AC 自动机

MengChunlei

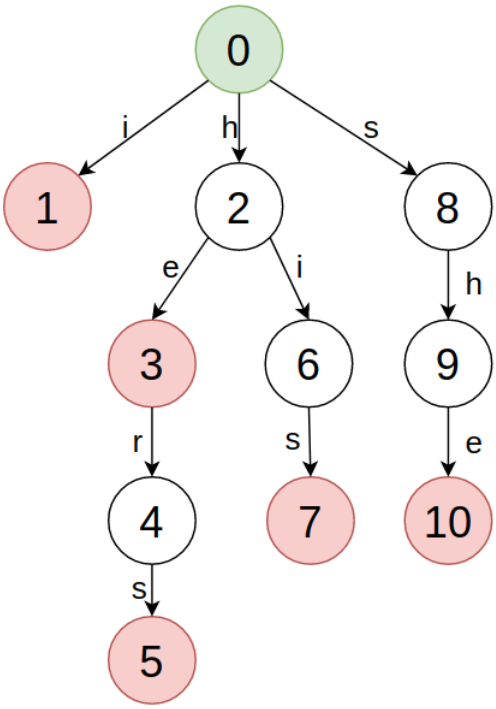
January 10, 2021

1 题目描述

给定一些串的集合 $S = \{s_1, s_2, ..., s_m\}$, 以及一个串 T. 问 T 包含了给定集合中的哪些串. 这是 AC 自动机要解决的问题.

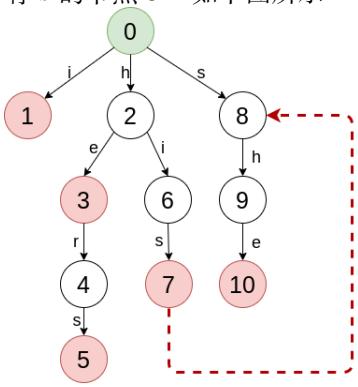
2 结构描述

算法的思路是首先为集合 S 建立 AC 自动机，然后在 AC 自动机上查询 T.
AC 自动机首先是一棵树. 假设集合 $S = \{i, he, his, she, hers\}$，那么对应的树为:

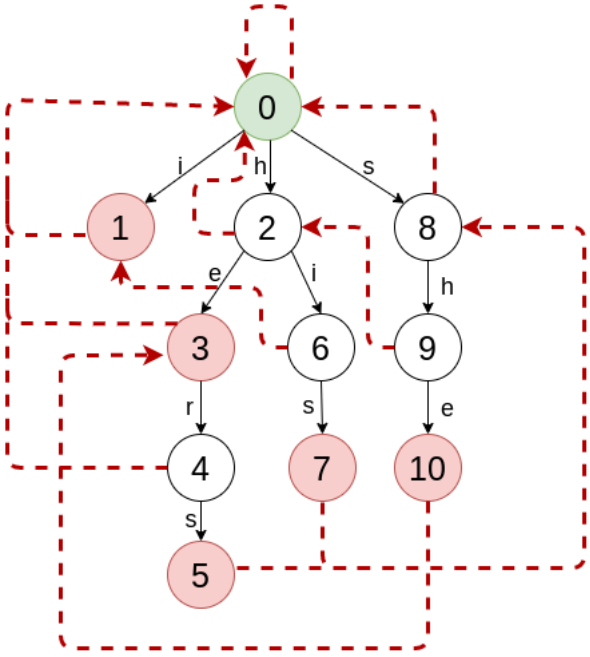


其中，绿色的 0 号节点为根节点，表示空串. 其他每个节点代表的串为从根节点到当前节点经过的边上的字符组成的串，比如节点 3 代表 he, 节点 6 代表 hi. 红色的节点代表的串为集合 S 中的串. 所以每个节点其实保存了每个字符对应的孩子节点，比如 $E(2, i) = 6$, 表示在节点 2 表示的串后加上字符'i' 所对应的新串的节点为 6 .

另外，每个节点还有一种边，叫做 $fail$ 边，$fail$ 边到达的节点所表示的串是当前节点表示的串的最长后缀. 在上面的例子中， 比如节点 7 代表的串为 his, 可以发现，它的 $fail$ 边会指向节点 8, 因为不存在 is 的节点，只有 s 的节点 8 . 如下图所示:



进而我们可以把所有的 $fail$ 边都加上：



现在来总结下这个节点的数据结构：

Listing 1: Node defination

```
struct Node {
  std::map<char, Node*> sons;
  Node* fail;
  int target_set_index; // The index word in S, -1 if not

  bool HasSon(char ch) const { return sons.count(ch) > 0; }
  const Node *GetSon(char ch) const { return sons.at(ch); }
  Node *GetSon(char ch) { return sons.at(ch); }
};
```

在这个节点的基础上，假设现在已经建立了给定 S 的 A C 自动机，那么查询 T 包含了 S 中的哪些的逻辑如下：

Listing 2: Query

```
1  std::set<int> Query(const Node *root, const std::string &T) {
2    std::set<int> result;
3    const Node *curr = root;
4    for (char ch : T) {
5      const Node *son = root;
6      do {
7        if (curr->HasSon(ch)) {
8          son = curr->GetSon(ch);
9          break;
10       }
11       curr = curr->fail;
12     } while (curr != root || curr->HasSon(ch));
13     for (const Node* p = son; p != root; p = p->fail) {
14       if (p->target_set_index >= 0) {
15         result.insert(p->target_set_index);
16       }
17     }
18     curr = son;
19   }
20   return result;
21 }
```

下面来解释一下这个函数．第 2 行的 $result$ 用来存储结果，即包含的目标串的编号．第 3 行设定当前节点为根节点，当前节点的含义是它表示的串是当前 T 的最长后缀．接下来这个主循环分两大部分，第一部分从第 5 行

到第 12 行，是为了找到一个节点使得这个节点代表的串是 T 的当前前缀 ($T[0,i]$) 的最长后缀. 由于 $curr$ 节点的串是 $T[0,i-1]$ 的最长后缀，所以第 7 行，如果 $curr$ 有一个孩子是 ch, 那么这个就节点就是 $T[0,i]$ 的最长后缀，第 8-9 行设置并退出循环. 否则第 11 行，$curr$ 跳转到它的 $fail$ 节点，因为它的 $fail$ 节点的串是 $curr$ 的最长后缀，如果这个节点有个 ch 的孩子，那么那个孩子节点就是目标，这样就进入了下一个循环. 最后第 12 行结束的另一个条件，到达了根节点并且根节点没有为 ch 的孩子，那么说明没有非空节点的串是 T 的后缀，这时当前节点就是根节点. 然后第二部分第 13-18 更新答案并重新设置 $curr$. 第 13 行到第 17 行的这个循环有个优化, 可以在节点上设定一个 $flag$, 如果遍历过, 就不再遍历, 在第 13 行的循环中直接跳出, 这样每个节点至多被访问一次.

下面给一个例子来模拟节点的跳转，假定 $T = ushersheishis$

T	$-$	u	s	h	e	r	s	h	e	i	s	h	i
$middle$	$-$	$-$	$-$	$-$	$-$	3	$-$	8	$-$	3,0	0	$-$	2,0
$node$	0	0	8	9	10	4	5	9	10	1	8	9	1

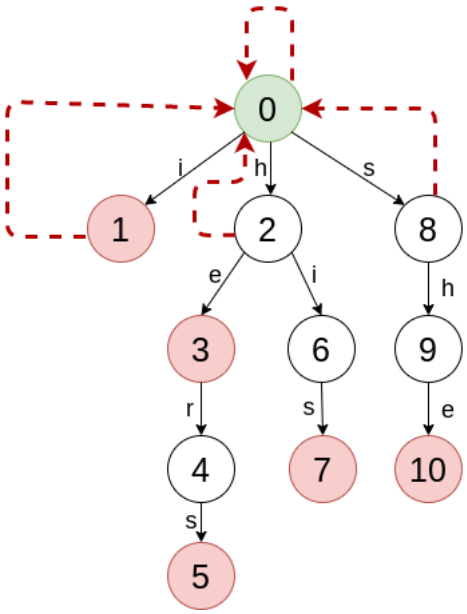
3 构造过程

分两部分，第一部分是插入字符串并建立孩子边，第二部分是建立 $fail$ 边. 第一部分比较简单，代码如下:

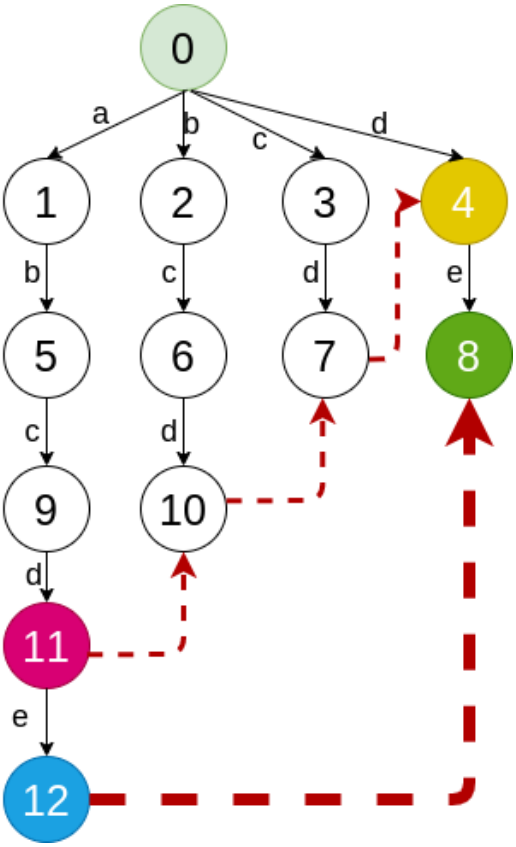
Listing 3: Insert

```
1  void Insert(const std::string &s, int id, Node* root) {
2    Node* curr = root;
3    for (char ch : s) {
4      if (!curr->HasSon(ch)) {
5        Node* p = new Node;
6        p->target_set_index = -1;
7        curr->sons[ch] = p;
8      }
9      curr = curr->GetSon(ch);
10   }
11   curr->target_set_index = id;
12 }
```

第二部分是 $fail$ 边的建立. 这个可以按照 bfs 的过程从深度由小到大进行建立. 首先，是根节点和深度为 1 的节点，因为这些节点的长度只有 1 ，所以没有更短的串是它们的后缀了，只有空串，所以它们的 $fail$ 都指向根节点.



从深度为 2 开始，来分析一个节点 $node$ 的 $fail$ 应该是谁. 假设 $node$ 的父节点是 $father$，并且 $father$ 到 $node$ 的边为字符 ch. 因为建立 $fail$ 是按照深度递增进行的，所以 $father$ 节点的 $fail$ 指针已经建立好了. 如果 $father$ 的 $fail$ 节点 q 有一个 ch 的孩子 p, 那么 p 就是 $node$ 的 $fail$ 节点. 因为 q 是 $father$ 的最长后缀. 否则只需要沿着 q 的 $fail$ 边一直向上找，直到找到一个节点 r, 使得 r 有一个 ch 的孩子 t, 那么 t 就是 $node$ 的 $fail$ 节点. 具体的示意图如下所示:



其中 *node* 节点是 12,*father* 节点是 11, 最后沿着 *fail* 边，$11 \to 10 \to 7 \to 4$, 最后找到节点 4 的时候，它有个 e 的孩子. 所以 12 的 *fail* 节点为 8.
下面是代码的实现：

Listing 4: Build

```
1  void Build(Node *root) {
2    std::queue<Node *> nodes;
3    for (auto &e : root->sons) {
4      e.second->fail = root;
5      nodes.push(e.second);
6    }
7    while (!nodes.empty()) {
8      Node *father = nodes.front();
9      nodes.pop();
10     for (auto &e : father->sons) {
11       char ch = e.first;
12       Node *node = e.second;
13       Node *fail = father->fail;
14       while (fail != root && !fail->HasSon(ch)) {
15         fail = fail->fail;
16       }
17       if (fail->HasSon(ch)) {
18         node->fail = fail->GetSon(ch);
19       } else {
20         node->fail = root;
21       }
22       nodes.push(node);
23     }
24   }
25 }
```

在上面的代码中的第 14 行寻找合适的 *fail* 节点的时候，这个循环可能会执行很多次，从而导致时间复杂度变大. 有一个解决的办法采用了空间换时间的概念，代码如下面所示. 它的改进在于，如果在插入完集合 S 中的字符串后，如果一个节点 p 没有孩子为 ch 的节点，那么将强行增加一条这样的边，它到达的节点所代表的串为节点 p 代表的串加上字符 ch 所得到的新串的最长后缀. 如下面的代码的第 32 行所示

Listing 5: FastBuild

```
struct Node {
  std::vector<Node *> sons;
  Node *fail;
  int target_set_index;

  Node() : sons(26, nullptr), fail(nullptr), target_set_index(01) {}

  bool HasSon(char ch) const { return sons[ch - 'a'] != nullptr; }
  Node *GetSon(char ch) { return sons.at(ch - 'a'); }
};

void FastBuild(Node *root) {
  std::queue<Node *> nodes;
  for (char ch = 'a'; ch <= 'z'; ++ch) {
    if (!root->HasSon(ch)) {
      root->sons[ch - 'a'] = root;
    } else {
      Node *son = root->GetSon(ch);
      son->fail = root;
      nodes.push(son);
    }
  }
  while (!nodes.empty()) {
    Node *father = nodes.front();
    nodes.pop();
    for (char ch = 'a'; ch <= 'z'; ++ch) {
      if (father->HasSon(ch)) {
        Node *son = father->GetSon(ch);
        son->fail = father->fail->GetSon(ch);
        nodes.push(son);
      } else {
        father->sons[ch - 'a'] = father->fail->GetSon(ch);
      }
    }
  }
}
```