Aho-Corasick Executive Summary

Brennen Muller, Natalia Wunder, Deevena Sista

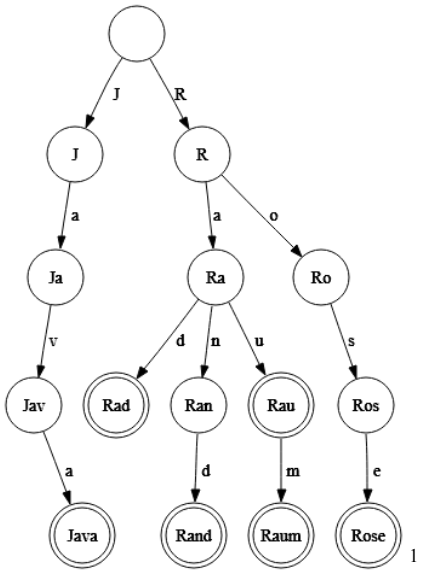
Introduction / Problem Statement:

Many modern applications - such as spam filters, PDF search tools, genetic sequencing algorithms, and intrusion detection systems - require a way to identify many patterns inside very large texts quickly. In the past, this problem was solved with "sliding window" algorithms (like the KMP or Boyer-Moore algorithm) that would compare a "window" or portion of the input to every pattern, and "slide" or traverse the window along the length of the text. This would run in $O(mn)$ time where m is the total length of all of the pattern words and n is the length of the input string we wish to search. This runtime explodes as the number of patterns increases (as m increases), making these algorithms impractical for situations where programs wish to look for a large set of patterns.

The Aho-Corasick algorithm was invented by Alfred V. Aho and Margaret J. Corasick in 1975 as a way to search strings in a linear runtime, and once served as the basis for the Unix command `fgrep`. To avoid searching the string by comparing against each pattern, the algorithm uses the given list of patterns to build a deterministic automaton. This way, the Aho-Corasick algorithm guarantees that each text character is examined once, and every pattern is checked in parallel. The process of constructing the automaton (trie with failure links) will be $O(m)$. This then allows for a linear search efficiency of $O(n + z)$ where z is the number of matches found. This means that overall, the Aho-Corasick algorithm has a runtime of $O(m + n + z)$.

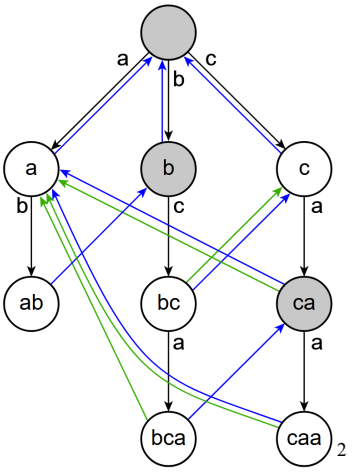
Overall Intuitive Approach / Solution:

The Aho-Corasick algorithm allows us to efficiently search for multiple patterns in any text, where, unlike naive approaches that check one pattern at a time, this algorithm preprocesses all patterns into a single automaton, enabling linear-time scanning of the text regardless of the number of patterns searched. The algorithm combines the trie data structure with the ideas of finite state automata to construct an automaton that can match multiple patterns in a single pass over the given text. There are three main components to the Aho-Corasick automaton: creating the trie, constructing the links, and searching the text.



[1]

First, we will use the trie data structure to insert the patterns into a trie. A trie can be thought of as a finite state machine where each node represents a prefix shared by one or more strings. The root node acts as a starting point with no character, where other nodes represent characters or parts of a string as they are connected by edges. Essentially, this structure allows us to capture our dictionary of patterns compactly. As seen in the trie above, there are no repetitions of substrings in prefixes, where “Rau” and “Raum” continue along one branch as they share a similar beginning.



[2]

Next, we will add more edges to the trie, specifically suffix links and terminal links, to improve the performance of our search. Once we have our trie structure, we can add suffix links, which tell us where to go if a mismatch occurs while scanning the text. For any node in the trie, its failure link points to the longest proper suffix of the current path that is also a prefix of another

[1] https://cp-algorithms.com/string/aho_corasick.html
[2] https://cp-algorithms.com/string/aho_corasick.html

pattern in the trie. For the root of the trie, the suffix link will point to itself. Above, we can see the blue arrows are suffix links, where for bca, a suffix link goes to ca, as the longest suffix of the current path of bca matches the prefix of the other pattern ca in the trie, while for ca, the longest suffix of the current path matches the prefix of the other pattern a. Essentially, when we hit a string where we cannot continue to read a character, we can fall to a suffix link where we can preserve as much context as possible. These links allow for the reuse of partial matches and to avoid unnecessary backtracking, allowing us to ensure our linear search.

We also need to create output or terminal links, where whenever we visit a node, we will output the string represented by the node at the end of the terminal, where by precomputing where we need to end up, we can instantly read off any extra patterns to emit, therefore outputting everything in the chain. The goal is to introduce output links so that when nodes are visited, the automaton outputs all the suffixes that end there. Above, we can see these as green arrows, where, for example, bca has a terminal link towards a, as bca encompasses a within itself.

Once we have our automaton created, we can search the text by scanning one character at a time, transitioning through the automaton. If there is no valid next step, we will transition through a suffix link. At each step, the algorithm will check if the current state corresponds to any completed patterns by checking terminal links.

Implementation:

To implement the Aho-Corasick algorithm, we construct our automaton based on a trie data structure along with suffix and terminal links to allow for efficient backtracking and multi-pattern recognition.

```
class TrieNode {
    string searchString;
    bool accept = false;
    int children[ALPHABET_SIZE];
    int parent;
    char lastChar;
    int suffixLink = UNSET;
    int terminalLink = NONE;
}

def addString(string str) {
    if (str.length() == 0)
        return;

    int state = ROOT;
        for (char character : str) {
```

```
            int index = character - 'a';

            if (trie[state].children[index] == NONE) {
                int newState = trie.size();
                trie.emplace_back(state, character);
                trie[state].children[index] = newState;
            }

            state = trie[state].children[index];
        }
     trie[state].accept = true;
     trie[state].searchString = str;
}
```

First, we begin by initializing the trie with a root node. Each pattern or keyword is inserted one character at a time. If a character edge does not exist from the current node, a new node is created and linked from that character. The final node in the path is marked as an accepting node and stores the original pattern. Each node stores its parent index and the character used to reach it, an array of child indices representing character transitions, booleans for whether it is an accepting state, the full pattern string, and links for suffix and terminal traversal. The pseudocode for the trie node and the insertion of a pattern operation are shown above.

```
def setSuffixLink(int state) {
    if (trie[state].parent == NONE) {
        trie[state].suffixLink = NONE;
        return;
    }

    if (trie[state].parent == ROOT) {
        trie[state].suffixLink = ROOT;
        return;
    }

    int index = trie[state].lastChar - 'a';
    int suffixChain = trie[trie[state].parent].suffixLink;

    while (trie[suffixChain].children[index] == NONE) {
        suffixChain = trie[suffixChain].suffixLink;

        if (suffixChain == NONE || suffixChain == UNSET) {
            trie[state].suffixLink = ROOT;
            return;
```

```
        }
    }

    trie[state].suffixLink = trie[suffixChain].children[index];
}
```

Once we have established the trie, we may add suffix links to our automaton. Suffix links serve as fallback paths when the current input does not match any direct child of the current node, where the suffix link points to the node that represents the longest possible proper suffix of the string that exists as a prefix in the trie. After all patterns are inserted, these links can be computed in a breadth-first search manner. For each node, if it is the child of the root, the suffix link is the root, otherwise, its suffix link is found by following the suffix link of its parent until a node is found with a child with the same character. If none is found, it defaults to the root. The pseudocode for setting suffix links is seen above.

```
def setTerminalLink(int state) {
    if (trie[state].suffixLink == NONE || trie[state].suffixLink == ROOT) {
        trie[state].terminalLink = NONE;
        return;
    }

    if (trie[trie[state].suffixLink].accept)
        trie[state].terminalLink = trie[state].suffixLink;

    else
        trie[state].terminalLink = trie[trie[state].suffixLink].terminalLink;
}
```

Now that we can efficiently fall back on matches, we need to report all the matches that end at or before a certain position of the input. If a node's suffix link points to an accepting state, it becomes its terminal link, otherwise, it inherits the terminal link of its suffix link recursively. This chain enables a quick retrieval of all matched patterns that end at the current state, which is useful for overlapping and nested patterns. The pseudocode for setting terminal links can be seen above.

Once the automaton is constructed, we search by reading the input string character by character, where if a matching edge exists, the automaton transitions normally, but if one does not, then it repeatedly follows suffix links until it either finds a match or returns to the root. At each state, it checks if the node is accepting or if there are any terminal links, therefore finding all matches and submatches in one parse.

Summary of Programming Challenge:

The programming challenge centers around building a dynamic library catalog search system. In the problem, the student has suddenly inherited a growing collection of books and must implement an efficient way to search through the titles using a dynamic list of keywords.

Although the students are only required to output the number of keyword matches in each title, the underlying infrastructure could also be used to sort book titles by "relevance" (number of keyword matches), categorize books by topic, or even search the contents of the books quickly (if available).

This assignment includes five key learning objectives:

- Learn about advanced string-search algorithms
- Implement an algorithm based on finite automata and trie-based data structures
- Analyze complexity classes that depend on multiple input parameters
- Devise mechanisms to add dynamic functionality to static data structures
- Render pseudo-code fragments and descriptions into functional programs

Although implementations of the Aho-Corasick algorithm are readily available online, the majority only present the immutable automaton. Thus, this assignment requires students to extend that functionality and allow for dynamic additions to the set of search strings. To implement this change, students must have an intimate understanding of the failure function, which is the key component of the automaton, and find efficient ways to reverse and reallocate links only as required.

This addition was proposed by Bertrand Meyer of the University of California, Santa Barbara, in the paper titled "Incremental String Matching"[3]. In the paper, Meyer proves that an extension using the inverse failure function only marginally increases the runtime complexity from $O(l_{total} + n + z)$ to $O(l_{max} \cdot l_{total} + n + z)$, where:

- l_{total} is the total length of all keywords
- l_{max} is the length of the longest keywords
- m is the number of keywords
- n is the length of the text
- z is the number of matches
- and K, the size of the alphabet, is assumed to be constant.

Note that the slower runtime is only necessary when new keywords are added after performing the first search, as the immutable version can be used until that point. Additionally, it is trivial to

[3] https://se.inf.ethz.ch/~meyer/publications/string/string_matching.pdf

show that this method is better than rebuilding the automaton each time, which costs $O(ml_{total})$ and will always perform at least as much work, often more, than the better solution. Consequently, the extension and its performance should be easy to measure with an autograder.

Although removing strings from the search set is likely possible by the same mechanism, we believed supporting both would likely be too complex for one homework assignment. As with other assignments, this problem also requires students to practice with I/O, string manipulation, recursion, and data structures (including various types of trees and arrays) in addition to the automata and failure links specific to the Aho-Corasick algorithm. It also makes them think critically about tradeoffs in data structure, including the efficiency of the implementation and making updates to a dynamic system.

Key Ideas for Solving Programming Challenge:

The key idea of the programming challenge is to implement the Aho-Corasick algorithm, which is an advanced string-matching automaton well-suited for finding multiple keywords in large texts, and support dynamic additions to the set of search strings without rebuilding the automaton.

Beginning from the standard, immutable version of the Aho-Corasick algorithm, the main challenge is efficiently determining how to update the failure links that are normally instantiated at the beginning.

In order to avoid the $O(ml_{total})$ case, students must find a way to identify which nodes' failure links need to be updated when each new node is added to the trie. Naive solutions that check every single node will fail to meet the desired runtime (performing $\Theta(l_{current\ total})$ work per string), and thus, students should analyze the structure and make some of the following key observations:

- All failure links to the new node must come from **lower in the tree**.
- Just as creating the original failure links can be done by tracing the parent node's failure links and searching for children, the failure links pointing to the new node can be found by tracing the **inverse path of failure links** from the new node's parent and checking for children.
- The **inverse failure links**, analogous to the original automaton, form a **suffix trie** with the same root as the original.

While this understanding can be difficult to grasp, implementing the **inverse failure trie**, either as a separate data structure or as additional data within the original trie, allows the algorithm to only check a subset of the original trie.

Even if new nodes are created for each character in the keyword string, all the inverse failure paths from the root will only be traversed once, so in the worst case, the cost to add a new string would be $O(l_{current\ total})$. However, it is often much less in practice; this is particularly true if the new nodes' most recent ancestor had few descendants compared to the overall size of the inverse failure trie.

Once implemented, this dynamically-updating automaton is used to search each book title and count the number of keyword matches. In addition to the main challenge of implementing the Aho-Corasick algorithm and the inverse failure trie, students must load and store the books titles (either as a list or one special-character-delimited string), perform case-insensitive matching, and handle non-word characters (including spaces and punctuation). Although these challenges are simple in isolation, they make the overall process of implementing the Aho-Corasick algorithm more closely replicate the conditions of a real-life keyword search engine.

Conclusion:

The Aho-Corasick algorithm solves the bottleneck problem of searching a string to match many different patterns. By pre-processing the list of patterns to build a deterministic automaton, the algorithm can scan each character of the input text once to search for all possible pattern matches in parallel. By pre-processing the patterns into a trie and constructing suffix links, we use $O(m)$ time to build the automaton, which gives us the linear search efficiency of $O(n + z)$. This gives us the Aho-Corasick algorithm an overall complexity of $O(m + n + z)$.

For the programming challenge, the group decided to modify the existing Aho-Corasick algorithm to allow for the dynamic ability to add additional pattern keywords into the trie without having to reconstruct it all over again. This allows students to go expand their knowledge beyond their general understanding of tries and finite-state automatons in order to allow for incremental updates with only marginal overhead costs.

Although the Aho-Corasick algorithm is a specialized algorithm, its efficiency in reducing multiple pattern matching from a quadratic to a linear runtime serves as a backbone for many large-scale text search applications used today.

References:

- Aho–Corasick algorithm - Wikipedia
- Aho-Corasick algorithm - Algorithms for Competitive Programming
- Aho-Corasick Algorithm for Pattern Searching | GeeksforGeeks
- Small02.pdf
- robert-bor/aho-corasick: Java implementation of the Aho-Corasick algorithm for efficient string matching
- PII: 0020-0190(85)90088-2