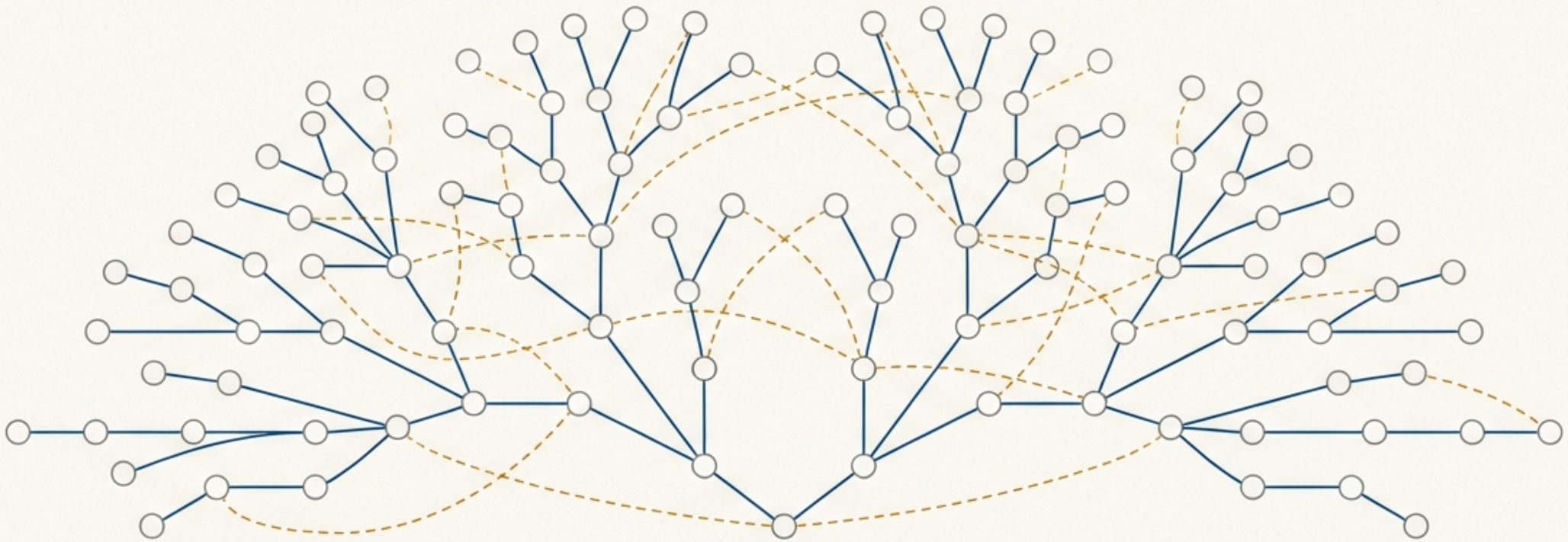
Aho-Corasick: High-Speed Multi-Pattern Matching

阿霍-科拉西克算法：高速多模式匹配

A finite automaton-based algorithm for finding all occurrences of a finite set of keywords in a text.

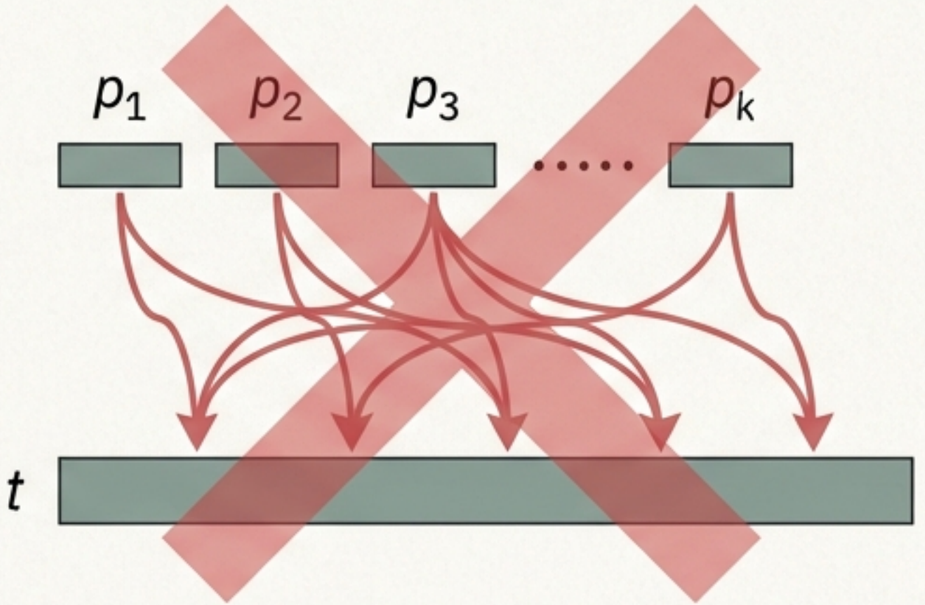
一种基于有限自动机的算法，用于在文本中查找一组有限关鍵词的所有出现。



The Challenge: Searching for Many Patterns

挑战：搜索大量模式串

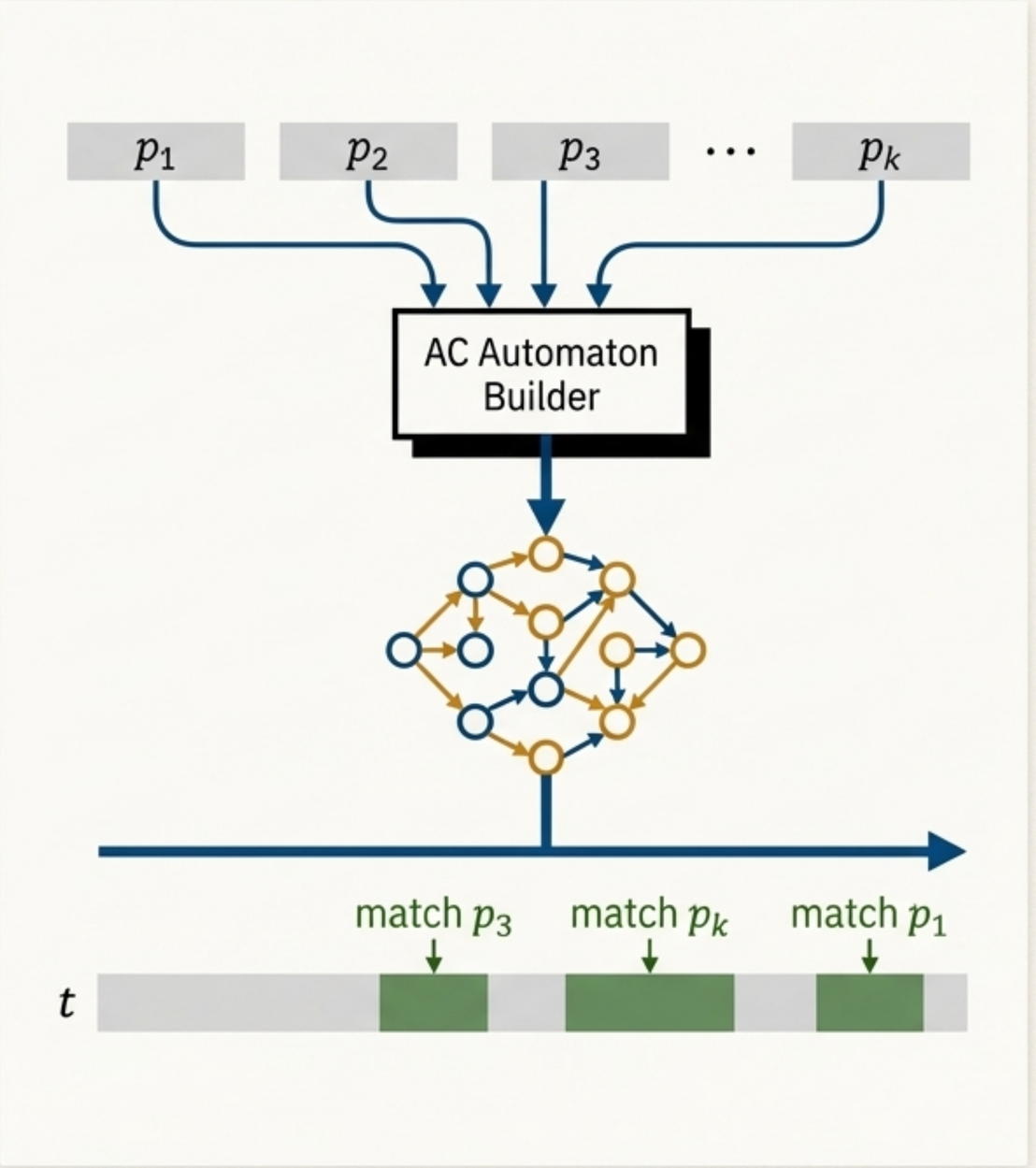
- **The Goal**: Given a set of k patterns, $P = \{p_1, p_2, ..., p_k\}$, and a text t, find all occurrences of every pattern p_i within t.
  目标：给定一个包含 k 个模式串的集合 $P = \{p_1, p_2, ..., p_k\}$ 和一个文本 t，在 t 中找出每个模式串 p_i 的所有出现位置。
- **The Obvious Approach**: Execute a single-pattern search algorithm (like KMP) k times, once for each pattern.
  显而易见的方法：对每个模式串执行一次单模式搜索算法（例如 KMP），共执行 k 次。
- **The Inefficiency**: The total time complexity becomes $O(kn + m)$, where m is the total length of all patterns. When k is large, this is prohibitively slow.
  效率瓶颈：总时间复杂度为 $O(kn + m)$，其中 m 是所有模式串的总长度。当 k 很大时，这个方法会非常慢。



The Aho-Corasick Solution: A Unified Automaton

Aho-Corasick 解决方案：一个统一的自动机

- The Aho-Corasick algorithm pre-processes all k patterns into a single, powerful finite automaton (a state machine). Aho-Corasick 算法将所有 k 个模式串预处理成一个强大的有限自动机（状态机）。
- This automaton scans the text t in a single pass, identifying all matches simultaneously.
该自动机通过对文本 t 进行单次遍历，同时识别所有匹配项。
- The Result: A dramatic performance improvement. The time complexity is $O(n + m + z)$, where z is the total number of matches found. The runtime is independent of k, the number of patterns.
结果：性能得到极大提升。时间复杂度为 $O(n + m + z)$，其中 z 是找到的匹配总数。运行时间与模式串的数量 k 无关。



Anatomy of the Aho-Corasick Automaton
Aho-Corasick 自动机剖析

1. The Goto Function (g): The Trie Backbone
1. 转移函数 (g)：Trie 骨架

Defines the primary forward transitions based on the next character in the text. This function forms a keyword tree (Trie) from the set of all patterns `P`.

定义了基于文本中下一个字符的基本前进状态转换。该函数由所有模式串集合 P 构成一个关键词树（Trie）。

2. The Failure Function (f): The Smart Fallback
2. 失败函数 (f)：智能回退

Engaged when a character does not match a `goto` transition. It points to an alternative state, preserving the longest possible partial match without rescanning the text. It is the core of the algorithm's efficiency, analogous to KMP's `next` array.

当一个字符与 `goto` 转换不匹配时启用。它指向一个备用状态，保留了最长的可能部分匹配，而无需重新扫描文本。这是该算法效率的核心，类似于 KMP 算法的 `next` 数组。

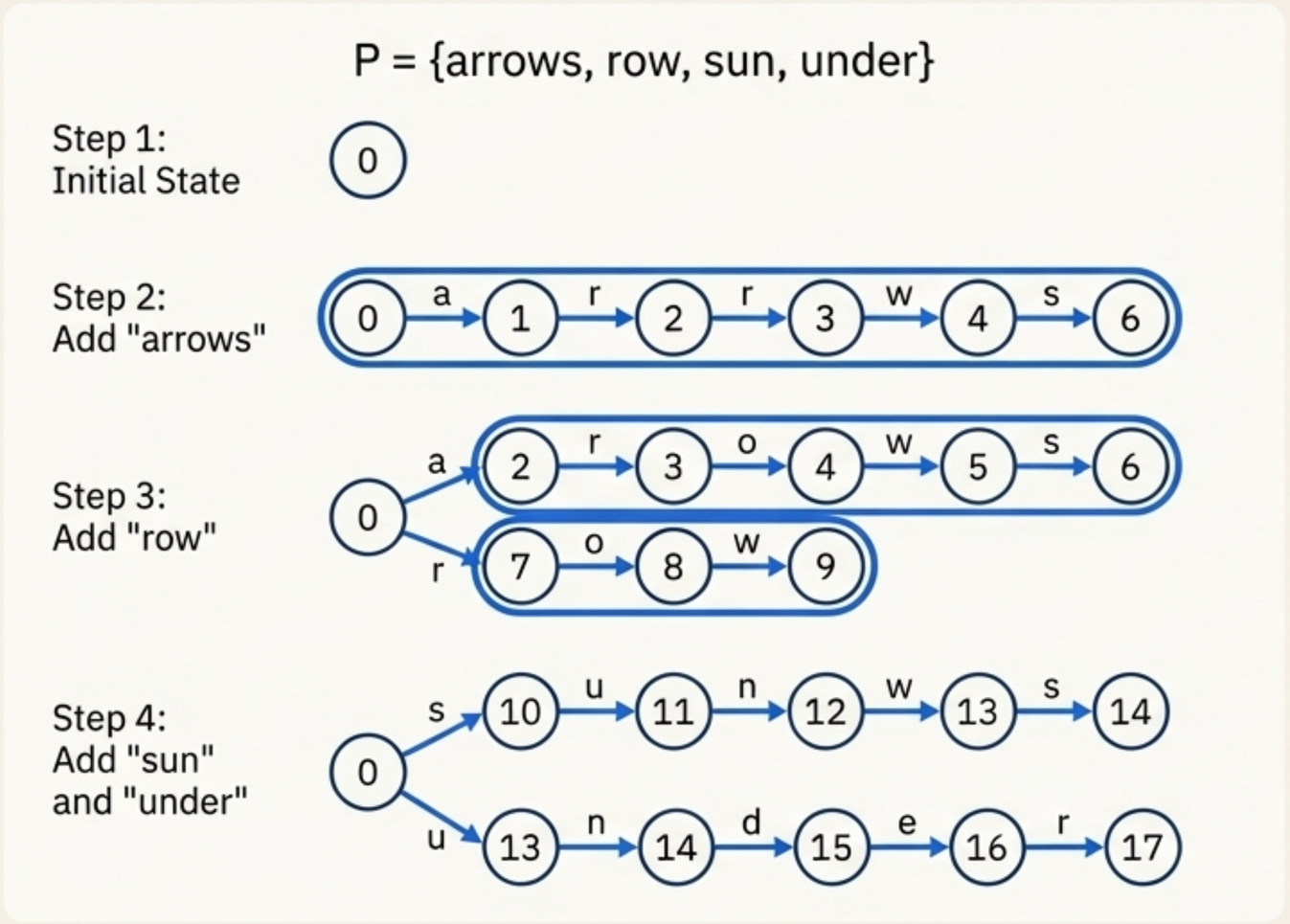
3. The Output Function (output): The Match Reporter
3. 输出函数 (output)：匹配报告器

Stores the set of patterns that end at a given state. When the automaton enters a state `s`, we consult `output(s)` to report any completed matches.

存储在给定状态结束的模式串集合。当自动机进入状态 s 时，我们查询 `output(s)` 来报告任何已完成的匹配。

Step 1: Building the Trie with the `goto` Function
第一步：使用 `goto` 函数构建 Trie

The `goto` function is built by inserting each pattern from the set `P` into a keyword tree (Trie). Shared prefixes are merged into common paths.
`goto` 函数通过将集合 P 中的每个模式串插入到一个关键词树（Trie）中来构建。共享的前缀被合并成公共路径。

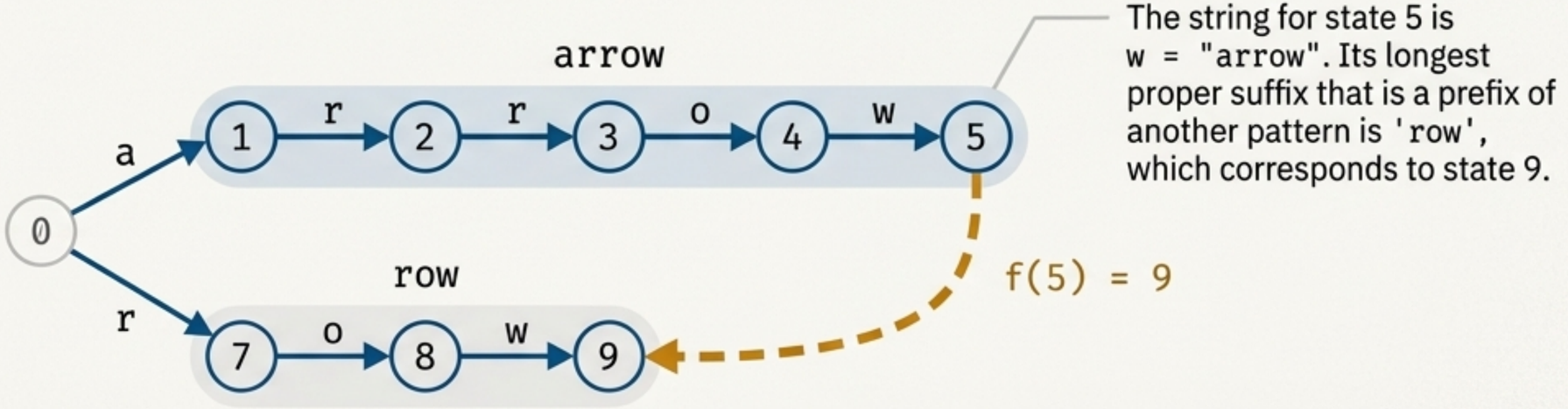



```cpp
// The fundamental node structure
// 基础节点结构
typedef struct node {
    int cnt;
    node* fail;
    node* go[dsize];
    vector<string> output;
} tnode;

// Function to insert a pattern into the trie
// 将一个模式串插入 Trie 的函数
void insert(const string& word) {
    tnode* cur = root;
    for(int i=0; i < word.length(); ++i) {
        if (!cur->go[idx[word[i]]])
            cur->go[idx[word[i]]] = newnode();
        cur = cur->go[idx[word[i]]];
    }
    cur->output.push_back(word);
    cur->cnt += 1;
}
```

Step 2: The Core Logic - The Failure Function
第二步：核心逻辑 - 失败函数

- **Formal Definition**: For any state s reached by a string w, the failure link f(s) points to the state corresponding to the *longest proper suffix* of w that is also a *prefix* of some pattern in P.
  形式化定义：对于由字符串 w 到达的任意状态 s，其失败链接 f(s) 指向的状态，对应于 w 的“最长真后缀”，且该后缀同时也是 P 中某个模式串的“前缀”。
- **In Plain English**: 'If I am at state s and the next character in the text does not match, where do I go? The failure link takes me to the most advanced state I could have been in, given the characters I've seen so far.'
  通俗解释："如果我当前在状态 s，而文本中的下一个字符不匹配，我该去哪里？失败链接会将我带到基于已见字符所能到达的最优备选状态。"



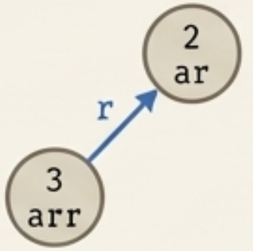
Constructing Failure Links with Breadth-First Search
使用广度优先搜索构建失败链接

We compute failure links level by level, starting from the states directly reachable from the root.
使用广度优先搜索构建失败链接，每你部位路在接趣曲找到方位。

The Process for a state `s = g(r, c)` (state `s` is reached from state `r` via character `c`):

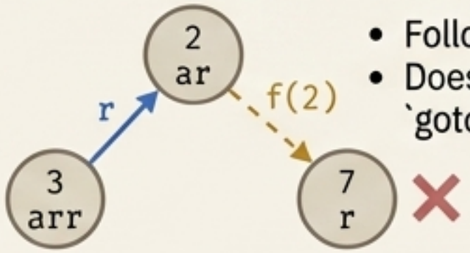
1. To find `f(s)`, we start at `s`'s parent, `r`.
2. We traverse the failure links of the parent: `p = f(r)`, `p = f(f(r))`, and so on, until we find a state `p` that has a valid `goto` transition for the character `c`.
3. The failure link `f(s)` is then set to `g(p, c)`.
4. If we reach the root and still find no transition for `c`, `f(s)` points back to the root.
4. We also augment the `output` set of `s` with the `output`
5. We also augment the `output` set of `s` with the `output` set of its failure state `f(s)`. This ensures we report patterns that are suffixes of other patterns (e.g., finding "row" when we match "arrow").

1 Start at Parent



- Find `f(3)`.
- Start at parent: node 2.

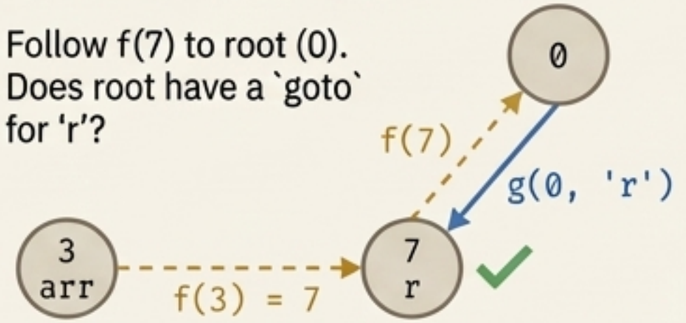
2 Traverse Parent's Failure Link



- Follow f(2) to node 7.
- Does node 7 have a `goto` for 'r'?

3 Find Transition & Set Link

- Follow f(7) to root (0).
- Does root have a `goto` for 'r'?



Code Deep Dive: `build_failure()`

代码深度解析：`build_failure()`

```
void build_failure() { // 构建失败指针
    queue<tnode*> q;
    root->fail = NULL;
    q.push(root);
    while(!q.empty()) {
        tnode* cur = q.front(); q.pop();
        for(int i=0; i < dsize; ++i) {
            if(cur->go[i]) { // State exists, find its failure link
                tnode* p = cur->fail;
                // Traverse failure links of parent to find fallback
                while(p && !p->go[i])
                    p = p->fail;
                if(p)
                    cur->go[i]->fail = p->go[i];
                else
                    cur->go[i]->fail = root;
                q.push(cur->go[i]);
            } else { // State does not exist, fill it in for faster search
                // This is a key optimization: pre-computing transitions
                // 这是一个关键优化：预计算状态转换
                cur->go[i] = (cur == root) ? root : cur->fail->go[i];
            }
        }
}   }
```

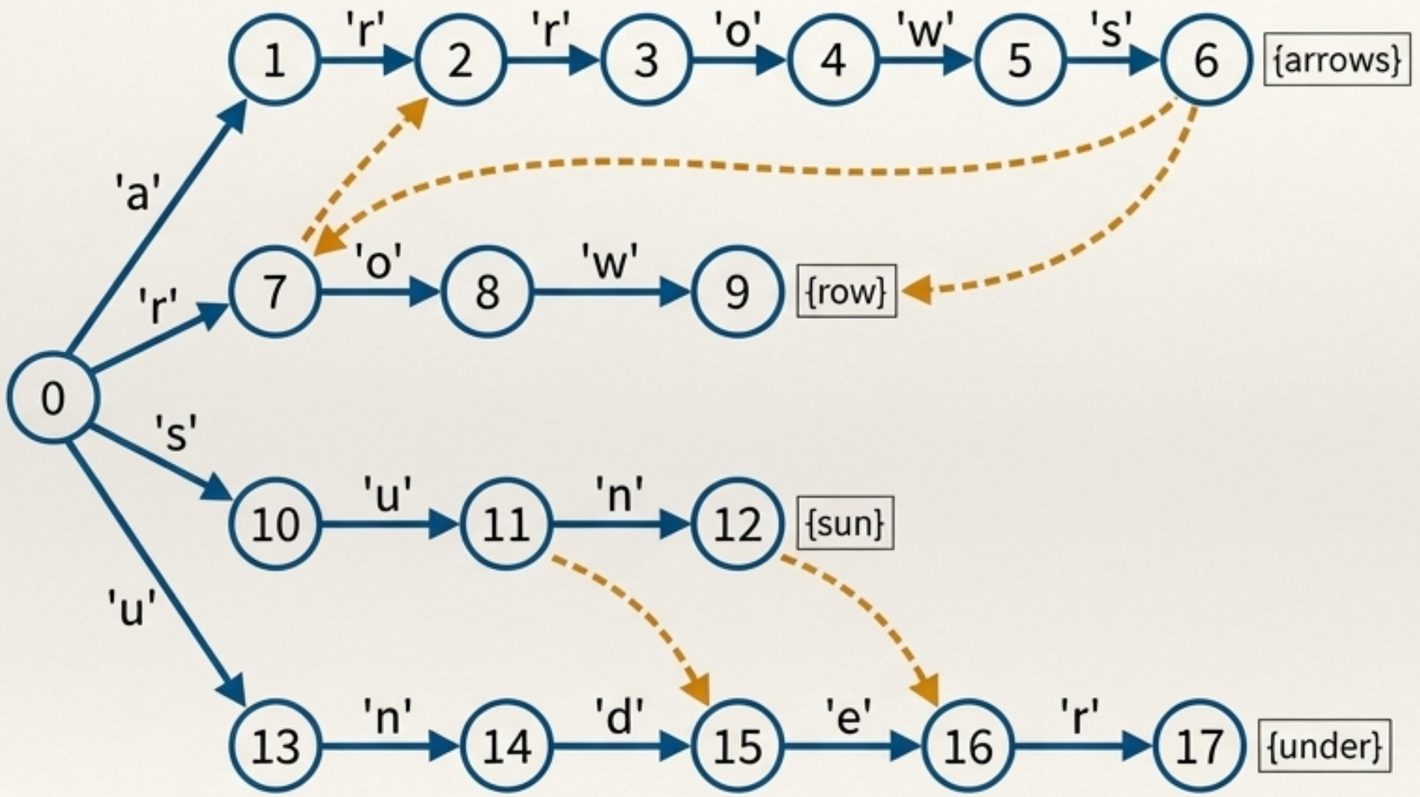
Standard BFS implementation using a queue.

The core loop: traversing the parent's failure chain to find the longest suffix-prefix.

Setting the failure link once a valid fallback is found.

CRITICAL OPTIMIZATION: Instead of leaving `go` transitions null, we fill them with the appropriate failure transition. This makes the final search code much simpler, as it never has to explicitly follow failure links.

The Fully Assembled Automaton
完全组装的自动机



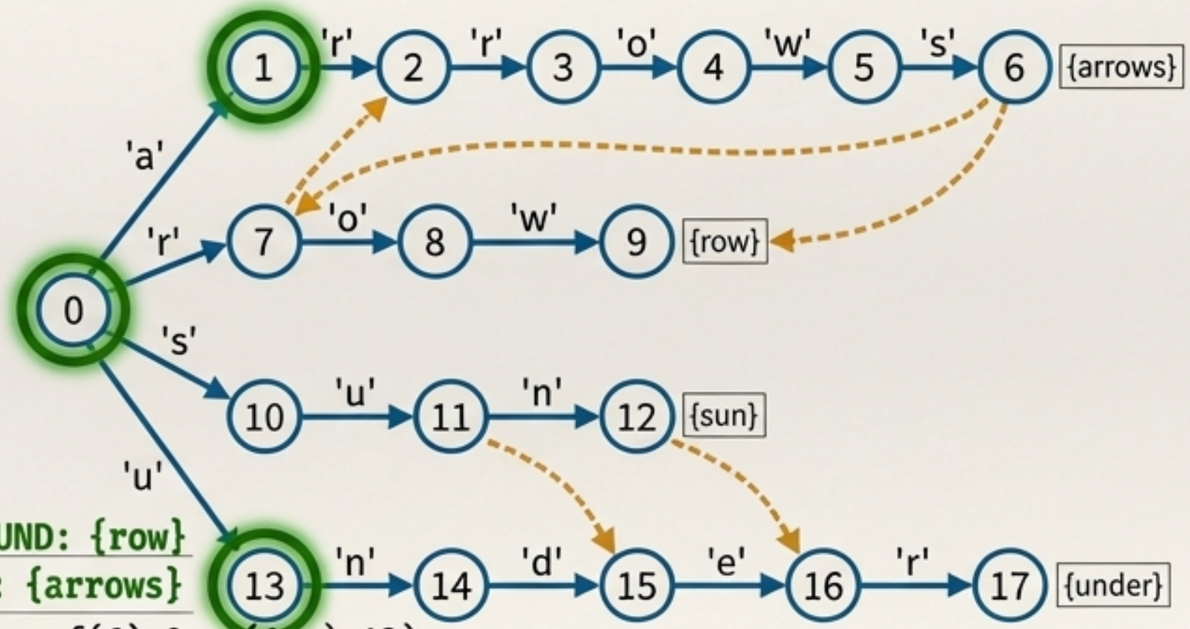
State i	g(i,c)	f(i)	output(i)
1 (a)	g(1,'r')=2	0	∅
5 (arrow)	g(5,'s')=6	9	∅
6 (arrows)	∅	0	{arrows}
9 (row)	∅	0	{row}
11 (su)	g(11,'n')=12	14	∅
12 (sun)	∅	15	{sun}

Step 3: Searching the Text in a Single Pass
第三步：单次遍历搜索文本

The search algorithm iterates through the text `t`, maintaining a pointer to the current state in the automaton. For each character, it follows the `goto` transition. Because of our `build_failure` optimization, we never need to explicitly follow failure links during the search.
搜索算法遍历文本 `t`，同时维持一个指向自动机当前状态的指针。对于每个字符，它都遵循 `goto` 转换。由于我们在 `build_failure` 中的优化，搜索过程中无需显式地跟随失败链接。

`t = "bcarrowsug"`

1.	char: 'b'	state: 0 → 0	
2.	char: 'c'	state: 0 → 0	
3.	char: 'a'	state: 0 → 1	
4.	char: 'r'	state: 1 → 2	
5.	char: 'r'	state: 2 → 3	
6.	char: 'o'	state: 3 → 4	
7.	char: 'w'	state: 4 → 5	Check output(5) (empty) Check output(f(5)=9) **MATCH FOUND: {row}**
8.	char: 's'	state: 5 → 6	Check output(6) **MATCH FOUND: {arrows}**
9.	char: 'u'	state: 6 → 13	(via failure logic baked into goto, f(6)=0, g(0,u)=13)
10.	char: 'g'	state: 13 → 0	(via failure logic)



Code Deep Dive: mult_search()
代码深度解析：mult_search()

A single loop through the text. The core of the O(n) search time.

The only transition logic needed. This single line handles both successful matches and failures thanks to the pre-filled go table from the optimization in build_failure().

```
int mult_search(const string& text) { // AC 多模式匹配
    int cnt = 0;
    tnode *cur = root;
    for(int i=0; i < text.length(); ++i) {
        // Simply follow the pre-computed goto transition.
        // The failure logic is already baked into the table.
        // 只需跟随预先计算好的 goto 转换。
        // 失败逻辑已经融入转换表中。
        cur = cur->go[idx[text[i]]];

        if(cur->cnt > 0) {
            cnt += cur->cnt;
            // A helper function 'outout' would iterate
            // through cur->output and print the matches.
            // 一个辅助函数 'outout' 会遍历 cur->output 并打印匹配项。
            outout(cur);
        }
    }
    return cnt;
}
```

Check for matches. cnt includes matches from the current state and all states reachable via failure links, which were aggregated during the build phase.

Performance Analysis: Blazing Fast and Scalable
性能分析：极速且可扩展

Phase	Time Complexity	Notes
Trie Construction	$O(m)$	Proportional to the total length of all patterns.
Failure Link Construction	$O(m)$	Each node is processed once in a BFS manner.
Search	$O(n+z)$	A single pass over the text, plus time to report z matches.
Total Time	$O(n+m+z)$	A linear-time algorithm.
Space Complexity	$O(m)$	To store the automaton.

Key Insight: The search time is fundamentally dependent on the text length (n) and total pattern length (m), but is **independent of the number of patterns (k)**. This is what makes Aho-Corasick the definitive solution for large-scale dictionary matching.

核心洞见：搜索时间根本上取决于文本长度（n）和模式串总长度（m），但与**模式串的数量（k）无关**。这使得 Aho-Corasick 成为大规模字典匹配的权威解决方案。

Aho-Corasick: The Final Word
Aho-Corasick 算法：总结

Summary

- Aho-Corasick elegantly combines a Trie data structure with KMP-style failure transitions to create a highly efficient multi-pattern matching machine.
  Aho-Corasick 算法优雅地将 Trie 数据结构与 KMP 风格的失败转换相结合，创造了一台高效的多模式匹配机器。
- Its core strength is processing text in a single pass to find all occurrences of a large dictionary of keywords simultaneously.
  其核心优势在于通过单次遍历文本，同时找到一个大型关键词字典中的所有出现。

Key Applications

Network Security: Intrusion Detection Systems scanning network traffic for thousands of malicious signatures in real-time.
网络安全：入侵检测系统实时扫描网络流量，以检测数千种恶意签名。

Computational Biology: Finding all occurrences of multiple DNA or protein sequences within a genome.
计算生物学：在基因组中寻找多个 DNA 或蛋白质序列的所有出现。

Data Processing & NLP: Content filters, spell checkers, and entity recognition systems that need to find all mentions of words from a large dictionary.
数据处理与自然语言处理：需要从大型词典中查找所有词语提及的内容过滤器、拼写检查器和实体识别系统。