Regular Expression Matching for XWraps Action Level Data

Vincentius Edwin Teguh

School of Computer Science and Software Engineering, University of Western Australia

CEED Partner: Defence Science Technology Organisation

Abstract

This project aims to enable search of user activities on XWraps data. The problem is that the raw XWraps data is of very low level so that it is difficult to obtain information about the user activities in the recordings. We translate the raw data into XWraps action level data. A user activity can be described as a pattern of user actions. Therefore, we use patterns of action level data to represent user activities. We use regular expressions to represent the patterns. We implement and adapt an existing regular expression search algorithm for text strings by Wu and Manber to our XWraps action level data.

1.0 Introduction

The X Windows Recording and Playback System (XWraps) is a recording and playback tool for X Windows developed by the Defence Science Technology Organisation (DSTO) Australia. It is usually used to record user activities in training and command applications. XWraps performs the recording by taking periodic screenshots (usually every one second) and tracking all the mouse and keyboard events. XWraps uses the program *vncrec* to record the screen, and *xnee* to track the mouse and keyboard events.

1.1 Client's Requirements

The primary goal of this project is to enable analysis on XWraps data. The problem with XWraps raw data is that it is of very low level so that directly deducing meaningful information is very difficult. Up to now, a human-analyst usually examines the data manually to identify interesting user activities. Considering the large volume of data recorded, this process is too slow, laborious, and inefficient for manual work. Therefore, this project aims to assist human in performing analysis of XWraps data by developing a tool that enables analysts to search for user activities in the data.

1.2 Our Solution

Using XWraps raw data, it is difficult to get any information about user activities. Therefore, it is helpful to translate the low-level XWraps raw data into a higher level data. We find it helpful to view the XWraps data at three different levels of abstraction.

1. *XWraps Raw Data*

2. *XWraps Widget Level Data*
   XWraps widget level data contains information about the states of the visual components that the user interact with on the screen. This information is useful because the state of the recorded system can be detected from the states of the visual components that are visible on the screen. Some examples of this data are "Button A is visible", "Button B is disabled", etc.

3. *XWraps Action Level Data*
   This data has to do with the actions undertaken by the user. These actions can be detected from user input events, such as mouse clicks over buttons, and the changes in the states of the visual components.

We translate the raw XWraps data into XWraps action level data because a user activity can be described as a sequence of actions performed by the user and action level data is a good representation of user actions. Hence, searching for a user activity can be done by searching a pattern of user actions.

In order to get the action level data, we firstly translate the raw data into widget level data. This can be achieved using the screenshots in the raw data. On each screenshot, we check the visibility of every visual component. As a result, we have a list of the visual components that are visible on every screenshot. We built an annotation tool to label the visual components in the screenshots. This annotation data is then used in performing the translation to widget level data.

The action level data can be derived from the widget level data. The actions happening between two subsequent screenshots can be detected by analysing the differences between them. For example, if button A is not visible on the first screenshot, but is visible on the next screenshot, we know that there has been an action that caused the button to become visible. Another type of action that we want to capture is mouse clicks on buttons. We can achieve this by checking the mouse clicks on the *xnee* data whenever we detect that a button is visible.

Once we have the list of actions, the next step is finding the best way to represent a pattern of user actions. We choose regular expressions because they allow flexibility in expressing the patterns. Operations such as union (|) and Kleene closure (*) are very useful in representing a wide range of patterns.

Regular expression matching is a classic problem in computer science. A lot of research has been conducted in this area. Most of them are focusing on matching text strings. The action level data has some similar characteristics with text strings. Therefore, we hypothesise that regular expression algorithms for strings can be adapted for XWraps action level data. Nevertheless, XWraps action level data also has some unique characteristics which make string regular expression matching algorithms difficult to apply. Therefore, we investigate the possibility of applying some existing regular expression matching algorithms to XWraps action level data and compare them to find the most suitable algorithm for XWraps action level data.

1.3 Regular Expressions

The set of regular expressions over an alphabet Σ are defined recursively as follows.

1. A character $x \in \Sigma \cup \{\varepsilon\}$ is a regular expression
2. If S and T are regular expressions, then so is the concatenation, $(S).(T)$, the union, $(S)|(T)$, and the Kleene closure, $(S)^*$. The concatenation symbol (.) is often omitted. For example, ST is equivalent to $S \,.\, T$.

The operations in the regular expressions are defined as the following.

1. Union (|) denotes alternation. For example, "(a | e)" matches both a and e so that "gr(a|e)y" can match "gray" and "grey".
2. Concatenation (.) combines two regular expressions together. For example, concatenating "bo" and "(x | b)" will result a regular expression that matches "box" and "bob".

3. Kleene Closure (*) is a quantifier that indicates there are 0, 1 or more of the previous expression. For example, "gr(ale)*y" matches "gray", "grey", "gry", "graey", "greay", "graaaaaaay", "graaaeeeaeaeaeaeaeaey", etc.

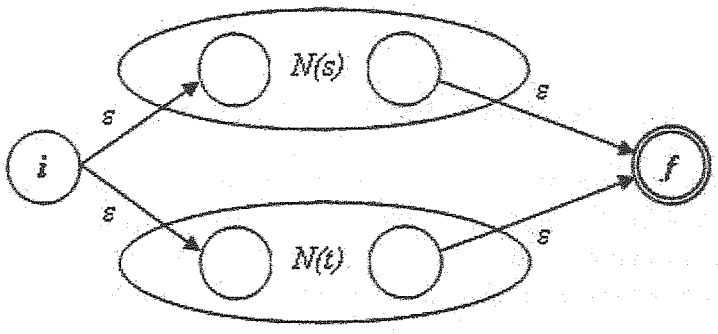
In this project, we adapt two algorithms that are most promising: Wu-Manber's algorithm and Navarro-Raffinot's algorithm. However, we only present Wu and Manber's algorithm in this paper because of the limitted space available.

2.0 Wu and Manber's Algorithm [4]

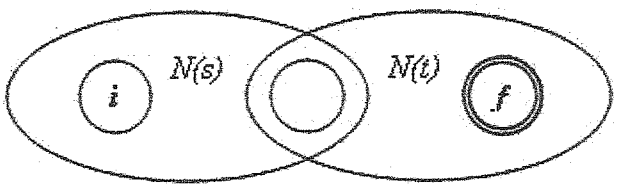
Wu and Manber's algorithm uses Thompson's Nondeterministic Finite Automata (NFA) to recognise the regular expressions. Thompson's NFA can be constructed in the following way [3]. The ellipse represents the automata of a sub-expression. There are two circles in every ellipse, which represent the start state and the final state of the automata, respectively.

1. For ε and a, we build the NFA in the following way

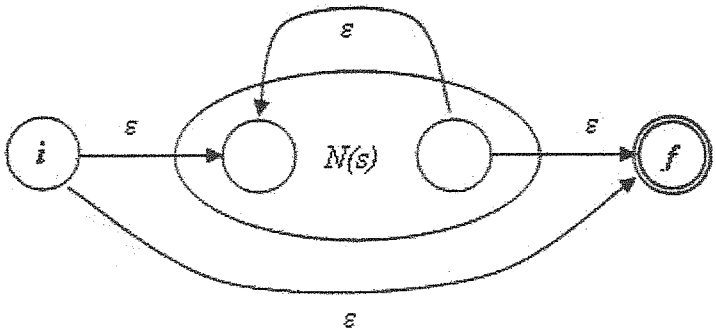
2. For regular expressions *S* and *T*, the automaton of *S*|*T* is built in the following way



3. For regular expressions *S* and *T*, the automaton of *S*.*T* is built in the following way



4. For regular expressions *S*, the automaton *S** is built in the following way



Thompson's NFA has the following properties.

1. It has a unique start and final state.
2. There is at most one non-ε transition can leave or arrive at any state.
3. The start state has no incoming transition.

4. The final state has no outgoing transition.
5. At most two transitions leave from and arrive at each state.
6. For regular expression of length *m* (including operators), there are at most *2m* states.
7. All the non-ε transitions are forward moves.
8. The ε moves can be backward moves and may "jump" arbitrarily.

Once we have constructed the NFA, we label the states with numbers so that all the non-ε transitions from state *i* are always forward moves to state *i+1*. An example of such numbering scheme is shown in Figure 1.

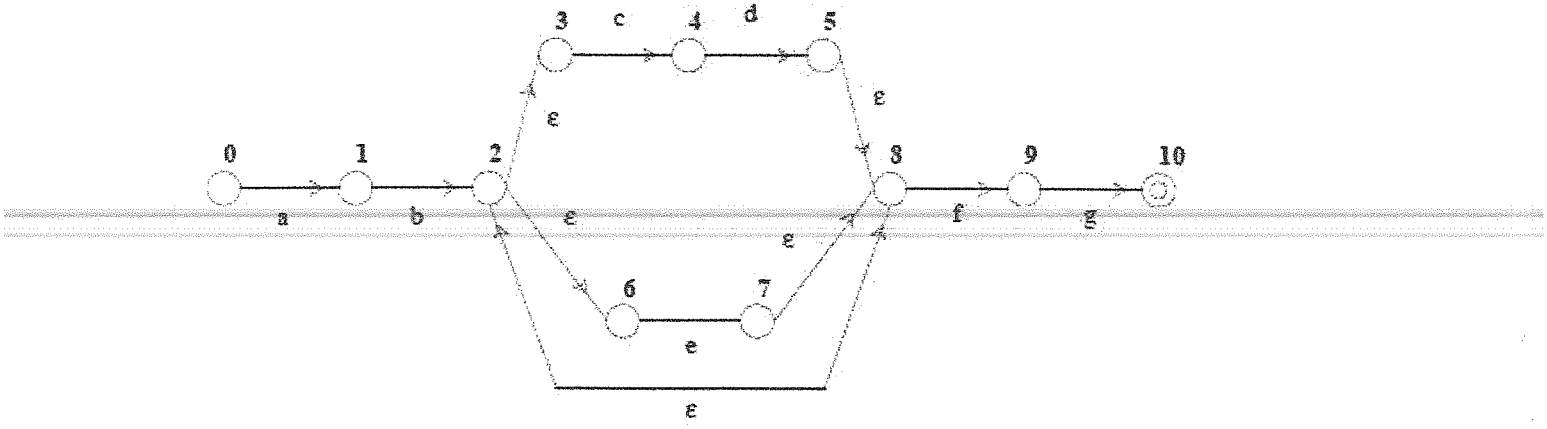


Figure 1 Thompon's NFA corresponding to ab(cdle)*fg

Using the labelled NFA, we can now represent the NFA using bit masks. The *i*-th bit of the bit mask represents the *i*-th state of the NFA. We use the following variables to represent the NFA.

1. **ε-move[*x*]**. This variable shows all the states reachable from state *x* by ε-transitions.
2. **ε-states**. This bit masks indicates the states that have outgoing ε-transitions.
3. **B[σ]**. For each character σ in the alphabet, the *i*-th bit of B[σ] is set to one if σ belongs to the *i*-th state of the NFA.

2.1.1. Simulating Non-ε Transitions

Because all the non-ε transitions are always forward move to the next state, the algorithm for exact string matching by Baeza-Yates and Gonnet [1] can be used. Simulating a transition by a character in the automata can be done by performing *left-shift* to the previous mask (setting the new bit to one), and then performing AND operation with the bit mask of the character. Figure 3 shows an example of running the algorithm on the automata shown in Figure 2.

$$S_{i+1} = (S_i << 1) \ \& \ B[x]$$

Figure 2 The automata representing "aabac".

	a	a	b	a	a	c	a	a	b	a	c	a	b
a	1	1	0	1	1	0	1	1	0	1	0	1	0
a	0	1	0	0	1	0	0	1	0	0	0	0	0
b	0	0	1	0	0	0	0	0	1	0	0	0	0
a	0	0	0	1	0	0	0	0	0	1	0	0	0
c	0	0	0	0	0	0	0	0	0	0	1	0	0

a.

a	b	c
1	0	0
1	0	0
0	1	0
1	0	0
0	0	1

b.

Figure 3 An example of exact matching on pattern "aabac" and the corresponding masks.

2.1.2 Simulating the ε Transitions

An ε transition from state *i* to state *j* means that if we reach state *i*, we also reach state *j* at the same time. Therefore, for every states that has at least one outgoing ε-transition, we need to calculate all the states that are reachable by ε moves. Given a set of states *S* that are currently active, simulating ε-transitions to each active state in the set will give us a new set of active states *S'* where $S \subset S'$. In order to do it quickly, a table *E* is pre-computed. Table *E* maps every possible set of active states to the new set of active states that can be reached by performing ε-moves. Therefore, the ε-transitions can be simulated on *S* by looking up the corresponding value of *S* in the *E* table.

Combining the non-ε and ε moves, the transition by a character *x* can be performed by:

$$S' = E[S << 1 \ \& \ B[x]]$$

If the left shift operation, AND operation, and accessing table *E* can be performed in constant time, the complexity of simulating the transition is also of constant time complexity. Therefore, this algorithm has *O(n)* search time, where *n* represents the size of the text.

3.0 Adapting the Algorithm to XWraps Action Level Data

Each frame of the action level data records the changes happening between two subsequent screenshots taken by XWraps. There is a particular time interval (usually one second) between the screenshots taken by XWraps. Within that interval, it is very probable that more than one action has happened. Therefore, a frame in the action level data may contain multiple actions. Such frames are said to be non-linear.

3.1 Non-linearity of XWraps Action Level Data

The non-linearity problem is described as follows. In order for a regular expression to match the data, it needs to match only one of the characters in the frames (if the frame contains more than one character). Consider the sample data in Figure 4. In that case, pattern "12367", "12467", and "12567" are all matches to the data.

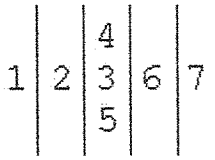


Figure 4 An example of non-linear data.

Making use of the bit-parallelism, each time a non-linear frame is read, we perform OR operation on all the bit masks of the characters in the frame. If *f* is a non-linear frame containing the characters $f_1, f_2, \ldots, f_i$, the NFA transition reading the frame *f* can be simulated as follows.

$$S' = E[\ (S << 1)\ \&\ M\ ]$$
where $M = (B[f_1] \mid B[f_2] \mid \ldots \mid B[f_i])$

3.2 Nondeterminic Ordering of Actions within a Single Frame

As have been mentioned before, there can be more than one action happening between two subsequent screenshots taken by XWraps. The problem is that the correct ordering of those actions is not known. It is the client's requirement that the regular expression search must be able to match every possible ordering of actions in such case. Therefore, every permutation of possible action sequence within non-linear frames should be checked.

All the possible permutations can be simulated by reading the same frame repeatedly until no new active state is found. Every time the same frame is read, we perform OR operation between the current set of active states and the new set of active states obtained by simulating the same frame.

$$S' = S \mid E[\ (S << 1)\ \&\ M\ ]\ \textbf{(until S' = S)}$$

When $S' = S$, the new set of active states is the set of active states after performing all the possible permutations of the actions in the non-linear frame. The advantage of doing this is that no backtracking is required so that the time complexity is linear.

By performing the above adaptations, we have our search algorithm as shown in Figure 5. Preprocessing time is the time required to build the automaton. Most of the preprocessing time is spent on precomputing the table E because of its large size.

4.0 Refining Thompson's NFA Construction

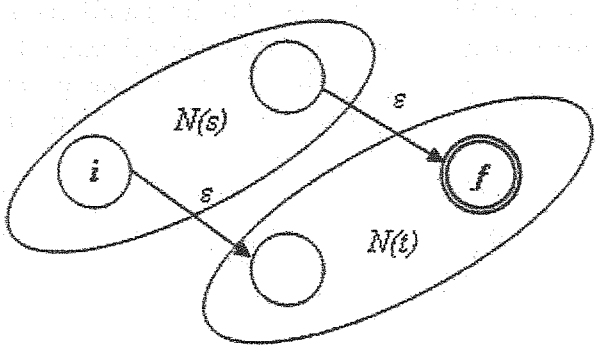
This algorithm relies only on the properties number 1, 2, 3, 4, 7, and 8 of Thompson's NFA so that the NFA contains some redundant states. We propose a way to refine Thompson's NFA construction algorithm so that it produces an NFA with smaller number of states while still preserving the properties needed for Wu and Manber's algorithm.

```
WuManberSearch (RE, T = t1 t2 ... tn)
1.    Preprocessing
2.        (vRE, m) ← Parse(RE)                  /* parse the regular expression */
3.        m' ← Thompson_variables(vRE)          /* build the variables of the Thompson's NFA*/
4.        E ← BuildE(ε-move, m')                /* m' is the number of states */
5     Searching
6         D ← E[1] /*the initial state */
7         For j ← 1 ... n Do
8             If tj non-linear                  /* if the frame is not linear */
9                 For k ∈ tj Do mask ← mask | B[tj[k]]
10                D ← E[D << 1 & mask]
11                While D' = D Do               /* checking the permutations */
12                    D' ← D
13                    D ← D | E[D << 1 & mask]
14                End of while
15            Else
16                D ← E[D << 1 & B[tj]]         /* simulate the transition */
17            End of if
18            If D & 10^(m-1) Then report an occurrence ending at j
19        End of for
```

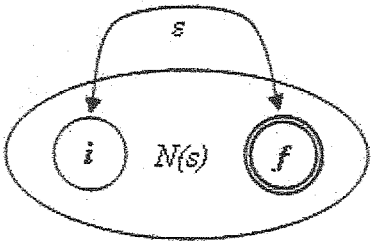
Figure 5 The pseudocode of our adaptation to Wu and Manber's algorithm

The following is our modification to Thompson's NFA construction.

1. For regular expressions S and T, the automaton of $S \mid T$ is built in the following way.



2. For regular expression S, the automaton of S^* is built in the following way.



Our refined NFA is guaranteed to have less or same number of states compared to the original Thompson's NFA. It has the advantage of having between $m+1$ and $2m$ states for m representing the size of the regular expression **without** counting the operator symbols. Reduction in the total number of states is advantegous because the size of table E becomes smaller so that the preprocessing time is quicker.

5.0 Horizontal Table Splitting

The space required by table E may be too large as it may require up to $m'2^{2m'+1}$ bits in the worst case where m' represents the number of states in the NFA. In order to reduce the space requirement of E, we split it horizontally using the method described by Navarro and Raffinot in [2].

We can split E into k tables so that $E = E_1 : \ldots : E_k$. Each subtable addresses roughly $(m'+1)/k$ bits. Therefore, E_i maps the bits from $\lfloor (i-1)(m'+1)/k \rfloor$ to $\lfloor i(m'+1)/k-1 \rfloor$. The full E table can be accessed by:

$$E[S_1S_2 \ldots S_k] = E_1[S_1] \mid E_2[S_2] \mid \ldots \mid E_k[S_k].$$

The total space for table E is now $O(2^{m'/k}m'k)$ bits and the cost of accessing the table is $O(k)$ so that the time for searching text of size n becomes $O(kn)$.

6.0 Conclusions and Further Work

We have shown that we can search for user activities in XWraps data by firstly translating the raw data into XWraps action level data, expressing the activities in patterns of action level data, and then performing search of the patterns in the action level data. To express patterns of action level data, we chose regular expressions because of their flexibility. Since the action level data has some similar characteristics with text strings, we hypothesised that a regular expression searching algorithm for text strings can be applied to XWraps action level data. However, action level data also has some unique characteristics so that some modifications are required so that the algorithms for text strings can be applied. We have shown a way to adapt a searching algorithm proposed by Wu and Manber. We use the bit-parallelism feature of the algorithm to handle the non-linear characteristics of XWraps action level data efficiently. We also provide an improvement to the algorithm by refining the NFA construction process. With smaller number of states, the preprocessing time is quicker because the size of the table to be precomputed is smaller.

Another feature that the client desires is to be able to express temporal constraints in the regular expression. For example, pattern e1{e2 e3* e4}:5 means searching for e1, followed by pattern e2 e3* e4 which must occur within 5 seconds. Further research needs to be done to implement this feature efficiently.

7.0 References

[1] Baeza-Yates, R., and Gonnet, G. H 1992, 'A New Approach to Text Searching', *Communications of the ACM*, vol. 35, no. 10, pp. 74–82.

[2] Navarro, G., and Raffinot, M 2004, 'New Techniques for Regular Expression Searching', *Algorithmica*, vol. 41, no. 2, pp. 89–116.

[3] Thompson, K 1968, 'Regular Expression Search Algorithm', *Communications of the ACM*, vol. 11, vol. 6, pp. 419–422.

[4] Wu, S., and Manber, U 1992, 'Fast Text Searching Allowing Errors', *Communications of the ACM*, vol. 35, vol. 10, pp. 83–91.