Thompson Digraphs: A Characterization*

Dora Giammarresi[1], Jean-Luc Ponty[2], and Derick Wood[3]

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

[2] 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

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

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 establish and prove a characterization of Thompson digraphs. 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.

1 Introduction

In 1968, Thompson [8] gave an inductive construction of finite-state machines from regular expressions that was motivated by `grep.` The resulting finite-state machines have sizes linear in the sizes of the original expressions. A resurge of interest in the implementation of machines has resulted in some new discoveries about the Thompson construction [2, 4].

We characterize the underlying digraphs of the machines resulting from the Thompson construction on empty-free regular expressions (they do not include the empty-set symbol): **Thompson digraphs** and **Thompson machines,** respectively.

First, we characterize Thompson digraphs that are obtained from empty-free, 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

* 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.

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 equivalent regular expression from the machine.

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 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].

[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.

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 an inductive construction [8] (see Fig. 1) to compile regular expressions into finite-state machines. In Fig. 2, we give the result of the

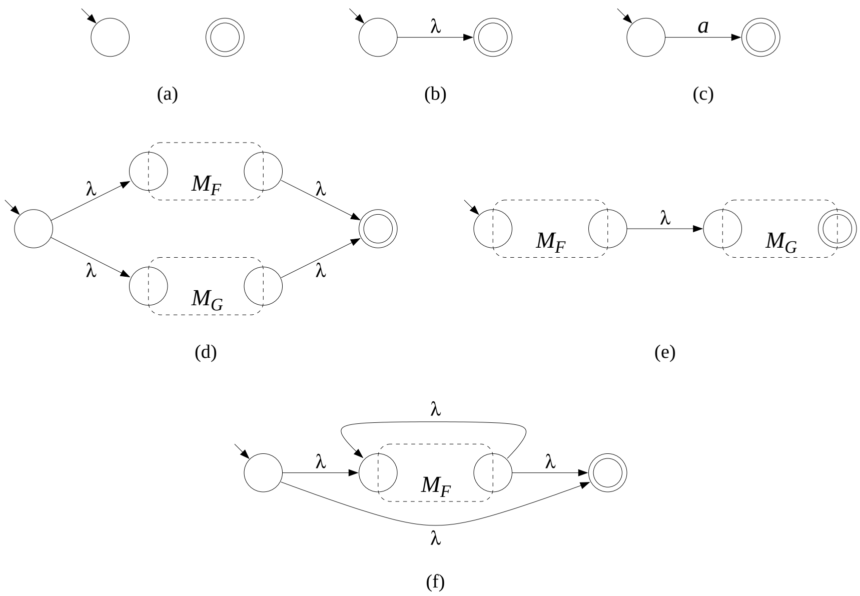

Fig. 1. The Thompson construction. The order of the figures corresponds to the order of the cases in the definition of regular expressions. The finite-state machines correspond to the regular expressions: a. $E = \emptyset$; b. $E = \lambda$; c. $E = a$, $a \in \Sigma$; d. $E = (F + G)$; e. $E = (F \cdot G)$; 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.

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 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)).

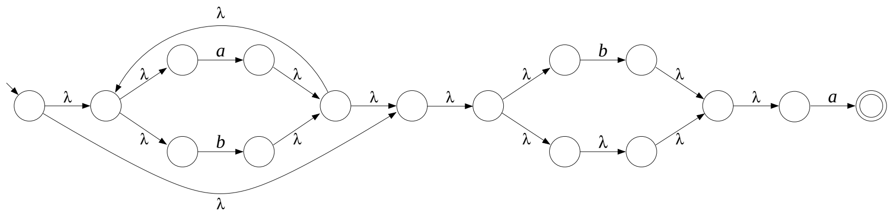

Fig. 2. The result of the Thompson construction on the running example expression $(((a+b)^*) \cdot ((b+\lambda) \cdot a))$.

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.

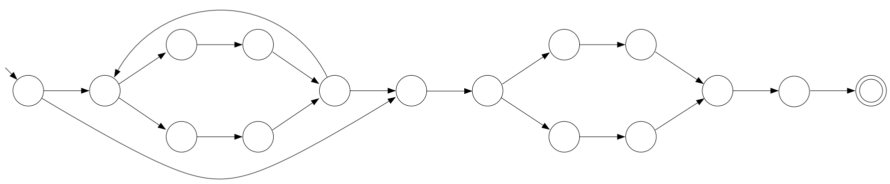

Fig. 3. The Thompson digraph given by the Thompson machine of Fig. 2.

We can obtain a Thompson digraph directly from a regular expression by modifying Thompson's construction appropriately.

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 $\bar{\Gamma}_t = \Gamma_t \cup \{\bowtie\}$, where $\bowtie$ is a special symbol used to label vertices of indegree one and outdegree one. 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 $\{1, \ldots, t\} \longrightarrow \{1, \ldots, t\}$

[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.

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 with respect to β 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, $1 \le i \le t$.*

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 with respect to β.
2. The substring $[_i\ ]_j$ does not appear in x, for any i and j, $1 \le i, j \le t$.
3. All maximal-length $\bowtie$-substrings of x have even length.

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.

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 $[_i$, $1 \le i \le t$; the $(1, 1)$ vertices with $\bowtie$; and the $(2, 1)$ vertices arbitrarily and uniquely with $]_j$, $1 \le j \le t$. 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, the first two are easy to check directly. We discuss the checking of the b-Dyck condition of Property 2 when we discuss how to check for Property 3 later in this section.

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

Proof. [4] 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.

[4] We omit most proofs—they are to be found in the full version of the paper [5].

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. 4. 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

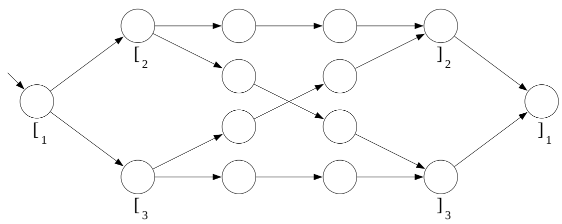

Fig. 4. An example for the insufficiency of Property 2.

reason is that we also need to verify that every path from a $[_i$-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.*

We are now in a position to characterize Thompson dags.

Theorem 1. *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 construct a regular expression E *such that H is isomorphic to H_E^T.*

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.

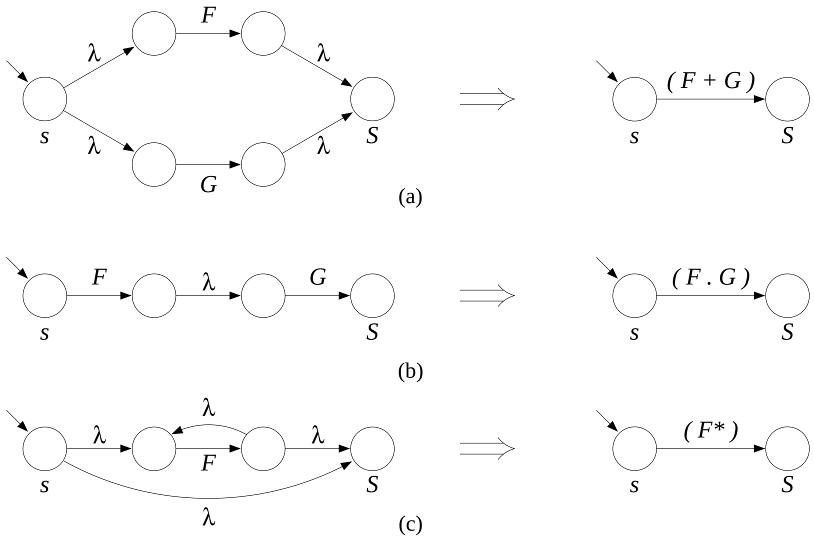

Fig. 5. Three Thompson digraph transformations: a. Replacement of plus unit. b. Replacement of dot unit. c. Replacement of star unit.

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 digraph 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 digraph 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. 6.

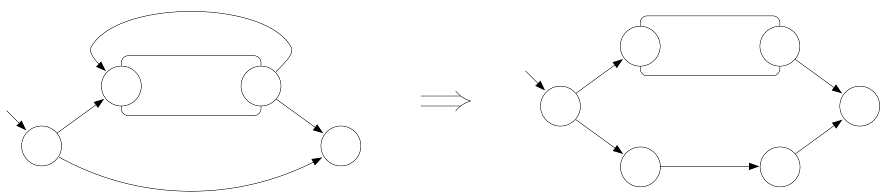

Fig. 6. An illustration of star reduction.

Theorem 2. *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 1. 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 the proof of Theorem 1.

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 elsewhere [5].

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 3. *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 can construct a parsing algorithm for Thompson digraphs that takes time linear in the size of a digraph [5]. 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 by the parsing algorithm; therefore, it takes time linear in the size of H. Moreover its correctness follows because H satisfies Properties 1, 2 and 3.

The parsing algorithm for Thompson digraphs is, essentially, the inverse of the Thompson construction. For, if we take a regular expression E, construct the Thompson machine M_E^T and then apply the parsing algorithm to M_E^T, we obtain a regular expression $\bar{E}$ that is equivalent to E. Note that, although $\bar{E}$ and E are equivalent, $\bar{E}$ 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. The earlier work of Caron and Ziadi [3] gives a second class of machines for which the construction of equivalent regular expressions can be carried out efficiently. 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.
   URL: http://www.cs.ust.hk/tcsc/RR/1998-11.ps.gz.
5. D. Giammarresi, J.-L Ponty, and D. Wood. A characterization of Thompson digraphs. Unpublished manuscript, January 1999.
   URL: http://www.cs.ust.hk/tcsc/RR/1999-2.ps.gz.
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.