Designing Search Features for Structured Data Using String Matching Algorithms: A Case Study on Unified Pokémon Information Search

Stefani Angeline Oroh - 13524064
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: angelineoroh15@gmail.com , 13524064@std.stei.itb.ac.id

***Abstract*—This paper presents Who's That Pokémon?, a web-based search system designed to study how string matching algorithms can support the development of search features for structured data. Pokémon data from PokéAPI is used as a case study because it provides structured and relatively complete Pokémon information, including names, types, abilities, and moves, making it suitable for evaluating search across multiple attributes. The system transforms these attributes into a normalized searchable corpus and applies substring matching so users can search using a single query across different fields. To keep the results useful, a rule-based ranking strategy is applied to prioritize more relevant matches, such as name and type matches, before ability and move matches. Four exact string matching algorithms, namely Naive String Matching, Knuth-Morris-Pratt, Boyer-Moore, and Rabin-Karp, are implemented and evaluated using preprocessing time, matching time, total execution time, and character comparisons. The results show that algorithm performance is affected not only by theoretical complexity, but also by corpus structure, pattern length, query type, preprocessing overhead, and browser-based implementation details. This study demonstrates that choosing a string matching algorithm for a search feature should consider both performance and search design factors, such as searchable attributes, result ranking, and user-facing responsiveness.**

Keywords—string matching, structured data search, search feature design, Naive algorithm, Knuth-Morris-Pratt, Boyer-Moore, Rabin-Karp, Pokémon, PokéAPI

I. Introduction

When users search for information, they often do not know the exact name of the item they are looking for. A user may remember only a category, a characteristic, or a related term. For instance, in a Pokémon information system, a user may search for "fire" without knowing whether the word refers to a Pokémon type, an ability, or a move. In this situation, a search feature that only checks Pokémon names is not sufficient. The system needs to understand that useful information may be stored across several structured attributes.

This problem is common in many structured data applications, such as product catalogues, digital libraries, movie databases, and game databases often store information in multiple fields, such as name, category, description, tags, and metadata. A good search feature should not only determine whether a query appears in the data, but also decide where to search, how to combine multiple attributes into a searchable form, and how to present the most relevant results first. Therefore, designing a search feature for structured data requires both algorithmic efficiency and result relevance.

String matching algorithms are one of the fundamental techniques that can be used to support this kind of search. Although several algorithms may return the same matching results, their performance can differ depending on the pattern length, corpus size, text structure, and preprocessing cost. In addition, the choice of algorithm can affect the responsiveness of the search interface, especially when the searchable corpus is built from multiple attributes. This makes string matching not only a theoretical algorithmic problem, but also a practical design consideration in building search systems.

This paper presents Who's That Pokémon?, a web-based unified Pokémon information search system that uses Pokémon data from PokéAPI as a case study. Pokémon data is suitable for this study because it contains structured and relatively complete information, including names, types, abilities, and moves. The system allows a single query to search across these attributes using substring matching, then ranks the results based on where the match is found. Afterwards, four string matching algorithms, namely Naive String Matching, Knuth-Morris-Pratt, Boyer-Moore, and Rabin-Karp, are implemented and evaluated to analyze how algorithm behavior can support the design of a practical search feature for structured data.

The main contribution of this research is not merely comparing which algorithm is the fastest, but studying how string matching algorithms can influence search feature design, which includes how structured attributes are transformed into a searchable corpus, how result ranking improves relevance, and how algorithm performance changes under different query types and corpus sizes. Through this approach, the study aims to connect string matching theory with the practical needs of user-facing search systems.

II. THEORETICAL BACKGROUND

A. String Matching

In computer science, string matching aims to find the occurrence of a pattern within a given text. Formally, given a text (T) of length (n) and a pattern (P) of length (m), the objective is to determine whether P appears in T and if it does, its position or all occurrences will be identified. This problem also commonly referred as pattern matching or exact string matching when the pattern must match the text substring exactly.

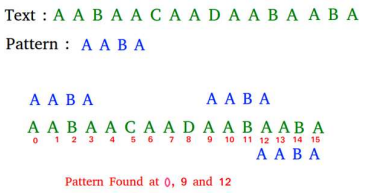


Fig 1. Pattern Matching Illustration (source: https://www.geeksforgeeks.org/dsa/what-is-pattern-searching/)

String matching is widely used in many application domains, including text editors, search engines, information retrieval, data mining, bioinformatics, and image processing. In the context of this research, the text is constructed from Pokémon information, while the pattern is the keyword entered by the user. This research focuses on substring matching rather than prefix-only matching. In prefix-only matching, a pattern is only considered a match if it appears at the beginning of a string. In substring matching, however, the pattern may appear at any position within the text. This makes the search system more suitable for evaluating string matching algorithms, because each query is treated as a pattern that must be located inside a larger textual corpus. For instance, the query "saur" can match "Bulbasaur", "Ivysaur", and "Venusaur", even though not all of these names begin with the query.

The performance of string matching algorithms depends not only on theoretical time complexity, but also on practical factors such as pattern length, text length, alphabet size, character distribution, preprocessing overhead, and implementation details. Therefore, this study compares four algorithms with different strategies: Naive String Matching, Knuth-Morris-Pratt, Boyer-Moore, and Rabin-Karp.

B. Naive String Matching

Naive String Matching is the simplest approach to exact string matching. The algorithm checks every possible starting position in the text and compares the pattern character by character. If all characters match, an occurrence is found. On the other hand, if a mismatch occurs, the algorithm shifts the pattern by one position and repeats the comparison from the beginning of the pattern [1].

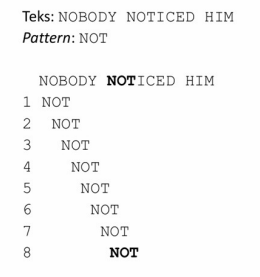


Fig 2. Naive String Matching (source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/23-Pencocokan-string-(2026).pdf)

Naive algorithm does not require preprocessing and is easy to implement. This makes it useful as a baseline in performance comparison. However, its worst-case performance can be inefficient because the same text characters may be compared repeatedly. In the worst case, the number of character comparisons can reach $m(n-m+1)$, which gives a time complexity of $O(nm)$ [1]. Despite this worst-case complexity, it can still perform well in practice when mismatches occur early or when the alphabet size is large. A larger alphabet increases the probability of early mismatches, reducing the number of comparisons needed at each alignment.

C. Knuth-Morris-Pratt Algorithm

The Knuth-Morris-Pratt algorithm (KMP), is an exact string matching algorithm that searches a pattern from left to right, similar to the brute force/naive approach. However, KMP improves the shifting process by using information from the pattern itself. Before the matching process begins, KMP builds a border function, also known as the failure function or LPS table [1,2]. For a pattern prefix ($P[0..k]$), the border function stores the length of the longest proper prefix of ($P[0..k]$) that is also a suffix of ($P[1..k]$). This means that when a mismatch occurs, the algorithm can determine which part of the pattern may still be reused instead of restarting the comparison from the beginning [1].

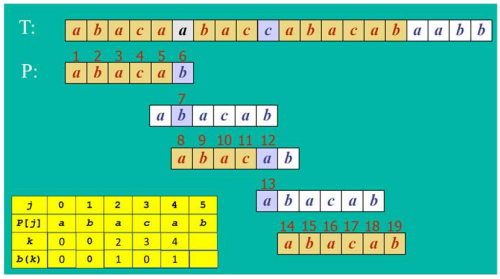


Fig 3. KMP Algorithm (source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/23-Pencocokan-string-(2026).pdf)

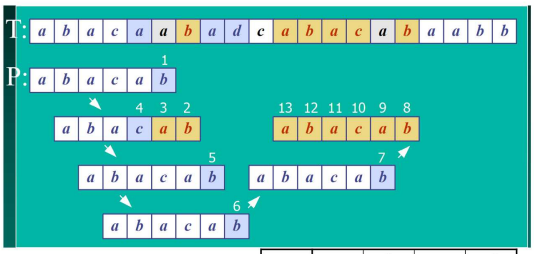
During matching, if the current pattern character ($P[j]$) does not match the current text character ($T[i]$), KMP uses the border function to update the pattern index. If ($j > 0$), the new comparison position becomes ($j = b[j-1]$), where (b) is the border function table. This avoids repeating

comparisons that are already known from previous matches. From Figure 3, the mismatch occurs at pattern index 5. Since the border value is $b(4) = 1$, KMP does not restart the comparison from the beginning of the pattern. Instead, the comparison resumes from pattern index 1, while the text index remains at the character that caused the mismatch.

Because the border function is computed in $(O(m))$ time and the text is scanned in $(O(n))$ time, the total time complexity of KMP is $(O(n+m))$ [1,2]. Another important property of KMP is that it never moves backward in the input text, making it suitable for large texts or stream-like input.

D. Boyer-Moore Algorithm

The Boyer-Moore algorithm is an exact string matching algorithm that compares the pattern from right to left. This approach is known as the looking-glass technique, because the comparison starts from the last character of the pattern rather than the first [1,3]. Before matching begins, Boyer-Moore preprocesses the pattern by building the Last Occurrence Function, denoted as $L(x)$. This function maps each character (x) in the alphabet to the largest index (i) such that $P[i] = x$. If the character does not appear in the pattern, then $L(x) = -1$ [1]. As shown in Figure 4, for the pattern abacab, the last occurrence values are $L(a) = 4$, $L(b) = 5$, $L(c) = 3$, and $L(d) = -1$.



x	a	b	c	d
L(x)	4	5	3	-1

Fig 4. Boyer-Moore Algorithm (source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/23-Pencocokan-string-(2026).pdf)

When a mismatch occurs at text index (i) and pattern index (j), Boyer-Moore uses the character-jump technique to determine the next alignment of the pattern. Let $x = T[i]$ be the mismatched character in the text. Using the Last Occurrence Function $L(x)$, the next text index computes with $(i = i + m - \min(j, 1 + L(x)))$, where (m) is the length of the pattern [1]. In Figure 4, the pattern is "abacab", so (m = 6), and the Last Occurrence Function gives $L(a) = 4$, $L(b) = 5$, $L(c) = 3$, and $L(d) = -1$. This table is built by scanning the pattern from left to right and storing the latest index of each character.

To illustrate, the leap at the sixth comparison occurs because the mismatched text character is 'd', while the current pattern character is 'b'. At this point, the mismatch happens at $(i = 8)$ and $(j = 5)$. Since d does not appear in the pattern, $(L(d) = -1)$. Therefore, the next position is calculated as $(i = 8 + 6 - min(5,1 + (-1)) = 8 + 6 - 0 = 14)$. This means the end of the pattern jumps from text index 8 to text index 14, shifting the pattern by six positions instead of only one. At comparison 7, another mismatch occurs with the character 'a', and since $(L(a) = 4)$, the pattern shifts by one more position. After that, the algorithm reaches the final alignment and compares the pattern from right to left, producing comparisons 8 to 13 until the pattern "abacab" is found.

The preprocessing step for building the Last Occurrence Function requires $(O(m + |\Sigma|))$ time, where m is the pattern length and $(|\Sigma|)$ is the alphabet size. In the worst case, Boyer-Moore can take $(O(nm))$ time, but in practice it is often much faster than brute force because its shifting rules allow the algorithm to skip many text positions [1,3].

E. Rabin-Karp Algorithm

Rabin-Karp algorithm uses hashing to find a pattern inside a text. Instead of comparing the pattern with every possible text window character by character, Rabin-Karp first converts the pattern and each text window of the same length into hash values. If the hash value of the current text window is different from the pattern hash, the algorithm can skip direct character comparison. However, if both hash values are equal, the algorithm still needs to verify the characters one by one because different strings may produce the same hash value, known as a hash collision. To make the process efficient, Rabin-Karp uses a rolling hash, which allows the hash of the next window to be computed by removing the first character of the previous window and adding the next character in the text [4].

```
Text    : charizard fire flying blaze fire punch
Pattern : fire

Pattern hash:
hash("fire") = H

Sliding windows with length 4:

char → hash ≠ H → skip
hari → hash ≠ H → skip
ariz → hash ≠ H → skip
...
fire → hash = H → verify characters → match
ire  → hash ≠ H → skip
re f → hash ≠ H → skip
...
fire → hash = H → verify characters → match
```

Fig 5. Rabin-Karp Algorithm (source: Author's Illustration)

For instance in Figure 5, suppose the searchable text is "charizard fire flying blaze fire punch" and the pattern is "fire". First, the algorithm computes the hash value of the pattern fire and the first four character window in the text. The window then slides one character at a time across the text. When the current window is fire, its hash value matches the pattern hash, so the algorithm verifies the characters and confirms a match. For other windows such as ire , re f, or e fl, the hash values are different, so no character verification is needed. In this paper, Rabin-Karp is useful to represent a hash-based string matching

approach. Its average-case time complexity is $O(n + m)$, where n is the text length and m is the pattern length. However, in the worst case, if many hash collisions occur, the time complexity can degrade to $O(nm)$. In the experiment, Rabin-Karp character comparisons only count the verification step after a hash match, while the cost of hash calculation is reflected in execution time.

F. Structured Data Search

Structured data is an organized data into several attributes or fields. In this research, each Pokémon is represented by multiple attributes, including name, type, ability, and move. This makes the search problem different from searching in a single plain text because a query may match different parts of the data. For example, the query "fire" may match a Pokémon type, a move name, or another related attribute. Therefore, a search feature for structured data must consider not only whether a query is found, but also which attribute contains the match. In addition, search systems commonly use normalization and ranking mechanisms to improve retrieval effectiveness [6].

Pokémon data is used as the case study because PokéAPI provides structured and relatively complete Pokémon information that can be accessed through an API [5]. These attributes make Pokémon data suitable for evaluating a unified search feature across multiple fields. Although the case study uses Pokémon, the same idea can also be applied to other structured data applications, such as product catalogs, digital libraries, movie databases, or game databases.

G. Searchable Corpus and Text Normalization

String matching algorithms work by finding a pattern inside a text. However, Pokémon data is stored in separate attributes, so the system first transforms these attributes into a unified searchable corpus. For example, one Pokémon entry can be represented as "charizard" | "fire flying" | "blaze solar power" | "fire punch" | "flamethrower" | "fire spin". This allows one query to search across multiple fields at the same time.

Before matching is performed, both the Pokémon data and the user query are normalized. In this system, normalization includes converting text to lowercase, replacing hyphens with spaces, and removing unnecessary spaces. For example, "fire-punch" is normalized into "fire punch", thus it can match a user query written with a space. This step helps make the search feature more flexible and consistent.

H. Result Relevance and Ranking

A string matching algorithm can determine whether a query appears in the searchable text, but it does not automatically decide which result is more relevant. This is important in structured data search because the same query may appear in different attributes. To illustrate, the query "fire" may appear as a Pokémon type or only as part of a move name. These two matches should not always be treated with the same priority.

To make the search results more useful, this system applies a rule-based ranking strategy. Matches are prioritized in the following order: name starts with the query, name contains the query, type match, ability match, and move match. With this ranking, a Pokémon whose name or type matches the query will appear before a Pokémon that only contains the query in its move list. This shows that search feature design is not only about finding matches, but also about presenting the most relevant results first.

I. Performance Metrics

To evaluate the implemented string matching algorithms, this research uses four performance metrics which are preprocessing time, matching time, total execution time, and character comparisons. Preprocessing time measures the time needed to prepare algorithm-specific data before matching begins. As an example, KMP builds a border function, Boyer-Moore builds a Last Occurrence Function, and Rabin-Karp computes hash-related values. Naive String Matching does not require preprocessing.

Matching time measures the time required to search the pattern in the corpus after preprocessing has been completed. Total execution time combines preprocessing time and matching time, making it useful for evaluating the overall cost of each algorithm. Character comparisons measure how many direct comparisons are performed between the pattern and the text during the search process.

These metrics are used because execution time alone may not fully explain algorithm behaviour. For example, Rabin-Karp may produce a low number of character comparisons because it only verifies characters when hash values match, but it still performs hash computations during the search. Therefore, both time-based metrics and comparison-based metrics are required to interpret the experimental results fairly. Through these metrics, the research can analyze why each algorithm behaves differently in a structured data search system.

III. IMPLEMENTATION

A. Result Relevance and Ranking

The system is implemented as a web-based single page application called Who's That Pokémon?. The application retrieves Pokémon data from PokéAPI, transforms selected attributes into searchable text, receives a user query, performs substring matching, ranks the matched Pokémon, and displays benchmark results for four string matching algorithms. The main purpose of the implementation is to demonstrate how string matching algorithms can be applied to a structured data search feature.

The system uses four main components. The first component is the data preparation module, which fetches and stores Pokémon details. Second, is the normalization utility, which ensures that Pokémon data and user queries use the same text format. Third, is the search and ranking process, which identifies where the query appears and orders the results based on relevance. Last, is the benchmark worker, which runs the

algorithms and measures their performance without blocking the user interface.

B. Text Normalization

Before the search process is performed, both Pokémon data and user input are normalized using the same function. The normalization process converts text into lowercase, replaces hyphens with spaces, removes repeated spaces, and trims unnecessary whitespace. This step is important because some data from PokéAPI is written using hyphens, such as fire-punch, while users may type the query using spaces, such as fire punch. In addition, applying the same normalization rule to both the dataset and the query, the system can reduce formatting differences that may prevent valid matches. This makes the search feature more consistent and flexible without changing the actual string matching algorithms.

```
export function normalizeSearchText(value) {
  return String(value ?? '')
    .toLocaleLowerCase('en-US')
    .replace(/-/g, ' ')
    .replace(/\s+/g, ' ')
    .trim();
}
```

Fig 6. normalize.js (source: Author's Code)

C. Searchable Corpus Construction

Attributes from each Pokémon detail, such as name, types, abilities, and moves are normalized separately and then combined into one searchable text using the format name | types | abilities | moves. This allows a single user query to be matched across multiple structured fields.

Although the attributes are combined for the matching process, the original structured data is still preserved, which is important because the system still needs to display Pokémon cards, type badges, abilities, moves, and match reasons separately. Therefore, the corpus is used for string matching, while the structured attributes are used for result presentation and relevance ranking.

```
function createSearchableDetail(data, sourceUrl = '') {
  const types = data.types?.map((entry) => entry.type.name) ?? [];
  const abilities = data.abilities?.map((entry) => entry.ability.name) ?? [];
  const moves = data.moves?.map((entry) => entry.move.name) ?? [];
  const searchableFields = {
    name: normalizeSearchText(data.name),
    types: normalizeSearchText(types.join(' ')),
    abilities: normalizeSearchText(abilities.join(' ')),
    moves: normalizeSearchText(moves.join(' ')),
  };

  return {
    cacheVersion: CACHE_VERSION,
    id: data.id,
    name: data.name,
    url: sourceUrl,
    image:
      data.sprites?.other?.['official-artwork']?.front_default
      || data.sprites?.other?.home?.front_default
      || data.sprites?.front_default
      || '',
    sprite: data.sprites?.front_default || '',
    types,
    abilities,
    moves,
    height: data.height ?? 0,
    weight: data.weight ?? 0,
    stats: data.stats?.map((entry) => ({
      name: entry.stat.name,
      value: entry.base_stat,
    })) ?? [],
    searchableFields,
    searchableText: [
      searchableFields.name,
      searchableFields.types,
      searchableFields.abilities,
      searchableFields.moves,
    ].join(' | '),
  };
}
```

Fig 7. pokemon-data.js (source: Author's Code)

D. Query Validation and Worker Execution

When a user enters a query, the system first normalizes and validates the input. Queries with fewer than two characters are not processed because they tend to produce overly broad results and unstable benchmark behaviour. This validation helps ensure that the search process remains meaningful for users. After validation, the search data and normalized pattern are sent to a Web Worker. The Web Worker is used to run the benchmark process in the background so that the main interface remains responsive. This prevents the website from freezing while the four string matching algorithms are being executed on the searchable corpus.

```
const worker = new Worker(new URL(`./worker.js?v=${APP_VERSION}`, import.meta.url), { type: 'module' });

worker.postMessage({
  records: state.pokemonSearchData.map((pokemon) => ({
    id: pokemon.id,
    name: pokemon.name,
    types: pokemon.types,
    abilities: pokemon.abilities,
    moves: pokemon.moves,
    searchableText: pokemon.searchableText,
  })),
  pattern,
  multiplier,
  sampleCount: BENCHMARK_SAMPLES,
});
```

Fig 8. app.js (source: Author's Code)

E. Match Detection and Result Ranking

After the matching process is completed, the system identifies where the query appears in each Pokémon entry. A match can come from the Pokémon name, type, ability, or move. This information is stored as match reasons and later displayed in the result cards so users can understand why a Pokémon appears in the search result.

Additionally, the system then ranks the results using a rule-based priority. A Pokémon whose name starts with the query receives the highest priority, followed by name contains, type match, ability match, and move match. This ranking strategy separates the search feature from a simple match-or-not-match process. The string matching algorithms detect

whether the query exists, while the ranking logic determines which matched Pokémon should be shown first.

```
let rank = 6;
if (nameStartsWith) rank = 1;
else if (nameContains) rank = 2;
else if (typeMatches.length > 0) rank = 3;
else if (abilityMatches.length > 0) rank = 4;
else if (moveMatches.length > 0) rank = 5;

return {
  nameStartsWith,
  nameContains,
  typeMatches,
  abilityMatches,
  moveMatches,
  totalMatches:
    (nameContains ? 1 : 0)
    + typeMatches.length
    + abilityMatches.length
    + moveMatches.length,
  rank,
  reasons,
};
```

Fig 9. worker.js (source: Author's Code)

F. Benchmark Procedure and Metrics

The benchmark process is executed inside a Web Worker using the same normalized pattern and the same searchable corpus for all algorithms. The Web Worker is used so that the benchmark computation can run in the background without blocking the main user interface [7]. Each algorithm creates a matcher object before the matching process begins. If the algorithm requires preprocessing, such as KMP, Boyer-Moore, or Rabin-Karp, the preprocessing time is recorded separately, while Naive String Matching does not require preprocessing.

The matching process is executed several times, and the median duration is used as the matching time. Execution time is measured using performance.now(), which provides high resolution timestamps for browser-based performance measurement [8]. The use of median duration helps reduce the effect of small timing fluctuations in the browser environment. The output of the benchmark includes preprocessing time, matching time, total execution time, character comparisons, and occurrence count. API fetching, image loading, and user interface rendering are not included in the benchmark, so the measured values focus only on the string matching process.

```
// If an algorithm requires preprocessing, such as KMP, Boyer-Moore,
// or Rabin-Karp, the preprocessing time is stored separately.
for (const algorithm of algorithms) {
  prepared[algorithm.key] = algorithm.createMatcher(pattern);
  preprocessingTimes[algorithm.key] = prepared[algorithm.key].preprocessingTime ?? 0;
}


// Each algorithm receives the same pattern, corpus, and corpus multiplier.
// The elapsed time of each run is stored so the median can be calculated later.
for (let sample = 0; sample < sampleCount; sample += 1) {
  for (const algorithm of order) {
    const start = performance.now();
    const stats = runRepeatedSearch(prepared[algorithm.key], corpus, multiplier);
    const elapsed = performance.now() - start;

    durations[algorithm.key].push(elapsed);
    latestStats[algorithm.key] = stats;
    checksum += stats.occurrences + stats.comparisons;
  }
}

// Matching time is taken from the median duration to reduce timing noise.
// Total time is calculated by adding preprocessing time and matching time.
const metrics = algorithms.map((algorithm) => {
  const matchingMs = median(durations[algorithm.key]);
  const preprocessingMs = preprocessingTimes[algorithm.key];

  return {
    algorithm: algorithm.key,
    preprocessingMs,
    matchingMs,
    totalMs: preprocessingMs + matchingMs,
    comparisons: latestStats[algorithm.key].comparisons,
    occurrences: latestStats[algorithm.key].occurrences,
    samples: sampleCount,
  };
});

```

Fig 10. Benchmark Procedure (source: Author's Code)

G. Rabin-Karp Character Comparison Counting

In the Rabin-Karp implementation, character comparisons are counted only during the verification step after the hash value of a text window matches the hash value of the pattern. If the hash values are different, the algorithm skips direct character verification. Rabin-Karp can produce a low number of character comparisons in the benchmark. However, hash computation is not counted as character comparison. The cost of computing and updating hash values is reflected in execution time instead. Therefore, Rabin-Karp results should be interpreted using both character comparisons and time-based metrics.

```
if (windowHash === patternHash) {
  let matched = true;

  for (let j = 0; j < m; j += 1) {
    comparisons += 1;

    if (text.charCodeAt(i + j) !== pattern.charCodeAt(j)) {
      matched = false;
      break;
    }
  }

  if (matched) {
    occurrences += 1;
  }
}
```

Fig 11. Comparison Counting in Rabin-Karp Algorithm(source: Author's Code)

IV. Testing and Results

A. Testing Setup

Testing was conducted using Pokémon data collected from PokéAPI, with four searchable attributes, namely name, type, ability, and move. The system was tested using functional queries to validate the search behaviour and benchmark queries

to compare the performance of Naive String Matching, Knuth-Morris-Pratt, Boyer-Moore, and Rabin-Karp. The benchmark was executed using two corpus scales, 1x and 100x, where 1x represents normal search usage and 100x represents a larger corpus stress test. The measured metrics include preprocessing time, matching time, total execution time, character comparisons, and result count. API fetching time, image loading time, and user interface rendering time were excluded from the benchmark so that the measurement focuses only on the string matching process.

B. Functional Testing

The purpose of this testing is to prove that the search feature works correctly. Here are the results:

Query	Purpose	Expected Result	Status
saur	Name substring	Returns Bulbasaur, Ivysaur, Venusaur	Pass
fire	Type/move search	Fire-type and fire-related Pokémon appear	Pass
blaze	Ability search	Pokémon with Blaze ability appear	Pass
fire punch	Move normalization	Matches fire-punch	Pass
solar power	Ability normalization	Matches solar-power	Pass
cc	Short valid query	Search runs	Pass
a	Invalid query	Search does not run	Pass
zzzz	No-match query	No result shown	Pass

Table 1. Functional Testing Result

C. Benchmark Summary

The benchmark was conducted using six representative queries and two corpus scales: 1x and 100x. The 1x scale represents normal search usage, while the 100x scale represents a larger corpus stress test. The main paper summarizes the most important findings based on the fastest total execution time and the fewest character comparisons. The complete benchmark results, including preprocessing time, matching time, total execution time, character comparisons, and result count for each algorithm, are provided in the Appendix.

Query	Scale	Fastest Algorithm	Fewest Comparisons	Result Count	Main Observation
saur	1x	Boyer-Moore	Rabin-Karp	3	Name substring query; Boyer-Moore performs well with fewer unnecessary checks.
saur	100x	Boyer-Moore	Rabin-Karp	3	Boyer-Moore remains the fastest as corpus size increases.
fire	1x	Boyer-Moore	Rabin-Karp	338	Broad structured query; many matches appear across type and move attributes.
fire	100x	KMP	Rabin-Karp	338	KMP becomes the fastest at larger scale, showing that the fastest algorithm can depend on corpus behavior.
fire punch	1x	Boyer-Moore	Rabin-Karp	166	Multi-word move query; normalization allows fire punch to match fire-punch.
fire punch	100x	Boyer-Moore	Rabin-Karp	166	Boyer-Moore benefits from the longer pattern and larger corpus.
hoothoot	1x	Boyer-Moore	Rabin-Karp	1	Repeated pattern query; KMP is not automatically the fastest.
hoothoot	100x	Boyer-Moore	Rabin-Karp	1	Boyer-Moore remains faster because it performs significantly fewer comparisons.
cc	1x	KMP	Rabin-Karp	4	Short valid query; KMP performs better even though Boyer-Moore uses fewer comparisons.

cc	100x	KMP	Rabin-Karp	4	KMP remains fastest for this short pattern at larger scale.
zzzz	1x	Boyer-Moore	Rabin-Karp	0	No-match query; Boyer-Moore is fastest, while Rabin-Karp performs no verification comparisons.
zzzz	100x	Boyer-Moore	Rabin-Karp	0	No-match stress test; Rabin-Karp still takes longer due to rolling hash computation.

Table 2. Benchmark Summary

D. Result Discussion

The functional and benchmark results show that the proposed system successfully supports structured data search. Queries such as "saur", "fire", "fire punch", and "hoothoot" demonstrate that the system can search not only Pokémon names, but also type, ability, and move attributes. The query "fire punch" is especially important because it shows the effect of text normalization. In PokéAPI, move names are commonly written using hyphens, such as fire-punch, while users may naturally type the query using spaces. By normalizing hyphens into spaces, the system becomes more user-friendly without changing the underlying string matching algorithms.

The query "fire" shows why structured data search needs ranking. Since fire can appear as a type, a move, or part of another attribute, simply finding the pattern is not enough. If all matches are treated equally, Pokémon that only have fire in one move may appear together with Pokémon whose type is Fire. The ranking strategy solves this by prioritizing matches based on where the query is found. This supports the main idea of the research: search feature design is not only about matching a string, but also about presenting results in a meaningful order.

In addition, from the performance perspective, Boyer-Moore performs strongly in most test cases. This is especially visible in "saur", "fire punch", "hoothoot", and "zzzz", where Boyer-Moore has much fewer character comparisons than Naive and KMP. The result supports the practical advantage of Boyer-Moore on a long and heterogeneous corpus, such as the combined Pokémon attributes. Since the corpus contains many different names, types, abilities, and moves, Boyer-Moore can often use its shifting strategy to skip unnecessary positions in the text.

However, the results also show that no single algorithm is always best. KMP becomes the fastest for the query "cc" at both 1x and 100x scale, and also for "fire" at 100x scale, which is an important finding because it shows that the best algorithm depends on the query and the corpus condition. A short query such as "cc" gives less room for Boyer-Moore to benefit from large shifts, while KMP can still scan the text consistently. Therefore, algorithm selection should not be based only on theoretical complexity, but also on expected query patterns.

Furthermore, the "hoothoot" test case provides another useful insight. Although hoothoot contains repeated structure, KMP is not the fastest in the benchmark. This means that a repetitive pattern alone does not guarantee that KMP will outperform other algorithms. KMP becomes more beneficial when the text also creates many partial matches that can reuse the prefix-suffix information. In this Pokémon corpus, Boyer-Moore still performs better because it can reduce the number of comparisons more effectively.

The "zzzz" query represents a no-match condition. This test case is useful because it shows how algorithms behave when the pattern does not appear in the corpus. Boyer-Moore remains the fastest because it can skip many positions during mismatches. Rabin-Karp shows zero character comparisons because no text window hash matches the pattern hash, thus no character verification is performed. Nevertheless, Rabin-Karp still has high execution time because it must compute rolling hashes across the corpus, which confirms that the character comparison count must be interpreted carefully.

Overall, the experiment shows that designing a search feature for structured data requires more than selecting the algorithm with the best theoretical complexity. A practical search feature must consider the structure of the data, the attributes included in the searchable corpus, the normalization process, the ranking strategy, the expected query type, and the performance behavior of the algorithm. In this study, Boyer-Moore is generally suitable for larger and heterogeneous searchable text, KMP can be effective for certain short or frequently matched queries, Naive remains useful as a simple baseline, and Rabin-Karp demonstrates that fewer direct character comparisons do not always produce faster execution time.

V. CONCLUSION

This research presented Who's That Pokémon?, a web-based search system designed to study how string matching algorithms can support search feature design for structured data. Using Pokémon data from PokéAPI, the system transformed multiple attributes, including name, type, ability, and move, into a searchable corpus. This allows a single query to search across different fields, making the search feature more flexible than a name only search.

The implementation shows that an effective search feature requires more than pattern detection. Text normalization is needed to handle different writing formats, such as matching "fire punch" with "fire-punch", whilst result ranking is required

because the same query may appear in different attributes. For instance, a type match should generally be considered more relevant than a match that only appears in a move name. Therefore, designers of structured data search systems should consider not only what fields are searchable, but also how matches from different fields are interpreted and ordered.

Based on the benchmark results, Boyer-Moore was the fastest algorithm in most test cases, especially for medium-length and longer queries on a heterogeneous corpus. Conversely, KMP performed best in some cases, such as short queries and the "fire" query at a larger corpus scale. Rabin-Karp produced the fewest character comparisons, but it was not the fastest because rolling hash computation still affected execution time. Then, Naive String Matching remained useful as a simple baseline. These results show that algorithm selection should depend on the expected query type, corpus size, match frequency, preprocessing overhead, and text structure.

The main lesson from this research is that designing a search feature for structured data should combine algorithmic performance with search usability. A practical search system should define searchable attributes clearly, normalize text consistently, rank results based on relevance, and evaluate algorithms under realistic query scenarios. Future work may extend this approach with fuzzy search, multi-keyword search, or indexing techniques such as inverted indexes and tries, especially for larger structured datasets such as product catalogs, movie databases, or digital libraries.

VI. Appendix

A. *Github Repository*

https://github.com/stefaniangeline/Makalah-STIMA

B. *Youtube*

https://youtu.be/egO1U_z_a3s?si=3bNsPp_1TvFGG2Oc

C. *Website*

https://makalah-stima-pokemon.vercel.app/

D. *Performance Benchmark Result*

https://drive.google.com/file/d/1KPH63BqoGc-2jTgsp2uUh58esln2s_Lu/view?usp=sharing

VII. Acknowledgement

First of all, the author would like to express gratitude to God Almighty for the strength and opportunity to complete this paper entitled "Designing Search Features for Structured Data Using String Matching Algorithms: A Case Study on Unified Pokémon Information Search." Then, the author also extends sincere appreciation to Ir. Rila Mandala, M.Eng., Ph.D., lecturer of IF2211 Algorithm Strategies, for his guidance and support throughout the learning process. Lastly, the author would also like to thank the author's parents, friends, and classmates for their encouragement, feedback, and moral support during the completion of this paper.

References

[1] R. Munir, "Pencocokan string," IF2211 Strategi Algoritma lecture slides, Institut Teknologi Bandung, Bandung, Indonesia, 2026. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/23-Pencocokan-string-(2026).pdf

[2] D. E. Knuth, J. H. Morris, Jr., and V. R. Pratt, "Fast pattern matching in strings," *SIAM Journal on Computing*, vol. 6, no. 2, pp. 323–350, Jun. 1977. [Online]. Available: https://www.cs.jhu.edu/~misha/Spring23/Knuth77.pdf

[3] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Communications of the ACM*, vol. 20, no. 10, pp. 762–772, Oct. 1977. [Online]. Available: https://www.cs.utexas.edu/~moore/publications/fstrpos.pdf

[4] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM Journal of Research and Development*, vol. 31, no. 2, pp. 249–260, Mar. 1987. [Online]. Available: https://didawiki.cli.di.unipi.it/lib/exe/fetch.php/bio/kr87.pdf

[5] PokéAPI, "PokéAPI documentation." Accessed: Jun. 19, 2026. [Online]. Available: https://pokeapi.co/docs/v2

[6] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge, U.K.: Cambridge University Press, 2008. [Online]. Available: https://nlp.stanford.edu/IR-book/information-retrieval-book.html

[7] MDN Web Docs, "Using Web Workers." Accessed: Jun. 18, 2026. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers

[8] MDN Web Docs, "Performance: now() method." Accessed: Jun. 18, 2026. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Performance/now

STATEMENT

I hereby declare that the paper I wrote is my own writing, not an adaptation or translation of someone else's paper, and it is not plagiarized.

Bandung, 19 June 2026

Stefani Angeline Oroh (13524064)