Process-based Aho-Corasick Failure Function Construction

Tinus Strauss[1] Derrick G. Kourie[2,4] Bruce W. Watson[2,4]
Loek Cleophas[2,3]

[1]Department of Computer Science, University of Pretoria, South Africa

[2]Department of Information Science, Stellenbosch University, South Africa

[3]Department of Computer Science, Umeå University, Sweden

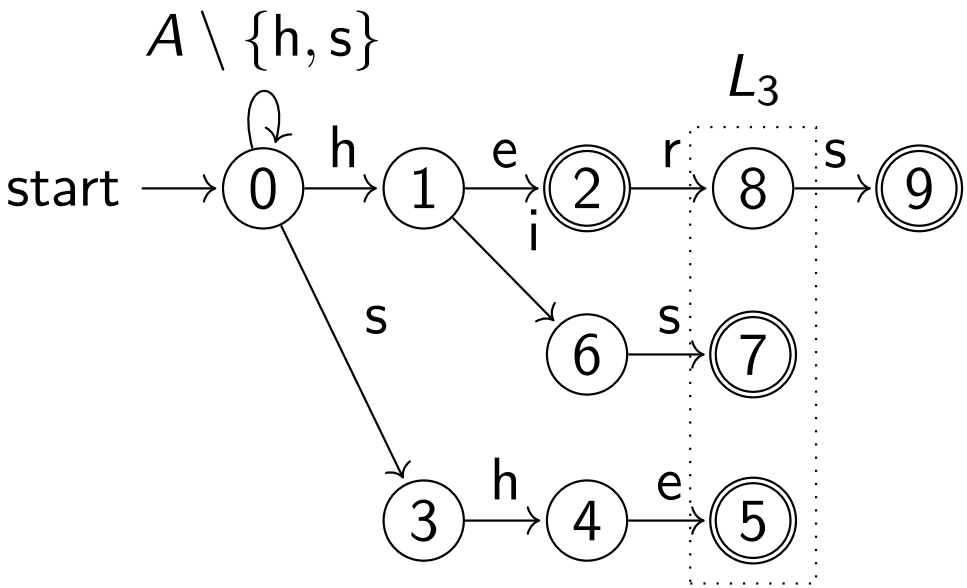
[4]Centre for Artificial Intelligence Research, CSIR Meraka Institute, South Africa

Communicating Process Architectures 2015

The Aho-Corasick algorithm

proc $AC(A, K, T) \rightarrow$
  { Construct automaton. }
  $\langle g, output \rangle := computeG(K)$;
  $\langle f, output \rangle := computeF(A, g, output)$;
  { Use automaton to do matching. }
  $q := 0$;
  for $(i : 0 \,..\, |T| - 1) \rightarrow$
    do $(g(q, T_i) = \text{fail}) \rightarrow q := f(q)$ **od**;
    $q := g(q, T_i)$;
    if $(output(q) = \emptyset) \rightarrow$ **skip**
    $[\!]$ $(output(q) \neq \emptyset) \rightarrow print(\text{`Match ending at '}, i)$;
                $print(output(q))$
    fi
  rof
corp

Trie after *computeG*



q	$output(q)$
2	{he}
5	{she}
7	{his}
9	{hers}

Computing the failure function

func $computeF(A, g, output) \rightarrow$
$queue := \emptyset;$
{ Phase 1: L_1 in queue and $\forall s \in L_1 : f(s) = 0$ }
for each $(a \in A) \rightarrow$
$s := g(0, a);$
if $(s = 0) \rightarrow$ **skip**
$[\!]$ $(s \neq 0) \rightarrow queue.enqueue(s);$
$f(s) := 0$
fi
rof;
{ Phase 2: }
...

Phase 2

func $computeF(A, g, output) \rightarrow$

$\ldots$

{ Phase 2: Determine L_d from L_{d-1}. }

do $(queue \neq \emptyset) \rightarrow$

$r := queue.dequeue();$

for each $(a \in A) \rightarrow$

$s := g(r, a);$

if $(s = \text{fail}) \rightarrow$ **skip**

$[\!]$ $(s \neq \text{fail}) \rightarrow q := f(r);$

do $(g(q, a) = \text{fail})) \rightarrow q := f(q)$ **od**;

$f(s) := g(q, a);$

$queue.enqueue(s);$

$output(s) := output(s) \cup output(f(s))$

fi

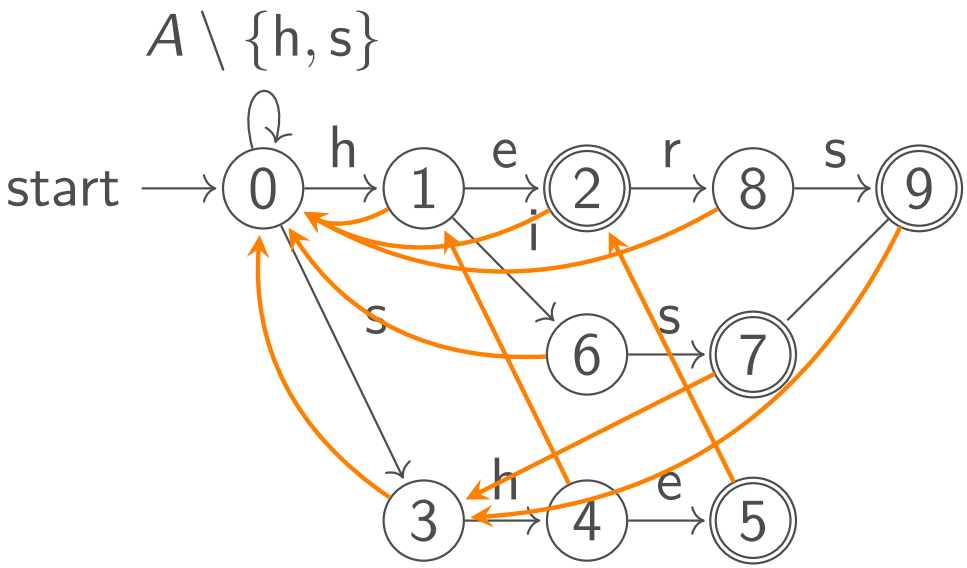
rof

od;

return $\langle f, output \rangle$

cnuf

Trie with failure function after *computeF*



q	$output(q)$
2	{he}
5	{she,he}
7	{his}
9	{hers}

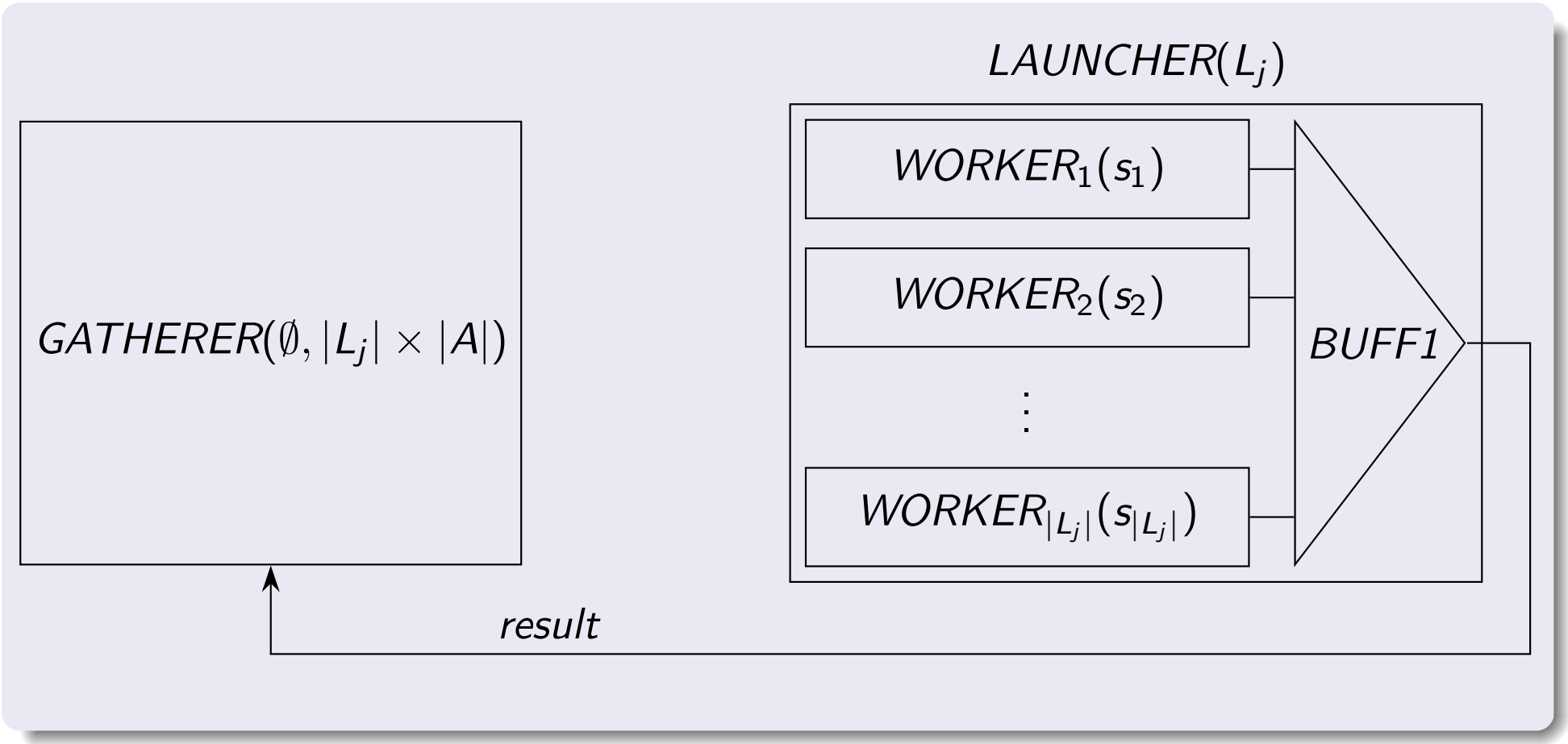
Overview

- Process levels sequentially.
- Within a level, nodes are independent.
- $LAUNCHER(L_1)\,;\, LAUNCHER(L_2)\,;\cdots\,;\, LAUNCHER(L_n)$
- $LAUNCHER(L_d) = |||_{\forall\, s \in L_d}\ WORKER(s)$

- Four variants of Phase 2.
- CSP descriptions.

Variant 1

- Dynamically created processes.
- Communicate next level elements via channel.



Variant 1

$$WORKER_i(s) = P(A, s)$$

$$P(S, s) =$$

if $(S \neq \emptyset)$ then

$$\Box_{a \in S}\, updateF.a.s \rightarrow out.i!g(s, a) \rightarrow P(S \setminus \{a\}, s)$$

else

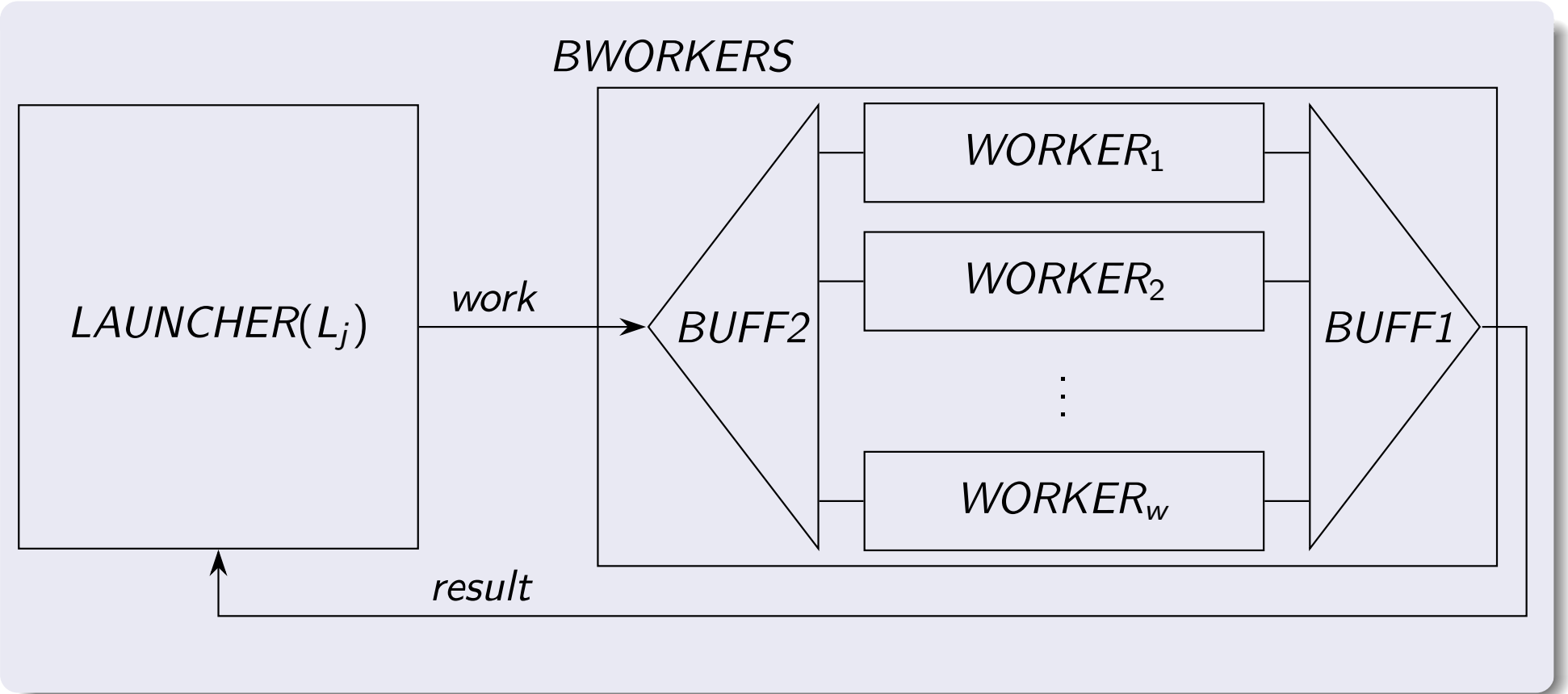
$$SKIP$$

Variant 1

$$
\begin{array}{l}
GATHERER(Q, Cnt) = \\
\quad \text{if}\,(Cnt > 0)\ \text{then} \\
\quad\quad result?r \rightarrow \\
\quad\quad\quad \text{if}\,(r \neq \text{fail})\ \text{then} \\
\quad\quad\quad\quad GATHERER(Q \cup \{r\}, Cnt - 1) \\
\quad\quad\quad \text{else} \\
\quad\quad\quad\quad GATHERER(Q, Cnt - 1) \\
\quad \ldots
\end{array}
$$

Variant 2 to 4

- Fixed number of *WORKER* processes.
- Receive nodes to process from channel.
- Communicate next level elements on channel.



Variant 2 to 4

$$WORKER_i = in.i?s \rightarrow P(A, s) \,;\, WORKER_i$$

$$SENDER(S) =$$
$$\quad \text{if}\,(S \neq \emptyset)\ \text{then}$$
$$\qquad \Box_{a \in S}\, work!a \rightarrow SENDER(S \setminus \{a\})$$
$$\quad \text{else}$$
$$\qquad SKIP$$

$$GATHERER(Q, Cnt) =$$
$$\quad \text{if}\,(Cnt > 0)\ \text{then}$$
$$\qquad result?r \rightarrow$$
$$\qquad\quad \ldots$$

Variant 2 to 4

Variant 2

$$LAUNCHER(L) = SENDER(L)\,;\, GATHERER(\emptyset, |L| \times |A|)$$

Variant 3

$$LAUNCHER(L) = work!a \rightarrow \cdots \Box\ result?r \rightarrow \cdots$$

Variant 4

$$LAUNCHER(L) = SENDER(L) \,|||\, GATHERER(\emptyset, |L| \times |A|)$$

Implementation

- Go programming language.
- `golang.org`
- Language supports channels.
- Synchronisation via channels.
- Concurrent processes implemented as go-routines.
- No buffer processes.

Experiments

- Keyword set sizes: 10, 100, 1000, 10 000, and 100 000 states.
- Keywords
  - Single symbol words (Two symbol alphabet)
  - English words (256 symbol alphabet)
- Go version 1.4.2
- Machine
  - Six-core Intel Xeon 2.6 GHz
  - 16 GB RAM
  - Linux kernel 3.10.17

Speedup?

Type	$\|K\|$	Variant 1	Variant 2	Variant 3	Variant 4
Single Symbol	10	0.18	0.14	0.13	0.10
	100	0.18	0.14	0.13	0.10
	1000	0.20	0.16	0.14	0.11
	10 000	0.57	0.54	0.52	0.46
English Unsorted	10	0.16	0.10	0.12	0.10
	100	0.15	0.15	0.15	0.15
	1000	0.18	0.18	0.18	0.18
	10 000	0.20	0.20	0.11	0.20
	100 000	0.23	0.14	0.12	0.13
English Sorted	10	0.17	0.07	0.09	0.07
	100	0.16	0.14	0.14	0.14
	1000	0.17	0.17	0.17	0.17
	10 000	0.18	0.18	0.11	0.18
	100 000	0.21	0.12	0.11	0.12

Reducing communication (Variant 1 example)

$WORKER_i(s) = P(A, s)$

$P(S, s) =$

if $(S \neq \emptyset)$ then

$\quad \square_{a \in S}\ updateF.a.s \rightarrow out.i!g(s,a) \rightarrow P(S \setminus \{a\}, s)$

else

$\quad SKIP$

$WORKER_i(s) = P(A, s, \emptyset)$

$P(S, s, R) =$

if $(S \neq \emptyset)$ then

$\quad \square_{a \in S}\ updateF.a.s \rightarrow P(S \setminus \{a\}, s, R \cup \{g(s,a)\})$

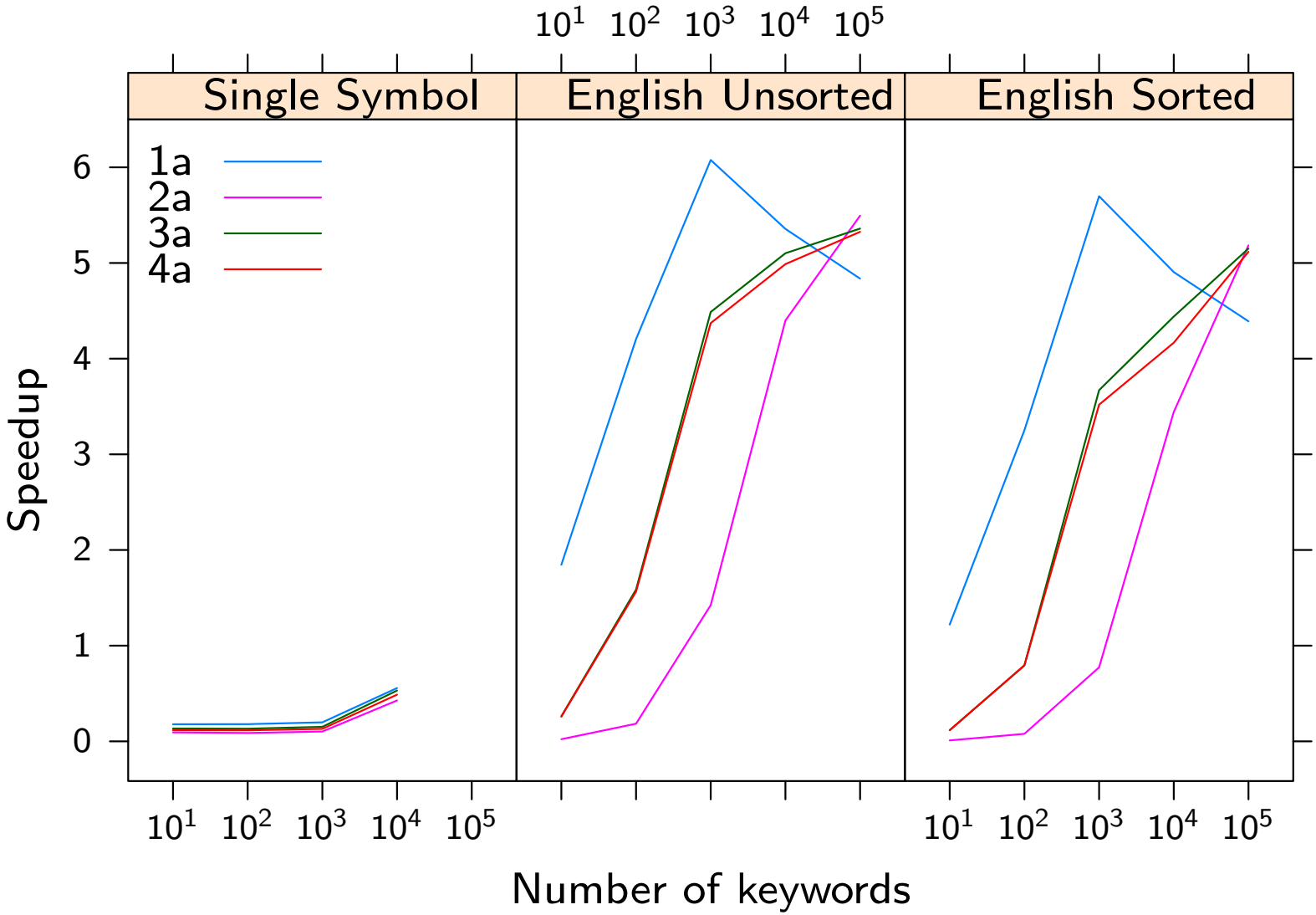
else

$\quad out.i!R \rightarrow SKIP$

Speedup for modified variants

Type	$\|K\|$	Variant 1a	Variant 2a	Variant 3a	Variant 4a
Single Symbol	10	0.18	0.09	0.13	0.12
	100	0.18	0.09	0.13	0.12
	1000	0.20	0.10	0.15	0.13
	10 000	0.56	0.43	0.53	0.49
English Unsorted	10	1.85	0.02	0.26	0.26
	100	4.20	0.18	1.59	1.56
	1000	6.08	1.42	4.49	4.37
	10 000	5.36	4.40	5.10	4.99
	100 000	4.84	5.49	5.36	5.33
English Sorted	10	1.22	0.01	0.12	0.12
	100	3.25	0.08	0.79	0.79
	1000	5.70	0.77	3.67	3.52
	10 000	4.90	3.44	4.44	4.17
	100 000	4.39	5.18	5.15	5.12

Speedup for modified variants



Conclusion

- Presented four process-based decompositions of the failure function construction algorithm.
- Presented the results of an experiment.
- Obtained speedup in some cases.
- Efficiency sometimes low.

- Next steps
  - Try to improve efficiency.
  - Other stringology algorithms such as Hopcroft's DFA minimisation algorithm.