**Keyword tree: Construction**

Construction for \( P = \{ P_1, \ldots, P_n \} \):

1. Begin with a root node only: Insert each pattern \( P_i \), one after the other:
2. Follow the path labeled by characters of \( P_i \) as long as possible:
   - If \( P_i \) exhausts at node \( v \), store an identifier of \( P_i \) at \( v \)
   - If the path terminates before \( P_i \), continue the path by adding new edges and nodes for the remaining characters of \( P_i \)
3. Takes clearly \( O(|P_1| + \cdots + |P_n|) = O(n) \) time

**Keyword tree: Lookup**

Lookup of a string \( F \): Starting at root, follow the path labeled by characters of \( P \) as long as possible:

1. If the path leads to a node with an identifier, \( P \) is a keyword in the dictionary.
2. If the path terminates before \( P \), the string is not in the dictionary.
3. Takes clearly \( O(|F|) \) time; efficient as a look-up method.
4. Naive application to pattern matching would lead to \( O(nm) \) time.
5. Next we extend a keyword tree into an automaton to support linear-time matching.

**Aho-Corasick automaton (1)**

States: nodes of the keyword tree

- **Initial state:** root (denoted 0)

The action of the automaton is determined by three functions defined for the states:

1. A **goto function** \( g(v, a) \) gives the state entered from current state \( v \) by matching text character \( a \)
2. If edge \( (u, v) \) is labeled by \( a \), then \( g(u, a) = v \)
3. \( g(0, a) = 0 \) for each \( a \) that does not label an edge out of the root
4. The automaton stays at the initial state while scanning non-matching characters

**Aho-Corasick automaton (2)**

2. A **failure function** \( f(s) \) gives the state entered at a mismatch
   - When \( s \) is the longest proper suffix of \( L(s) \) such that \( s \) is a prefix of some pattern, \( f(s) \) is the node labeled by \( s \) or \( s \) itself: we do not miss any potential occurrence by a failure transition.
3. An **output function** \( out(s) \) gives the set of patterns recognized when entering state \( s \).

Efficiency of AC search

Theorem: Searching text $T[1 \ldots n]$ with an AC automaton takes time $O(n + z)$, where $z$ is the number of pattern occurrences.

Proof: For each text char, we perform a goto, and possibly a number of fail transitions. Each goto either stays at the root, or the depth of the current state ($q$) increases by 1. As the depth of $q$ is increased at most $n$ times, each fail moves $q$ closer to the root, the number of them can be at most $n$. The $z$ occurrences can be reported in $O(z)$ time (say, as pattern identifiers and start positions of occurrences).

Constructing an AC automaton (I)

An AC automaton can be constructed in two phases:

Phase I:
1. Construct the keyword tree for $P$:
   * For each $P \in P$ added to the tree, if $v$ is the node labeled by $P$, set $\text{out}(v) := \{P\}$.
2. Complete the goto function for the root by setting $g(\{\}, a) = 0$ for each $a \in \Sigma$.
   * If the alphabet is fixed, Phase I takes time $O(n)$.

Result of Phase I

Idea of AC construction Phase II

Functions $\text{fail}$ and $\text{output}$ are computed for the nodes of the trie in a breadth-first order:

* when considering a node, nodes that are closer to the root have been treated.

Consider nodes $v$ and $w = g(r, a)$, that is, $r$ is the parent of $u$ and $L(u) = L(r) u$.

Now what should $f(u)$ be?

A: The deepest node labeled by a proper suffix of $L(u)$.

This is found by trying nodes labeled by shorter and shorter suffixes of $L(r)$, until a node is found for which $g(r, a)$ is defined and gets assigned to $f(u)$.

(Note that $v$ and $g(r, a)$ may be the root.)

Completing the output functions

What about $\text{out}(u) = \text{out}(v) \cup \text{out}(f(u))$?

This is done because any patterns recognized at $f(u)$ (and only those) are proper suffixes of $L(u)$, and shall thus be recognized at state $v$ also.
**Efficiency of the AC construction (1)**

Phase II can be implemented to run in time $O(n)$, too: The breadth-first traversal alone takes time proportional to the size of the tree, which is $O(n)$.

How much work is done for following $f$ transitions (in the inner-most loop)?

**AC construction: Unions of output functions**

Is it costly to unite output functions (that is, to perform $\text{out}(u) := \text{out}(u) \cup \text{out}(f(u))$)?

No: The sets can be implemented as linked lists, and a union thus in constant time
(Any patterns in out($f(u)$) are shorter than $C(u)$, which is (possibly) the only member of out($u$) before the assignment)

---

**AC construction: Number of fail transitions**

Consider the nodes $u_1, \ldots, u_t$ on a path created by entering a pattern $a_1 \ldots a_t$ to the tree, and the depths of their $f$ nodes, denoted by $df(u_1), \ldots, df(u_t)$

Now $df(u_{max}) \leq df(u) + 1$, which means that the $df$ values can increase at most $t$ times along the path. Now each execution of $v := f(v)$ decreases the value of $df(u)$ by one at least

$\Rightarrow$ in total, at most $t$ fail transitions (for a pattern of length $t$) $\Rightarrow$ the $f$ links are followed, in total, at most $n$ times

---

**Biological applications**

1. Matching against a library of known patterns
   A Sequence-tagged-site (STS) is, roughly, a DNA string of 200-300 bases whose left and right ends occur only once in the entire genome
   ESTS (expressed sequence tags) are STSs that participate in gene expression, and thus belong to genes
   Hundreds of thousands of STSs and tens of thousands of ESTs (by mid-90’s) are stored in databases, and used to compare against new DNA sequences
   $\Rightarrow$ set matching in time independent of the number of patterns is highly useful

2. Matching with wild cards
   Let $\phi$ be a wild card that matches any single character
   For example, $ab\phi c\phi d$ occurs at positions 2 and 8 of $xab\phi ab\phi c\phi d$ 
   A transcription factor is a protein that binds to specific locations of DNA and regulates its transcription to RNA
   Many transcription factors are separated into families characterized by substrings with wild cards
   Example: Signature for a common transcription factor of Zinc Finger: $C\phi\phi\phi\phi\phi\phi\phi\phi\phi H\phi\phi H$

3. Matching with wild cards (2)
   If the number of wild cards is bounded by a constant, patterns with wild-cards can be matched in linear time by counting occurrences of non-wild-card substrings of $P$:
   Let $P = \{P_1, \ldots, P_k\}$ be the substrings of $P$ separated by wild-cards, and let $t_1, \ldots, t_k$ be their start positions in $P$:
   1. for $i := 1$ to $m$ do $C[i] := 0$;
   2. Using AC, locate occurrences of patterns in $P^*$:
      When an occurrence of $P_j$ is found to start at position $j$ of $T$, increment $C[j - t_j]$ by one;
   3. Report any $i$ with $C[i] = k$ as a start of an occurrence

---

**Complexity of AC-based wild-card matching**

Preprocessing: $O(m + n)$ ($\approx \frac{1}{2} |P| \leq |P| - n$)
Search: $O(m + z)$, where $z$ is the number of occurrences
Each occurrence increments a cell of $C$ by one, and each cell is incremented at most $k$ times
$\Rightarrow$ there can be at most $kn$ occurrences ($= O(n)$ if $k$ is bounded by a constant)

**Theorem 3.5.1** If the number of wild-cards in pattern $P$ is bounded by a constant, exact matching with wild-cards can be performed in time $O(n + m)$