Project report

Sujet : Automaton. Cloning egrep command supporting simplified ERE.

Member :
Runlin ZHOU
Weida LIU

16 Sep 2024 - 06 Oct 2024

Contents

1 Introduction of the project

This project focuses on implementing pattern recognition for regular expressions (RegEx) as defined by the Extended Regular Expressions (ERE) specifications. The goal is to search for patterns in a text file, processing each line individually.
This project focuses on comparing the performance differences between the three algorithms, including time complexity as well as space complexity.

- egrepLike: Self-programmed code modelled on egrep
- egrep: the default linux command
- KMP: the Knuth-Morris-Pratt algorithm

2 EgrepLike

2.1 Data Structures

2.1.1 NDFA Representation

The `NonDetFiniAuto` class is used to represent a non-deterministic finite automaton (NDFA). It consists of:

- `startState`: The initial state of the NDFA.
- `acceptState`: The final accept state of the NDFA.
- `transitions`: A list of `Transition` objects representing the state transitions, which may include ϵ-transitions.

Each `Transition` object represents a transition from one state to another on a specific input character or ϵ (epsilon transitions). This structure allows the representation of the non-deterministic nature of the automaton.

2.1.2 DFA Representation

The `DetFiniAuto` class represents a deterministic finite automaton (DFA), consisting of:

- `startState`: The start state of the DFA.
- `acceptStates`: A set of accepting states.
- `transitions`: A map from states to a map of character inputs and their corresponding target states.

The DFA is deterministic, meaning each state has at most one transition for each symbol in the alphabet, and no ϵ-transitions are allowed.

2.2 Algorithm Logic

2.2.1 Thompson's Construction

The code first converts a regular expression into a non-deterministic finite automaton (NDFA) using Thompson's construction algorithm. This construction is implemented in the `Construction` class. The `convert()` method recursively builds the NDFA for each component of the regular expression:

- `base()`: Constructs an NDFA for a single character.
- `concat()`: Handles concatenation of two sub-automata.
- `altern()`: Handles alternation (union) between two sub-automata.
- `star()`: Constructs an NDFA for the Kleene star operation.
- `plus()`: Similar to `star()`, but guarantees at least one occurrence of the pattern.
- `dot()`: Matches any single character (dot operator).

Each of these methods constructs transitions between states, potentially including ϵ-transitions, resulting in an NDFA that corresponds to the input regular expression.

2.2.2 Subset Construction (Powerset Algorithm)

After constructing the NDFA, the next step is converting it to a DFA. This is done using the subset construction algorithm (also known as the powerset construction) in the `convertToDFA()` method of the `Construction` class.

The algorithm begins by computing the ϵ-closure of the start state of the NDFA. It then systematically explores all possible transitions for each set of NDFA states on each input symbol. For each new set of NDFA states encountered, a new DFA state is created and added to the DFA transition table.

The ϵ-closure is a critical part of this algorithm, ensuring that all states reachable via ϵ-transitions are included in each new DFA state. The `epsilonClosure()` and `move()` methods are used to compute the closure and move functions, respectively.

2.2.3 Hopcroft's Algorithm for DFA Minimization

Once the DFA has been constructed, it is minimized using Hopcroft's algorithm, implemented in the `minimizeDFA()` method. The key steps of this algorithm are as follows:

- The states are initially partitioned into two sets: accepting and non-accepting states.
- For each input symbol, the algorithm refines these partitions by splitting states based on how they transition on that symbol.
- The refinement process continues until no further splitting is possible, resulting in the minimal DFA.

The algorithm maintains a work queue to track sets of states that need to be processed. At each iteration, the current set of states is examined to determine whether it can be split into smaller sets based on their transitions.

2.3 Exemple of usage: EgrepLike

```java
RegEx regEx1 = new RegEx();
RegExTree tree = regEx1.toTree(regEx);  // transform the regEx into a tree
Construction construction = new Construction();    // init the construction
NonDetFiniAuto ndfa = construction.convert(tree);  // using Thompson construct algo to
    generate NFA
DetFiniAuto dfa = construction.convertToDFA(ndfa);  // turn NFA into DFA
DetFiniAuto minimizeDFA = construction.minimizeDFA(dfa); // turn DFA into mDFA
// apply the mDFA to match the context in the file
try (BufferedReader br = new BufferedReader(new FileReader(file))) {
    String line; int lineNumber = 0;
    while ((line = br.readLine()) != null) {
        lineNumber++;
        if (matchesDFA(minimizeDFA, line)) {
            System.out.println(lineNumber + ":␣" + line); // print the matching line
        }
    }
} catch (IOException e) { e.printStackTrace(); }
```

3 Knuth-Morris-Pratt

The Knuth-Morris-Pratt algorithm, also known as the KMP algorithm, is a string matching algorithm. Its basic idea is that when there is a string mismatch, you can know which part of the text must match.
The core idea of the KMP algorithm is to use known information to reduce the number of matches as much as possible. Specifically, it can be roughly divided into two steps:

3.1 Construction of the KMP Partial Matching Table

The KMP algorithm can generate an LST array from a preprocessed pattern string and then convert the LST array to a carryover array, where carryover[i] indicates the position at which the pattern string should jump to continue matching if the i-th character in the pattern string does not match a character in the text string.

Algorithm 1 ComputeCarryover

```
Inputs
    • $pattern$ = a character array
Initializations
    • $len$ = length of $pattern$
    • $LST$ = an array of size $(len + 1)$, initialized with $LST[0] = -1$
    • $i = 0, j = -1$
Calculations
for $i \leftarrow 0$ to $len - 1$ by 1 do
    if $j == -1$ $pattern[i] == pattern[j]$ then
        $i \leftarrow i + 1$
        $j \leftarrow j + 1$
        $LST[i] \leftarrow j$
    end
    else
        $j \leftarrow next[j]$
    end
end
for $i \leftarrow 1$ to $len$ by 1 do
    if $LST[i] \geq 0$ $i < len$ $pattern[i] == pattern[LST[i]]$ then
        $LST[i] \leftarrow LST[LST[i]]$
    end
end
Outputs
    • The final $LST$ array
```

3.2 KMP Matching Process

The algorithm uses a pre-calculated jump table (carryover) to handle the matching process efficiently. Whenever a match fails, the algorithm quickly adjusts the pattern pointer through the jump table to continue searching for a match. If a match is successful, the match is recorded and the pointer is reset to continue searching for the next occurrence. Finally, the matched content is returned.

Algorithm 2 KPM (Knuth-Morris-Pratt)

```
Inputs
    • $factor$ = a string (the pattern to search for)
    • $file$ = a string (the file path)
Initializations
    • $factorChar$ = Convert $factor$ to a character array
    • $carryover$ = Result of calling $computeCarryover(factorChar)$
    • $lineNumber = 0$
Calculations
while there is a new line in the file do
    $lineNumber \leftarrow lineNumber + 1$
    $line$ = read the next line from the file
    $i \leftarrow 0$, $j \leftarrow 0$
    $textLen$ = length of $line$
    $factorLen$ = length of $factorChar$
    while $i < textLen$ do
        if $j == -1$ $line[i] == factorChar[j]$ then
            $i \leftarrow i + 1$
            $j \leftarrow j + 1$ if $j == factorLen$ then
                $j \leftarrow carryover[j]$ {Pattern found, reset $j$ using $carryover$}
            end
        end
        else
            $j \leftarrow carryover[j]$
        end
    end
end
Outputs
    • The matches found in the file (if any) with the corresponding line number
```

4 Performance Analysis

The analysis of the performance of three different search algorithms (EgrepLike, KMP, and Egrep) was conducted based on two main metrics: *execution time* and *memory usage*. The goal is to understand how these algorithms behave as the size of the input text files increases. Below, we outline the methods used for data analysis and present a comparative analysis of the results.

Data Analysis Methods

1. Performance Metrics

We measured two key performance metrics for each algorithm:

- **Execution Time (ms)**: The time taken by the algorithm to complete its task, measured in milliseconds.
- **Memory Usage (KB)**: The memory consumed by the algorithm during execution, measured in kilobytes.

2. Test Setup

For each algorithm, the following steps were followed:

1. Five text files of varying sizes, ranging from small to large (in terms of number of characters), were chosen.
2. Each algorithm was run 10 times on each text file, and the average execution time and memory usage were calculated to mitigate the impact of outliers and system noise.
3. The results were saved in a CSV file (`result.csv`) for further analysis.

Comparative Analysis of Results

From the **Execution Time vs. File Size** plot, we observe the following trends:

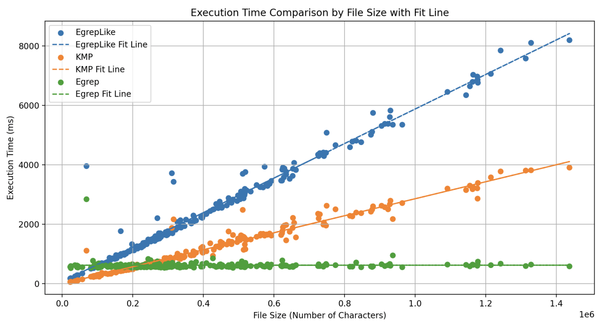

Figure 1: Time complexity

1. Execution Time Analysis

- **EgrepLike** exhibits the steepest slope, indicating that its execution time increases significantly as the file size grows. This suggests that EgrepLike's performance is more sensitive to larger file sizes.
- **KMP** demonstrates a moderate slope, suggesting better scalability compared to EgrepLike, but its performance also degrades as the file size increases.
- **Egrep** shows the least slope, meaning that its execution time remains relatively constant even for larger files, which points to superior efficiency in handling larger datasets.

The linear regression lines confirm these observations, as the slope of the EgrepLike regression line is much higher compared to KMP and Egrep.

2. Memory Usage Analysis

In the **Memory Usage vs. File Size** plot, we see that:

- **EgrepLike** again shows the highest memory usage, indicating that it requires more memory resources as the file size increases.
- **KMP** follows a similar trend but with a lower memory usage compared to EgrepLike.
- **Egrep** maintains the lowest memory usage, with its regression line remaining nearly flat, indicating minimal changes in memory usage even as file sizes increase.

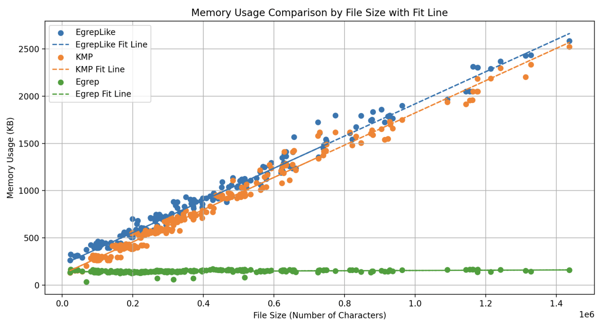

Figure 2: Memory Complexity

3. Overall Performance Comparison

From the results, it is clear that **Egrep** consistently outperforms the other algorithms in terms of both execution time and memory usage, especially for larger files. **KMP** presents a balanced performance but still lags behind Egrep, whereas **EgrepLike** performs the worst in terms of scalability for both metrics.