

A Simple Heuristic for Deciding Intersection Non-Emptiness for Regular and ω -Regular Automata

Jake Ginesin

1 Introduction

Intersection non-emptiness is a classical decision problem in automata theory. For n regular or ω -regular language-accepting Büchi automata A_1, A_2, \dots, A_n that share a common alphabet Σ , the intersection non-emptiness problem decides whether their respective accepted languages are disjoint. That is, letting $L(A)$ denote the language accepted by A :

$$L(A_1) \cap L(A_2) \cap \dots \cap L(A_n) = \emptyset$$

Or equivalently:

$$\exists w \in \Sigma^* \text{ such that } w \in \bigcap_{i=1}^n L(A_i)$$

In general, for DFAs, NFAs, and Büchi Automata, this problem is known to be PSPACE-complete [7]. In this note, we introduce a cheap and natural heuristic for deciding the intersection non-emptiness problem in practical scenarios.

We have two primary motivations for constructing such a heuristic. First, the SPIN model checker reduces deciding safety and liveness properties on Promela models to deciding Büchi Automata intersection non-emptiness [14]. This is done via a translation from the defined Promela model to a Büchi Automata and a highly optimized (but, exponential complexity in the worst-case) translation from Linear Temporal Logic to another Büchi Automata [3], as shown in figure 1. Second, the Büchi Automata language inclusion problem – a closely related PSPACE-complete automata decision problem with many practical applications – is often reduced to the intersection non-emptiness problem by constructing the complement of one of the Büchi Automata [9]. Even though the Büchi Automata complementation

problem is in general exponential in complexity, there is a long line of work optimizing the procedure and creating practical implementations [13].

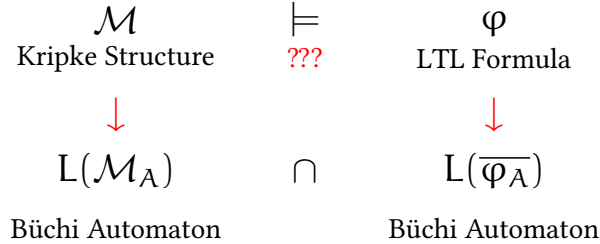


Figure 1: The SPIN model checker translates user-defined Promela models into Kripke structures, which is then translated into a Büchi Automata. The LTL formula is turned into a never claim (as to avoid an expensive complementation procedure), then translated into a Büchi Automata. Then, intersection non-emptiness is checked.

2 Classic Intersection Non-Emptiness Methods

We begin by introducing the classical methods and heuristics for deciding the intersection non-emptiness problem. For the remainder of this note, we use the following semi-formal definition: a (standard) finite, regular language-accepting automata is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ is a finite alphabet, $\delta : Q \times \Sigma \rightarrow Q$ is the transition function over the states and input alphabet, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of accepting states.

The canonical method for deciding intersection non-emptiness between n finite automata A_1, \dots, A_n involves constructing a composite automata $A_1 \parallel A_2 \parallel \dots \parallel A_n$ whose acceptance states are the intersection of each sub-automata's acceptance states. Then, this composite automata is exhaustively searched. This method is generally infeasible in practice, as it requires constructing the entire automata – which is $|A_1| \times \dots \times |A_n|$ in size – in memory.

Therefore, every practical intersection non-emptiness method employs *on-the-fly composition* to search the composite automata without explicitly constructing it. The state of the composite automata is denoted $(q^{(A_1)}, \dots, q^{(A_n)})$ where $q^{(A_i)} \in Q^{(A_i)}$. Similarly, the transition function becomes:

$$\delta : \left((\delta^{(A_1)} : Q^{(A_1)} \times \Sigma) \times \dots \times (\delta^{(A_n)} : Q^{(A_n)} \times \Sigma) \right) \rightarrow (Q^{(A_1)} \times \dots \times Q^{(A_n)})$$

Explicitly, the state space of the composite automata is just:

$$(Q^{(A_1)} \times \dots \times Q^{(A_n)})$$

For simplicity, we denote the transition function of the composite automata δ^c and the state space Q^c . Observe, in order to take a transition over δ^c in the composite automata for a given $a \in \Sigma$, that transition over $\delta^{(A_1)}, \dots, \delta^{(A_n)}$ given $a \in \Sigma$ must also exist. Writing this relationship out:

$$\text{for } a \in \Sigma, \delta^c(Q^c, a) \rightarrow \left(\delta^{(A_1)}(q^{(A_1)}, a), \dots, \delta^{(A_n)}(q^{(A_n)}, a) \right)$$

From here, modern solvers employ a number of techniques to reduce the search space and improve efficiency. This includes bit-state hashing [5], exploiting commutativity of certain transitions to reduce state-space [6], using multiple cores [8], and utilizing binary decision diagrams [12]. We observe, however, classical heuristic graph-search techniques such as A^* have seldom been applied to this problem. To this end, we introduce a simple heuristic to more efficiently navigate the search space to find counterexamples to intersection non-emptiness.

3 A Simple Heuristic

At each iteration of the state-space search of Q^c , we need to choose which path to extend (that is, which valid character in Σ to apply to δ^c). Classically, A^* selects the path that minimizes $f(n) + g(n) + h(n)$, where n is the next node on the path, $g(n)$ is the cost of the path from the start node to n , and $h(n)$ is a *heuristic* that estimates the cost of the cheapest path from n to the goal, in this case the shortest path to an acceptance state. Thus, for our purposes, $h(n)$ becomes:

$$h\left((q^{(A_1)}, \dots, q^{(A_n)})\right) = \sum_{i=1}^n q^{(A_i)}\text{'s distance from acceptance condition in } A_i$$

It must be noted, the *acceptance condition* for a regular automata versus a ω -regular Büchi automata is different; a regular automata requires discovering an acceptance state, while a Büchi automata requires discovering an acceptance state *within* a cycle. Real-world verification scenarios generally reduce to Büchi automata rather than regular automata, and solvers such as SPIN that reduce verification problems to Büchi automata search the state-space via a nested depth first search-based method [1].

For each A_1, \dots, A_n , we seek to minimal *approximate* the distance from any node to an acceptance condition. This heuristic can be simply realized through a variety of techniques, including pre-processing approaches and solver-based approaches.

3.1 Preprocessing-based Approach

We observe as compared to reasoning about a composite automata of size $|A_1| \times \dots \times |A_n|$, reasoning about each automata A_1, \dots, A_n individually is vastly preferable complexity-wise.

Specifically, for some $A \in \{A_1, \dots, A_n\}$ we seek to *label* each $q \in A$ with its *minimal* distance to any given acceptance condition. For regular automata, this distance is defined by the least distance from the current state to an acceptance state. For Büchi automata, this distance is defined by the least distance from the current state to an acceptance state plus the length of the acceptance cycle.

To achieve this labeling for regular automata and ω -regular automata, we employ a modified BFS-based technique that simultaneously and iteratively expands from each acceptance state, labeling each state with a minimal distance value. For regular automata, the distance value for each acceptance state is labeled "0", while for ω -regular automata, the distance value for each acceptance state is labeled with the length of the shortest cycle containing the state. Once this labeling is constructed for each A_1, \dots, A_n , we simply compute the shortest total distance for each possible expansion of each automata. This sum calculation can be effectively memoized with an efficient enough bit-state hashing implementation, and thus is efficient to compute in practice.

Algorithm 1 Simultaneous BFS to Compute Distance to Acceptance Cycle

- 1: **Input:** Büchi automaton $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, where Q is the set of states, Σ is the input alphabet, δ is the transition function, q_0 is the initial state, and F is the set of accepting states.
 - 2: **Output:** Distance mapping $d : Q \rightarrow \mathbb{N} \cup \{\infty\}$ indicating the shortest distance from each state to an acceptance cycle.
 - 3: Initialize a queue Q_x containing all states in F with their distance set to the minimum acceptance cycle distance.
 - 4: Initialize the distance mapping d such that:
 - $d(q) = 0$ for all $q \in F$ (initial distance for acceptance states),
 - $d(q) = \infty$ for all $q \in Q \setminus F$.
 - 5: **while** Q_x is not empty **do**
 - 6: Remove the current state q_{curr} from the front of Q_x .
 - 7: Let $d(q_{\text{curr}})$ be the distance associated with q_{curr} .
 - 8: Let $R_{\text{reachable}}(q_{\text{curr}})$ be the set of states that can reach q_{curr} via a transition.
 - 9: **for each** $(q_r, \sigma) \in R_{\text{reachable}}(q_{\text{curr}})$ **do**
 - 10: **if** $d(q_{\text{curr}}) + 1 < d(q_r)$ **then**
 - 11: Set $d(q_r) = d(q_{\text{curr}}) + 1$.
 - 12: Add q_r to the queue Q_x .
 - 13: **end if**
 - 14: **end for**
 - 15: **end while**
 - 16: **Return** the distance mapping d .
-

3.2 Constraint Solver-based Approaches

While doing a pre-processing approach front-loads the cost of this heuristic, computing the minimal distance between any given state and an acceptance condition can also be done as-needed to potentially avoid unnecessary work. This can be naively accomplished by running and re-running depth-first search. However, we also experiment with using constraint solvers for this purpose.

We observe that finite automata that accept a language are naturally viewed as a system of flows. Thus, to encode a constraint solver to find the minimal distance between any given state $q_0 \in Q$ in some automata \mathcal{A} , we add the following constraints to our solver:

- Initial state $q_0 \Rightarrow 1 + \sum q$'s incoming transitions = $\sum q$'s outgoing transitions
- For all accepting states $\alpha \in F \Rightarrow \sum \alpha$'s incoming transitions = $1 + \sum \alpha$'s outgoing transitions

- Other states $q \in Q \Rightarrow \sum q$'s incoming transitions = $\sum q$'s outgoing transitions
- **Minimize** \sum all transitions

Alternatively, for a Büchi automata where acceptance states must be within cycles, we can modify the accepting state constraint to be:

For all accepting states $a \in F \Rightarrow 1 + \sum a$'s incoming transitions = $2 + \sum a$'s outgoing transitions

We implement our constraints using Z3 [2] and Gurobi [4]. However, we observe that many constraints can be re-used between runs as only the initial state changes. Similarly, most solver runs have very similar constraints and only require a few modifications – Z3 does not allow adding and removing specific constraints, while Gurobi does. Gurobi also supports warm starting, allowing previous solves with similar constraints to inform future solves. In our experience, Gurobi with warm starting enabled has seen the best performance.

This concept of using constraint solving to reason about the languages of regular language automata is not new – constraint solving has been previously employed to find the lower-bound parikh images [11] and decide intersection non-emptiness of NFAs [10]. However, constraint solver-based methods haven't seen much use for Büchi automata.

4 Implementation

We implement our approaches in `AB`, our experimental automata library. Our code is publicly available in our Github repository: <https://github.com/JakeGinesin/ab>

However, our implementation should not be used for real-world verification purposes, as it is implemented in Python. Python is interpreted and thus generally much less efficient than a standard compiled language.

5 Conclusion

In this note, we introduce a simple heuristic for efficiently solving the intersection non-emptiness problem for regular and ω -regular automata. Our approach is aimed at practical implementations for solving this problem, and in the future we hope to integrate and evaluate our heuristics into more traditional solvers and assess them against standard benchmarks.

6 Acknowledgements

The author would like to acknowledge the helpful advice, support, and mentorship of Christoph Haase.

References

- [1] *The SPIN Verification System*, volume 32 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, Providence, Rhode Island, May 1997.
- [2] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 4963:337–340, 2008.
- [3] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. *Simple On-the-fly Automatic Verification of Linear Temporal Logic*, page 3–18. Springer US, Boston, MA, 1996.
- [4] LLC Gurobi Optimization. *Gurobi Optimizer Reference Manual*, 2023.
- [5] Gerard J. Holzmann. *An analysis of bitstate hashing*, page 301–314. IFIP Advances in Information and Communication Technology. Springer US, Boston, MA, 1996.
- [6] Gerard J. Holzmann and Doron Peled. *An Improvement in Formal Verification*, page 197–211. IFIP Advances in Information and Communication Technology. Springer US, Boston, MA, 1995.
- [7] Dexter Kozen. Lower bounds for natural proof systems. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, page 254–266, Providence, RI, USA, September 1977. IEEE.
- [8] Alfons Laarman and Anton Wijs. *Partial-Order Reduction for Multi-core LTL Model Checking*, volume 8855 of *Lecture Notes in Computer Science*, page 267–283. Springer International Publishing, Cham, 2014.
- [9] Sven Schewe. Büchi complementation made tight.
- [10] Amanda Stjerna and Philipp Rümmer. A constraint solving approach to parikh images of regular languages. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA1):1235–1263, April 2024.

- [11] Anthony Widjaja To. Parikh images of regular languages: Complexity and applications. (arXiv:1002.1464), February 2010. arXiv:1002.1464 [cs].
- [12] Tom van Dijk. The parallelization of binary decision diagram operations for model checking.
- [13] Moshe Y. Vardi. *The Büchi Complementation Saga*, volume 4393 of *Lecture Notes in Computer Science*, page 12–22. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.
- [14] Moshe Y. Vardi and Pierre Wolper. *An automata-theoretic approach to automatic program verification*. IEEE Computer Society, 1986.