RICE UNIVERSITY

Evaluating Performance of Automaton Universality Checking Algorithms

by

Corey Fisher

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Master of Science

Approved, Thesis Committee:

Moshe Vardi, Chair
Karen Ostrum George Distinguished
Service Professor in Computational
Engineering

Luay Nakhleh
J.S Abercrombie Professor of Computer
Science

Swarat Chaudhuri
Associate Professor of Computer Science

Houston, Texas

March, 2019

ABSTRACT

Evaluating Performance of Automaton Universality Checking Algorithms

by

Corey Fisher

NFA universality is an important problem in formal verification, since it is an effective proxy for complementation of NFAs - a key operation that underlies most verification algorithms. However, because complemented automata are extremely large, many modern tools use symbolic representations to perform complementation and universality checking. One state-of-the-art tool for NFA universality, ALASKA, symbolically represents automata using binary decision diagrams (BDDs) to more efficiently complement them with the subset construction.

The algorithm usually only represents a small number of subset-constructed states at a time, relative to the vast state space. Zero-suppressed decision diagrams (ZDDs) have the same semantics as BDDs, but are more efficient when representing sparse solution sets. We used this advantage in constructing a new ZDD-based tool, ALASKA-ZDD, which completely replaces ALASKA's symbolic representation with a ZDD-based one. We then experimentally compared it with ALASKA, using random automata generated with the widely-used Tabakov-Vardi (T-V) random model due to a lack of practical benchmarks. We found that while ALASKA is more efficient on average, ALASKA-ZDD had fewer timeouts due to difficult problems.

But how do we know the T-V model gives robust results? The model was originally adopted due to lack of practical benchmarks, but this also prevents checking its

reliability against real examples. While it statistically guarantees certain universality properties about the automata it produces, no further work has been done to verify its results. Therefore, it is unclear if tests on the T-V model are sufficient. In graph theory, many different random models are used for representing different problems - would that be an appropriate approach for verification? We introduce three new random models, and show that their results for the NFA universality question are the same as T-V. We also compare multiple solutions to the Büchi universality problem on these models, and find that their results are the same as T-V. Therefore, in addition to showing ALASKA-ZDD is competitive, we show that T-V can be used as a robust random model for verification, across multiple problems, verifying many previous results with the model.

Contents

Abstract ii

1 Introduction 1

2 Background 7

2.1 Automata theory 8

2.1.1 Types of automata 8

2.1.2 Treating NFAs as AFAs 10

2.1.3 Languages and complementation 11

2.1.4 Evaluating automata-theoretic algorithms 14

2.2 Decision Diagrams 16

2.2.1 Binary Decision Diagrams (BDDs) 17

2.2.2 Zero-Suppressed Decision Diagrams (ZDDs) 18

3 ALASKA-ZDD 20

3.1 ALASKA 21

3.1.1 Algorithm 21

3.2 ALASKA-ZDD 25

4 Random Models 27

4.1 Random Models 28

4.2 Experiments - NFA 36

4.3 Experiments - Büchi Automata 40

4.3.1 Methodology 40

4.3.2 Results 41

5 Conclusion **47**

Bibliography **50**

Chapter 1

Introduction

Automata-theoretic formal verification is an approach to the problem of guaranteeing that a program (in software or hardware) conforms to its specification in which conformance is reduced to the problem of language containment. By representing both programs and specifications as automata and then proving that the specification contains the program, we can prove conformance [1]. This connection to automata theory motivated an extensive research program into the algorithmic theory of automata of infinite words, cf. [2], and the focus of this program is often on algorithms that perform well in practice, cf. [3]. The popularity of this format - due to the intuitiveness of specifications given in linear temporal logic and converted to automata - has also led to a demand for specification of finite properties. To accomodate this, the field has recently expanded to include finite-word automata.[4]

We focus here on the NFA *universality-checking* problem, a simplified case of containment checking, the canonical verification problem [1], which preserves the most difficult part of containment checking: constructing the complement. An automaton A is *universal* iff it accepts all input words; equivalently A is universal iff its complement $\overline{A}$ is *empty*, i.e., it accepts no input words. One way to check universality of A is to check emptiness of $\overline{A}$, which can be reduced to reachability analysis of $\overline{A}$'s state-transition graphs. Such approaches have to deal with the exponential blow-up of complementation, so extant algorithms for universality use a variety of heuristics to check emptiness of $\overline{A}$ without constructing it in full, cf. [5].

We focus on performance in practice because of the large gap between worst-case complexity and performance in practice for many automata-theoretic algorithms. For example, the universality problem for NFAs is known to be worst-case PSPACE-complete[6], but heuristic improvements allow many tools to perform much better in practice.

The tool ALASKA[7] implements one such heuristic NFA universality algorithm, using two primary techniques to efficiently represent automata complemented using the subset construction: 1) binary decision diagrams (BDDs) and 2) subsumption. BDDs are a heuristically compact method of representing Boolean functions - by interpreting sets of states, or macrostates, as bitstrings, we can symbolically represent a set of states from a subset construction as a Boolean function. Subsumption is a process by which some 'smaller' elements (as determined by some preorder) are subsumed by other, 'greater' elements, when only the greatest elements affect the solution - allowing the algorithm to discard all but the greatest elements. In the case of the subset construction, supersets of states are subsumed in their subsets, and can be discarded from consideration. The overhead for subsumption is high, and the operation is often too expensive unless removing many elements. But by discarding large amounts of unnecessary macrostates and efficiently representing the rest in one data structure, ALASKA performs efficient reachability analysis on NFAs.

The successful use of subsumption in ALASKA raises an interesting quetion. Since this results in a smaller set of macrostates, more sparse relative to the exponential complemented state space, could a symbolic representation optimized for sparse solution sets perform better than BDDs?

Fortunately, such a representation already exists - zero-suppressed decision diagrams, or ZDDs, which share semantics with BDDs but more efficiently represent sparse solution sets, at the cost of poorly representing dense ones. Using ZDDs, we should be able to efficiently represent the sparse sets created by subsumption.

Therefore, we introduce ALASKA-ZDD - a complete overhaul of ALASKA's universality-checking tool to internally use ZDDs for symbolic representation of automata, including adding support for new ZDD operations from the EXTRA library

to PyCUDD. While preserving identical semantics to ALASKA, the adaption to a ZDD-based algorithm changes the heuristics of the tool hoping to increase performance in practice.

Unfortunately, the quest for automata-theoretic algorithms that perform well in practice is hampered by a shortage of benchmark instances of automata that arise in industrial verification. To overcome this challenge, Tabakov and Vardi proposed a model for generating random automata on which different algorithms can be evaluated and compared [8, 9]. The model has three parameters: (1) the size (number of states) of the automaton, (2) the *density* of transitions (ratio of transitions to states), and (3) The *density* of accepting states (ratio of accepting to total number of states). Subject to these parameters, the model generates automata randomly. The Tabakov-Vardi (T-V, for short) model is attractive for NFAs [8] for two reasons: First, the model gives rise to an interesting *universality terrain*, which describes the relationship between the probability of automaton universality (which means that all input words are accepted) and the density parameters. Second, the model gives rise to an interesting *performance terrain*, which describes the relationship between algorithmic performance and the density parameters. (We discuss these two terrains in detail in the body of the paper.) In subsequent years, this model has become the standard model for the evaluation of complementation tools, cf. [10, 5, 11, 12].

We compare ALASKA and ALASKA-ZDD using the T-V model, and find that ALASKA-ZDD is slower than ALASKA, but more consistent - with a slowdown in average time taken, but fewer results overrunning their maximum allowed time.

The T-V model, however, is just one specific random model, based on a specific, and quite simple model of random graphs [13]. Several other models of random graphs have been studied over the years [14, 15]. While the T-V model has the advantage

of simplicity, it is not a priori clear that performance analyses conducted on this model are robust, as it is entirely possible that analogous analyses over other random models would yield different conclusions. Since performance analyses over random models are used in this context as a substitute to such analyses over a benchmark suite of real-life problem instances, it is desirable at least to know whether analyses over random models yield robust conclusions - both for our own work, and for the entire body of other work using the Tabakov-Vardi model!

To address this problem, we introduce three* novel models of structured random automata, based on existing random graph models – the *vertex-copying* model [14], the *Frank-Strauss* model [15], and the *co-accessible* model [16]. These models are based on different models that have been proposed for random graphs. While the T-V model is uniformly random, generating unstructured automata, these new models constrain randomness in some way to provide structural guarantees about the resulting automata: The vertex-copying model guarantees a power-law degree distribution, the Frank-Strauss model restricts which transitions are valid, and the co-accessible model guarantees that each state in the resulting automaton can reach an accepting state.

These structural properties help the models represent a wide variety of possible types of problem instances that might be encountered in the real world. Furthermore, these model generate problem instances that are quite unlikely to be generated by the T-V model. Our goal is to compare performance analysis on the T-V model against performance analysis on the three new models. If performance analyses on the a variety of different models all reach similar conclusions, then we can conclude that these

*Preliminary results for other models deemed too similar for inclusion can be found in the appendix.

conclusions are likely robust. If, on the other hand, performance analyses on different models reach different conclusions, then we would gain a deeper understanding of how structure affects algorithmic performance and learn that the choice of algorithm should depend on the structure of the problem instance being solved.

By generating large corpora of random automata and checking how likely they are to produce universal automata, we first show that the new models possess the same useful properties for universality as the T-V model. We then replicate our earlier results using all four models, showing that even when tested with other models, ALASKA-ZDD remains competitive with ALASKA for NFA universality, and the results of the Tabakov-Vardi model are robust. Additionally, we perfom tool comparisons with our new models on another problem to demosntrate robustness - comparing the tools Rank[5], Ramsey [5], and RABIT 2.3[†] for Büchi universality. These experiments found that the robustness of Tabakov-Vardi results across models holds for multiple problems, and that the more modern RABIT solver is vastly superior to older options.

[†]`http://languageinclusion.org/doku.php?id=tools`

Chapter 2

Background

2.1 Automata theory

2.1.1 Types of automata

NFAs A *non-deterministic finite state automaton*, or NFA, is a tuple $A = (\Sigma, Q, Q_0, \delta, F)$, where Σ is a finite alphabet, Q is the finite set of states, $Q_0 \subseteq Q$ is the set of initial states, $\delta : Q \times \Sigma \to 2^Q$ is the transition function, and $F \subseteq Q$ is the set of accepting states. NFAs take words from Σ^* as input. A *run* of an NFA on a word $w_0, w_1, ...w_n \in \Sigma^*$ is a finite sequence $q_0, q_1, ...q_n \in Q^*$ such that $q_0 \in Q_0$, and $(\delta(q_i, w_i) = q_{i+1})$. A run is *accepting* if the last state in the run is accepting - that is, $q_n \in F$. An NFA A accepts a word w if there is some run of A on w that is accepting. The language of A, or $L(A)$, is the set of all words that A accepts.

Büchi automata A *Büchi automaton* is a variant of an NFA that takes infinite words from Σ^ω as input. A run of a Büchi automaton is on an infinite word $w_0, w_1, ... \in \Sigma^\omega$, and is an infinite sequence $q_0, q_1, ... \in Q^\omega$ of states, obeying the same rules as a run of an NFA. A run of a Büchi automaton is accepting if some accepting state $q_i \in F$ occurs infinitely often in the run.

Alternating automata An *alternating automaton*, or AFA, is a more powerful variant of an NFA that can visit multiple states at once, rather than only visiting one state at a time in each run. We use a symbolic variant of AFAs, originally defined by De Wulf et. al.[17] To accomplish this, it uses a more powerful transition function, where the set of next states is determined using a positive Boolean formula.

Given a set of propositions P, we define $Lit(P) = P \cup \{\neg p \mid p \in P\}$ to be the set of literals over P, and $B^+(P)$ to be the set of positive boolean formulae over P - formulae built from elements in $P \cup \{true, false\}$ using Boolean connectives $\wedge$ and

$\vee$. Given $R \subseteq P$ and $\phi \in B^+(P)$, we write $R \models \phi$ iff the truth assignment assigning *true* to the elements of R and false to the elements of $P \setminus R$ satisfies ϕ.

An alternating automaton is a tuple $A = (P, \Sigma, Q, I, \delta, F)$ consisting of a set of atomic propositions P, an alphabet $\Sigma = 2^P$, a set of states Q, a set of possible initial states I given by a logical formula in $B^+(Q)$, a transition function $\delta : Q \rightarrow B^+(Lit(P) \cup Q)$, and a set of accepting states F. A run of A on a finite word $w = \sigma_0, \sigma_1, \sigma_2, \ldots, \sigma_{n-1} \in \Sigma^*$ is a directed acyclic graph (DAG) $G = (V, E)$ such that -

- For each $v \in V$, $Lvl(v)$ is the length of the path from an initial node to v.
- The set $S \subseteq V$ of initial nodes satisfies I - i.e., $S \models I$.
- For each $v \in V$ and the set $V' = \{v' \mid (v, v') \in E\}$ of successors to V, $V' \cup \sigma_{Lvl(v)} \models \delta(v)$.
- If $(v, v') \in E$, then $Lvl(v) + 1 = Lvl(v')$.

A run of A on w is accepting iff all paths $p = s_0, s_1, \ldots, s_i$ through G where s_0 is an initial node and s_i has no successors are accepting. A path p is accepting iff $\sigma_i \models \delta(s_i)$, or $i = n$ and $s_i \in F$.

Intuitively, an alternating automaton takes a truth assignment as input, and accepts if certain subsets of possible nondeterministic runs through the automaton are accepting - choosing nondeterministically on $\vee$, and requiring all successors to accept on $\wedge$. It accepts if all the nondeterministic runs considered in an alternating run accept - with all extant runs ending in accepting states after the input, and allowing some runs to terminate early if the input satisfies their transition.

2.1.2 Treating NFAs as AFAs

While we will describe ALASKA's algorithm in Section 3.1 in terms of NFAs, ALASKA's algorithm accepts alternating automata, AFAs, as input. Therefore, we must convert NFAs into AFAs to use them as input. For simplicity, we consider here the special case where the alphabet Σ of the NFA is $\Sigma = 2^P$ for some set P of atomic propositions, that is, a truth assignment to P. An equivalent translation can be acquired for other NFAs using an injective function $f : \Sigma \rightarrow 2^p$.

Converting an NFA $A = (2^P, Q, Q_0, \delta, F)$ to an AFA $A' = (2^P, Q, I, \delta', F)$ that accepts the same language is fairly simple. Note that both A and A' use $2^P, Q$, and F, needing no conversion.

Recall that AFAs transition on a positive Boolean formula, rather than a direct transition function. We convert NFAs to AFAs by making sure that for each run of the NFA, there is a corresponding run of the AFA. To do this, we can ensure that if $q' \in \delta(q, \tau)$, then $\tau \cup q' \models \delta'(q)$, and that $\forall q_0 \in Q_0 : q_0 \models I$. Since this ensures each possible transition in A satisfies the transition function in A',and that for each state $q_0 \in Q_0, q_0 \models I$, if a run exists in A, then it also exists as a path in a run of A' with only a single path. This is because, by construction of δ', one path is sufficient to satisfiy δ' at each transition - more are unnecessary. We construct such an I and δ' and as follows:

$$I = \bigvee_{q_0 \in Q_0} q_0$$

$$\forall q \in Q : \delta'(q) = \bigvee_{\tau \in 2^P} (\tau \wedge \bigvee_{q_i \in \delta(q, \tau)} q_i)$$

Note that if there is no transition $\delta(q, \tau)$, then $\delta'(q)$ includes $\tau \wedge false$, since $\bigvee$ becomes false when there is nothing in the disjunction.

Since each run of A also exists in A', and $F = F'$, if there exists an accepting run of A on a word $w = \tau_0, \tau_1, \ldots, \tau_{n-1}$, then there is an accepting run of A' on w. By the construction of δ', an accepting run of A' must always transition from a state q on some τ_i to at least one q' such that $q' \in \delta(q, \tau_i))$. Since this holds for every transition, there must always be at least one path in a run of A' on w that is equivalent to a valid run of A. Since all paths must be accepting for a run of A' on w to accept, if A' has an accepting run of w, then one of its paths is an accepting run of A. Therefore, if there is an accepting run of A' on w, there is an accepting run of A on w. Therefore, A accepts w iff A' accepts w.

2.1.3 Languages and complementation

The set of all words an automaton A accepts is called the *language of* A, or $L(A)$. A *complement* $\overline{A}$ of an automaton A is an automaton whose language is $\Sigma^\omega \setminus L(A)$. Finding the complement of an automaton is called *complementation*.

An automaton A is *contained* in an automaton B when $L(A) \subseteq L(B)$. In automata-theoretic verification [1], we prove that a program satisfies a specification by modeling the program as a automaton A and the specification as a automaton B, and then proving that A is contained in B. To check this containment, we check that the intersection of $L(A)$ with $L\left(\overline{B}\right)$ is empty. If it is not empty, then a word in the intersection is a trace of A that violates the specification B. In practice, efficient containment algorithms do not explicitly construct the complement $\overline{B}$, using instead various strategies for on-the-fly complementation and symbolic construction, cf. [5]. Nevertheless, because these strategies are still fundamentally based on complementation, there is a close link between the efficiency of complementation and the efficiency of containment. Since the hard step in containment checking is the need to

construct (at least implicitly) $\overline{B}$, papers on the subject, e.g. [5, 8, 9], usually focus on *universality checking*, where $L(A) = \Sigma^*$ or Σ^ω– that is, checking if $L(B)$ contains the set of all words, by checking if $L(\overline{B})$ is empty.

Complementing alternating automata In Chapter 3, we use ALASKA's AFA emptiness checker to verify the universality of NFA. As described in Section 2.1.2, we can convert NFAs to AFAs easily. However, to use an emptiness checker, we must complement the resulting AFA. This is a fairly straightforward procedure - converting to an AFA moves a difficult step of NFA complementation, namely determinization, to the emptiness checking step, leaving complementation much simpler. We consider specifically the case of an AFA constructed from an NFA, with a transition function of the form

$$\delta(q) = \bigvee_{\tau \in 2^P} (\tau \wedge \bigvee_{q_i \in f(q,\tau)} q_i)$$

for some function $f : Q \times 2^P \to 2^Q$, and an initial constraint of the form

$$I = \bigvee_{q_0 \in Q_0} q_0$$

for some $Q_0 \subseteq Q$.

An AFA $A = (2^P, Q, I, \delta, F)$ accepts a word w iff there exists a run of A on w such that every path is accepting. Therefore, $\overline{A} = (2^P, \overline{Q}, \overline{I}, \overline{\delta}, \overline{F}$ accepts w iff every run of A on w has some path that is not accepting.

Intuitively, an $\vee$ in the output of the transition function represents a split into multiple runs, and an $\wedge$ represents the split of a run into multiple paths. By De Morgan's laws, by negating an $\vee$ we can acquire an $\wedge$, and vice versa. Therefore, to check some path in every run, rather than every run in some path, we negate outputs

of the transition function, each run of $\overline{A}$ choosing nondetermnistically between the paths in runs of A to find paths that do not accept.

Thus, we define $\overline{\delta}(q) = \neg\delta(q) : q \in Q$. We define $\neg q = \overline{q} : q \in Q, \overline{q} \in \overline{Q}$ - i.e., a state from A becomes the equivalent state from $\overline{A}$.* We equivalently define $\overline{I} = \neg I$. After fully resolving De Morgan's, the products are of the form:

$$\overline{I} = \bigwedge_{q_0 \in Q_0} \overline{q_0}$$

$$\forall \overline{q} \in \overline{Q} : \overline{\delta}(\overline{q}) = \bigwedge_{\tau \in 2^P} (\neg\tau \vee \bigwedge_{q_i \in f(q,\tau)} \overline{q_i})$$

Note that $\neg\tau \vee \bigwedge_{q_i \in f(q,\tau)} \overline{q_i} = \tau \rightarrow \bigwedge_{q_i \in f(q,\tau)} \overline{q_i}$ - implicitly, at each point where the original NFA $A_N$would have transitioned to one state, $\overline{A}$ transitions to all states. Also, note that if $f(q,\tau)$ is empty, then $\bigwedge_{q_i \in f(q,\tau)} = true$, resulting in $\neg\tau \vee true$ - satisfying the formula if the input is τ, since all other components of the conjunction are satisfied by it being a different truth assignment.

Therefore, a run of $\overline{A}$ on a word w has a path for every run of A_N on w. These paths are accepting if they end early - when they would not have been accepting runs. However, they will end in the same states that runs of A_N would have, and be accepting if the original runs were accepting. Since we wish to accept only if no run of A_N was accepting, we define $\overline{F} = \{\overline{q} \mid q \notin F\}$. Thus, a run of $\overline{A}$ on w contains a path for each run of A_N on w, and accepts iff all paths either end early, or end in states that were not accepting states in A_N or in A. Therefore, $L(\overline{A}) = \overline{L(A)}$.

*Recall that δ is defined over the literals of the propositions, but not of the states - states cannot be directly negated.

Using AFAs in ALASKA The input format to ALASKA can be found in full at [18]. It accepts as input a Python dictionary, with assignments to the values *automaton_type*, *locations*, *propositions*, *initial_constraint*, *transition_function*, and *accepting_locations*. *automaton_type* = "*sAFW*", *locations* is an array of integers which are used as the states of the automaton, *propositions* is an array of strings, *initial_constraint* is a string consisting of states separated by || and &, *transition_function* is a second dictionary from the states to strings of states and propositions using ~, ||, and &, and *accepting_locations* is a set of integers.

2.1.4 Evaluating automata-theoretic algorithms

The quest for automata-theoretic algorithms that perform well in practice is hampered by a shortage of benchmark instances of automata that arise in industrial verification. The automaton B above corresponds to a formal specification of intended design functionality. Industrial specifications are typically proprietary and not openly available. To overcome this challenge, Tabakov and Vardi (T-V) proposed a model for generating random automata on which different algorithms can be evaluated and compared [8, 9]. In subsequent years, this model has become the standard model for the evaluation of automata-theoretic tools, cf. [10, 5, 11, 12].

The T-V model generates automata using the *uniformly random* choice of elements from a set. The T-V model takes three parameters - an integral *size* n, a positive real *transition density* r, and a real *accepting-state density* f between 0 and 1. The transition density is the average out-degree of each state in the result automaton per input symbol. The accepting-state density is the percentage of states in the result automaton that are accepting states. Formally, a (n, r, f) T-V random automaton is defined as follows. Each random automaton $A = (\Sigma, Q, Q_0, \delta, F)$ has the alphabet

$\Sigma = \{1, 0\}$ and set of states $Q = \{0, \ldots, n-1\}$. The set Q_0 of initial states is $\{0\}$. For each $\sigma \in \Sigma$, the model generates a digraph (directed graph) D_σ over the nodes $\{0, \ldots, n-1\}$ with $n * r$ edges chosen uniformly at random from the set of all possible edges $(u, v) \in Q \times Q$. The transition function δ is then defined as $\delta(u, \sigma) = \{v \mid (u, v) \in D_\sigma\}$. The accepting states F comprise $\lfloor n * f \rfloor$ states selected uniformly at random from Q. Note that each element of D_σ is a random digraph - specifically, a Karp [13] random digraph. Thus, we say that the T-V model *lifts* the Karp model of random digraphs into automata.

The T-V model is attractive for performance evaluation for two reasons [8, 9]: First, the useful properties of its *universality terrain*, which describes the relationship between the probability of automaton universality (which means that all input words are accepted) and the density parameters. When transition and accepting-state densities are low, the probability for universality is low, while at higher densities the probability steadily increases. Thus, the model provides a way to evaluate the performance of universality-checking algorithms on both universal and non-universal automata. We call a model "*interesting*" when its universality probabilities vary with the input parameters and increase from low to high probability. Second, the model gives rise to an interesting *performance terrain*, which describes the relationship between algorithmic performance and the density parameters. Specifically, at low and high densities universality checking is easier than at intermediate densities. Thus, the model provides a way to evaluate the performance of universality-checking tools on both easy and hard problems. We take these two features, universality terrain and performance terrain to be desiderata that we expect to have in other models of random automata.

2.2 Decision Diagrams

A *decision diagram* is a canonical method of representing Boolean functions with rooted, edge-labelled directed acyclic graphs [19]. It consists of *internal vertices*, which correspond to the variables of the function, and *terminal vertices*, which correspond to the output values 0 and 1. A decision diagram is deterministic, and every internal vertex v has exactly two outgoing edges - $(v, 0, v_0)$ and $(v, 1, v_1)$. We often abbreviated these edges as $LO(v)$ and $HI(v)$ respectively, and extend that abbreviation to refer to v_0 and v_1 when referring to nodes instead of edges.

Let f be a Boolean function, defined over the tuple of variables $var(f)$. A decision diagram G is a tuple $(V, E, \rho, <_G)$. G consists of a set $V = I \cup \{0, 1\}$ of vertices where I are internal vertices and { 0,1 } are terminal vertices, a set E of labelled edges $(v, \sigma, v') : v, v' \in V, \sigma \in \{0, 1\}$, a labelling function $\rho : V \to var(f)$, and a total order $<_G$ over $var(f) \cup \{0, 1\}^\dagger$.

No terminal vertices v_T have outgoing edges $(v_T, \sigma, v') \in E$.

G is ordered by $<_G$. For every $(v, \sigma, v') \in E$, it holds that either $v_2 \in \{0, 1\}$, or $\rho(v_2) <_G \rho(v)$. Thus, an edge can only progress to an internal vertex with a lower ranked label, or a terminal vertex. Since all internal vertices must have a HI and a LO, and rank is both finite and constantly-decreasing, every path must eventually terminate in a terminal vertex. The variable order chosen can greatly impact the size of the resulting decision diagram[20].

All decision diagrams use *reduction rules* to eliminate unnecessary vertices and produce a canonical form. One reduction rule is shared by all decision diagrams - elimination of identical nodes. If $\exists v_1, v_2 \in V$ s.t. $\rho(v_1) = \rho(v_2), HI(v_1) = HI(v_2), LO(v_1) =$

---

†We assume without loss of generality that $var(f)$ is disjoint from $\{0, 1\}$, and that I is disjoint from $\{0, 1\}$.

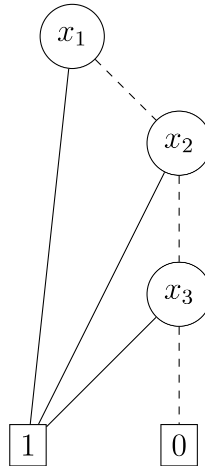

Figure 2.1 : The DD that represents the logical or of three variables if interpreted as a BDD, or the logical xor of three variables if interpreted as a ZDD. Note that this means precisely that the interpretation where missing nodes are true leads to 0 in the ZDD, but not in the BDD.

$LO(v_2)$, then remove v_2 from V, and redirect all incoming edges $(v, \sigma, v_2) \in E$ to (v, σ, v_1). Other rules vary based on the decision diagram used.

Decision diagrams represent the computation of a Boolean function as paths from the root of the diagram to a vertex. When tracing such a path through the diagram, the set of edges followed - and thus, the states reached - yields an assignment τ to $var(f)$, and the terminal vertex reached yields $f(\tau)$. We interpret $HI(v)$ as an assignment of 1 to $\rho(v)$ and $LO(v)$ as assignment of 0. For example, in Figure 2.1, if we follow the path LO, LO, HI, then we assign $x_1 = x_2 = 0$ and $x_3 = 1$, and this is an assignment that satisfies f.

Not all paths will have an assignment to all variables. There are multiple ways to interpret an unassigned variable. This depends on the kind of decision diagram used.

2.2.1 Binary Decision Diagrams (BDDs)

The most common kind of decision diagram is a binary decision diagram. In a BDD,

1. An unassigned variable on a path is interpreted as not affecting the output on that path - it can be freely assigned to true or false.

2. We use the new reduction rule: if $HI(v) = LO(v) : v \in V$, then remove v from V and redirect all incoming edges $(v', \sigma, v) \in E$ to $(v', \sigma, LO(v))$.

Since they eliminate nodes that are irrelevant to the outcome, BDDs are capable of efficiently representing the "don't-care" literal - the fewer variable assignments need to be known to determine the outcome, the more compact the BDD can be. A BDD is useful when a large number of similar variable assignments result in an output of 1, allowing its heuristics to more tightly compact the representation. In 2.1, we can see that BDDs are able to efficiently represent the or operation.

2.2.2 Zero-Suppressed Decision Diagrams (ZDDs)

Zero-suppressed decision diagrams are an alternative to binary decision diagrams, which are more efficient with sparse solution sets. In a ZDD,

1. An unassigned variable on a path is interpreted as an assignment to 0. If the variable would be assigned 1, then the output is 0.

2. We use the new reduction rule: if $HI(v) = 0 : v \in V$, then remove v from V and redirect all incoming edges $(v', \sigma, v) \in E$ to $(v', \sigma, LO(v))$.

Since they eliminate nodes which would falsify the output if true, ZDDs are capable of efficiently representing the "false" literal - if fewer variable assignments output 1, then the ZDD can be more compact. A ZDD is most useful when very few variable assignments result in 1, and many of them are different - satisfying assignments are scattered sparsely across the solution space. In these cases, its heuristics can more

tightly compact the representation. In 2.1, we can see that ZDDs are able to efficiently represent the xor operation.

Chapter 3

ALASKA-ZDD

3.1 ALASKA

Antichains for **L**ogic, **A**utomata and **S**ymbolic **K**ripke Structures **A**nalysis, or ALASKA[21], is a broad-ranging Python tool for automata-theoretic verification, implementing a number of algorithms including emptiness of finite and infinite word automata, satisfiability and validity of LTL formulae, and model checking of LTL specifications. In this work, we focus on its tool for testing emptiness of alternating automata using backwards reachability. We can fairly trivially use this to check the universality of NFAs, by complementing them as alternating automata as described in Sections 2.1.2 and 2.1.3. (Recall that the complement is empty iff the automaton is universal.) A more complete description of the algorithm and proof of it is available at [22].

3.1.1 Algorithm

Consider an AFA $A = (2^P, Q, I, \delta, F)$, which we wish to check for emptiness. We prove that A is empty by using the subset construction to show a lack of backwards reachability - that from the final states, we cannot reach the initial states. We use backwards reachability since previous results have found backwards reachability to be more efficient than forward reachability[22].

We implicitly use the subset construction on A to produce an NFA $A' = (2^P, 2^Q, 2^Q_0, \delta', 2^F)$, where 2^Q_0 is the set of all sets of states that satisfy I, and for $S \in 2^Q$ and $p \in 2^P$, $\delta'(S,p) = S'$ such that $S' = \{s' \mid s \in S, s' \in \delta(s,p)\}$. It is well-known that the subset construction does not change the language of an automaton.

Beginning in the set of macrostates 2^F, we recursively visit predecessor macrostates of our current set of macrostates SS as defined by the operation $PRE(SS) : 2^{2^Q} \to 2^{2^Q} = \bigcup_{S \in SS} PRE(S)$, where $PRE(S) : 2^Q \to 2^{2^Q} = \bigcup_{p \in 2^P} \{\{s | \delta(s,p) = S\}\}$ - that is, the union of the predecessors on each possible set of propositions for each

macrostate within SS.

By recursively visiting the predecessors, the algorithm will eventually visit every backwards-reachable state within the state space. If at any point there exists a $Q_0 \in 2^Q_0$ s.t. $Q_0 \in SS$, then the initial states are backwards-reachable from the final states. Since every backwards transition was equivalent to some forward transition on an assignment p to the propositions, this means the final states can also be reached from the initial states.

By adding a history of previously-visited macrostates and removing those that have been visited at each step, we can ensure that the algorithm eventually terminates when all backwards-reachable states have been reached.

Subsumption The macrostate space of the subset-constructed automaton is exponential in the size of the original automaton. If many of the macrostates are reachable, then finding the reachable macrostates becomes extremely inefficient - reducing the number of macrostates under consideration can therefore offer significant performance improvements. Other approaches attempt to minimize the size of the automaton [21]. We instead take advantage of subsumption properties of the subset construction - by showing that the answer does not change when considering only a subset of the macrostates, we can discard unnecessary macrostates.

The relationship between the macrostates of A' and the states of A creates a *subsumption relation* over the macrostates of the A', specifically $\subseteq$. A macrostate $S_1 \in 2^Q$ subsumes a macrostates $S_2 \in 2^Q$ iff $S_2 \subseteq S_1$. This means that S_2 provides no information not provided by S_1 for emptiness checking, or more specifically, about the reachability of an accepting path. We provide a sketch of the proof, found more completely using simulation relations in [17] and [22].

Consider two macrostates $S_1, S_2 \in 2^Q$ s.t. S_1, S_2 are backwards reachable and $S_1 \subseteq 2_1$. Then, because of the properties of the subset construction as proved in [22], if $\exists C \in 2^{2^Q}, p \in 2^P$ s.t $C \in \delta'(S_3, p)$ and $S_1 \in C$, then there exists S_4 s.t. $S_2 \in C'$ for all $C' \in \delta'(S_4, p)$, and $S_3 \subseteq S_4$. By repeated application, we can show that for any arbitrary sequence of predecessor macrostates of S_1 such that S_i is a predecessor of S_1, there exists a predecessor S_j of S_2 such that $S_i \subseteq S_j$.

A' is nonempty iff 2^Q_0 can be reached. We show that if S_1 can reach an accepting macrostate, then so can S_2.

Consider the case where $S_i \cap 2^Q_0 \neq \emptyset$. Since $S_i \subseteq S_j$, $S_j \cap 2^Q_0 \neq \emptyset$. Note that S_i is a predecessor of S_1, and S_j is a predecessor of S_2. Therefore, if S_1 can reach the initial set, then S_2 can reach the initial set. Since we only care about the presence of an accepting path, S_2 provides all the information that S_1 does for emptiness checking.

Given this, we can successfully check emptiness using *antichains* - a canonical maximal representation of the set of reachable macrostates, which only contains the largest subsets. We can compute the antichain S' of a set R of reachable macrostates by discarding all macrostates S_1 s.t. $\exists S_2 \in R : S_1 \subset S_2$. While this operation can be very expensive, it improves efficiency of emptiness checking when discarding a large number of macrostates. By taking the antichain of the newly-reached macrostates each time we visit a new set of macrostates, we can improve our original algorithm.

Semi-Symbolic Representation Fully representing the antichain and performing operations on it is still quite expensive, due to the exponential size macrostate space. Because of this, explicitly constructing the complemented automaton is extremely inefficient. ALASKA approaches this problem by symbolically representing sets of macrostates using BDDs.

Assume without loss of generality that $Q = \{0, 1, \ldots, n-1\}$. Consider a bitstring $B = b_1, b_2, \ldots, b_{n-1}$ of length $|Q|$. B can be interpreted as a macrostate $v \in 2^Q$, where $\forall q \in Q$, $b_q = 1 \leftrightarrow q \in v$. Call this bitstring $B(v)$. Therefore, a Boolean function $f(x_1, x_2, \ldots, x_{n-1})$, which takes as arguments bitstrings representing macrostates, represents a set S of macrostates - $\forall v \in 2^Q : f(B(v)) = 1 \leftrightarrow v \in S$.

These functions can track the all sets of macrostates in our algorithm, such as the initial set 2^Q_0 and the set of visited states. Since BDDs are semantically equivalent to Boolean functions, we can efficiently symbolically represent these values, as well as the transition relation, using BDDS, considerably speeding up useful set operations, such as checking the intersection of the visited set with 2^Q_0.

Nevertheless, we do not use a fully symbolic construction. We instead convert BDDs back into states when visiting a new set of states - while the input and output of this operation are both BDDs, we convert S into an explicit set of macrostates, then find their successors as BDDs. This *semi-symbolic* is more efficient than the fully symbolic approach, which requires the BDDs to contain variables representing macrostates to compute the set of successors. Thus, the number of variables needed grows logarithmically with the number of macrostates in the semi-symbolic approach, while it grows linearly in the fully symbolic approach[17].

Using all of these, we produce the following algorithm, with the working set $FRONTIER$ and visited set $VISITED$ both initialized to 2^F:

1. $FRONTIER \leftarrow PRE(SUBSUMPTION(FRONTIER))$
2. $FRONTIER \leftarrow FRONTIER \setminus VISITED$
3. $IF\ FRONTIER = \emptyset, RETURN\ TRUE$
4. $IF\ FRONTIER \cap 2^Q_0 \neq \emptyset, RETURN\ FALSE$

5. $VISITED \leftarrow VISITED \cup FRONTIER$

All operations are symbolic and all values are represented as BDDs, except inside of PRE, which converts the BDD input to an explicit representation of the current working set and finds its successors.

3.2 ALASKA-ZDD

By using subsumption, ALASKA minimizes the working set, and the number of macrostates represented. Therefore, after subsumption, the minimum possible number of inputs to the decision diagram produce 1. ZDDs are efficient when representing functions with few outputs of 1, raising the natural question of whether ZDDs can outperform BDDs for ALASKA. We implement a new overhaul of ALASKA, ALASKA-ZDD, which replaces all BDD operations in ALASKA with equivalent ZDD operations, and compare it to ALASKA. All set operations in ALASKA's algorithm, and all values represented with BDDs, are replaced with ZDDs. This replaces almost every value in the algorithm, since it is symbolic except within PRE.

Like ALASKA, ALASKA-ZDD is a Python tool built on top of the DD library PyCUDD[23], a Python wrapper for the C DD library CUDD[24]. In addition to direct replacement of BDD operations with ZDD semantic equivalents, we implemented operations that would replace BDD operations for which there was no direct ZDD equivalent, or the ZDD equivalent was inefficient, while preserving ALASKA's overall semantics. We also fixed a long-standing bug preventing PyCUDD from functioning with ZDDs, and we implemented a Python wrapper for the EXTRA ZDD extension library for CUDD[25], integrating new ZDD operations into PyCUDD when equivalent operations existed, but were not already implemented in CUDD.

We compare the performance of ALASKA and ALASKA-ZDD experimentally in Section 4.2.

Chapter 4

Random Models

4.1 Random Models

Our goal in this work is to compare the T-V model to other models of random automata as a framework for evaluating the performance of universality-checking algorithms. We take advantage of the fact that the Tabakov-Vardi technique of *lifting* digraphs into automata is not limited to Karp random digraphs. By substituting other random-digraph models, we can generate new models of random automata. Since Büchi automata are identical to NFAs except for the interpretation of final states, these models can generate both.

The Tabakov-Vardi lifting is as follows. A random automata model that lifts a random digraph model has all of the parameters of the digraph model, plus an accepting-state density parameter f. Each random automaton is a tuple $(\Sigma, Q, Q_0, \delta, F)$, with the elements defined as follows. We take the alphabet $\Sigma = \{0, 1\}$ for all models. For each character $\sigma \in \Sigma$, create a random digraph D_σ using the digraph parameter values of the automaton model. The set Q of states of the random automaton is equivalent to the set N of D_σ's nodes, usually $N = \{0, \ldots, n-1\}$, where n is the size parameter. The initial state set $Q_0 \subseteq Q$ is a singleton set containing one state from Q, usually 0. The transition relation δ is the union of all sets $\{(q, \sigma, r) \mid (q, r) \in D_\sigma\}$ for $\sigma \in \Sigma$ - equivalently, δ can be considered a transition function where $\delta(q, \sigma)$ is the set of all r related to q by σ in the relation. Finally, the set $F \subseteq Q$ of accepting states consists of $\lfloor |N| * f \rfloor$ elements of Q chosen uniformly at random (without repetition). Not all models we study use the Tabakov-Vardi lifting; see details below.

In the rest of this section, we introduce three* new models based on this lifting - the *vertex-copying* model, the *Frank-Strauss* model, and the *co-accessible* model. The first two models are based on existing models of structured random digraphs which

*Other models can be found in the appendix.

have found common use in other disciplines, and the co-accessible model guarantees a particular automaton property. While the lack of existing benchmarks makes it difficult to compare these models directly to industrial problem instances, we can use a variety of structured random models to more fully explore the problem space. If these models disagree with the Tabakov-Vardi model, then the T-V model is not rich enough to fully represent the space on its own – if they agree, then it is likely that the conclusions of the T-V model are quite robust.

We show each of the models to have a Büchi universality terrain that is somewhat similar but not identical to that of the T-V model, using experiments run on the DAVinCI cluster[†] at Rice University. To show that each model has an interesting universality terrain, we present with each model a terrain plot showing how likely the Büchi automata generated by the model are to be universal when made with certain parameters. We generated and tested 100 automata using the parameters at each point on the plot. The universality terrains show that that the random models we introduce generate automata whose likelihood of being universal ranges from 0 to 1, just as in the T-V model.

Vertex-Copying Automata The random vertex-copying model presented here is a simplification of the model defined by Kleinberg *et al.* [14]. A vertex-copying digraph starts out as an empty set of nodes, and adds edges over time. By sometimes choosing edges at random, and at other times copying edges from one node to another, it creates a heavy-tailed distribution – a "rich get richer" effect as nodes with many edges steadily gain more and more edges. This copying is intended to model hyperlinks on the Web – links are often created when someone discovers a link to a

[†] `http://www.rcsg.rice.edu/sharecore/davinci/`

site they're interested in on another site, then adds a link to it on their own website, thus "copying" the link from one site to another. This approach may also model code reuse - when a code block is reused, then calls to functions are duplicated.

An (n, b, r) vertex-copying random digraph takes as parameters the *size* n, the *copying probability* b, and the *transition density* r. The vertices are $\{0, \ldots, n-1\}$. The model begins with no edges and adds edges (u, v) to the graph one at a time until there are $\lfloor n * r \rfloor$ edges. Each time it does so, it has a probability b of copying an edge from one node to another, and a probability $1 - b$ of simply generating an edge uniformly at random. If it copies, then it chooses an edge $(u, v) \in E$ and a node $u' \in V \setminus u$ uniformly at random. It then adds (u', v) to E. If it generates the edge at random, it acts as in the T-V model. This digraph model extends to automata by directly using the standard lifting. Its universality terrain is given in Figure 4.1.

Frank-Strauss Automata The Frank-Strauss random graph model, based on an approach by Frank and Strauss[‡] [15], limits the space of possible edges. Instead of the vertices being integers, vertices are unordered pairs of integers. The Frank-Strauss model permits edges only between vertices that share an element – the vertex $(0, 1)$ can connect to $(0, 3)$ and $(1, 3)$, but not to $(2, 3)$. Within this space, edges are generated uniformly at random. The Frank-Strauss model can represent systems that require some relationship between actors. For example, it can be used to represent binary relationships between individuals in a social setting. Alternatively, we may have a program such that if one module calls another, then there must be some relation between them – for example, operating on shared data.

An (l, r) Frank-Strauss random graph takes as parameters a *label size* l and a

[‡]Referred to in their paper as a "Markov graph".

transition density r. The set V of vertices is the set $\{(i, j) \mid i, j \in 0, \ldots, l-1\}$ of unordered pairs of elements. Since we allow the case where $i = j$, there are $\binom{l+1}{2} = \frac{l(l+1)}{2}$ such vertices. We generate $\lfloor |V| * r \rfloor$ edges. To generate each edge, first choose a vertex (u_1, u_2) uniformly at random as the source, and then choose a vertex $(v_1, v_2) \in \{u_1, u_2\} \times \{0, \ldots, l\}$ uniformly at random as the destination. This digraph model extends to automata directly by using the standard lifting. The universality terrain is presented in Figure 4.2.

Co-accessible Automata The co-accessible model of random automata is so named because it guarantees that the resulting automata are co-accessible, where an automaton is *co-accessible* if all states $q \in Q$ are co-accessible, that is, can reach an accepting state. Because this property is meaningful only for automata, the co-accessible model cannot be based on lifting a model of random digraphs. It is loosely based on Leslie's generation of connected automata [16]. Automata possessing this property correspond to useful program properties – for example, a co-accessible automaton may specify that the program can recover and perform its intended function from every state.

The co-accessible model takes as parameters a *size* n, a *transition density* r, and an *accepting state density* f. The co-accessible model does not define the transition relation based on an underlying digraph. Instead, we start with a set $Q = \{0, \ldots, n-1\}$ of states and initial and accepting state sets Q_0 and F as in the T-V model. The transition relation δ is initially empty.

To fill in δ, we construct a random spanning inverted forest over Q. This is a set of trees over the automaton which contains every state, each rooted at an accepting state, and where edges go from children to parents instead of parents to children. A

forest can be found as follows: make a set of co-accessible states $C = F$ and states that are not yet co-accessible $U = Q \setminus F$, then select some $u \in U$, $c \in C$ and $\sigma \in \Sigma$ uniformly at random. Add (u, σ, c) to δ, then remove u from U and add it to C, repeating until U is empty.

Once the spanning forest has been constructed, the model must fill in the rest of the transition relation. It then ensures that each character $\sigma \in \Sigma$ is associated with exactly $\lfloor n * r \rfloor$ edges. If some σ_0 has more than $\lfloor n * r \rfloor$ transitions, replace random transitions (u, σ_0, v) with (u, σ_1, v) for $\sigma_0 \neq \sigma_1$ and $\sigma_1 \in \Sigma$. Then generate new edges uniformly at random, as in the T-V model, for each character with fewer than $\lfloor n * r \rfloor$ transitions. We assume $r \geq 1$. The universality terrain is given in Figure 4.3.

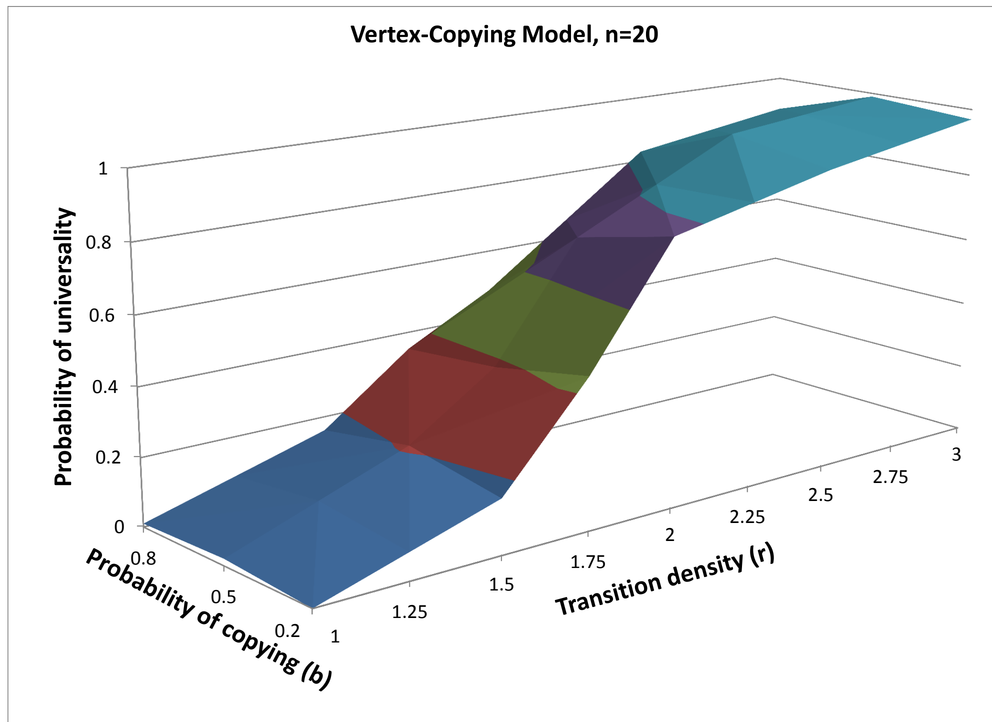

Figure 4.1 : A vertex-copying Büchi universality terrain for $n = 20$. The transition density r ranges from 1 to 3, and the copying probability b ranges from 0.2 to 0.8. The accepting-state density f was set to 0.3. The universality probability is comparable to that of the T-V model for most values of r. Note that increasing b does not monotonically increase universality probability – after a certain point it actually reduces it. This may be because all transitions go to a small number of states, with few transitions leaving them, increasing the likelihood of rejection.

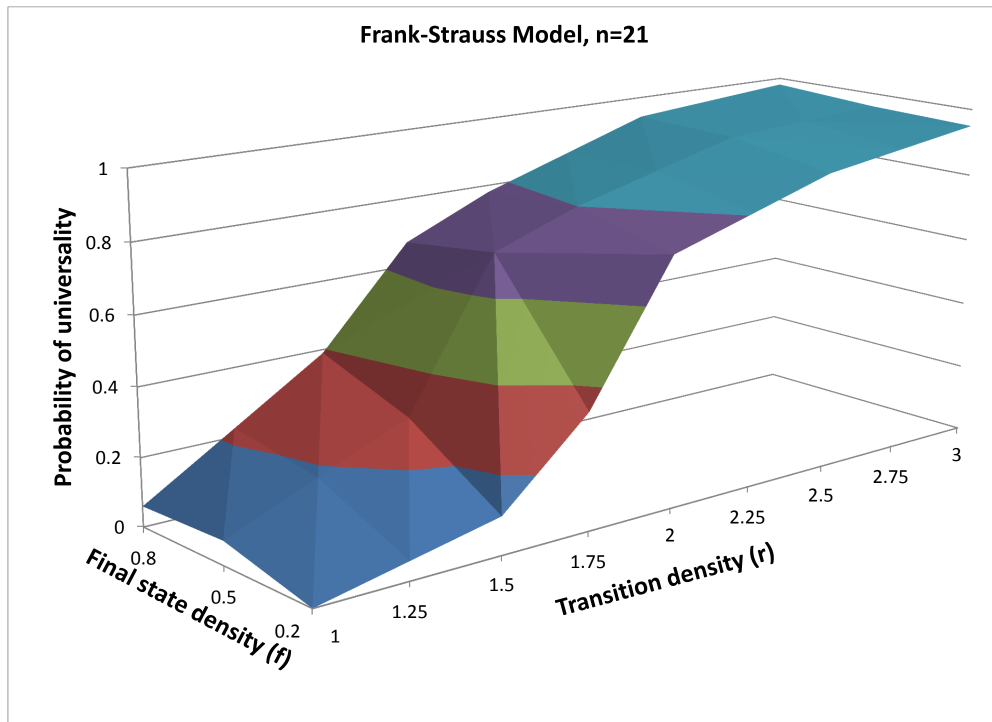

Figure 4.2 : A Frank-Strauss Büchi universality terrain for $l = 21$. r ranges from 1 to 3 and f ranges from 0.2 to 0.8. While the universality probably scales more quickly with r than in the T-V model, there are still a number of points where universality is neither nearly guaranteed nor always absent.

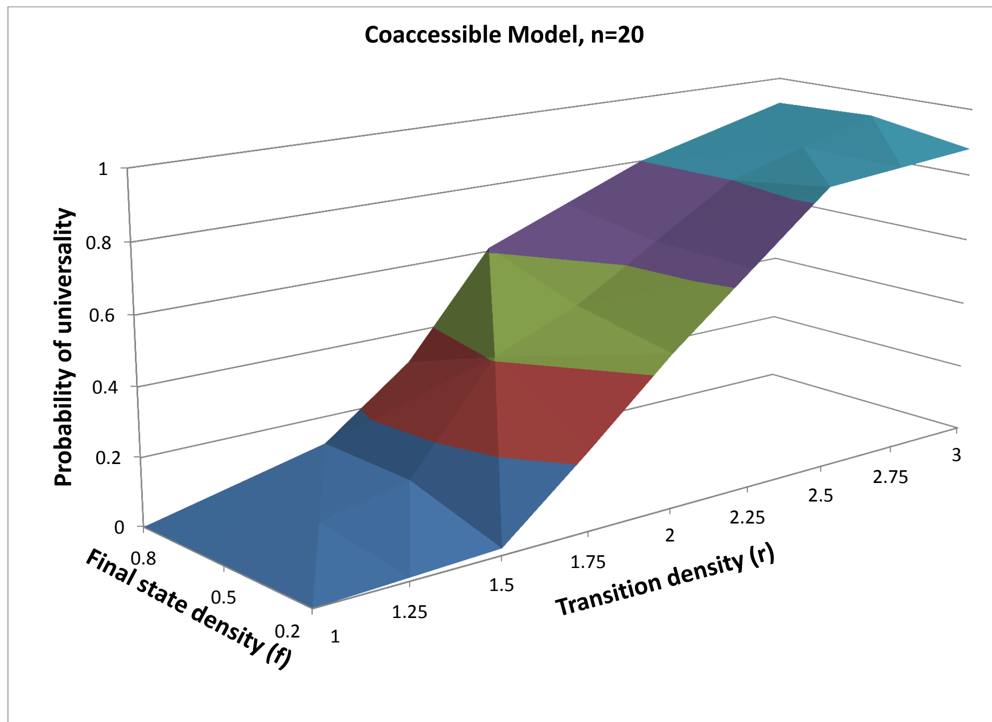

Figure 4.3 : A co-accessible Büchi universality terrain for $n = 20$. The transition density r ranges from 1 to 3, and f ranges from 0.2 to 0.8. Notice that the slope is much shallower than in previous models. This gives us an extremely wide range of useful configurations for testing.

4.2 Experiments - NFA

Having defined three new random models and, via universality testing, proven them to be interesting for performance evaluation, we then use these models to run timing experiments for ALASKA vs ALASKA-ZDD. As in the PREvious section, experiments were run on the DAVinCI cluster at Rice University, which consists of many Westmere nodes with 2.83 GHz processors and 48 GB of memory per node. We limit each job to 30GB of memory and 10 minutes of time. Jobs that did not finish were marked as timeouts.

As in the previous section, we run terrain experiments. In terrain experiments, the size of the automata is held constant, and two other parameters are changed to see the effects on running time. We generate 100 automata using each combination of parameter values, and report median running time.

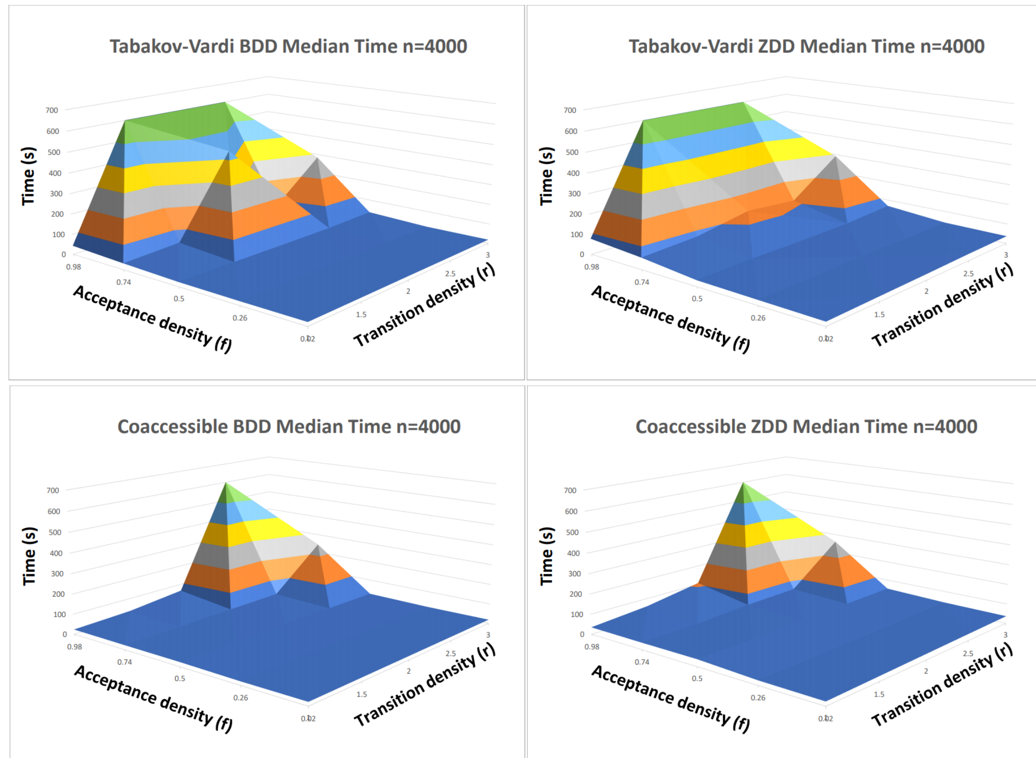

Figure 4.4 : For terrain experiments on the Tabakov-Vardi and coaccessible models, we tested parameter values of $n = 4000$, $r \in \{1, 1.5, 2, 2.5, 3\}$, and $f \in \{0.02, 0.26, 0.5, 0.74, 0.98\}$. These graphs show results for ALASKA and ALASKA-ZDD. Note that ALASKA performs better in most, but not all, cases.

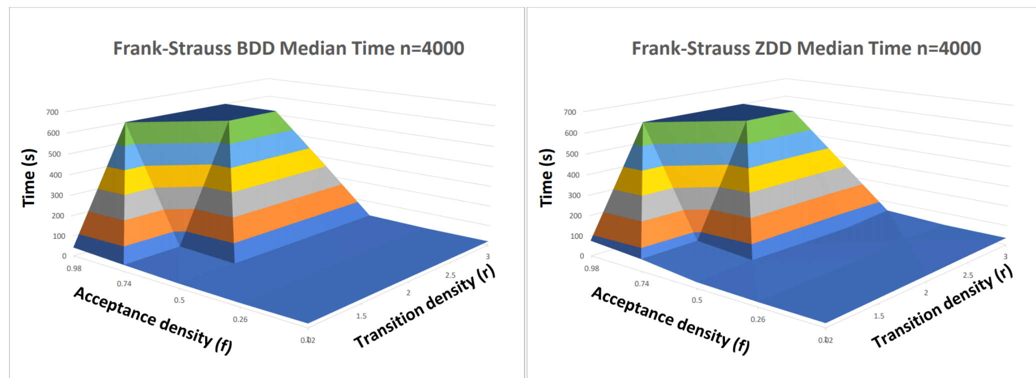

Figure 4.5 : For the Frank-Strauss model, we tested parameter values of $n = 4005$, or $l = 89$, $r \in \{1, 1.5, 2, 2.5, 3\}$, and $f \in \{0.02, 0.26, 0.5, 0.74, 0.98\}$.

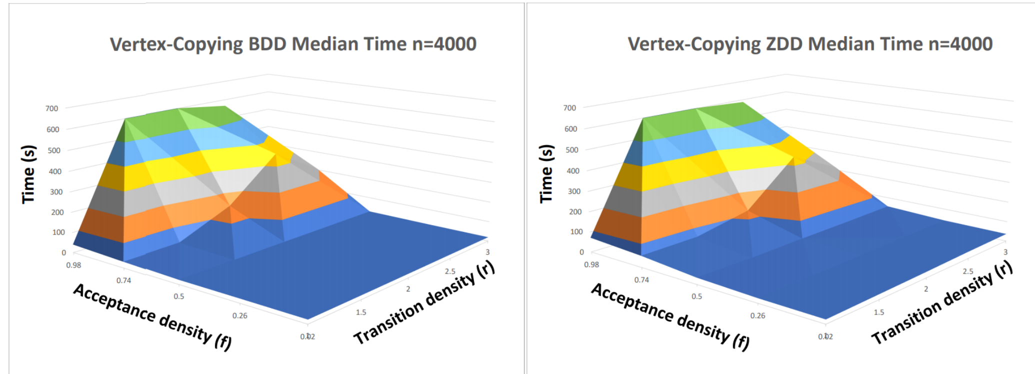

Figure 4.6 : For the vertex-copying model, we tested parameter values of $n = 4000$, $r \in \{1, 1.5, 2, 2.5, 3\}$, $f \in \{0.02, 0.26, 0.5, 0.74, 0.98\}$, and $b \in set 0.2, 0.5, 0.8$. We combine data for all values of b in this chart, to fit it into three dimensions.

Based on Figs. 4.4 through 4.6, ALASKA-ZDD does not consistently outperform ALASKA on random automata. However, we do find that, averaged across all models, ALASKA-ZDD has 5.9% fewer overall timeouts than ALASKA. Every individual model showed at least a 5.5% improvement. This suggests that ALASKA-ZDD may perform more consistently on the most difficult subset of problems. These limited results may because the calculation of PRE in ALASKA and ALASKA-ZDD uses existential abstraction on the variables representing the states to determine which predecessor macrostates a given state belongs in. Existentially abstracting out most variables produces a Boolean function that produces true for many inputs, which are difficult to represent with ZDDs. It may be possible to improve the algorithm for ZDDs to avoid this construction in the future.

We also find that all random models agree extremely closely on the results for ALASKA and ALASKA-ZDD, and which areas are easier or harder. Since many different random models produce the same results, this strongly reinforces the hypothesis that Tabakov-Vardi is a sufficiently robust random model with accurate

results. Nevertheless, this only shows evidence that Tabakov-Vardi is sufficient for NFA universality. We also test with Büchi universality, another simple verification problem that uses very different algorithms, to see whether Tabakov-Vardi is robust across problems.

4.3 Experiments - Büchi Automata

4.3.1 Methodology

While Tabakov-Vardi seems to be robust on NFAs, it is not clear if this carries over to infinite words. To examine this, we used our models to run timing experiments for three universality checkers. We first compared the Rank and Ramsey tools[§] from [5], expanding on previous work and seeing if our new models agreed with their results. To acquire a more recent picture of the comparison between algorithms, we also compared these tools with the RABIT 2.3 tool[¶], a more recent Ramsey-based containment checker. As in previous sections, experiments were run on the DAVinCI cluster at Rice University, which consists of many Westmere nodes with 2.83 GHz processors and 48 GB of memory per node. We limited each job to 30GB of memory and one hour of time. Jobs that did not finish were marked as timeouts.

We ran two types of experiments: terrain experiments and scaling experiments. In terrain experiments, the size of the automata is held constant, and two other parameters are changed to see the effects on running time. In scaling experiments, all parameters are held constant except those affecting the size of the automaton, and we steadily increase the size to see how the implementations respond to larger problems. We conduct scaling experiments with parameters that are particularly difficult for at least one tool to handle, as determined by the terrain experiments, to test practical worst-case performance. We generated 100 automata using each combination of parameter values in both kinds of experiments, and report median running time.

[§] `https://www.cs.rice.edu/CS/Verification/Software/software.html`

[¶] `http://www.languageinclusion.org/doku.php?id=tools`

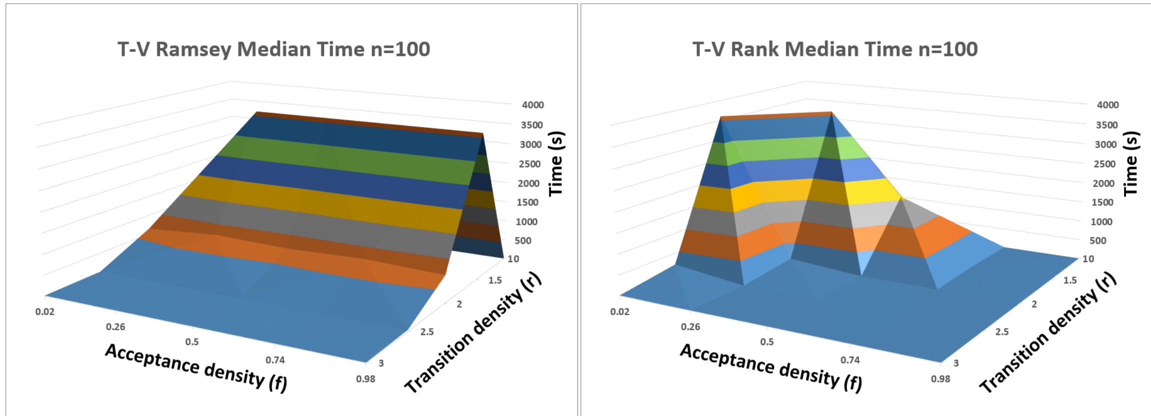

Figure 4.7 : For terrain experiments on the Tabakov-Vardi model, we tested parameter values of $n = 100$, $r \in \{1, 1.5, 2, 2.5, 3\}$, and $f \in \{0.02, 0.26, 0.5, 0.74, 0.98\}$. These graphs show results for the Rank and Ramsey tools. Note that Rank and Ramsey are not directly comparable - Ramsey tends to be slower at points where $r = 1.5$ and $r = 2$, while Rank tends to be slower at $f = 0.02$ and $f = 0.26$. This agrees with previous results [5] using the Tabakov-Vardi model.

4.3.2 Results

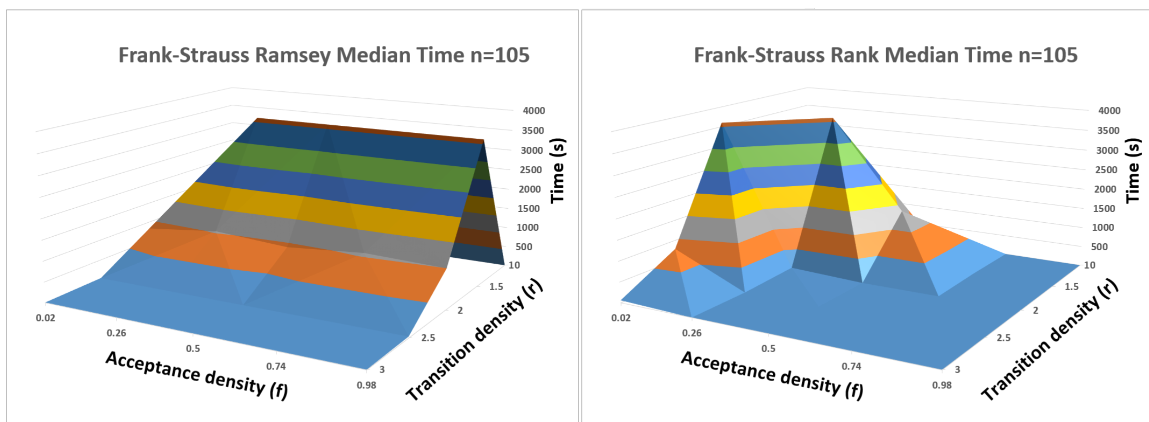

Figure 4.8 : For terrain experiments on the Frank-Strauss model, we tested parameter values of $n = 105$, or $l = 14$, $r \in \{1, 1.5, 2, 2.5, 3\}$, and $f \in \{0.02, 0.26, 0.5, 0.74, 0.98\}$. These graphs show results for the Rank and Ramsey tools. Again, the Rank model tends to perform the slowest at low f and low r, while Ramsey is slowest at $r = 2$. This agrees with our results on the Tabakov-Vardi model, as do the terrains of other models found in the appendix.

We find both that choice of model does not seriously impact tool comparisons, and that RABIT noticeably outperforms Rank and Ramsey.

In both terrain (Figs. 4.7, 4.8, 4.9) and scaling (Fig. 4.10) experiments, we find that the relative efficiency of tools is very similar across models. All models show that, as in the Tabakov-Vardi model in [5], the Rank and Ramsey are not directly comparable – which parameters are used to generate an automaton determine which tool solves it most efficiently, as seen in the terrain experiments in Figure 4.8. Since all models agree with T-V here, it is reasonable to use the T-V model to compare tools. Nevertheless, while models agree on the comparison between tools, they do not have the same running time. For example, in Figure 4.10, we see on a log scale that there is a factor of 10 difference between the running time of Ramsey on the Tabakov-Vardi and co-accessible models. While Tabakov-Vardi's tool comparisons seem to be robust in this case, its precise runtimes are not. Thus, the T-V model should be relied on for relative comparisons, but not for predicting runtimes.

Since there was little difference in comparison between models, Rank and Ramsey compare similarly to their results in [5]. Yet, when we compare Rank to RABIT, we saw a massive speedup at all difficult points – sometimes thousands of times faster. At $n = 100$, the terrain was flat, with most cases terminating in just over a tenth of a second. Therefore, the improved modern Ramsey tools are more suited for practical use than Rank-based ones. However, as seen in Figure 4.9, random models can still provide interesting performance terrain on the more efficient tools by scaling up the size of the problems.

There is one noticeable difference between algorithms not shown – both Ramsey-based algorithms used much more memory than Rank did. When provided with 5 gigabytes of memory, the Rank tool performed acceptably, but Ramsey and RABIT

crashed regularly. 30GB of memory provided was necessary to avoid crashes due to running out of memory.

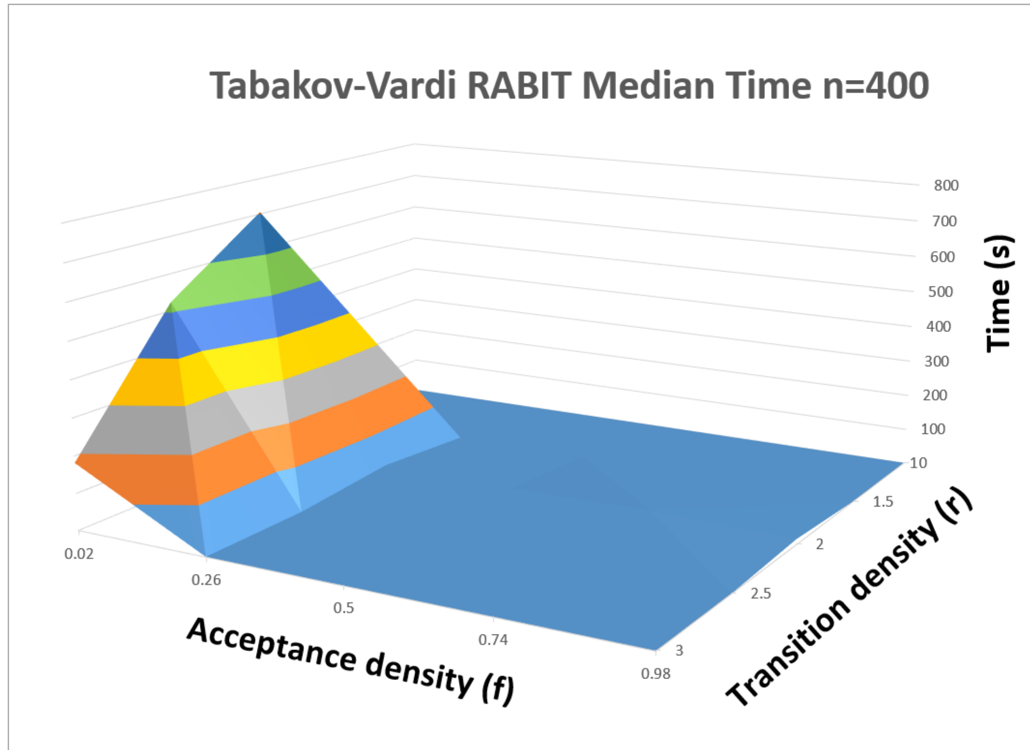

Figure 4.9 : For all terrain experiments at $n = 100$ for RABIT, we found that the terrain was entirely flat - very few problems took more than one second to terminate. Therefore we show results for RABIT on $n = 400$, instead, with parameter values $r \in \{1, 1.5, 2, 2.5, 3\}$, and $f \in \{0.02, 0.26, 0.5, 0.74, 0.98\}$. Note that the maximum Y-axis value is only 800 seconds, because at no point was the median result a timeout. RABIT has the most difficulty at high transition density and extremely low acceptance densities, with orders of magnitude slower performance on $f = 0.02$. While it does not appear on this graph, we also find that RABIT takes about two orders of magnitude more time at $r = 2.0$ and high f than other areas, and one order of magnitude less than the extremely difficult areas. Also, we find that at $r = 1.5$, we consistently had a small (5%) chance of timeouts at all values of n tested with few to no timeouts elsewhere, though the median time taken was no higher.

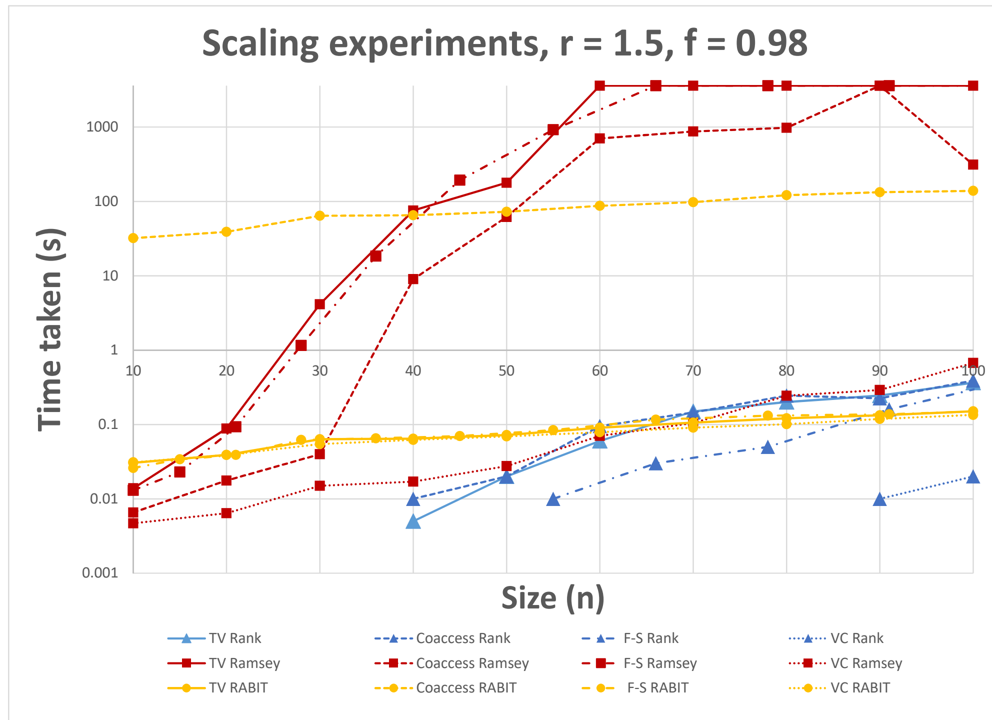

Figure 4.10 : For this set of scaling experiments, we set $r = 1.5$ and $f = 0.98$, and scale n from 10 to 100. In the Frank-Strauss model, l scales from 4 to 14. This point was chosen for scaling because it is particularly difficult for Ramsey. On this log-scale plot, different tools (indicated by shared color and marker shape) tend to have similar slopes regardless of model (indicated by shared line style). Notably, an obvious exponential gap exists between other models and Ramsey at these parameters for every model except the trivally-easy vertex-copying model. Since f is high, this is an easy point for Rank. The relationship between tools found by T-V is also reflected in the other random models shown here.

Chapter 5

Conclusion

ALASKA provides a powerful tool for NFA universality based on subsumption and symbolic representation with BDDs. Since subsumption reduces the space being considered, it seemed plausible that ZDDs, useful for representing sparse functions, could be more effective than BDDs for symbolically solving NFA universality. This did not result in a general improvement in median time taken over BDDs in practice. However, using ZDD-based approaches, we found a noticeable decrease in the number of timeouts, suggesting that ZDDs avoid worst-case behavior in some scenarios.

While formal verification provides important software tools, it has been unclear whether these tools are efficient enough to be used in practice. The Tabakov-Vardi random model is a powerful tool for automata-theoretic formal verification, allowing us to test the efficiency of algorithms for determining conformance to a specification. Due to concerns about whether the model accurately reflected real-world performance, we tested other models to see if the structure of a problem would influence the results; we found that it did not, on both NFA unversality and Büchi universality. Future work in the area can proceed to test algorithms and tools on the T-V model, more confident that it is robust and that its results are widely applicable. This work gives reason to believe that the Tabakov-Vardi model is a robust model with results that are likely to be close to the real-world. Complementation, and thus containment checking, should be practical on real-world problems.

We also discovered an improvement of many orders of magnitude in modern Büchi containment checkers using a Ramsey-based approach. RABIT outperformed both older Ramsey and rank-based tools significantly, and can scale to much higher input sizes. Since little work has been done on rank-based solvers since 2010, current heuristics-driven Ramsey-based approaches are the best available options for containment checking for Büchi automata.

Acknowledgements Work supported in part by NSF grants CCF-1319459 and IIS-1527668, by NSF Expeditions in Computing project "ExCAPE: Expeditions in Computer Augmented Program Engineering", as well as the Data Analysis and Visualization Cyberinfrastructure funded by NSF grant OCI-0959097 and Rice University.

Bibliography

[1] M. Vardi and P. Wolper, "An automata-theoretic approach to automatic program verification," in *Proceedings of the First Symposium on Logic in Computer Science*, pp. 322–331, IEEE Computer Society, 1986.

[2] M. Y. Vardi, "The Büchi complementation saga," in *Proc. 24th Sympo. on Theoretical Aspects of Computer Science*, vol. 4393 of *Lecture Notes in Computer Science*, pp. 12–22, Springer, 2007.

[3] M. Tsai, S. Fogarty, M. Vardi, and Y. Tsay, "State of Büchi complementation," in *Implementation and Application of Automata*, pp. 261–271, Springer, 2011.

[4] G. De Giacomo and M. Y. Vardi, "Linear temporal logic and linear dynamic logic on finite traces.," in *IJCAI*, vol. 13, pp. 854–860, 2013.

[5] S. Fogarty and M. Vardi, "Efficient Büchi universality checking," in *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 205–220, Springer Berlin Heidelberg, 2010.

[6] J. E. Hopcroft, *Introduction to automata theory, languages, and computation.* Pearson Education India, 2008.

[7] L. Doyen, J. Raskin, K. Chatterjee, L. Doyen, T. Henzinger, and J. Raskin, "Alaska: Antichains for logic, automata and symbolic kripke structures analysis," *ATVA: Automated Technology for Verification and Analysis*, vol. 3, pp. 153–168,

2008.

[8] D. Tabakov and M. Vardi, "Experimental evaluation of classical automata constructions," *LPAR*, pp. 396–411, 2005.

[9] D. Tabakov and M. Vardi, "Model checking Büchi specifications," in *Proc. 1st Int'l Conf. on Language and Automata Theory and Applications*, pp. 565–576, 2007.

[10] L. Doyen and J. Raskin, "Antichains for the automata-based approach to model-checking," *arXiv preprint arXiv:0902.3958*, 2009.

[11] M. De Wulf, L. Doyen, T.A. Henzinger, and J. Raskin, "Antichains: A new algorithm for checking universality of finite automata," in *Computer Aided Verification*, pp. 17–30, Springer, 2006.

[12] P.A. Abdulla, Y. Chen, L. Clemente, L. Holík, C. Hong, R. Mayr, and T. Vojnar, "Advanced Ramsey-based Büchi automata inclusion testing," in *CONCUR 2011–Concurrency Theory*, pp. 187–202, Springer, 2011.

[13] R. M. Karp, "The transitive closure of a random digraph," *Random Structures & Algorithms*, vol. 1, no. 1, pp. 73–93, 1990.

[14] J. Kleinberg, R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins, "The web as a graph: Measurements, models, and methods," in *Computing and combinatorics*, pp. 1–17, Springer, 1999.

[15] O. Frank and D. Strauss, "Markov graphs," *Journal of the American Statistical Association*, vol. 81, no. 395, pp. 832–842, 1986.

[16] T. Leslie, "Efficient approaches to subset construction," tech. rep., University of Waterloo, Canada, 1995.

[17] M. De Wulf, L. Doyen, N. Maquet, and J.-F. Raskin, "Antichains: Alternative algorithms for ltl satisfiability and model-checking," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 63–77, Springer, 2008.

[18] "ALASKA user manual." `http://web.archive.org/web/20161009183809/http://lit2.ulb.ac.be/alaska/usermanual.html`. Accessed: 2018-06-12.

[19] R. E. Bryant, "Symbolic boolean manipulation with ordered binary-decision diagrams," *ACM Computing Surveys (CSUR)*, vol. 24, no. 3, pp. 293–318, 1992.

[20] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient implementation of a bdd package," in *Design Automation Conference, 1990. Proceedings., 27th ACM/IEEE*, pp. 40–45, IEEE, 1990.

[21] M. De Wulf, L. Doyen, N. Maquet, and J.-F. Raskin, "Alaska," in *International Symposium on Automated Technology for Verification and Analysis*, pp. 240–245, Springer, 2008.

[22] L. Doyen and J.-F. Raskin, "Antichain algorithms for finite automata," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 2–22, Springer, 2010.

[23] S. Haynal, "Pycudd, python wrapper for cudd." `http://web.archive.org/web/20171210091429/https://bears.ece.ucsb.edu/pycudd.html`. Accessed: 2018-06-15.

[24] F. Somenzi, "CUDD package, release 2.4.1." http://vlsi.colorado.edu/~fabio/CUDD/.

[25] A. Mishchenko, "Extra, zdd extension package for cudd." http://web.archive.org/web/20180615161227/https://people.eecs.berkeley.edu/~alanmi/research/extra/. Accessed: 2018-06-15.