Mechanization of a Proof of String-Preprocessing in Boyer-Moore’s Pattern Matching Algorithm

Milos Besta
(Joint work with Frank Stomp)
Department of Computer Science
Wayne State University
Detroit, MI 48201, USA
{besta, fstomp}@cs.wayne.edu
Outline

- Motivation
- String-Preprocessing Algorithm
- Correctness Proof of the Algorithm
- Mechanization of the Correctness Proof
- Conclusions
Motivation
Boyer-Moore’s Pattern Matching Algorithm

- One of the fastest pattern matching algorithms
- Pattern matched from right to left
- Pattern shifted by a large number of positions
- Shifts-preprocessing based on pattern only
  - Single character heuristics
  - Substring heuristics
Motivation
Substring Heuristics

text

next test mostly fails ...

pattern

test rest

shift
test rest

shift
test rest
Motivation
String Preprocessing

- Mismatch may occur at any position in the pattern
- For every mismatched position the shift can be pre-computed independently of the text
- A shift is the distance of the rightmost plausible reoccurrence of a matched substring in the pattern
- Expression \( \text{Match}(p, i) \) denotes the shift of pattern \( p \) if a mismatch occurs at position \( i \)
String Preprocessing Algorithm

Complete Reoccurrence

\[ \text{Match}(p, i) = \max \{ r \mid (m - i < r < m \land p[i] \neq p[r - m + 1]) \land \forall j.(i + 1 \leq j \leq m \Rightarrow p[j] = p[r - m + j]) \land (0 \leq r \leq m - i) \land \forall j.(1 \leq j \leq r \Rightarrow p[j] = p[m - r + j]) \} \]

- Whole matched substring reoccurs in the pattern
- The preceding characters are different
String Preprocessing Algorithm
Partial Reoccurrence

LongJump\((p, i)\) = \(2^*m - i - \text{Match}(p, i)\)

\[
\text{Match}(p, i) = \max\{r \mid (m - i < r < m \land p[i] \neq p[r - m + 1] \land \forall j. (i + 1 \leq j \leq m \Rightarrow p[j] = p[r - m + j])) \\
\lor (0 \leq r \leq m - i \land \forall j. (1 \leq j \leq r \Rightarrow p[j] = p[m - r + j]))\}
\]

- Part of the matched substring reoccurs from the beginning of the pattern
String-Preprocessing Algorithm

The Program

\[
\begin{align*}
\ell_0: & \quad k = m; \\
\ell_1: & \quad q = m + 1; \\
\ell_2: & \quad \textbf{while } k > 0 \\
& \quad \textbf{do } \ell_3: \quad \textbf{back}[k] = q; \\
& \quad \textbf{do } \ell_4: \quad \textbf{while } q \leq m \land p[k] \neq p[q] \\
& \quad \quad \textbf{do } \ell_5: \quad \textbf{MatchJump}[q] = \min(\textbf{MatchJump}[q], m - k); \\
& \quad \quad \ell_6: \quad q = \textbf{back}[q] \\
& \quad \textbf{od}; \\
& \quad \ell_7: \quad k = k - 1; \\
& \quad \ell_8: \quad q = q - 1 \\
& \textbf{od}; \\
\ell_9: & \quad k = 1; \\
\text{Continues on the right}
\end{align*}
\]

\[
\begin{align*}
\ell_{10}: & \quad \textbf{while } k \leq q \\
& \quad \textbf{do } \ell_{11}: \quad \textbf{MatchJump}[k] = \min(\textbf{MatchJump}[k], m + q - k); \\
& \quad \ell_{12}: \quad k = k + 1 \\
& \textbf{od}; \\
\ell_{13}: & \quad \textbf{qq} = \textbf{back}[q]; \\
\ell_{14}: & \quad \textbf{while } q < m \\
& \quad \textbf{do } \ell_{15}: \quad \textbf{while } q \leq \textbf{qq} \\
& \quad \quad \textbf{do } \ell_{16}: \quad \textbf{MatchJump}[q] = \min(\textbf{MatchJump}[q], \textbf{qq} - q + m); \\
& \quad \quad \ell_{17}: \quad q = q + 1 \\
& \quad \textbf{od}; \\
& \quad \ell_{18}: \quad \textbf{qq} = \textbf{back}[\textbf{qq}] \\
& \textbf{od}
\end{align*}
\]
Although the program looks simple, what the program does is not obvious.

String-preprocessing uses dynamic programming techniques.

The program computes the values of

$$LongJump(p, i) = 2m - i - Match(p, i)$$

Those values are stored in array $MatchJump$

Is this really what the program computes?
Correctness Proof
Motivation

- Does the program compute correct values?
  - Since 1977 several errors were found
  - Errors were found also in corrected versions
  - Recently a minor error was found and corrected in the algorithm

- Is the suggested code correction correct?

- Does the correction introduce new errors?

- Is the algorithm correct or does the algorithm contain undiscovered errors?
Correctness Proof
Preliminaries

- A correctness proof can give the ultimate answer as to whether a program is correct or not
- Linear Time Temporal Logic (LTL) is a formalism well suited for reasoning about sequential and parallel programs
- Eventuality properties can be expressed by the eventually (◊) operator in LTL
- Program invariants can be expressed by the always (□) operator in LTL
Correctness Proof
Program Specification

- Program specification must be given
- Proof that the specification is satisfied by the program must be developed

- Initial Condition
  - $|p| = m \land m > 0 \land \text{MatchJump}[i] = 2*m - i$

- Program Specification
  - $p = P \Rightarrow \Box(p = P)$
  - $\Diamond \ell_{19}$
  - $\Box(\ell_{19} \Rightarrow \forall i.(1 \leq i \leq m \Rightarrow \text{MatchJump}[i] = \text{LongJump}(p, i)))$
Mechanization of the Proof

Motivation

- The correctness proof was developed using pencil and paper
- The proof comprises 17 lemmata and theorems
- Many lemmata consist of several properties
- Is the complicated proof valid?
- Does the program satisfy the specification?
- Mechanical verification of the proof can answer these questions
Mechanization of the Proof

Description

- We have finished mechanization of all lemmata and theorems of the pencil-and-paper proof using the PVS system.
- Here, we focus on mechanization of program invariants.
- We did find some errors and omission in the pencil-and-paper proof.
- None of the errors was so serious that it would invalidate the whole proof.
Mechanization of the Proof
Program Representation in PVS

Original program code

\[
\ell_{10}: \textbf{while } k \leq q \\
\textbf{do } \ell_{11}: \text{MatchJump}[k] = \text{min}(\text{MatchJump}[k], m + q - k) \\
\ell_{12}: k = k + 1 \\
\textbf{od};
\]

UNITY like representation in PVS

guard_a14(s: state): bool = s`loc = L11 \\
\text{AND s`k > 0 AND s`k }\leq M \text{ AND s`k }\leq s`m + s`q \\
body_a14: state_transition = (lambda(s: state):
\text{IF guard_a14(s) THEN} \\
s WITH [`loc := L12, `MatchJump :=
\text{update(s`MatchJump, s`k,}
\text{min(s`MatchJump(s`k), s`m + s`q - s`k))}]
\text{ELSE s ENDIF}')
Mechanization of the Proof

Omissions in the Proof

- Expressions used in the hand-written proof were formulated in PVS
- Minor omissions were found and corrected
- Omissions are common in hand-written proofs

\[
\begin{array}{ccccccc}
1 & k + 1 & m - r + k & q & r + 1 & m \\
\end{array}
\]

RightReocc(q, k) =
\[
\min \{ r \mid (q < r \leq m \land k \leq q \land \forall j. (r + 1 \leq j \leq m \Rightarrow p[j] = p[j - r + k])) \lor r = m + 1 \}
\]
Mechanization of the Proof

Invariants of the Program

- Program properties described as invariants
- Invariants must be strong enough to prove the specification

\[ \ell_{10}: \quad \text{while} \ k \leq q \]
\[ \quad \text{do} \quad \ell_{11}: \quad \text{MatchJump}[k] = \min(\text{MatchJump}[k], m + q - k); \]
\[ \quad \ell_{12}: \quad k = k + 1 \]
\[ \quad \text{od}; \]

Lemma 12

\[ \square((\bigvee_{i=9}^{13} \ell_i) \Rightarrow (q = \text{RightReocc}(0, 0) \land \text{RightMatch}(0))) \]
\[ \square((\ell_{11} \lor \ell_{12}) \Rightarrow 1 \leq k \leq q) \]
\[ \square((\bigvee_{i=10}^{19} \ell_i) \Rightarrow 1 \leq k \leq q + 1) \]
Mechanization of the Proof
Errors in Program Invariants

- Proof mechanization reveals errors that are common in pencil-and-paper proofs

**Lemma 13: Original invariants**

\[
\square (\ell_{13} \implies (q \leq m \land q < m + 1))
\]
\[
\square (\bigvee_{i=15}^{17} \ell_i \implies (q \leq m \land q < m))
\]

**Lemma 13: Corrected invariants**

\[
\square (\bigvee_{i=9}^{13} \ell_i \implies q \leq m)
\]
\[
\square (\ell_{14} \implies (q = m \lor q = m + 1 \lor q < m))
\]
\[
\square (\bigvee_{i=15}^{18} \ell_i \implies (q \leq m + 1 \land q < m))
\]
Mechanization of the Proof

Use of Proof Strategies

- Mechanization of invariants involves development of proofs that have similar structure
- We have designed 8 strategies to automate proofs
- Strategy ht-grind-cases discharges all trivial branches of an invariant proof

```
(defun grind-ht-cases ()
  (spread (case "i=0 OR i=1 OR … OR i=23")
    (branch (split)
      (try
        (try (grind) (fail) (skip))
        (skip)
      (skip))))
  (then (typepred "i") (grind))))
```
Mechanization of the Proof
Refinement of the Proof

- Hand-written lemmata are often proved in big steps and details are omitted
- When a proof is mechanized, all fine details must be understood
- Sometimes it is discovered that the proof steps were too big and not understandable

Lemma 16
\[ \forall k.1 \leq k \leq m \Rightarrow \Diamond (MatchJump[k] = LongJump(p, k)) \]
Mechanization of the Proof

Results

- Mechanization of the complete hand-written proof including liveness properties was finished using PVS
- Errors and omissions were found in the hand-written proof and corrected
- Finally, the preprocessing algorithm is correct
- PVS proof will be submitted to SRI site
- Detailed results will be presented in an expanded (journal) version of the paper
Conclusions

- A rigorous formal proof justifies correctness of a (complicated) algorithm
- Errors are often found in hand-written proofs
- Mechanization of a hand-written proof validates correctness of that proof
- Mechanization of a proof reveals fine details, omissions, and errors in a proof
- Proof mechanization desirable whenever correctness and reliability are important
References


Milos Besta and Frank Stomp
(C) 2002