HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI
MATEMAATTIS-LUONNONTIETEELLINEN TIEDEKUNTA
MATEMATISK-NATURVETENSKAPLIGA FAKULTETEN
FACULTY OF SCIENCE

EXPERIMENTAL COMPARISON OF MULTIPLE EXACT STRING MATCHING ALGORITHMS

Paul Saikko

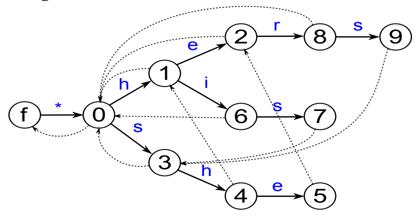
The *multiple exact string matching* problem asks us to search a text T of length n for a set of patterns $\mathcal{P}$ with total length $\|\mathcal{P}\| = m$. Obviously we could search the text for each pattern in P using some exact string matching algorithm, but much better algorithms for this task exist. Here we give an overview of three such algorithms: Aho-Corasick, Shift-And, and Karp-Rabin. Each provides a different approach to solving the problem. We compare the theoretical performance of these algorithms to experimental results and attempt to identify the strenghts and weaknesses of each approach, assuming a constant alphabet size of σ and equal-length patterns, $|P_j| = l$ for every $P_j \in \mathcal{P}$.

THE ALGORITHMS

AHO-CORASICK

The algorithm. We can construct a trie from the patterns $\mathcal{P}$, then for each node, add a link to the node that represents the longest proper suffix of the node and make note of pattern occurrences. The automaton can be then simulated with the text T as input to find occurrences.

Aho-Corasick automaton. For the patterns $\mathcal{P} = \{\text{he}, \text{she}, \text{his}, \text{hers}\}$ we can produce the following automaton:



Time complexity.

- Preprocessing: $O(m)$
- Search: $O(n)$

SHIFT-AND

The algorithm. Shift-And scans the text, it maintaining a bitvector D which keeps track of the longest prefixes of patterns it has encountered.

```
# preprocessing
for (i = 0; i < m; i++):
  B[P[i/l][i%l]] += 2^i
for (i = 0; i < m; i += l):
  pBegin += 2^i
for (i = l - 1; i < m; i += l):
  pEnd += 2^i
# search
for (i = 0; i < n; i++):
  D = ((D << 1) | pBegin) & B[T[i]]
  if D & pEnd ≠ 0: yield i
```

Bitparallelism. The bitvectors D, pBegin, and pEnd can be represented in $\lceil m/w \rceil$ machine words. B can be stored in $\sigma\lceil m/w \rceil$ and D can be updated in $O(\lceil m/w \rceil)$ bitwise operations at each step.

Time complexity.

- Preprocessing: $O(m)$
- Search: $O(n\lceil m/w \rceil)$

KARP-RABIN

Karp-Rabin hash function. For some fixed positive integers r and q, the karp-rabin hash of a string $S = s_0 s_1 \dots s_{m-1}$ is

$$H(S) = \sum_{i=0}^{m-1} (s_i r^{m-1-i}) \mod q.$$

This is an example of a rolling hash function – if we know the hash $H(T_{[i \dots i+m]})$, we can compute $H(T_{[i+1 \dots i+1+m]})$ in constant time.

Multiple string matching. We can precompute the hash value for every pattern and store them in a data structure that supports constant-time lookups. Potential occurrences in the text can be found by computing $H(T_{[i \dots i+l]})$ for each $i \in [0 \dots n-l)$, and comparing them to the precomputed pattern hashes. Every potential occurrence must be checked, which leads to poor worst-case behavior.

Average time complexity.

- Preprocessing: $O(m)$
- Search: $O(n+m)$

EXPERIMENTAL RESULTS

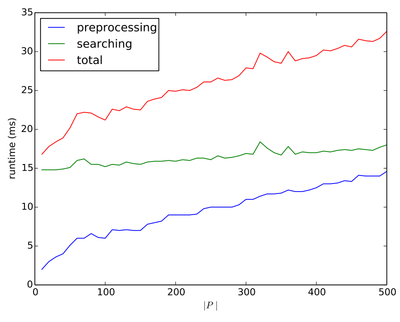
METHODOLOGY

Implementations. The algorithms introduced above were implemented with java version 1.7. The source code for the implementations can be viewed on github at `https://github.com/psaikko/string-algorithms-project`.

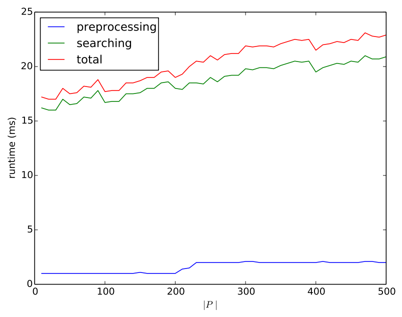
Experiments. The experiment data was gathered using python scripts to automate running the algorithms. Every plotted data point is an average of 10 runs of the algorithm.

COMPARISON WITH THEORETICAL PERFORMANCE

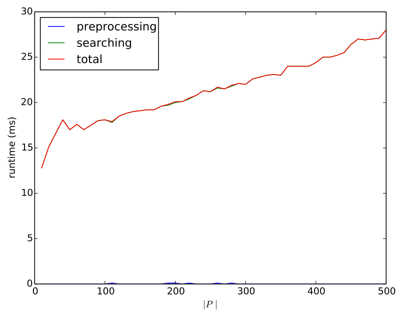
With constant text length. We plot the preprocessing and search times for each algorithm as the number of patterns to search for is increased.



Aho-Corasick. We see search time staying roughly constant as expected, while preprocessing time increases linearly as we add more patterns.

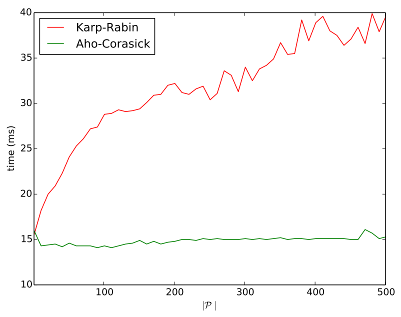


Karp-Rabin. The algorithm generally performs well despite poor worst-case behavior for search, preprocessing is quite fast.

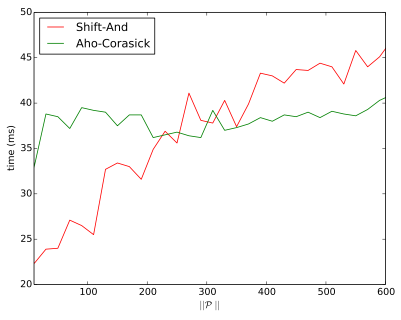


Shift-And. The preprocessing time for Shift-And is much shorter than the other algorithms, despite the same asymptotic time complexity.

EDGE CASES



Many short patterns. A combination of the Karp-Rabin algorithm's $O(n+m)$ time complexity and hash collisions leads it to perform poorly.



Small total pattern length. Although Shift-And scales poorly with total pattern length, the bitparallel algorithm is fast when $\|\mathcal{P}\|$ is not large.