Knuth-Morris-Pratt String Matching Algorithm

I. Introduction

The Knuth-Morris-Pratt (KMP) algorithm is a string-searching algorithm that efficiently finds all occurrences of a pattern string within a larger text string. The problem, in a more procedural sense is: given a text of length n and a pattern of length m, find every starting index in the text where the pattern appears as a substring. This type of operation is important in applications such as text editors, search engines, and even things like DNA sequence analysis and network intrusion detection systems.

The naive approach to this problem scans the text character by character, and for each position attempts to match the full pattern. Whenever a mismatch occurs partway through the pattern, the text pointer is reset to just one position past where the last attempt began, and the pattern pointer restarts from the beginning. In the worst case, for example, searching for the pattern “AAAAB” in a text of all “A” characters, nearly every character in the text is compared against nearly every character in the pattern, which leads to a worst-case time complexity of O(n * m).

The primary challenge is that the naive algorithm throws away useful information each time it resets. When a mismatch occurs after successfully matching several characters, those matched characters already tell us something about the structure of the text, which is that we know exactly what characters in the text were just examined, because they matched a prefix of the pattern. If the pattern itself contains repeating substructures, like “ABABAB,” we may be able to skip ahead rather than restart from scratch. The goal is an algorithm that never needs to move the text pointer backward, scanning each text character at most once, for an overall runtime of O(n + m).

II. Overall Intuitive Approach

The intuition behind KMP is that when a mismatch occurs at some position in the pattern, we can consult a precomputed Fallback Table to determine the longest proper prefix of the matched portion that is also a suffix of it. We can then shift the pattern pointer to that position, preserving all the information the matched characters already gave us, and continue comparing without moving the text pointer backward.

To see why this works, consider searching for the pattern “ABABAC” in some text. Suppose we have successfully matched the first five characters “ABABA” and then encounter a mismatch on the sixth. Instead of restarting from index 0 in the pattern, we see that the matched prefix “ABABA” has “ABA” as its longest proper prefix that is also a suffix. We therefore shift the pattern pointer back to index 3 (after “ABA”) and resume. No characters in the text are re-examined, we just reposition within the pattern.

This idea is encoded in the Fallback Table, which is built from the pattern alone before the search begins. For each index i in the pattern, the table stores the length of the longest proper prefix of the substring pattern[0..i] that is also a suffix of it. A value of -1 at index 0 is a sentinel signaling that even the first character of the pattern did not match, so the text pointer should simply advance. The result is that both the table-construction phase and the search phase run in linear time for an overall complexity of O(n + m).

III. Implementation

Building the Fallback Table: Given a pattern W of length m, we allocate an array T of size m + 1. We initialize T[0] = -1 and set a position pointer pos = 1 and a candidate pointer cnd = 0. At each step, if W[pos] == W[cnd], we set T[pos] = T[cnd]. The current position can inherit

the same fallback as its candidate prefix, because any mismatch here is equivalent to a mismatch at cnd. Otherwise we record T[pos] = cnd and walk cnd backward through the table until we find a match or go through all options. Finally, we increment both pos and cnd and repeat. The last entry T[m] records the length of the longest proper prefix-suffix for the full pattern, which is used to resume the search after a complete match is found. The pseudocode is:

```
BUILD-TABLE(W):
  T[0] = -1,  pos = 1,  cnd = 0
  while pos < length(W):
    if W[pos] == W[cnd]:
      T[pos] = T[cnd]
    else:
      T[pos] = cnd
      while cnd >= 0 and W[pos] != W[cnd]:
        cnd = T[cnd]
    pos++,  cnd++
  T[pos] = cnd
  return T
```

Running the search: Using the Fallback Table, we scan the text S using index j (text pointer) and track our position in the pattern with index k (pattern pointer), both starting at 0. When S[j] == W[k], both pointers advance. When k reaches m, a complete match has been found at index j - k, and we set k = T[k] to prepare for the next potential match without losing the suffix overlap. When a mismatch occurs, we look at the table: k = T[k]. If k drops below 0, the pattern could not be shifted to recover any prefix, so we advance j by one and reset k to 0. The text pointer j never moves backward, so each character is examined at most once.

```
KMP-SEARCH(S, W):
  T = BUILD-TABLE(W),  j = 0,  k = 0
```

```
while j < length(S):
  if W[k] == S[j]:
    j++,  k++
    if k == length(W):
      record match at (j - k)
      k = T[k]
  else:
    k = T[k]
    if k < 0:  j++,  k++
```

IV. Summary of Programming Challenge

The programming challenge, themed as "Circuit Board Inspection," extends KMP one dimension to two. Cast as an industrial quality control scenario, students are given a large 2D circuit board scan represented as a grid of characters (up to 5,000×5,000) and a smaller 2D defect pattern, and must efficiently find all occurrences of the pattern in the map, returning the count and top-left coordinates of each match. The learning objective is for students to understand KMP enough to apply it as a building block in a different setting, rather than simply calling it as a black box.

The twist is that pattern matching is now two-dimensional, so a single pass of KMP over the text is no longer enough. Students should think about how to decompose the problem into two independent 1D KMP searches, one horizontal and one vertical, and how to encode intermediate results in a form that the second KMP pass can use. This design forces students to engage with what KMP actually compares (sequences of comparable elements) rather than treating it as string-specific magic. The challenge is appropriately scoped since it does not require any algorithm beyond KMP itself, but the two-pass decomposition and the state-encoding step are non-obvious and reward careful thought about the algorithm's structure.

V. Key Ideas for Solving Programming Challenge

The main insight is that 2D pattern matching can be reduced to two sequential 1D KMP searches. In the first horizontal pass, for each unique row of the pattern, run KMP across every row of the image to locate all starting columns where that pattern row appears. Each unique pattern row is assigned an integer state ID, and a state matrix of the same dimensions as the image is populated: cell (i, j) receives the state ID if the corresponding pattern row begins at column j in image row i, and -1 otherwise. Mapping repeated pattern rows to the same state ID avoids redundant KMP searches and is an important optimization for images with many repeated structures.

In the second vertical pass, for each candidate starting column j, extract the column of state IDs from the state matrix and run KMP on it using the vertical sequence of state IDs from the pattern as the search word. A match in this vertical search at row i means the pattern's full grid of rows aligns with the image starting at position (i, j), so (i, j) is recorded as a valid top-left coordinate.

The main challenge students must overcome is recognizing that KMP operates on any sequence of comparable elements, not just characters. Once the horizontal results are encoded as integer state IDs, the vertical KMP search treats those integers exactly as it would treat characters in a string. The overall time complexity of this approach is O(u * C * R), where R * C is the image size and u is the number of unique rows in the defect pattern. In the worst case, where every row in the pattern is distinct, this becomes O(r * R * C).

VI. Conclusion

The Knuth-Morris-Pratt algorithm focuses on not throwing away work you have already done, which is the key difference from the naive approach, which discards information after a partial match. KMP recognizes that the characters that did match tell you what was just seen in the text. If the pattern has internal repetition, a suffix of the matched portion may align with a prefix of the pattern, allowing the search to resume mid-pattern instead of starting over. This efficient shift is encoded in the precomputed Fallback Table. As a result, the text pointer continuously advances, examining each character at most once, which yields a total runtime of $O(n + m)$, significantly better than the naive $O(mn)$.

The programming challenge then extends these KMP ideas into two dimensions by encoding each row of the pattern as an abstract state and treating columns of those states as sequences to be searched. This two-dimensional approach allows the same KMP algorithm to be applied vertically after a horizontal pass, since the two passes are independent. This extension is designed to provide students with experience in abstracting and encoding structure, leading to a much stronger understanding of the Fallback Table's mechanics and where these ideas might be generally applied.