HKUST Theoretical Computer Science Center Research Report HKUST-TCSC-99-02

A Characterization of Thompson Digraphs*

Dora Giammarresi† Jean-Luc Ponty‡ Derick Wood§

February 11, 1999

Abstract

A finite-state machine is called a Thompson machine if it can be constructed from a regular expression using Thompson's construction. We call the underlying digraph of a Thompson machine a Thompson digraph. We characterize Thompson digraphs and, as one application of the characterization, we give an algorithm that generates an equivalent regular expression from a Thompson machine in time linear in the number of states. Although the construction is simple, it is novel in that the usual constructions of equivalent regular expressions from finite-state machines produce regular expressions that have size exponential in the size of the given machine, in the worst case. The construction provides a first step in the construction of small expressions from finite-state machines.

1 Introduction

In 1968, Thompson [8] introduced his inductive construction of a finite-state machine from a regular expression. Thompson's construction is elegant and efficient. Although Kleene [5] gave an inductive construction when proving that finite-state-machine languages are regular, his method is not efficient and does not yield a simple implementation. Thompson was motivated by the implementation of the first version of `grep` to provide a simple and efficient construction. The resulting finite-state machine has size linear in the size of the original expression and each state has no more than two ingoing transitions and two outgoing transitions. It is somewhat surprising that until recently there has been little follow-up work on Thompson's construction and the resulting machines. A resurge of interest in the implementation of machines has resulted in some new discoveries about the Thompson construction [2, 4].

*This research was supported under a grant from the Research Grants Council of Hong Kong SAR. It was carried out while the first and second authors were visiting HKUST.

†Dipartimento di Matematica Applicata e Informatica, Università Ca' Foscari di Venezia, via Torino, 155 - 30173 Venezia Mestre, Italy. Email: `dora@dsi.unive.it`

‡L.I.F.A.R., Université de Rouen, Faculté des Sciences et des Techniques, Place Émile Blondel, 76821 Mont-Saint-Aignan Cedex, France. Email: `ponty@dir.univ-rouen.fr`

§Department of Computer Science, Hong Kong University of Science & Technology, Clear Water Bay, Kowloon, Hong Kong SAR. Email: `dwood@cs.ust.hk`

We characterize the underlying digraphs of the machines resulting from the Thompson construction on empty-free regular expressions that do not include the empty-set symbol, **Thompson digraphs** and **Thompson machines,** respectively. The characterization is different from the one of Caron and Ziadi [3] for Glushkov digraphs although it is necessarily related to it indirectly. We could characterize Thompson digraphs directly in terms of their inductive construction—see Fig. 1—but such a characterization yields little new insight into their structure. We take a different approach.

First, we characterize Thompson digraphs that are obtained from star-free regular expressions. These digraphs are, acyclic; therefore, we call them **Thompson dags.** We use Dyck languages defined by source–sink paths in a Thompson digraph in the characterization.

Second, since Thompson digraphs are **hammocks**, we can easily find all back edges with a depth-first traversal in linear time. Therefore, in linear time, we can determine where the star units occur and we can then transform the digraph into a dag with what we call **star reduction.** The resulting dag is a Thompson dag if and only if the original digraph is a Thompson digraph.

The characterization provides us with a means of generating small regular expressions from some finite-state machines. We first determine whether a given finite-state machine is Thompson and, if it is, we construct a small regular expression E from the machine such that when we apply the Thompson construction to E we obtain a finite-state machine that is identical to M apart from a renaming of states. The work of Caron and Ziadi [3] gives a second class of machines for which the construction of equivalent regular expressions can be solved efficiently. We expect similar results to hold for Mirkin's construction [6] and the SSS construction [7].

2 Notation and terminology

We recall the basics of digraphs, finite-state machines and regular expressions and introduce the notation that we use.

A directed graph or **digraph** $G = (V, E)$ consists of a finite set V of vertices and a set E of directed edges of the form (u, v), where u and v are vertices. A **path** is a sequence $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$ of edges; it is a **cycle** if $v_0 = v_k$ and $k \geq 1$. A path is a **simple path** if it contains no cycles. A digraph that has no cycles is called a directed acyclic graph or **dag.** The size of a digraph is the sum of the number of vertices and number of edges.

We are particularly interested in digraphs that have a single designated **source vertex s** that has no edges entering it, a single designated **sink vertex S** that has no edges exiting it, and each of its vertices occurs on some simple path from the source to the sink. Such digraphs are called **hammocks**[1] and we denote them by a tuple (V, E, s, S).

A **finite-state machine**[2] consists of a finite set Q of states, an input alphabet Σ, a start state $s \in Q$, a final state $f \in Q$ and a transition relation $\delta \subseteq Q \times \Sigma_\lambda \times Q$, where λ denotes the null string and $\Sigma_\lambda = \Sigma \cup \{\lambda\}$. Clearly, we can depict the transition relation of such a

[1] In the literature hammocks are often called *st*-digraphs.

[2] Normally, a finite-state machine is allowed to have more than one final state and, sometimes, more than one start state. The formulation we have chosen is appropriate for the study of Thompson machines.

machine as an edge-labeled digraph (the labels are symbols from Σ_λ); it is usually called the state or transition digraph of the machine. If we drop the edge labels of a state digraph and ignore multiple edges, we obtain a digraph, the underlying digraph of the machine. The size of a machine is the number of its transitions.

Let Σ be an alphabet. Then, we define a regular expression E over Σ inductively as follows:

$E = \emptyset$, where $\emptyset$ is the empty-set symbol;

$E = \lambda$, where λ is the null-string symbol;

$E = a$, where a is in Σ;

$E = (F + G)$, where F and G are regular expressions;

$E = (F \cdot G)$, where F and G are regular expressions;

$E = (F^*)$, where F is a regular expression.

We define the language of E inductively, in the usual way [1, 9].

We say that a regular expression E is **empty free** if E does not contain any appearance of the empty-set symbol. We say that a regular expression is **star free** if it has no Kleene-star subexpression. The size $|E|$ of a regular expression E is the total number of appearances in E of symbols from $\Sigma \cup \{\lambda, \emptyset\}$.

3 Thompson digraphs

Thompson developed a construction [8] to compile regular expressions into a form that is suitable for pattern matching in text files. The construction is defined inductively (see Fig. 1) and it gives a finite-state machine that has a number of pleasing properties *when we restrict the given regular expression to be empty free:*

1. There is one start state that is **only exiting;** that is, there are no transitions that enter it.

2. There is one final state that is **only entering;** that is, there are no transitions that leave it.

3. Every state is on a simple path from the start state to the final state.

4. Each state has at most two ingoing transitions and at most two outgoing transitions.

5. The size of the machine is at most three times the size of the given regular expression.

In Fig. 2, we give the result of the Thompson construction on the regular expression $(((a + b)^*) \cdot ((b + \lambda) \cdot a))$.

We define a **Thompson machine** to be a finite-state machine that is obtained by the Thompson construction on an empty-free regular expression and we define a **Thompson**

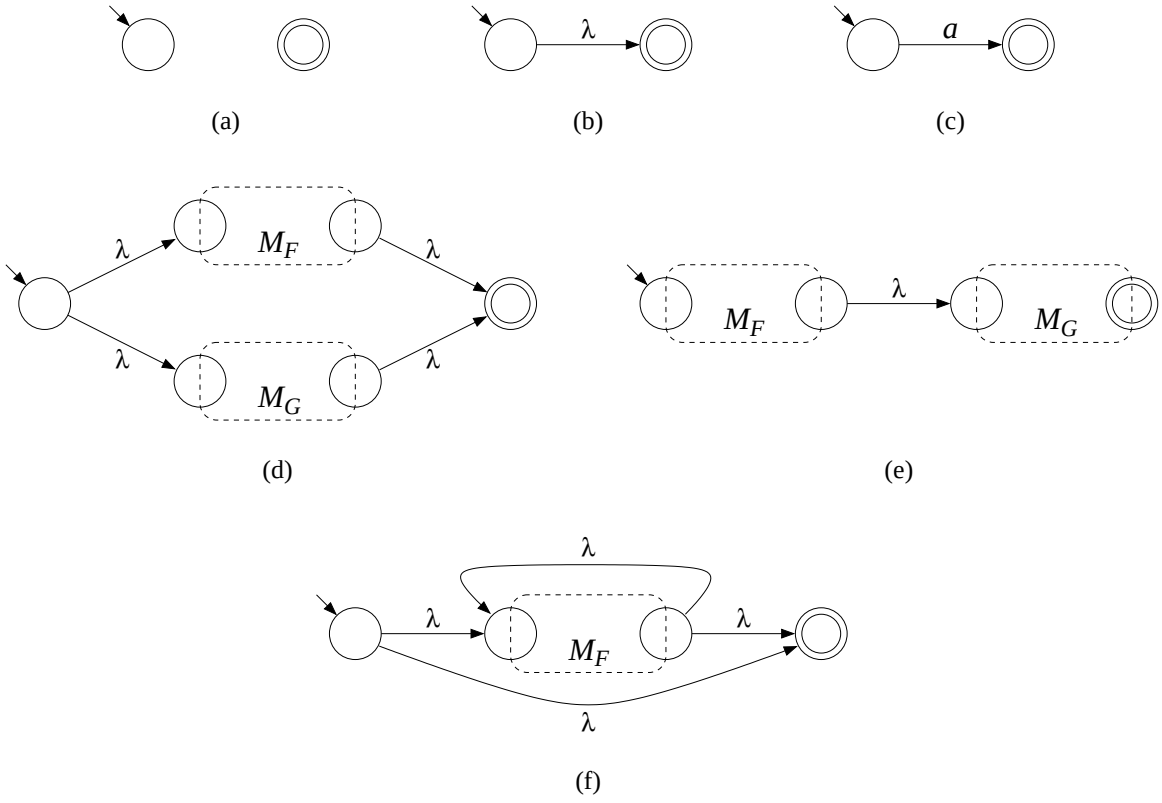

Figure 1: The Thompson construction. The order of the figures corresponds to the order of the cases in the definition of regular expressions. The finite-state machines correspond to the regular expressions: a. $E = \emptyset$; b. $E = \lambda$; c. $E = a$, $a \in \Sigma$; d. $E = (F + G)$; e. $E = (F \cdot G)$; and f. $E = (F^*)$. When a given regular expression E is empty free, (a) is never used by the Thompson construction. We include (a) for completeness.

digraph to be the underlying digraph of a Thompson machine. We name the units that the Thompson construction uses to assemble a Thompson machine as follows: the **base units** (Fig. 1(b) and (c)); the **plus unit** (Fig. 1(d)); the **dot unit** (Fig. 1(e)); and the **star unit** (Fig. 1(f)).

Observe that a Thompson digraph is a hammock[3] and it has exactly the same number of edges as the number of transitions in the original Thompson machine. The Thompson machine of Fig. 2 yields the Thompson digraph of Fig. 3.

We can obtain a Thompson digraph directly from a regular expression by modifying Thompson's construction appropriately: see Fig. 4. We use the same visual cues to indicate the source and sink vertices as we do to indicate the start and final states of finite-state machines.

[3]We can establish this observation formally using the inductive construction. If we allow regular expressions to contain the empty-set symbol, then the digraph of the resulting machine is not a hammock.

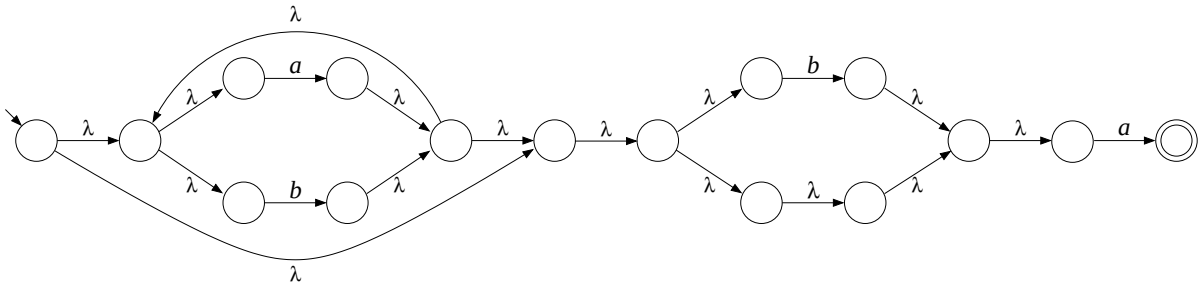

Figure 2: The result of the Thompson construction on the running example expression $(((a+b)^*)\cdot((b+\lambda)\cdot a))$.

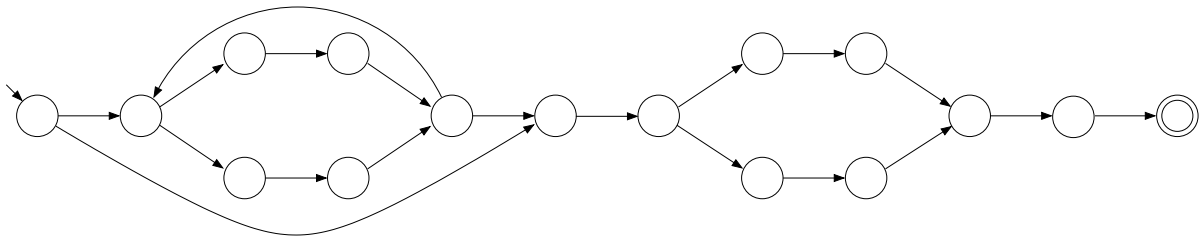

Figure 3: The Thompson digraph given by the Thompson machine of Fig. 2.

4 The characterization theorem

Not only is a Thompson digraph a hammock, but also the vertices of a Thompson digraph have indegree and outdegree at most two. We call hammocks that satisfy this additional restriction **two hammocks.**

We characterize the two hammocks that are Thompson digraphs.

We begin with a modification of the usual definition of Dyck strings. For an integer $t \geq 1$, let $\Gamma_t^L = \{[_i : 1 \leq i \leq t\}$, $\Gamma_t^R = \{]_i : 1 \leq i \leq t\}$, $\Gamma_t = \Gamma_t^L \cup \Gamma_t^R$ and $\bar{\Gamma}_t = \Gamma_t \cup \{\bowtie\}$. The notion of Dyck string over Γ_t is well known; such a string is well-balanced with respect to the pairing of $[_i$ with $]_i$, for all i, $1 \leq i \leq t$. We need to allow the pairing to be given by any bijection $\{1,\ldots,t\} \longrightarrow \{1,\ldots,t\}$ not only by the identity bijection. We call such strings, **b-Dyck strings** and we define them as follows. Given an alphabet Γ_t and a bijection $\beta : \{1,\ldots,t\} \longrightarrow \{1,\ldots,t\}$, we define b-Dyck strings over Γ_t inductively as follows: *The null string λ is a b-Dyck string over Γ_t and whenever x and y are b-Dyck strings over Γ_t, xy is a b-Dyck string over Γ_t and $[_i x]_{\beta(i)}$ is a b-Dyck string over Γ_t, for all i, $1 \leq i \leq t$.*

We can now define a fundamental set of strings for Thompson machines. Given an alphabet Γ_t and a bijection $\beta : \{1,\ldots,t\} \longrightarrow \{1,\ldots,t\}$, a string x over $\bar{\Gamma}_t$ is a **Thompson string** if it satisfies the following three conditions:

1. If we erase all appearances of $\bowtie$ from x, we obtain a b-Dyck string over Γ_t.
2. The substring $[_i\ ]_j$ does not appear in x, for any i and j, $1 \leq i, j \leq t$.
3. All maximal-length $\bowtie$-substrings of x have even length.

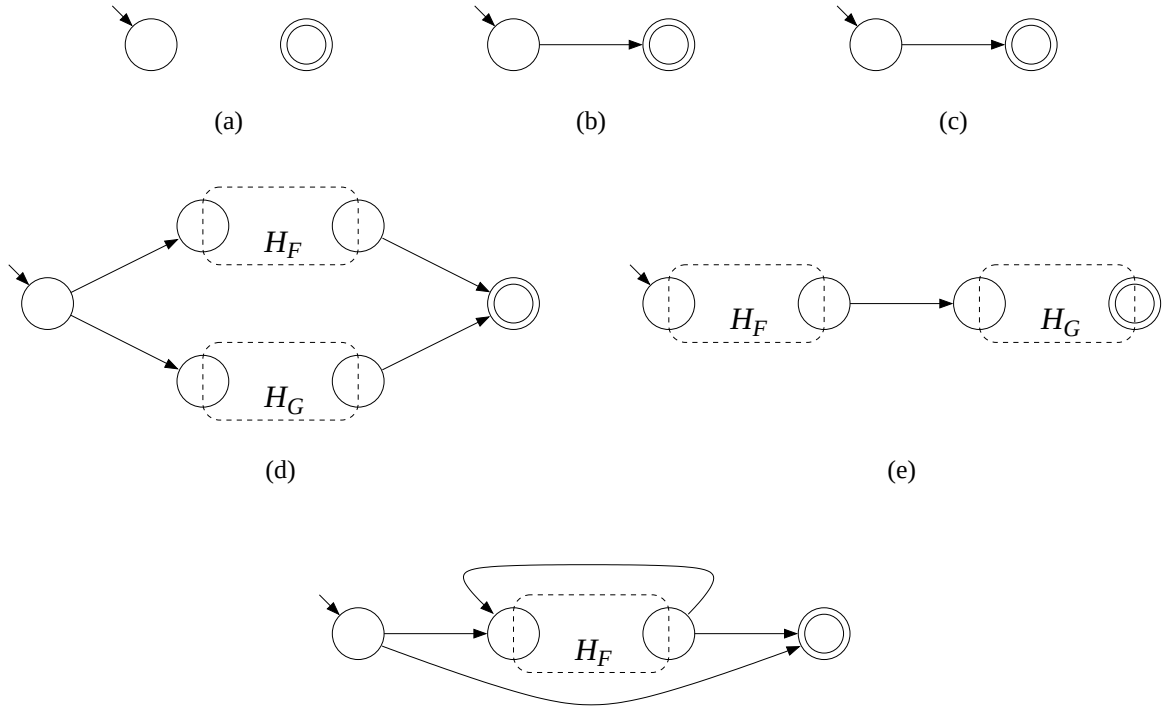

Figure 4: The Thompson digraph construction. The order of the figures corresponds to the order of the cases in the definition of regular expressions. The digraphs correspond to the regular expressions: a. $E = \emptyset$; b. $E = \lambda$; c. $E = a$, $a \in \Sigma$; d. $E = (F + G)$; e. $E = (F \cdot G)$; and f. $E = (F^*)$.

We develop a characterization of Thompson digraphs by first considering the Thompson digraphs that are obtained from star-free expressions. In this case, we obtain **Thompson dags.** Since Thompson digraphs are two hammocks, we refer to a vertex with indegree i and outdegree j as an **(i,j) vertex.** There are no $(2, 2)$ vertices in Thompson dags. Indeed, $(2, 2)$ vertices are produced only by the Kleene star of a plus subexpression that has the form $((F + G)^*)$. For convenience we assume that the source and the sink vertices have a dummy in-edge and a dummy out-edge, respectively. Then, we have the following necessary condition for two-hammock H to be a Thompson dag.

Property 1: A Thompson dag has only $(1, 2)$, $(1, 1)$, and $(2, 1)$ vertices, it has an even number of $(1, 1)$ vertices and it has as many $(1, 2)$ vertices as it has $(2, 1)$ vertices.

Clearly, we can check whether an acyclic two hammock satisfies Property 1 in time linear in the size of the hammock. We simply examine each edge in the hammock, counting the numbers of $(1, 2)$ and $(2, 1)$ vertices and checking whether any vertex is a $(2, 2)$ vertex.

As a result of Property 1, let the number of $(1, 2)$ vertices (or, equivalently, the number

of $(2,1)$ vertices) be t. We label the $(1,2)$ vertices arbitrarily and uniquely with $[_i$, $1 \leq i \leq t$; the $(1,1)$ vertices with $\bowtie$; and the $(2,1)$ vertices arbitrarily and uniquely with $]_j$, $1 \leq j \leq t$. The resulting digraph is a $\bar{\Gamma}_t$-labeled two hammock. Note that, if $t = 0$, then the two hammock is a line digraph.

We can now state a second necessary condition for a two-hammock H to be a Thompson dag.

Property 2: There is a bijection $\beta : \{1, \ldots, t\} \longrightarrow \{1, \ldots, t\}$, such that every source–sink path in a $\bar{\Gamma}_t$-labeled two hammock spells out a Thompson string.

A Thompson string satisfies three conditions, two of them are easy to check directly. First, a substring $[_i]_j$ occurs in an acyclic two hammock H if and only if a target vertex of a $(1,2)$ vertex is a $(2,1)$ vertex and this property is easy to check. Second, every maximal-length $\bowtie$-substring of a candidate Thompson string has even length. Since such a substring corresponds to a maximal path of $(1,1)$ vertices, we can enumerate such paths and do so efficiently. Each such path must begin with the source vertex or with a target vertex of a $(1,2)$ or a $(2,1)$ vertex. Essentially, we partition a depth-first traversal of the two hammock into several disjoint traversals. Both checks take time linear in the size of the hammock. All that remains is to check the b-Dyck condition of Property 2 which we do in conjunction with the check for Property 3 later in this section.

Lemma 1 *Let H be a Thompson dag. Then, H satisfies Properties 1 and 2.*

Proof: Let E be a star-free and empty-free regular expression such that H is the Thompson digraph obtained from E. We prove the result by induction on the size of E. If $|E| = 1$, then $E = a$ or $E = \lambda$ and, in both cases, there is only one source–sink path in H and it consists of an edge from the source vertex to the sink vertex. The corresponding Thompson string $\bowtie \bowtie$ has even length.

Now, assume that the result holds for all star-free and empty-free expressions E of size at most n, for some $n \geq 1$, and consider a star-free and empty-free expression E of size $n+1$. Then, either $E = (F \cdot G)$ or $E = (F + G)$, where $|F| < n$ and $|G| < n$. We examine the two cases separately. Notice that, by the Thompson construction, the dot operation for two subexpression F and G replaces the dummy edges from the final state f_F of M_F^T and into the start state s_G of M_G^T and adds a new transition (f_F, λ, s_G). The state f_F still has outdegree one and the state s_G still has indegree one; thus, the resulting Thompson dag satisfies Property 1. Second, each string spelled out by a source–sink path in the Thompson dag for E has the form $x_F x_G$, where x_F and x_G are Thompson strings, which is also a Thompson string.

Now consider the case $E = (F + G)$. By the Thompson construction, every source–sink path in the digraph $M_{(F+G)}^T$ corresponds to a string $[_i w]_j$, where $\beta(i) = j$ and w is a Thompson string of a source–sink path either in M_F^T or in M_G^T. Thus, each string $[_i w]_j$ is also a Thompson string. In addition, the source and sink vertices satisfy Property 1; therefore, all vertices satisfy Property 1. □

Notice that Properties 1 and 2 are not sufficient for an acyclic two hammock be a Thompson dag; for example, consider the dag of Fig. 5. Although all source–sink paths in this dag

spell out Thompson strings, the dag cannot be obtained by applying the Thompson construction to any star-free regular expression. The reason is that we also need to verify that

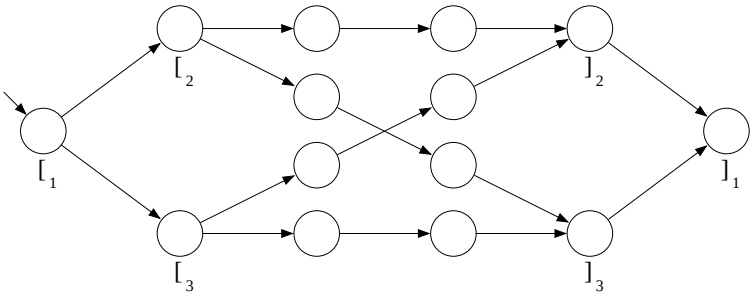

Figure 5: An example for the insufficiency of Property 2.

every path from an $[_i$-vertex passes through the same matching $]_j$-vertex. We express this condition using specific decompositions of b-Dyck strings.

Property 3: For a $(1,2)$ vertex p in a labeled acyclic two hammock, let L_p be the set of strings spelled out on paths from the source vertex to the sink vertex that pass through p. Then, we can decompose each string x in L_p into $u[_i v]_j w$, where $[_i$ is the label of p and $]_j$ is the matching symbol of $[_i$.

Lemma 2 *Let H be a Thompson dag. Then, H satisfies Property 3.*

Proof: Let E be a star-free, empty-free regular expression such that $H = M_E^T$. If two vertices p and q in H correspond to a matching $([_i,]_j)$ pair, then they correspond to the source and the sink, respectively, of a Thompson machine $M_{F+G}^T = H'$ such that $(F+G)$ is a subexpression of E. Moreover, by the definition of the Thompson construction, the subgraph H' is connected to $H - H'$ by H''s source and sink; that is, by p and q. Thus, each path in H from p to H's sink passes through q. □

We are now in a position to characterize Thompson dags.

Theorem 3 *An acyclic two hammock is a Thompson dag if and only if it satisfies Properties 1, 2 and 3.*

Proof: We already proved, by Lemmas 1 and 2 that, if H is a Thompson dag, then it satisfies Properties 1, 2 and 3. We now prove the converse. Let H be an acyclic two hammock that satisfies Properties 1, 2 and 3. To demonstrate that H is a Thompson dag, we must construct a regular expression E such that H is isomorphic to H_E^T.

We label each edge in H (to give H') either with the null-string symbol or with a unique positive integer in the following manner. Let e be the number of edges in H. We will use the integers up to at most e as possible edge labels. If (p,q) is an edge that enters or leaves a $(1,2)$-vertex or a $(2,1)$-vertex, then we label it with the null-string symbol. (We do not label the dummy edges into the source vertex or from the sink vertex.) In addition, for each maximal $\bowtie$-path of $\bowtie$-vertices, we label its edges with either the null symbol or

with unique integers such that the null symbols and the integers alternate, starting with an integer label. Note that by Property 2, there must be a nonzero, even number of $\bowtie$-vertices in such a path; hence, there is an odd number of edges. More precisely, given a maximal $\bowtie$-path $p_1, p_2, p_3, \ldots, p_{2l-1}, p_{2l}$, where $l \geq 1$, we label its edges as follows: Assume that we have already used integer labels up to and including some integer $k \geq 0$. We then label the edges in the path as $(p_1, k+1, p_2)$, (p_2, λ, p_3), $(p_3, k+2, p_4)$, ..., $(p_{2l-1}, k+l, p_{2l})$ and replace k by $k+l$.

The expression E' is over the alphabet $\{1, \ldots, k\}$ and the number $\mathrm{nn}(H')$ of nonnull-labeled edges in H' is exactly k. We prove by induction on the value of $\mathrm{nn}(H')$ that there is a regular expression E such that $H_E^T \cong H'$.

Basis: $\mathrm{nn}(H') = 1$. In this case, H' has one edge $(s, 1, S)$. Clearly, $E = 1$ and $H_E^T \cong H$.

Induction hypothesis: Assume the result holds for all H' such that $\mathrm{nn}(H') \leq l$, for some $l \geq 1$.

Induction step: Consider an H' such that $\mathrm{nn}(H') = l + 1$. Since $l \geq 1$, H' must contain at least one plus unit or one dot unit. Consider both alternatives.

1. H' contains a plus unit. Replace the plus unit as shown in Fig. 6(a) to give a new labeled dag H'' such that $\mathrm{nn}(H'') = l$ and the new label $(F + G)$ is treated as a compound symbol.

2. H' contains a dot unit. Replace the dot unit as shown in Fig. 6(b) to give a new labeled dag H'' such that $\mathrm{nn}(H'') = l$ and the new label $(F \cdot G)$ is treated as a compound symbol.

By the induction hypothesis, there is a regular expression $\bar{E}$ such that $H_{\bar{E}}^T \cong H''$, where the new compound symbol is not expanded in the Thompson construction. We now expand the compound symbol $(F + G)$ or $(F \cdot G)$ using Fig. 6(b) or (c), respectively, which expands H'' to give H' and we obtain $H_E^T \cong H'$. □

We now characterize the two hammocks that are Thompson digraphs. Let H be a two hammock. We perform a depth-first traversal of H starting from the source vertex. It determines a set of **back edges** $B = \{(y_1, x_1), \ldots, (y_k, x_k)\}$, where $k \geq 0$. Notice that when (y_i, x_i) is a back edge, the vertex x_i has indegree two. For, if x_i has indegree one, then there is no simple source–sink path that contains x_i. Similarly, y_i has outdegree two. For, if y_i has outdegree one, then there is no simple source–sink path that contains y_i. We now state another necessary property for a two hammock to be a Thompson digraph.

Property 4: For each back edge (y, x) in a Thompson digraph H, there are two vertices w and z such that w, x, y and z are distinct and edges (w, x), (y, z) and (w, z) are in H.

Based on Property 4, we define a graph transformation that removes each back edge (y, x) and expands each edge (w, z) into three edges. Thus, if a two hammock has only Thompson-like cycles, the graph transformation will transform them to give a dag. We define the **star-reduction dag** $\bar{H}$ as follows: for each back edge (y_i, x_i), remove the edge (y_i, x_i) and replace the edge (w_i, z_i) with the three edges (w_i, u_i), (u_i, v_i) and (v_i, z_i), where u_i and v_i are new vertices; see Fig. 7.

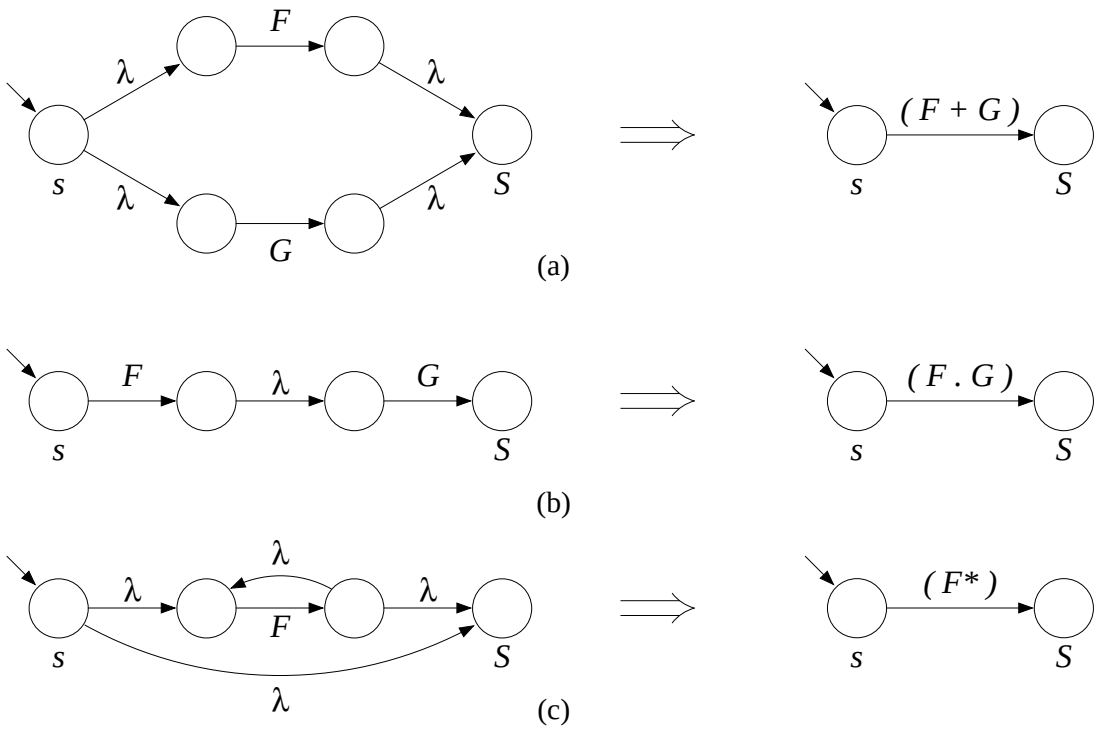

Figure 6: Three Thompson digraph transformations: a. Replacement of plus unit. b. Replacement of dot unit. c. Replacement of star unit.

Theorem 4 *A two-hammock H is a Thompson digraph if and only if H satisfies Property 4 and $\bar{H}$ is a Thompson dag.*

Proof: It is easy to verify that, by definition, if H is a Thompson digraph, then each back edge has corresponding forward edges as specified in Property 4 and $\bar{H}$ is a Thompson dag.

Conversely, suppose that we have a two-hammock H that satisfies Property 4 and $\bar{H}$ is a Thompson dag. We can construct a regular expression $\bar{E}$ corresponding to $\bar{H}$ as in the proof of Theorem 3. Then, we can obtain a regular expression E for H from $\bar{E}$ by replacing subexpressions of the form $(F + \lambda)$, produced by star reduction, with (F^*). We omit the formal inductive proof that $H_E^T \cong H$ as it is similar to that of proof of Theorem 3. □

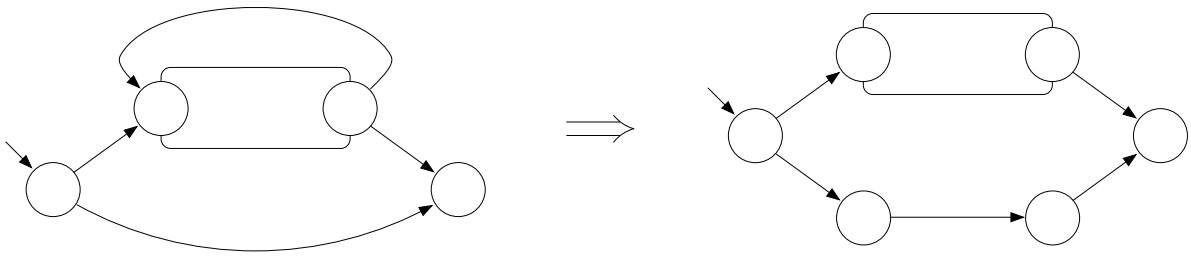

Figure 7: An illustration of star reduction.

Given a digraph H, we can verify whether it is a two hammock in time linear in the size of H. Moreover we can check whether it is a Thompson digraph in time linear in the size of H in two steps as follows: First, using a depth-first traversal of H from the source vertex, we detect all back edges and then check whether Property 4 holds. Second, if Property 4 holds, then we apply star reduction to H to give a dag H' and then check by depth-first traversal whether H' is a Thompson dag. We give a more formal description of the nontrivial portion of a Thompson-dag recognition algorithm in Fig. 8.

5 Regular expressions from Thompson machines

Once we have confirmed that a given edge-labeled digraph is an edge-labeled Thompson digraph, we can construct a regular expression from it whose size is linear in the size of the given machine and we can do so in linear time.

Theorem 5 *Given a Thompson machine M,we can construct an equivalent regular expression from it in time linear in the size of M.*

Proof: The idea behind the proof of this result is that we, first, parse the underlying edge-labeled Thompson digraph using a depth-first traversal to obtain an expression tree and, second, perform a depth-first traversal of the expression tree to produce a correctly parenthesized regular expression. Clearly, the second step is straightforward and can be implemented to run in time linear in the size of the expression tree. We claim that the size of the expression tree is of the same order as the size of the Thompson digraph. We specify, in Fig. 9, a parsing algorithm `exptree` for Thompson digraphs that takes time linear in the size of a digraph. The algorithm assumes that for each starting vertex of a unit, including a base unit, the β map gives the corresponding ending vertex.

It is easy to verify that each edge of H is traversed exactly once and therefore the algorithm `exptree` takes time linear in the size of H. Moreover its correctness follows because H satisfies Properties 1, 2 and 3.

□

The algorithm `exptree` is an inverse of the Thompson construction. For, if we take a regular expression E, construct the Thompson machine M_E^T and then apply `exptree` we obtain a regular expression $\bar{E}$ that is equivalent to E. Note that, although $\bar{E}$ is equivalent to E it may be differently parenthesized and the order of subexpressions of plus units may be different.

6 Concluding remarks

We have established a characterization of Thompson digraphs that enables us to unambiguously parse such digraphs and reconstruct regular expressions from them that have the same sizes as their digraphs. The interesting fact is that we ignore the transition labels completely in the characterization. Can similar characterizations be established for other inductively

constructed finite-state machines? We conjecture that such results hold for Mirkin's construction [6] and the SSS construction [7]. In any case, we conjecture that the finite-state machines given by these constructions can be unambiguously parsed. One interesting problem is whether we can unambiguously parse the machines given by other constructions in the literature including Kleene's original construction.

A tantalizing open problem is to characterize the largest class of finite-state machines that have small expressions easily computable from the machines. A less ambitious goal is to identify nontrivial classes of finite-state machines that yield small expressions.

References

[1] A.V. Aho and J.D. Ullman. *The Theory of Parsing, Translation, and Compiling, Vol. I: Parsing*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1972.

[2] A. Brüggemann-Klein and D. Wood. The validation of SGML content models. *Mathematical and Computer Modelling*, 25:73–84, 1997.

[3] P. Caron and D. Ziadi. Characterization of Glushkov automata. *Theoretical Computer Science*, 1998. To appear.

[4] D. Giammarresi, J.-L Ponty, and D. Wood. The Glushkov and Thompson constructions: A synthesis. Unpublished manuscript, July 1998.

[5] S.C. Kleene. Representation of events in nerve nets and finite automata. In C.E. Shannon and J. McCarthy, editors, *Automata Studies*, pages 3–42, Princeton, NJ, 1956. Princeton University Press.

[6] B. G. Mirkin. An algorithm for constructing a base in a language of regular expressions. *Engineering Cybernetics*, 5:110–116, 1966.

[7] S. Sippu and E. Soisalon-Soininen. *Parsing Theory: I Languages and Parsing*. EATCS Monographs on Theoretical Computer Science, Volume 15. Springer-Verlag, New York, NY, 1988.

[8] K. Thompson. Regular expression search algorithm. *Communications of the ACM*, 11:419–422, 1968.

[9] D. Wood. *Theory of Computation*. John Wiley & Sons, Inc., New York, NY, second edition, 1998. In preparation.

```
bool Thdag(vertex u; stack left)
    We check whether Properties 2 and 3 hold
    since Property 1 is easy to check.

if u is marked
    if u is a (2,1) vertex
        if beta(pop(left)) = u
            return true
        else
            return false
    else
        return true
else u is not marked
    mark u;
    if u is a (1,1) vertex and u is the sink vertex
        return true;
    elseif u is a (1,1) vertex and (u,v) in E
        return Thdag(v,left);
    elseif u is a (1,2) vertex
        /* (u,v), (u,w) in E */
        return Thdag(v,push(u,left)) and Thdag(w,push(u,left));
    elseif u is a (2,1) vertex
        /* (u,v) in E */
        s = pop(left);
        if beta(s) is undefined
            beta(s) = u;
            if u is the sink vertex
                return true;
            else
                return Thdag(v,left);
        else
            return beta(s) = u;
```

Figure 8: A depth-first traversal algorithm to check whether a two hammock spells out b-Dyck strings.

```
tree exptree(vertex u; digraph H)
The beta map is extended to handle the base units as
    well as the plus, dot and star units.

if beta(u) is a sink vertex
    if u is a (1,1) vertex
        return a node labeled A;
    elseif u is a (1,2) vertex
        /* (u,lambda,v), (u,lambda,w) in E */
        if beta(u) = v
            /* u and v bracket a star unit */
            /* (p,lambda,v) is in E, p != u */
            make p a sink vertex
            return a subtree that has a root labeled *
                and whose only subtree is exptree(w,H)
        elseif beta(u) = w
            /* u and w bracket a star unit */
            /* (p,lambda,w) is in E, p != u */
            make p a sink vertex
            return a subtree that has a root labeled *
                and whose only subtree is exptree(v,H)
        else /* u begins a plus unit */
            /* (p,lambda,beta(u)), (q,lambda,beta(u)) are in E */
            make p and q sink vertices
            return a subtree that has a root labeled +,
                the root's left subtree is exptree(v,H) and
                the root's right subtree is exptree(w,H)
else /* beta(u) is not a sink vertex */
    make beta(u) a sink vertex
    return a subtree that has a root labeled ·,
        the root's left subtree is exptree(u,H) and
        the root's right subtree is exptree(w,H),
        where (beta(u),lambda,w) is in H.
```

Figure 9: An algorithm to generate an expression tree from a Thompson machine.