Experimental Results

Register constructions, or related constructions for asynchronous interprocess communication, are used in current hardware and software.

Cross References

- ▶ Asynchronous Consensus Impossibility
- ▶ Atomic Broadcast
- ▶ Causal Order, Logical Clocks, State Machine Replication
- ▶ Concurrent Programming, Mutual Exclusion
- ▶ Linearizability
- ▶ Renaming
- ▶ Self-Stabilization
- ▶ Snapshots in Shared Memory
- ▶ Synchronizers, Spanners
- ▶ Topology Approach in Distributed Computing

Recommended Reading

1. Bloom, B.: Constructing two-writer atomic registers. IEEE Trans. Comput. **37**(12), 1506–1514 (1988)
2. Burns, J.E., Peterson, G.L.: Constructing multi-reader atomic values from non-atomic values. In: Proc. 6th ACM Symp. Principles Distr. Comput., pp. 222–231. Vancouver, 10–12 August 1987
3. Dolev, D., Shavit, N.: Bounded concurrent time-stamp systems are constructible. SIAM J. Comput. **26**(2), 418–455 (1997)
4. Haldar, S., Vitanyi, P.: Bounded concurrent timestamp systems using vector clocks. J. Assoc. Comp. Mach. **49**(1), 101–126 (2002)
5. Israeli, A., Li, M.: Bounded time-stamps. Distribut. Comput. **6**, 205–209 (1993) (Preliminary, more extended, version in: Proc. 28th IEEE Symp. Found. Comput. Sci., pp. 371–382, 1987.)
6. Israeli, A., Shaham, A.: Optimal multi-writer multireader atomic register. In: Proc. 11th ACM Symp. Principles Distr. Comput., pp. 71–82. Vancouver, British Columbia, Canada, 10–12 August 1992
7. Kirousis, L.M., Kranakis, E., Vitányi, P.M.B.: Atomic multireader register. In: Proc. Workshop Distributed Algorithms. Lect Notes Comput Sci, vol 312, pp. 278–296. Springer, Berlin (1987)
8. Lamport, L.: On interprocess communication—Part I: Basic formalism, Part II: Algorithms. Distrib. Comput. **1**(2), 77–101 (1986)
9. Li, M., Tromp, J., Vitányi, P.M.B.: How to share concurrent wait-free variables. J. ACM **43**(4), 723–746 (1996) (Preliminary version: Li, M., Vitányi, P.M.B. A very simple construction for atomic multiwriter register. Tech. Rept. TR-01–87, Computer Science Dept., Harvard University, Nov. 1987)
10. Peterson, G.L.: Concurrent reading while writing. ACM Trans. Program. Lang. Syst. **5**(1), 56–65 (1983)
11. Peterson, G.L., Burns, J.E.: Concurrent reading while writing II: The multiwriter case. In: Proc. 28th IEEE Symp. Found. Comput. Sci., pp. 383–392. Los Angeles, 27–29 October 1987
12. Singh, A.K., Anderson, J.H., Gouda, M.G.: The elusive atomic register. J. ACM **41**(2), 311–339 (1994) (Preliminary version in: Proc. 6th ACM Symp. Principles Distribt. Comput., 1987)
13. Tromp, J.: How to construct an atomic variable. In: Proc. Workshop Distrib. Algorithms. Lecture Notes in Computer Science, vol. 392, pp. 292–302. Springer, Berlin (1989)
14. Vitányi, P.M.B., Awerbuch, B.: Atomic shared register access by asynchronous hardware. In: Proc. 27th IEEE Symp. Found. Comput. Sci. pp. 233–243. Los Angeles, 27–29 October 1987. Errata, Proc. 28th IEEE Symp. Found. Comput. Sci., pp. 487–487. Los Angeles, 27–29 October 1987

Regular Expression Indexing

2002; Chan, Garofalakis, Rastogi

CHEE-YONG CHAN[1], MINOS GAROFALAKIS[2], RAJEEV RASTOGI[3]

[1] Department of Computer Science, National University of Singapore, Singapore, Singapore

[2] Computer Science Division, University of California – Berkeley, Berkeley, CA, USA

[3] Bell Labs, Lucent Technologies, Murray Hill, NJ, USA

Keywords and Synonyms

Regular expression indexing; Regular expression retrieval

Problem Definition

Regular expressions (REs) provide an expressive and powerful formalism for capturing the structure of messages, events, and documents. Consequently, they have been used extensively in the specification of a number of languages for important application domains, including the XPath pattern language for XML documents [6], and the policy language of the *Border Gateway Protocol* (BGP) for propagating routing information between autonomous systems in the Internet [12]. Many of these applications have to manage large databases of RE specifications and need to provide an effective matching mechanism that, given an input string, quickly identifies all the REs in the database that match it. This RE retrieval problem is therefore important for a variety of software components in the middleware and networking infrastructure of the Internet.

The RE retrieval problem can be stated as follows: Given a large set S of REs over an alphabet Σ, where each RE $r \in S$ defines a regular language $L(r)$, construct a data structure on S that efficiently answers the following query: given an arbitrary input string $w \in \Sigma^*$, find the subset S_w of REs in S whose defined regular languages include the string w. More precisely, $r \in S_w$ iff $w \in L(r)$. Since S is a large, dynamic, disk-resident collection of REs, the data structure should be dynamic and provide efficient support of updates (insertions and deletions) to S. Note that this problem is the opposite of the more traditional RE search problem where $S \subseteq \Sigma^*$ is a collection of strings and the task is to efficiently find all strings in S that match an input regular expression.

Notations

An RE r over an alphabet Σ represents a subset of strings in Σ^* (denoted by $L(r)$) that can be defined recursively as follows [9]: (1) the constants ϵ and $\emptyset$ are REs, where $L(\epsilon) = \{\epsilon\}$ and $L(\emptyset) = \emptyset$; (2) for any letter $a \in \Sigma$, a is a RE where $L(a) = \{a\}$; (3) if r_1 and r_2 are REs, then their union, denoted by $r_1 + r_2$, is a RE where $L(r_1 + r_2) = L(r_1) \cup L(r_2)$; (4) if r_1 and r_2 are REs, then their concatenation, denoted by $r_1.r_2$, is a RE where $L(r_1.r_2) = \{s_1 s_2 \mid s_1 \in L(r_1), s_2 \in L(r_2)\}$; (5) if r is a RE, then its closure, denoted by r^*, is a RE where $L(r^*) = L(\epsilon) \cup L(r) \cup L(rr) \cup L(rrr) \cup \cdots$; and (6) if r is a RE, then a parenthesized r, denoted by (r), is a RE where $L((r)) = L(r)$. For example, if $\Sigma = \{a, b, c\}$, then $(a + b).(a + b + c)^*.c$ is a RE representing the set of strings that begins with either a "a" or a "b" and ends with a "c". A string $s \in \Sigma^*$ is said to match a RE r if $s \in L(r)$.

The language $L(r)$ defined by an RE r can be recognized by a *finite automaton (FA) M* that decides if an input string w is in $L(r)$ by reading each letter in w sequentially and updating its current state such that the outcome is determined by the final state reached by M after w has been processed [9]. Thus, M is an FA for r if the language accepted by M, denoted by $L(M)$, is equal to $L(r)$. An FA is classified as a *deterministic finite automaton* (DFA) if its current state is always updated to a single state; otherwise, it is a *non-deterministic finite automaton* (NFA) if its currant state could refer to multiple possible states. The trade off between a DFA and an NFA representations for a RE is that the latter is more space-efficient while the former is more time-efficient for recognizing a matching string by checking a single path of state transitions. Let $|L(M)|$ denote the size of $L(M)$ and $|L_n(M)|$ denote the number of length-n strings in $L(M)$. Given a set $\mathcal{M}$ of finite automata, let $L(\mathcal{M})$ denote the language recognized by the automata in $\mathcal{M}$; i. e., $L(\mathcal{M}) = \bigcup_{M_i \in \mathcal{M}} L(M_i)$.

Key Results

The RE retrieval problem was first studied for a restricted class of REs in the context of content-based dissemination of XML documents using XPath-based subscriptions (e. g., [1,3,7]), where each XPath expression is processed in terms of a collection of path expressions. While the XPath language [6] allows rich patterns with tree structure to be specified, the path expressions that it supports lack the full expressive power of REs (e. g., XPath does not permit the RE operators *, $+$ and $\cdot$ to be arbitrarily nested in path expressions), and thus extending these XML-filtering techniques to handle general REs may not be straightforward. Further, all of the XPath-based methods are designed for indexing main-memory resident data. Another possible approach would be to coalesce the automata for all the REs into a single NFA, and then use this structure to determine the collection of matching REs. It is unclear, however, if the performance of such an approach would be superior to a simple sequential scan over the database of REs; furthermore, it is not easy to see how such a scheme could be adapted for disk-resident RE data sets.

The first disk-based data structure that can handle the storage and retrieval of REs in their full generality is the *RE-tree* [4,5]. Similar to the R-tree [8], an RE-tree is a dynamic, height-balanced, hierarchical index structure, where the leaf nodes contain data entries corresponding to the indexed REs, and the internal nodes contain "directory" entries that point to nodes at the next level of the index. Each leaf node entry is of the form (id, M), where id is the unique identifier of an RE r and M is a finite automaton representing r. Each internal node stores a collection of finite automata; and each node entry is of the form (M, ptr), where M is a finite automaton and ptr is a pointer to some node N (at the next level) such that the following *containment property* is satisfied: If $\mathcal{M}_N$ is the collection of automata contained in node N, then $L(\mathcal{M}_N) \subseteq L(M)$. The automaton M is referred to as the *bounding automaton* for $\mathcal{M}_N$. The containment property is key to improving the search performance of hierarchical index structures like RE-trees: if a query string w is not contained in $L(M)$, then it follows that $w \notin L(M_i)$ for all $M_i \in \mathcal{M}_N$. As a result, the entire subtree rooted at N can be pruned from the search space. Clearly, the closer $L(M)$ is to $L(\mathcal{M}_N)$, the more effective this search-space pruning will be.

In general, there are an infinite number of bounding automata for $\mathcal{M}_N$ with different degrees of precision from the least precise bounding automaton with $L(M) = \Sigma^*$ to the most precise bounding automaton, referred to as the *minimal bounding automaton*, with $L(M) = L(\mathcal{M}_N)$. Since the storage space for an automaton is dependent on its complexity (in terms of the number of its states and transitions), there is a space-precision tradeoff involved in the choice of a bounding automaton for each internal node entry. Thus, even though minimal bounding automata result in the best pruning due to their tightness, it may not be desirable (or even feasible) to always store minimal bounding automata in RE-trees since their space requirement can be too large (possibly exceeding the size of an index node), thus resulting in an index structure with a low fan-out. Therefore, to maintain a reasonable fan-out for RE-trees, a space constraint is imposed on the maximum number of states (denoted by α) permitted for each bounding automaton in internal RE-tree nodes. The automata stored in RE-tree nodes are, in general, NFAs with a minimum number of states. Also, for better space utilization, each individual RE-tree node is required to contain at least m entries. Thus, the RE-tree height is $O(\log_m(|S|))$.

RE-trees are conceptually similar to other hierarchical, spatial index structures, like the R-tree [8] that is designed for indexing a collection of multi-dimensional rectangles, where each internal entry is represented by a minimal bounding rectangle (MBR) that contains all the rectangles in the node pointed to by the entry. RE-tree search simply proceeds top-down along (possibly) multiple paths whose bounding automaton accepts the input string; RE-tree updates try to identify a "good" leaf node for insertion and can lead to node splits (or, node merges for deletions) that can propagate all the way up to the root. There is, however, a fundamental difference between the RE-tree and the R-tree in the indexed data types: regular languages typically represent *infinite* sets with no well-defined notion of spatial locality. This difference mandates the development of novel algorithmic solutions for the core RE-tree operations. To optimize for search performance, the core RE-tree operations are designed to keep each bounding automaton M in every internal node to be as "tight" as possible. Thus, if M is the bounding automaton for $\mathcal{M}_N$, then $L(M)$ should be as close to $L(\mathcal{M}_N)$ as possible.

There are three core operations that need to be addressed in the RE-tree context: (P1) selection of an optimal insertion node, (P2) computing an optimal node split, and (P3) computing an optimal bounding automaton. The goal of (P1) is to choose an insertion path for a new RE that leads to "minimal expansion" in the bounding automaton of each internal node of the insertion path. Thus, given the collection of automata $\mathcal{M}(N)$ in an internal index node N and a new automaton M, an optimal $M_i \in \mathcal{M}(N)$ needs to be chosen to insert M such that $|L(M_i) \cap L(M)|$ is maximum. The goal of (P2), which arises when splitting a set of REs during an RE-tree node-split, is to identify a partitioning that results in the minimal amount of "covered area" in terms of the languages of the resulting partitions. More formally, given the collection of automata $\mathcal{M} = \{M_1, M_2, \cdots, M_k\}$ in an overflowed index node, find the optimal partition of $\mathcal{M}$ into two disjoint subsets $\mathcal{M}_1$ and $\mathcal{M}_2$ such that $|\mathcal{M}_1| \geq m$, $|\mathcal{M}_2| \geq m$ and $|L(\mathcal{M}_1)| + |L(\mathcal{M}_2)|$ is minimum. The goal of (P3), which arises during insertions, node-splits, or node-merges, is to identify a bounding automaton for a set of REs that does not cover too much "dead space". Thus, given a collection of automata $\mathcal{M}$, the goal is to find the optimal bounding automaton M such that the number of states of M is no more than α, $L(\mathcal{M}) \subseteq L(M)$ and $|L(M)|$ is minimum.

The objective of the above three operations is to maximize the pruning during search by keeping bounding automata tight. In (P1), the optimal automaton M_i selected (within an internal node) to accommodate a newly inserted automaton M is to maximize $|L(M_i) \cap L(M)|$. The set of automata $\mathcal{M}$ are split into two tight clusters in (P2), while in (P3), the most precise automaton (with no more than α states) is computed to cover the set of automata in $\mathcal{M}$. Note that (P3) is unique to RE-trees, while both (P1) and (P2) have their equivalents in R-trees. The heuristics solutions [2,8] proposed for (P1) and (P2) in R-trees aim to minimize the number of visits to nodes that do not lead to any qualifying data entries. Although the minimal bounding automata in RE-trees (which correspond to regular languages) are very different from the MBRs in R-trees, the intuition behind minimizing the area of MBRs (total area or overlapping area) in R-trees should be effective for RE-trees as well. The counterpart for area in an RE-tree is $|L(M)|$, the size of the regular language for M. However, since a regular language is generally an infinite set, new measures need to be developed for the size of a regular language or for comparing the sizes of two regular languages.

One approach to compare the relative sizes of two regular languages is based on the following definition: for a pair of automata M_i and M_j, $L(M_i)$ is said to be larger than $L(M_j)$ if there exists a positive integer N such that for all $k \geq N$, $\sum_{l=1}^{k} |L_l(M_i)| \geq \sum_{l=1}^{k} |L_l(M_j)|$. Based on the above intuition, three increasingly sophisticated measures are proposed to capture the size of an infinite regular language. The *max-count measure* simply counts the number of strings in the language up to a certain size λ; i. e., $|L(M)| = \sum_{i=1}^{\lambda} |L_i(M)|$. This measure is useful for applications where the maximum length of all the REs to be indexed are known and is not too large so that λ can be set to some value slightly larger than the maximum length of the REs. A second more robust measure that is less sensitive to the λ parameter value is the *rate-of-growth measure* which is based on the intuition that a larger language grows at a faster rate than a smaller language. The size of a language is approximated by computing the rate of change of its size from one "window" of lengths to the next consecutive "window" of lengths: if λ is a length parameter that denote the start of the first window and θ is a window-size parameter, then $|L(M)| = \sum_{\lambda+\theta}^{\lambda+2\theta-1} |L_i(M)| / \sum_{\lambda}^{\lambda+\theta-1} |L_i(M)|$. As in the max-count measure, the parameters λ and θ should be chosen to be slightly greater than the number of states of M to ensure that strings involving a substantial portion of paths, cycles, and accepting states are counted in each window. However, there are cases where the rate-of-growth measure also fails to capture the "larger than" relationship between regular languages [4]. To address some of the shortcomings of the first two metrics, a third information-theoretic measure is proposed that is based on Rissanen's Minimum description length (MDL) principle [11]. The intuition is that if $L(M_i)$ is larger than $L(M_j)$, then the per-symbol-cost of an MDL-based encoding of a random string in $L(M_i)$ using M_i is very likely to be higher than that of a string in $L(M_j)$ using M_j, where the per-symbol-cost of encoding a string $w \in L(M)$ is the ratio of the cost of an MDL-based encoding of w using M to the length of w. More specifically, if $w = w_1.w_2.\cdots.w_n \in L(M)$ and $s_0, s_1, \ldots, s_n$ is the unique sequence of states visited by w in M, then the MDL-based encoding cost of w using M is given by $\sum_{i=0}^{n-1} \lceil \log_2(n_i) \rceil$, where each n_i denotes the number of transitions out of state s_i, and $\log_2(n_i)$ is the number of bits required to specify the transition out of state s_i. Thus, a reasonable measure for the size of a regular language $L(M)$ is the expected per-symbol-cost of an MDL-based encoding for a random sample of strings in $L(M)$.

To utilize the above metrics for measuring $L(M)$, one common operation needed is the computation of $|L_n(M)|$, the number of length-n strings in $L(M)$. While $|L_n(M)|$ can be efficiently computed when M is a DFA, the problem becomes #P-complete when M is an NFA [10]. Two approaches were proposed to approximate $|L_n(M)|$ when N is an NFA [10]. The first approach is an unbiased estimator for $|L_n(M)|$, which can be efficiently computed but can have a very large standard deviation. The second approach is a more accurate randomized algorithm for approximating $|L_n(M)|$ but it is not very useful in practice due to its high time complexity of $O(n^{\log(n)})$. A more practical approximation algorithm with a time complexity of $O(n^2|M|^2 \min\{|\Sigma|, |M|\})$ was proposed in [4].

The RE-tree operations (P1) and (P2) require frequent computations of $|L(M_i \cap M_j)|$ and $|L(M_i \cup M_j)|$ to be performed for pairs of automata M_i, M_j. These computations can adversely affect RE-tree performance since construction of the intersection and union automaton M can be expensive. Furthermore, since the final automaton M may have many more states than the two initial automata M_i and M_j, the cost of measuring $|L(M)|$ can be high. The performance of these computations can, however, be optimized by using sampling. Specifically, if the counts and samples for each $L(M_i)$ are available, then this information can be utilized to derive approximate counts and samples for $L(M_i \cap M_j)$ and $L(M_i \cup M_j)$ without incurring the overhead of constructing the automata $M_i \cap M_j$ and $M_i \cup M_j$ and counting their sizes. The sampling techniques used are based on the following results for approximating the sizes of and generating uniform samples of unions and intersections of arbitrary sets:

Theorem 1 (Chan, Garofalakis, Rastogi, [4]) *Let r_1 and r_2 be uniform random samples of sets S_1 and S_2, respectively.*

1. $(|r_1 \cap S_2||S_1|)/|r_1|$ *is an unbiased estimator of the size of* $S_1 \cap S_2$.
2. $r_1 \cap S_2$ *is a uniform random sample of* $S_1 \cap S_2$ *with size* $|r_1 \cap S_2|$.
3. *If the sets S_1 and S_2 are disjoint, then a uniform random sample of $S_1 \cup S_2$ can be computed in $O(|r_1| + |r_2|)$ time. If S_1 and S_2 are not disjoint, then an approximate uniform random sample of $S_1 \cup S_2$ can be computed with the same time complexity.*

Applications

The RE retrieval problem also arises in the context of both XML document classification, which identifies matching DTDs for XML documents, as well as BGP routing, which assigns appropriate priorities to BGP advertisements based on their matching routing-system sequences.

Experimental Results

Experimental results with synthetic data sets [5] clearly demonstrate that the RE-tree index is significantly more effective than performing a sequential search for matching REs, and in a number of cases, outperforms sequential search by up to an order of magnitude.

Recommended Reading

1. Altinel, M., Franklin, M.: Efficient filtering of XML documents for selective dissemination of information. In: *Proceedings of 26th International Conference on Very Large Data Bases*, Cairo, Egypt, pp. 53–64. Morgan Kaufmann, Missouri (2000)
2. Beckmann, N., Kriegel, H.-P., Schneider, R., Seeger, B.: The R*-Tree: An efficient and robust access method for points and rectangles. In: *Proceedings of the ACM International Conference on Management of Data*, Atlantic City, New Jersey, pp. 322–331. ACM Press, New York (1990)
3. Chan, C.-Y., Felber, P., Garofalakis, M., Rastogi, R.: Efficient filtering of XML documents with XPath expressions. In: *Proceedings of the 18th International Conference on Data Engineering*, San Jose, California, pp. 235–244. IEEE Computer Society, New Jersey (2002)
4. Chan, C.-Y., Garofalakis, M., Rastogi, R.: RE-Tree: An efficient index structure for regular expressions. In: *Proceedings of 28th International Conference on Very Large Data Bases*, Hong Kong, China, pp. 251–262. Morgan Kaufmann, Missouri (2002)

5. Chan, C.-Y., Garofalakis, M., Rastogi, R.: RE-Tree: An efficient index structure for regular expressions. *VLDB J.* **12**(2), 102–119 (2003)
6. Clark, J., DeRose, S.: XML Path Language (XPath) Version 1.0. W3C Recommendation, http://www.w3.org./TR/xpath, Accessed Nov 1999
7. Diao, Y., Fischer, P., Franklin, M., To, R.: YFilter: Efficient and scalable filtering of XML documents. In: *Proceedings of the 18th International Conference on Data Engineering*, San Jose, California, pp. 341–342. IEEE Computer Society, New Jersey (2002)
8. Guttman, A.: R-Trees: A dynamic index structure for spatial searching. In: *Proceedings of the ACM International Conference on Management of Data*, Boston, Massachusetts, pp. 47–57. ACM Press, New York (1984)
9. Hopcroft, J., Ullman, J.: *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Massachusetts (1979)
10. Kannan, S., Sweedyk, Z., Mahaney, S.: Counting and random generation of strings in regular languages. In: *Proceedings of the 6th ACM-SIAM Symposium on Discrete Algorithms*, San Francisco, California, pp. 551–557. ACM Press, New York (1995)
11. Rissanen, J.: Modeling by Shortest Data Description. Automatica **14**, 465–471 (1978)
12. Stewart, J.W.: *BGP4, Inter-Domain Routing in the Internet*. Addison Wesley, Massacuhsetts (1998)

Regular Expression Matching

2004; Navarro, Raffinot

LUCIAN ILIE
Department of Computer Science, University of Western Ontario, London, ON, Canada

Keywords and Synonyms

Automata-based searching

Problem Definition

Given a *text string* T of length n and a *regular expression* R, the **regular expression matching** problem **(REM)** is to find all text positions at which an occurrence of a string in $L(R)$ ends (see below for definitions).

For an alphabet Σ, a *regular expression* R over Σ consists of elements of $\Sigma \cup \{\varepsilon\}$ (ε denotes the empty string) and operators $\cdot$ (concatenation), | (union), and $*$ (iteration, that is, repeated concatenation); the set of strings $L(R)$ represented by R is defined accordingly; see [5]. It is important to distinguish two measures for the size of a regular expression: the *size*, m, which is the total number of characters from $\Sigma \cup \{\cdot, |, *\}$, and Σ*-size*, m_Σ, which counts only the characters in Σ. As an example, for $R = (\mathtt{A}|\mathtt{T})((\mathtt{C}|\mathtt{CG})*)$, the set $L(R)$ contains all strings that start with an A or a T followed by zero or more strings in the set {C, CG}; the size of R is $m = 8$ and the Σ-size is $m_\Sigma = 5$. Any regular expression can be processed in linear time so that $m = \mathcal{O}(m_\Sigma)$ (with a small constant); the difference becomes important when the two sizes appear as exponents.

Key Results

Finite Automata

The classical solutions for the REM problem involve finite automata which are directed graphs with the edges labeled by symbols from $\Sigma \cup \{\varepsilon\}$; their nodes are called states; see [5] for details. Unrestricted automata are called *nondeterministic finite automata (NFA)*. *Deterministic finite automata (DFA)* have no ε-labels and require that no two outgoing edges of the same state have the same label. Regular expressions and DFAs are equivalent, that is, the sets of strings represented are the same, as shown by Kleene [8]. There are two classical ways of computing an NFA from a regular expression. Thompson's construction [14], builds an NFA with up to $2m$ states and up to $4m$ edges whereas Glushkov–McNaughton–Yamada's automaton [3,9] has the minimum number of states, $m_\Sigma + 1$, and $\mathcal{O}(m_\Sigma^2)$ edges; see Fig. 1. Any NFA can be converted into an equivalent DFA by the *subset construction*: each subset of the set of states of the NFA becomes a state of the DFA. The problem is that the DFA can have exponentially more states than the NFA. For instance, the regular expression $((\mathtt{a}|\mathtt{b})*)\mathtt{a}(\mathtt{a}|\mathtt{b})(\mathtt{a}|\mathtt{b})\ldots(\mathtt{a}|\mathtt{b})$, with k occurrences of the $(\mathtt{a}|\mathtt{b})$ term, has a $(k+2)$-state NFA but requires $\Omega(2^k)$ states in any equivalent DFA.

Classical Solutions

A regular expression is first converted into an NFA or DFA which is then simulated on the text. In order to be able to search for a match starting anywhere in the text, a loop labeled by all elements of Σ is added to the initial state; see Fig. 1.

Searching with an NFA requires linear space but many states can be active at the same time and to update them all one needs, for Thompson's NFA, $\mathcal{O}(m)$ time for each letter of the text; this gives Theorem 1. On the other hand, DFAs allow searching time that is linear in n but require more space for the automaton. Theorem 2 uses the DFA obtained from the Glushkov–McNaughton–Yamada's NFA.

Theorem 1 (Thompson [14]) *The REM problem can be solved with an NFA in $\mathcal{O}(mn)$ time and $\mathcal{O}(m)$ space.*

Theorem 2 (Kleene [8]) *The REM problem can be solved with a DFA in $\mathcal{O}(n + 2^{m_\Sigma})$ time and $\mathcal{O}(2^{m_\Sigma})$ space.*