String Matching algorithms

A study

Literature survey + code

PES1UG20CS249 – Mohammed Ghouse

PES1UG20CS260 - Naman Kashyap

**1.0. Naive:**

**Theory:**

It is the brute force approach to string matching. It iterates through the pattern for each substring (of the same length as the pattern) in the text.

NOTE: n is the length of the text, and m is the length of the pattern. This convention is followed throughout this document.

Time complexity: **O(n2)**

Space complexity:**O(1)**

**Implementation:**

This algorithm is implemented using just 2 strings, and nested for/while loops to iterate through the text and pattern respectively.

**Code:**

\* Insert implementation here \*

**SWOT analysis:**

* Strengths: Simple and easy to implement. Constant memory.
* Weaknesses: As the name suggests, it is naïve. It is slow and not smart.
* Opportunities: Does the job well enough for small text and patterns. A good algorithm to start with.
* Threats: Can lead to Time Limit Exceeded in situations where the time complexity of the algorithm/code is of importance.
  1. **Accelerated naive:**

**Theory:**

It almost feels like a hack to able to do string matching in **O(1)** time AND space complexities! And that too in such a simple manner! But in life things that are too good to be true often are too good to be true. The catch in this algorithm is that all the characters *must be unique*.

Time complexity: **O(n)**

Space complexity:**O(1)**

**Implementation:**

This algorithm is implemented using a for/while loop that iterates over the text once and finds the pattern match.

**Code:**

\* Insert implementation here \*

**SWOT analysis:**

* Strengths: Simple, fast, efficient for unique text.
* Weaknesses: Doesn’t work, gives erroneous results for non-unique texts.
* Opportunities: Almost a “hack” for unique texts. A great lesson on simplicity in solving particular problems.
* Threats: If used for non-unique texts, it could lead to disaster.

**2.Boyer-Moore:**

**Theory:**

\* Insert theory here \*

**Implementation:**

\* Insert implementation here \*

**Code:**

\* Insert implementation here \*

**General testcases:**

* Testcases:
* Performance:

**Specific testcases:**

* Testcases:
* Performance:

**SWOT analysis:**

* Strengths:
* Weaknesses:
* Opportunities:
* Threats:

**3.Rabin-Karp:**

**Theory:**

Rabin–Karp algorithm is a string-matching algorithm created by Richard M. Karp and Michael O. Rabin (1987) that uses hashing to find an exact match of a pattern string in a text. It uses a *rolling hash* to quickly filter out positions of the text that cannot match the pattern, and then checks for a match at the remaining positions. Generalizations of the same idea can be used to find more than one match of a single pattern, or to find matches for more than one pattern.

**Implementation:**

It is implemented using the concept of hashing. The hash value of the pattern is calculated using an appropriate hash function. Then a window of the text (L to R) is taken and the hash value of that substring of the text is calculated. If there is a match, then each character in the pattern is matched with each character of the substring.

The ”*sliding window*” approach is used, and hence the concept of “*rolling hash*” is appropriated to find new hash values efficiently as the window slides.

**Code:**

\* Insert implementation here \*

**SWOT analysis:**

* Strengths:
* Weaknesses:
* Opportunities:
* Threats:

**4.KMP:**

**Theory:**

\* Insert theory here \*

**Implementation:**

\* Insert implementation here \*

**Code:**

\* Insert implementation here \*

**SWOT analysis:**

* Strengths:
* Weaknesses:
* Opportunities:
* Threats:

**5.Finite State Machine:**

**Theory:**

A finite-state machine (FSM) or finite-state automaton is a mathematical model of computation. It is an abstract machine that can be in exactly one of a finite number of states at any given time. The FSM can change from one state to another in response to some inputs; the change from one state to another is called a transition. An FSM is defined by a list of its states, its initial state, and the inputs that trigger each transition. Finite-state machines are of two types—deterministic finite-state machines and non-deterministic finite-state machines. A deterministic finite-state machine can be constructed equivalent to any non-deterministic one.

**Implementation:**

An FSM must be created for the given pattern. The text is passed as input to the FSM which either accepts or rejects the text.

**Code:**

\* Insert implementation here \*

**SWOT analysis:**

* Strengths: Logically easy to visualize and construct.
* Weaknesses: An FSM is unique to a pattern. Pre-processing time is hence, high.
* Opportunities: Can be used to physically implement in Automata related projects.
* Threats: Not really practical in code.

**6.0. Suffix Trie:**

**Theory:**

\* Insert theory here \*

**Implementation:**

\* Insert implementation here \*

**Code:**

\* Insert implementation here \*

**SWOT analysis:**

* Strengths:
* Weaknesses:
* Opportunities:
* Threats:

**6.1. Suffix Tree:**

**Theory:**

\* Insert theory here \*

**Implementation:**

\* Insert implementation here \*

**Code:**

\* Insert implementation here \*

**SWOT analysis:**

* Strengths:
* Weaknesses:
* Opportunities:
* Threats:

**7.Fuzzy algorithm:**

**Theory:**

\* Insert theory here \*

**Implementation:**

\* Insert implementation here \*

**Code:**

\* Insert implementation here \*

**General testcases:**

* Testcases:
* Performance:

**Specific testcases:**

* Testcases:
* Performance:

**SWOT analysis:**

* Strengths:
* Weaknesses:
* Opportunities:
* Threats: