Applying a trace-based framework to the Zhu-Takaoka string matcher

Masters exam in computer science
Aarhus University
7. November 2011

Student: Dan Amlund Thomsen – 20040943
Supervisor: Olivier Danvy
External examiner: Peter Sestoft

Keywords

- String matchers
- Trace-based framework

The question

Consider a string matcher, eg, from the handbook chapter, that you haven't considered in your thesis, and give it the treatment of your thesis, the way someone else would do if (s)he had read your thesis and wanted to apply its results.

What conclusions can be drawn from the treatment?

Key phrases of the question

- Haven't considered in your thesis
  - General framework
  - Extendable
- Treatment of your thesis
  - How to apply the framework
- Conclusions from the treatment
  - New data point
  - Confirm the thesis' conclusions

The answer

- Zhu-Takaoka's string-matching algorithm
  - String-matching algorithm
- Add Zhu-Takaoka to the framework
  - Trace
- Identify Zhu-Takaoka's string-matching concepts
  - String-matching concepts
- Build binding-time separated Zhu-Takaoka matcher
  - Partial evaluation
  - KMP Test
- Compare string-matching algorithms
  - Evolutionary tree over matchers

My thesis

My thesis is that trace-based frameworks make it possible to compare string matchers, and that this comparison reveals new methods of investigating, understanding and building string-matching algorithms.

- Reveal the concepts of Zhu-Takaoka
- Design Zhu-Takaoka matchers
- Compare Zhu-Takaoka with other string-matching algorithms

Background of my thesis (1/2)

- Motivation: Compare string matchers
- Verify the KMP test
- Ager, Danvy, Rohde 02 used a formal proof
- Not practical
- Rohde automated the negative proof
- Compare traces on a set of inputs

Background of my thesis (2/2)

- Measuring the Propagation of Information in Partial Evaluation
  Henning Rohde 2005
  - Compare string matchers
  - Compose concepts into string matchers
  - Examine the literature
- My work
  - Released framework as an easy-to-use tool
  - Comparison methods
    - Evolutionary tree
  - Overview over string matchers

Plan

- Zhu-Takaoka's string-matching algorithm
- Add Zhu-Takaoka to the framework
- Identify Zhu-Takaoka's string-matching concepts
- Build a binding-time separated Zhu-Takaoka matcher
- Compare Zhu-Takaoka with other algorithms
- Conclusion

String-matching algorithms

- String-matching algorithms find the first occurrence of a string (the pattern) in another string (the text)
- String matchers are implementations of string-matching algorithms
- Naive algorithm
  - Left to right
  - Checks everything
  - $O(n^2)$

```
txt abbaba
pat aba
    ==!
     aba
     !
      aba
      !
       aba
       ===
```

Boyer-Moore and Zhu-Takaoka

- Boyer-Moore
  - Right to left
  - Average sub-linear
  - Good-suffix
  - Bad character shift heuristic

```
txt abbaba
pat aba
      !
     aba
     !==
       aba
       ===
```

- Zhu-Takaoka
  - Two-characters wide
    bad character shift heuristics

```
txt abbaba
pat aba
     !!
       aba
       ===
```

Plan

- Zhu-Takaoka's string-matching algorithm ✓
- Add Zhu-Takaoka to the framework
- Identify Zhu-Takaoka's string-matching concepts
- Build a binding-time separated Zhu-Takaoka matcher
- Compare Zhu-Takaoka with other algorithms
- Conclusion

Add Zhu-Takaoka to the framework

- Motivation: Something to compare with
- Trace
- Tracing a Zhu-Takaoka matcher

Trace

- A trace is the sequence of text indices compared when a matcher searches for a pattern in a text

```
idx 012345
txt abbaba
pat aba
      !
     aba
     !==
       aba
       ===
```

Boyer-Moore example.

Trace: 2, 3, 2, 1, 5, 4, 3

- Matchers are trace-equivalent if they have the same trace on all patterns and texts

Tracing string matchers

- Not trivial
- Traces need to be comparable
- Assumptions ensure different implementations use uniform tracing
  - Stop after first occurrence
  - Must be at least one occurrence
  - Pattern lengths at least 2
  - No duplicate indices in the same matching phase

Zhu-Takaoka C matcher

```
j = 0;
while (j <= n - m) {
  i = m - 1;
  while (i < m && x[i] == y[i + j])
    --i;
  if (i < 0) {
    OUTPUT(j);
  } else
    j += MAX(bmGs[i],
             ztBc[y[j + m - 2]]
                 [y[j + m - 1]]);
 }
```

Traced Zhu-Takaoka C matcher

```
j = 0;
while (j <= n - m) {
  start_pruning_duplicates();
  i = m - 1;
  while (i < m && i >= 0 &&
         x[i] == trace_get(y, i + j))
    --i;
  if (i < 0) {
    OUTPUT(j);
  } else
    j += MAX(bmGs[i],
             ztBc[trace_get(y, j + m - 2)]
                 [trace_get(y, j + m - 1)]);
  stop_pruning_duplicates();
```

Plan

- Zhu-Takaoka's string-matching algorithm ✓
- Add Zhu-Takaoka to the framework ✓
- Identify Zhu-Takaoka's string-matching concepts
- Build a binding-time separated Zhu-Takaoka matcher
- Compare Zhu-Takaoka with other algorithms
- Conclusion

Identify Zhu-Takaoka's concepts

- Motivation: Understand Zhu-Takaoka
- String-matching concepts
- Expand the framework
- Composing concepts to build a Zhu-Takaoka matcher
- Comparing the composed Zhu-Takaoka and Boyer-Moore matchers

String-matching concepts

- Traversal order
  - Left to right
  - Right to left
- Positive/negative information
  - Boyer-Moore's good-suffix
- Bad character shift heuristics
  - Two characters

Naive algorithm

```
txt abbaba
pat aba
    ==!
     aba
     !
      aba
      !
       aba
       ===
```

Zhu-Takaoka

```
txt abbaba
pat aba
     !!
       aba
       ===
```

Expand the framework

- Two-character bad character shift heuristics
- Ability to match all indices despite mismatches

```
(define (match-general orderer pruner is-table stop-on-mismatch)
  ...
  (if stop-on-mismatch
      (list #f cache' trace' i)
      (walk (cdr pat-indices) cache'
            trace'))
  ...)

(define (match-table-full orderer pruner)
  (match-general orderer pruner #t #f))

(define (match-table-full-shifts orderer pruner)
  (match-shifts (match-table-full orderer pruner) pruner))
```

Composing Zhu-Takaoka

- Guess concepts from description and implementation
- Use framework to compare with correct Zhu-Takaoka matcher
  - Verify correctness
  - Get counter-example

```
fw_zhu_takaoka is different from fw_boyer-moore
   pattern 'aaa' and text 'aabaaa'
   cl_zhu_takaoka trace: (2 1 5 4 3)
   fw_boyer-moore trace: (2 5 4 3)
```

Composed Zhu-Takaoka

```
(define zhu-takaoka
  (make-matcher
   (match-backtracking
    (match-basic-shifts order-right-to-left
                        (prune-older-than 1))
    (match-table-full-shifts order-last-two
                             (prune-older-than 1)))))


 (define boyer-moore
   (make-matcher
    (match-skew
     (match-basic-shifts order-right-to-left
                         (prune-older-than 1))
     (match-table-shifts order-last-only
                         (prune-older-than 1)))))
```

Plan

- Zhu-Takaoka's string-matching algorithm ✓
- Add Zhu-Takaoka to the framework ✓
- Identify Zhu-Takaoka's string-matching concepts ✓
- Build a binding-time separated Zhu-Takaoka matcher
- Compare Zhu-Takaoka with other algorithms
- Conclusion

Build a binding-time separated matcher

- Motivation: Apply framework to literature
- Partial evaluation
- KMP Test
- Building a binding-time separated Zhu-Takaoka matcher

Partial Evaluation

- Specialize a program with respect to part of its input in order to improve running time
- KMP Test: Specialize a slow naive matcher into a fast known matcher
- Binding-time separated matcher

Binding-time separated Zhu-Takaoka matcher

- Modified Danvy & Rohde's IPL06 binding-time separated Boyer-Moore matcher

```
(define (compute-offset p t j k)
  (+ (- pl j 1)
     (max (rematch-gs p j (sub1 pl) (- pl 2))
          (let ((last-txt-pat-index (+ pl (- j) k (- 1))))
            (shift p (txt-ref (- last-txt-pat-index 1))
                     (txt-ref last-txt-pat-index))))))
(define (shift p c1 c2)
  (if (equal? c2 (string-ref p 0))
      (min (- pl 1) (rematch p 1 c1 c2))
      (rematch p 1 c1 c2)))
(define (rematch p i c1 c2)
  (if (= i pl)
      i
      (if (and (equal? c1 (string-ref p (- pl i 1)))
               (equal? c2 (string-ref p (- pl i))))
          (- i 1)
          (rematch p (1+ i) c1 c2))))
```

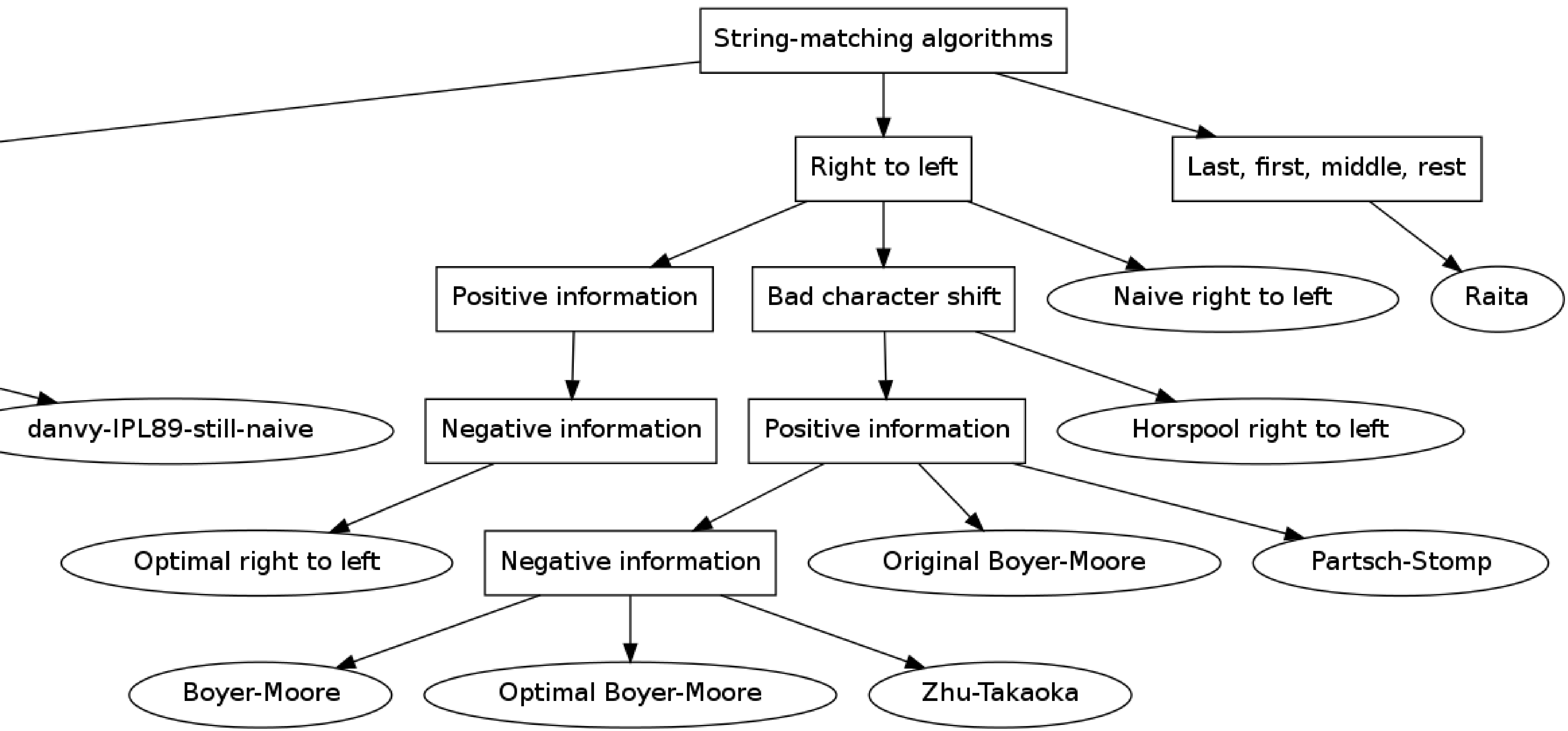
Plan

- Zhu-Takaoka's string-matching algorithm ✓
- Add Zhu-Takaoka to the framework ✓
- Identify Zhu-Takaoka's string-matching concepts ✓
- Build a binding-time separated Zhu-Takaoka matcher ✓
- Compare Zhu-Takaoka with other algorithms
- Conclusion

Compare Zhu-Takaoka with other algorithms

- Motivation: Finding Zhu-Takaoka's place in the world of string-matching algorithms
- Comparing concepts
  - Tree over algorithms grouped by concepts
- Comparing behavior
  - Similarity of matchers
  - Evolutionary tree over matchers

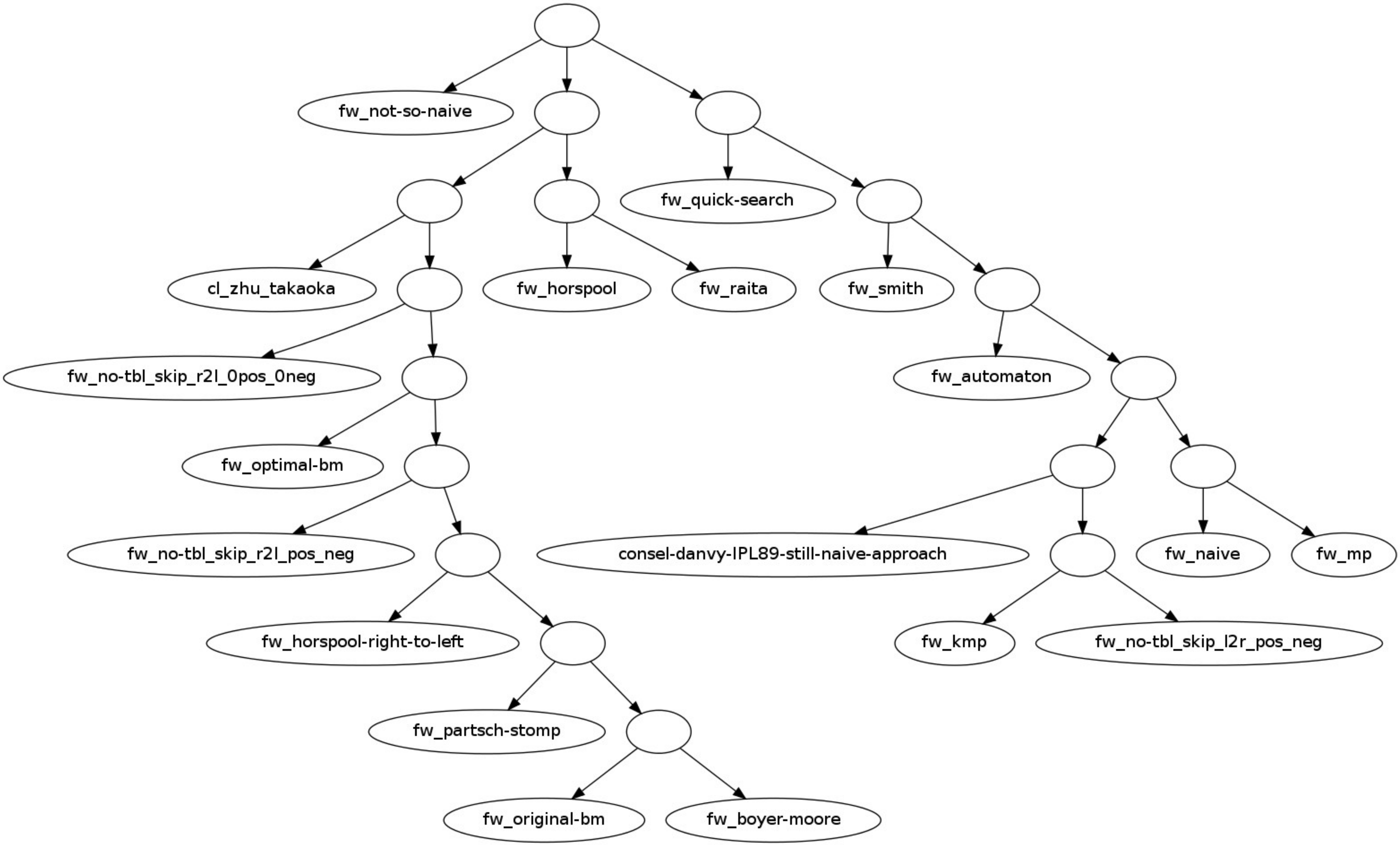
Tree over string-matching concepts



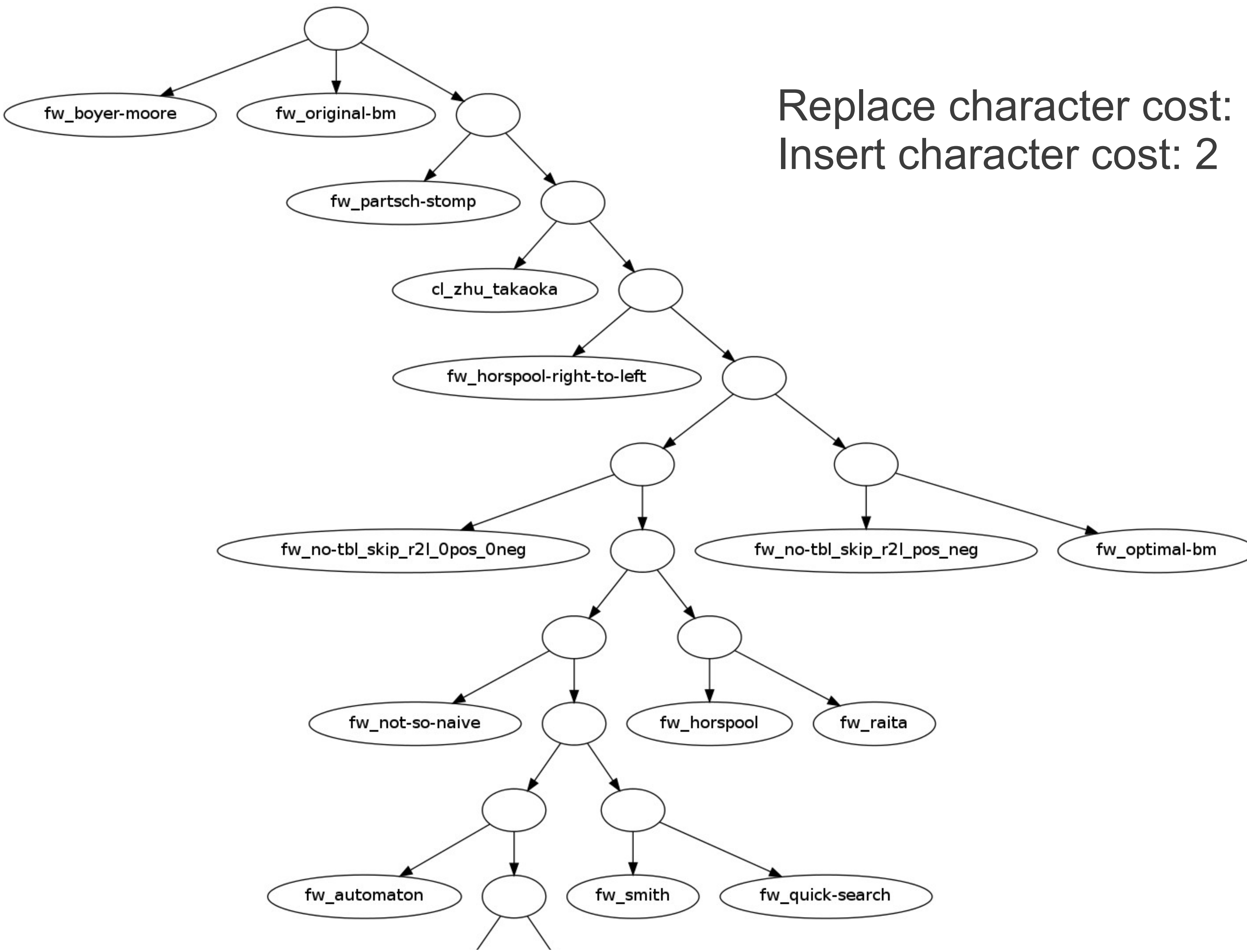
Similarity of matchers

- Similarity of traces over a set of inputs
- Naive: number of equal traces
- Pairwise-alignment: similarity of traces
  - How many insertions and changes of characters are needed to convert one trace into another
- Evolutionary tree
  - Matchers are similar the closer they are in the tree

Naive evolutionary tree



Pairwise-alignment Evolutionary tree



Replace character cost: 5
Insert character cost: 2

Plan

- Zhu-Takaoka's string-matching algorithm ✓
- Add Zhu-Takaoka to the framework ✓
- Identify Zhu-Takaoka's string-matching concepts ✓
- Build a binding-time separated Zhu-Takaoka matcher ✓
- Compare Zhu-Takaoka with other algorithms ✓
- Conclusion

Conclusion (1/3)

- Question: Conclusions from applying the framework to Zhu-Takaoka's algorithm?
- We have
  - Traced a Zhu-Takaoka matcher
  - Identified concepts of Zhu-Takaoka's algorithm
  - Built a binding-time separated Zhu-Takaoka matcher
  - Compared Zhu-Takaoka with other algorithms
- Conclusion
  - The trace-based framework helped me to
    - Investigate
    - Understand
    - Build

Conclusion (2/3)

- Henning Rohde's framework
  - Goal: Help development of partial evaluators
  - Contribution: Fast comparison of string matchers
- My framework
  - Goal: Release the framework as an easy-to-use tool
  - Contribution: Overview over string matchers
- Answer to the question
  - Demonstrates usage of the framework

Conclusion (3/3)

This work have revealed a new approach to understanding string-matching algorithms.

This new approach involves:

- Focusing on string-matching concepts
- Combining concepts in novel ways
- Inventing new concepts
- Understand ideas; not individual algorithms

Thank You