将其长度加上1即可
- 否则说明T的第$i+1$个前缀最长的（是S的子串的）后缀比xc短
- 考虑不断删去xc的首字符，直到它是S的子串为止
- 这也相当于不断删去x的首字符，直到它有c的转移边为止，当然要注意特判删到空以后仍然没有c的转移边的情况
- 在删首字符的时候注意判断是否要跳到所在节点的父亲，当然你也可以把删去首字符改成直接跳父亲并修改当前串长度为父亲的max来节省常数

例题

- 这里所有例题的字符串的字符集为小写字母，串长总和不超过1e6
- 这里先给三个板子题
- 1.已知两个字符串S和T，求S和T的最长公共子串
- 直接建出S的后缀自动机，用T在S上跑匹配，取长度最长的一个状态即可
- 2.已知字符串S，求S本质不同的非空子串个数
- ~~你说什么是本质不同的子串？~~
- 定义字符串A和B本质不同当且仅当A与B长度不相同或存在一个i使得A的第i位与B的第i位不相同
- 做法很简单，建出S的后缀自动机后，直接把每个非根节点接受的串的个数，即每个非根节点自己的max减去父亲的max加起来就是答案

例题

- 3.已知两个字符串S,T，求S和T本质不同的公共非空子串个数
- 考虑枚举T的每个本质不同的子串询问其是否是S的子串
- 先建出T的后缀自动机，然后考虑每个节点接受的字符串中有哪些是S的子串
- 由于一个节点接受的字符串都是某个子串的后缀，所以我们可以找到这些字符串在T中的一个右端点（即节点的at），然后查询一下T的长度为at的前缀最长的（是S的子串的）后缀的长度len
- 这样我们就可以知道T的以at为右端点的子串中长度不超过len的都是S的子串，长度大于len的都不是S的子串
- 于是我们就可以把(0,len]和节点接受字符串长度的区间（即(父亲的max,自己的max]）的交中的整数个数计入答案
- 至于如何求len，只要预处理的时候建出S的自动机，用T在上面跑一遍就可以了

例题

- 如果你会做这三个问题，那么你就能在NOI2018D1T3中拿到68分
- 那剩下的呢？
- 我们来看一下第4个问题
- 4.已知字符串S和若干个字符串T1,T2,…,Tm，有m个询问，第i个询问给定li,ri，求S[li:ri]与Ti本质不同的公共非空子串个数，允许离线
- 还是按照刚才的思路，刚才的时间复杂度除了建S的后缀自动机这一步其余都是$O(|T|)$的
- 所以我们主要考虑如何快速地对Ti的每个前缀求出最长的（是S[li:ri]的子串）的后缀
- 由于S[li:ri]的子串也是S的子串，所以我们可以考虑利用S的后缀自动机来完成这件事情

例题

- 设Ti的第j个前缀的最长的（是S[li:ri]的子串）的后缀是Lj（Lj是个字符串）
- 如果我们知道了Lj以及其所在节点，我们要怎么求L(j+1)呢？
- 仍然考虑之前的双指针扫描法，从Lj开始，不断删去其首字符，直到它加上T的第j+1位是S[li:ri]的子串
- 原先我们的判断方法是判断当前状态所在的节点是否有T(j+1)的转移边
- 但是现在不能这么判了，因为转移边指向的状态（如果有的话）是S的子串但不一定是S[li:ri]的子串
- 于是我们重新想一想这个问题

例题

- 我们的问题是，如何判断一个状态x是否在S[li:ri]中出现
- 考虑x在S中出现的所有（右端点的）位置，即x所在节点的Right集合
- Right集合中什么样的位置p才会让x在S[li:ri]中出现呢？
- x在S[li:ri]中出现当且仅当x所在节点的Right集合中存在p使得 $p \le ri$ 并且 $p-len(x)+1 \ge li$（其中len(x)是x的长度）
- 那么就是判断Right集合中不超过ri的p中最大的(p-len(x)+1)是否大于等于li
- 当然，(p-len(x)+1)最大当且仅当p最大
- 这是个经典的二维偏序问题
- 一维是$p \le ri$，另一维是at值为p的节点必须要在x所在节点的子树中（这样Right集合中才会有p）

例题

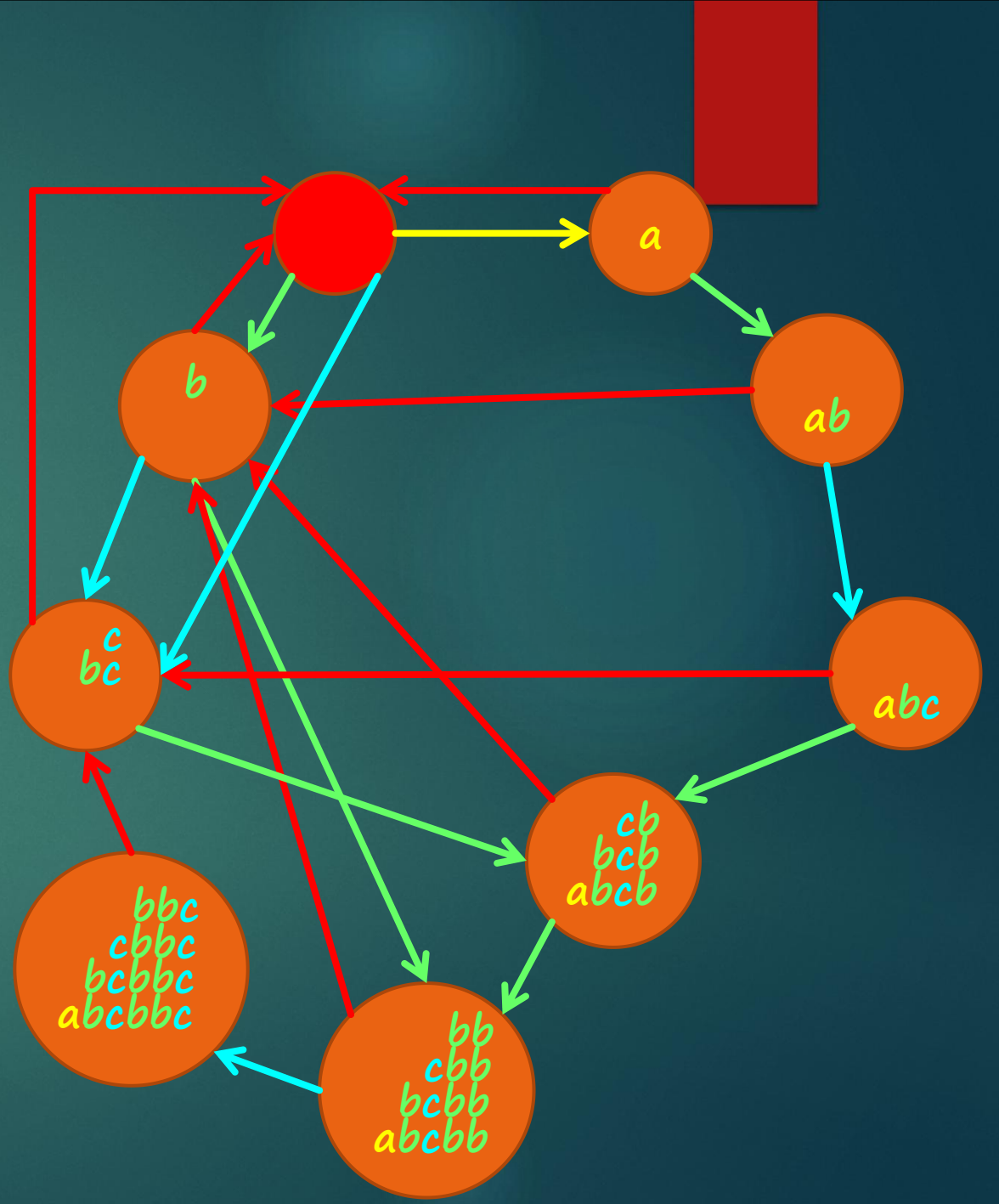
- 于是就可以在$O(|Ti|log|S|)$的时间内求出每个Lj（的长度）
- 于是就可以在$O(|S|+\Sigma|Ti|log|S|)$的时间内解决所有询问
- 于是你就可以A掉「NOI2018」你的名字 了

例题

- 我们还有第5个问题
- 这个问题来自2018年省队集训中InvUsr的题
- 5.已知字符串S，你需要进行|S|次操作，每次操作可以删去字符串的首字母或尾字母（不能都删），最后会删到空。你要最大化每次操作前的字符串在原串S中作为子串的出现次数之和（比如aba的最优操作之一是aba->ba->a->空串，其中aba出现1次，ba出现1次，a出现2次，和是4次）
- 首先，一个子串在S中的出现次数可以用后缀自动机的Right集合大小得到
- 考虑任意的操作过程中出现的字符串都是S的子串，所以我们可以使用后缀自动机中的状态，对一个状态x定义f(x)为从字符串x删到空累计和的最大值
- 那么答案就是f(S)

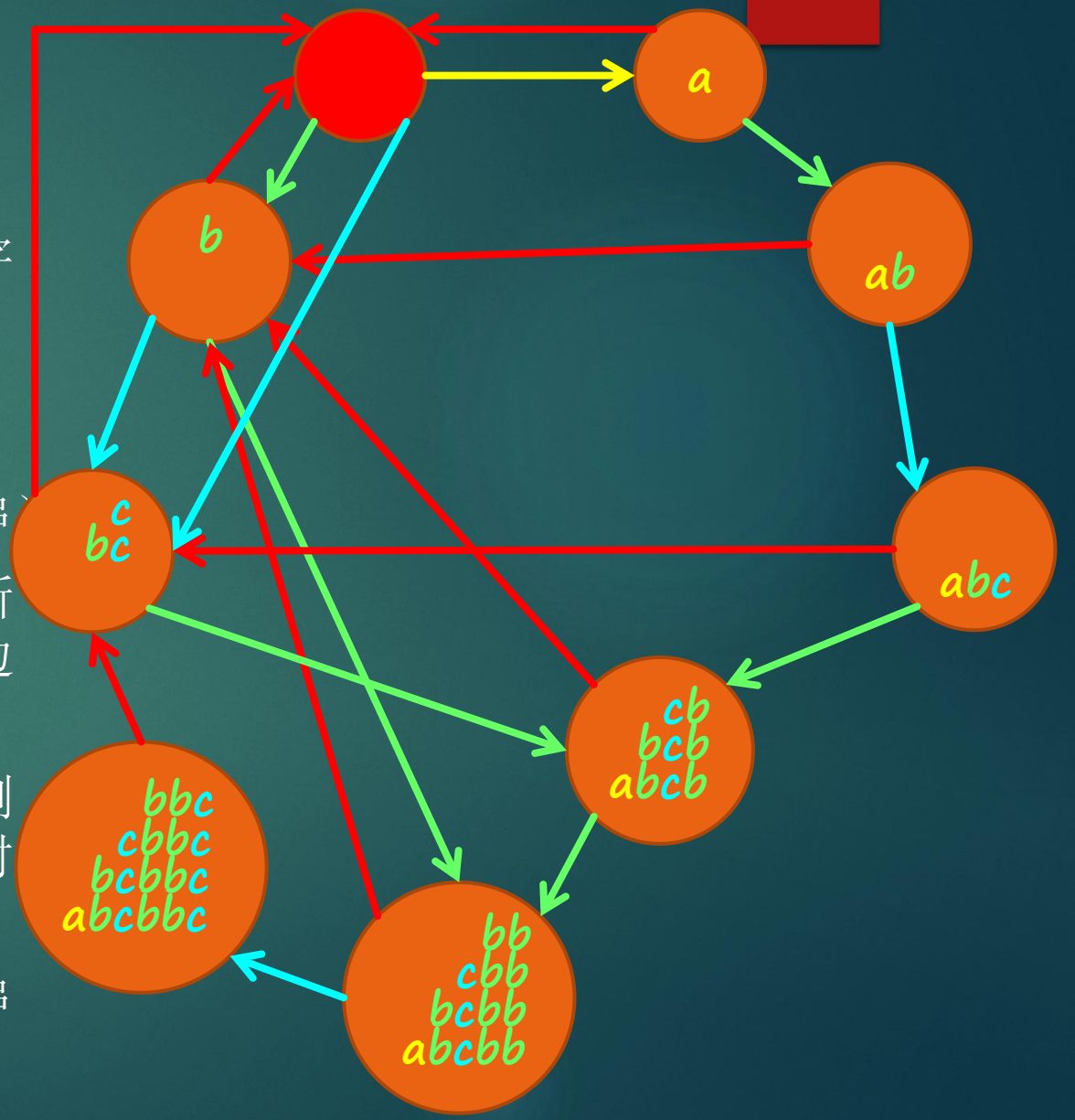
例题

- 显然f(空串)=0
- 否则考虑枚举删去x的哪个字母
- 如果是首字母，x的所在节点可能原地不动或者跳到它的父亲
- 如果是尾字母，x会转移到某一个有指向它的转移边的节点
- 因为相同节点的状态有着相同的出现次数，所以我们可以考虑按节点转移
- 我们考虑离开S的所在节点时走的是哪条边
- 如果是父亲边，那就说明是连续删去S的首字母直到状态变为父亲接受的最长的字符串



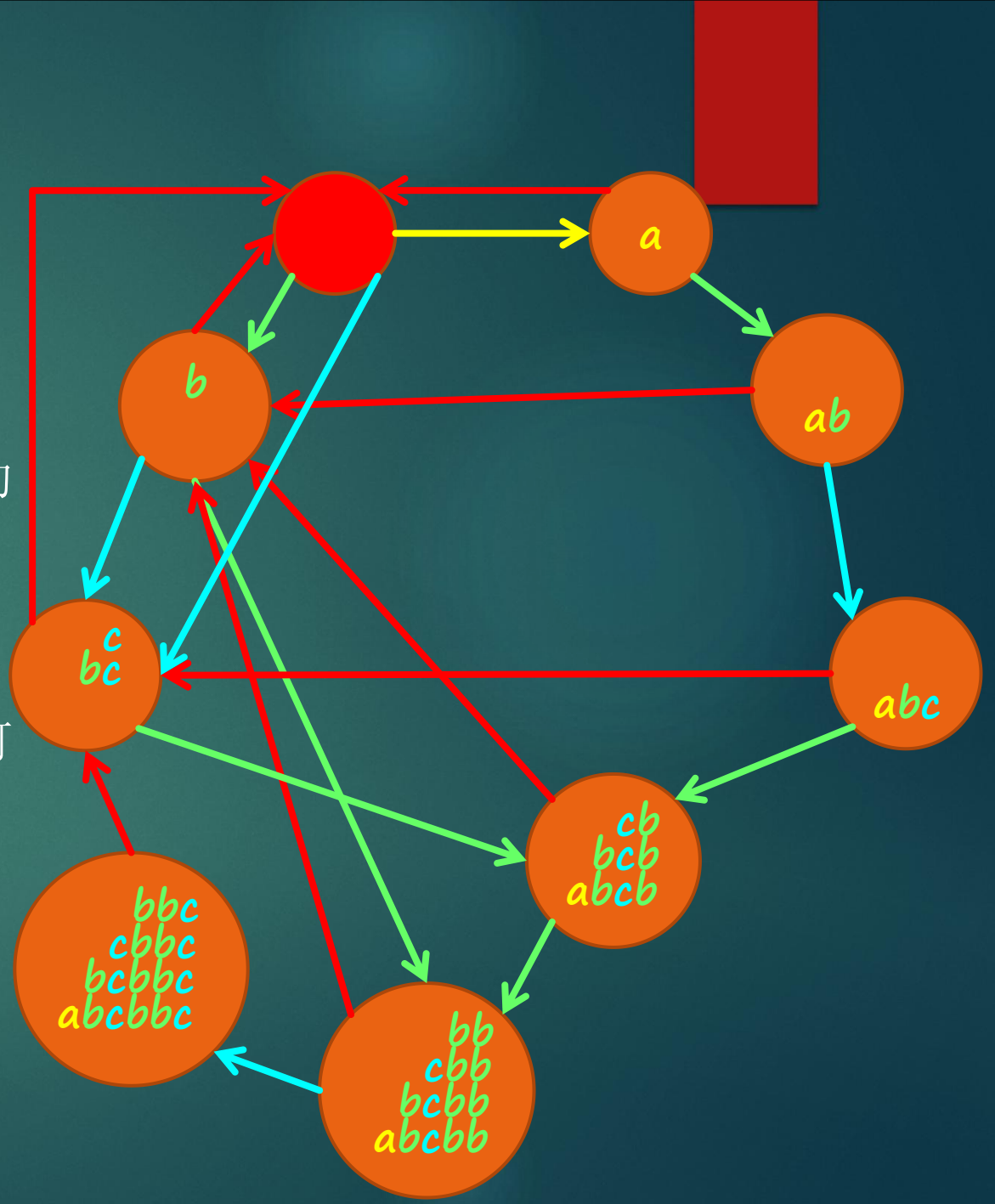
例题

- 如果是转移边，那么可以选择删去若干个首字母直到长度减1在目标节点接受的范围内时再删尾字母
- 注意到操作过程中不可能会使字符串在S中的出现次数变小（因为每次都是变成自己的子串）
- 又因为状态所在节点不变出现次数也不变，所以一定存在一种最优情况使得在走这条转移边之前删去尽量少的首字母
- 若不是，我们可以把多删的那些首字母留到删去尾字母之后再删，这样不会让删这些字母对答案的贡献变小
- 于是这样就会转移到目标节点中最长的字符串



例题

- 又因为S也是某个节点接受的最长的字符串
- 所以我们发现只需要对每个节点接受的最长的字符串计算$f(x)$就可以了
- 由于每个x只会贡献到严格更长的状态
- 所以转移是无环的，所以你只需要按照max从小到大的顺序枚举节点，再枚举边转移就可以了，贡献是很好统计的
- 时间复杂度是$O(n\times$字符集大小$)$



例题

- 终于来到了树形数据结构部分
- 1.有一排n个甘蔗，它们有不同的生长速度（每天长高多少）和限制高度（一旦达到限制高度就不再生长），初始高度均为0
- 有m个询问，每次给出t,l,r，表示从初始或上一次询问过了t天之后，砍下在l到r区间的甘蔗（再补种上，初始高度为0），询问收获的甘蔗的总长
- 前面的询问会对后面的询问有影响
- 强制在线
- n,m≤10^5，其它不超过10^9
- Source:非原创，但我也不知道其真正的Source

例题

- 我们先考虑每次割所有甘蔗的情况
- 一段时间后，有些甘蔗还在自由地生长，而有一些受到了限制
- 考虑把甘蔗按照达到限制的时间排序
- 那么小于等于t的部分就是达到限制的，其余部分就是未达到限制的
- 达到限制的部分对答案的贡献是限制高度之和
- 未达到限制的部分对答案的贡献是生长速度之和乘以t
- 这两部分都可以用前缀和维护
- 那么如果每次割的不一定是所有甘蔗呢？

例题

- 考虑到上一次收割时间相同的甘蔗的生长状况是类似的
- 那么我们可以把甘蔗划分成若干个区间，使其每个区间中上次收割的时间是相同的
- 收割的时候把l到r内的所有区间（或是区间的一部分）计算答案后加上l,r这一区间
- 开始时只有一个区间
- 每次收割后最多增加两个区间，并且可能会删去一些区间
- 由于增加的区间个数是$O(m)$的，所以删去的区间总数也是$O(m)$的
- 我们可以用map维护这些区间
- 然后计算答案变成了询问区间中达到限制时间大于/小于等于t的生长速度/限制高度和
- 这是个二维偏序，因为强制在线，所以考虑用可持久化线段树预处理维护区间的和
- 总时间复杂度为$O((n+m)\log n)$

例题

- 2.有n个栈和m个操作，每个操作形如：
- - 把同一个元素x插入到第l到r个栈中
- - 弹出某一个栈的栈顶
- - 询问第l到第r个栈中栈顶元素的和
- 强制在线
- 保证所有操作都合法
- n,m≤5e5，其它数字不超过1e9
- 时限4s，空间限制1G
- Source: UOJ#218 火车管理

例题

- 如果没有弹栈的操作，那么就是很简单的区间赋值区间求和的线段树问题
- 如果我们能知道每次弹栈后该栈的栈顶元素，那么我们一样能用一棵区间赋值区间求和的线段树来解决求和的询问
- 考虑维护每个栈栈顶元素的插入时刻，那么弹栈之后栈顶元素就是被弹元素插入时刻的前一时刻时栈顶的元素
- 于是我们可以用可持久化线段树来维护每个时刻每个栈栈顶元素的插入时刻
- 入栈操作是区间赋值，弹栈操作是单点修改
- 所以我们要实现区间赋值的功能
- 如果直接打标记不仅常数大而且有*MLE*的风险
- 有一种方法是把原来需要打标记的节点直接换成一个左右儿子都指向自己的节点

例题

- 3.有一棵所有节点要么是叶子要么有两个儿子的二叉树
- 每个叶子都有一个权值
- 每个非叶子节点以p的概率等于权值较大的儿子的权值，以$1-p$的概率等于权值较小的儿子的权值，不同节点的p可能不同
- 求根节点权值的分布列（即取到每一种权值的概率）
- 对大质数取模
- 节点数不超过$3e5$
- Source：非原创

例题

- 暴力的做法是直接DP计算每个节点的分布列
- 时间复杂度O(n^2)
- 由于一个节点权值可能的取值只可能是子树中某个叶子的权值
- 所以可以考虑与合并有关的算法
- 那么合并两个子节点的分布列会发生什么呢？
- 一个权值的概率会*=另一个节点权值比自己小的概率*p+(1-另一个节点权值比自己小的概率)*(1-p)
- 所以可以用线段树维护分布列的区间和
- 于是我们可以在线段树合并的过程中在外面分别维护原来两棵线段树当前的前缀和，然后打上区间乘的标记即可

完结撒花

GL&HF