TEST DRIVEN REGULAR EXPRESSION PROGRAM SYNTHESIS

An Undergraduate Research Scholars Thesis

by

STELLA YANG

Submitted to the Undergraduate Research office at
Texas A&M University
in partial fulfillment of the requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by
Faculty Research Advisor: Dr. Philip Ritchey

May 2025

Major: Computer Science

RESEARCH COMPLIANCE CERTIFICATION

Research activities involving the use of human subjects, vertebrate animals, and/or biohazards must be reviewed and approved by the appropriate Texas A&M University regulatory research committee (i.e., IRB, IACUC, IBC) before the activity can commence. This requirement applies to activities conducted at Texas A&M and to activities conducted at non-Texas A&M facilities or institutions. In both cases, students are responsible for working with the relevant Texas A&M research compliance program to ensure and document that all Texas A&M compliance obligations are met before the study begins.

I, Stella Yang, certify that all research compliance requirements related to this Undergraduate Research Scholars thesis have been addressed with my Faculty Research Advisor prior to the collection of any data used in this final thesis submission.

This project did not require approval from the Texas A&M University Research Compliance & Biosafety office.

TABLE OF CONTENTS

ABSTRACT

Test Driven Regular Expression Program Synthesis

Stella Yang
Department of Computer Science and Engineering
Texas A&M University

Faculty Research Advisor: Dr. Philip Ritchey
Department of Computer Science and Engineering
Texas A&M University

This thesis investigates the problem of program synthesis in the constrained domain of regular expression (regex) generation. We compare three distinct methodologies: AlphaRegex, a heuristic-guided search-based synthesizer; the L* algorithm, a procedural learner rooted in automata theory; and large language models (LLMs), which generate regex through few-shot prompting. Our evaluation framework assesses each approach on correctness, efficiency, and complexity, using a diverse benchmark set spanning easy to hard regex synthesis tasks.

We find that AlphaRegex benefits from tunable cost heuristics that allow control over the balance between simplicity and runtime but suffers in scalability on complex patterns. The L* algorithm reliably produces correct regex through interaction with a teacher but can become rigid and verbose due to its dependence on counterexamples and DFA construction. LLMs perform surprisingly well on simple problems that resemble training data patterns but degrade with compositional complexity and struggle with formal constraints. Additionally, we analyze AlphaRegex under varying heuristic weight configurations and explore the limitations and benefits of hybrid (or chained) synthesis pipelines.

This work contributes a detailed experimental comparison across methodologies, a public

implementation of each, and reflections on what it means to "ask good questions" when synthesizing code. It lays groundwork for future synthesis tools that integrate heuristic, procedural, and neural strategies, offering insights into usability, test-driven development, and scalable human-in-the-loop programming workflows.

ACKNOWLEDGMENTS

Contributors

I would like to express my sincere gratitude to my faculty advisor, Dr. Philip Ritchey, for his exceptional guidance and support throughout the duration of this project. Dr. Ritchey's insightful advice and helpful pointers have been invaluable to me during my entire college career at Texas A&M University, and especially so in my interactions with him on this project. His expertise and practical approach have greatly enhanced my learning experience.

I would like to thank my friends Teddy Heinen, Nathan Nguyen, Jasmine Pan, Lucy Zhang, Chris Kelley, Michael McCarthy, Winnie Phung, and Matt Krehel, whose friendship and support have been an incredible source of joy and inspiration. I would also like to thank the department faculty and staff for making my time at Texas A&M University a great experience.

Last but definitely not least, I am deeply grateful to my partner Emmie Teng for her patience, love, and all the help through everything.

Funding Sources

I have not received funding for this undergraduate research project.

1. INTRODUCTION

A central goal of software engineering is to faithfully translate user intent into executable code that runs correctly on a machine. This transformation—bridging human ideas with computational logic—is at the heart of program synthesis, a field dedicated to the automatic generation of purposeful, correct code from high-level specifications. This presents major benefits—ranging from reducing the cost and complexity of software development to improving code quality and safety. Central to this effort is the notion of program synthesis, which seeks to transform high-level specifications into executable programs, ideally requiring minimal human intervention. This thesis engages with program synthesis from both a theoretical and practical perspective, aiming to build a system that generates functional, minimal programs based on test-driven constraints.

Program synthesis reduces the cognitive and logistical burden of programming. Writing software is often tedious, error-prone, and time-consuming. Developers repeatedly solve similar problems in slightly different ways across domains. By leveraging automation, we can offload tasks like boilerplate generation, routine logic, and even algorithm design to synthesis systems. This makes programming more accessible to non-experts and allows experienced developers to focus on higher-level design concerns. Importantly, the synthesis process has the potential to produce correct-by-construction programs, mitigating bugs and inconsistencies early in development.

Unlike traditional programming, where a developer manually translates requirements into code, synthesis systems aim to infer intent from a set of examples, specifications, or natural language descriptions. The challenge is to ensure that the output program not only satisfies all provided constraints but also aligns with the user's implicit expectations—such as readability, generality, and performance. This introduces a trade-off between formal correctness and human-aligned utility.

At a high level, program synthesis is a search problem over a (usually infinite) space of candidate programs. Each candidate is evaluated against constraints—often defined as input-output

examples, logical formulas, or type annotations—and strategies are applied to explore the search space efficiently. Exhaustive enumeration is rarely feasible, so modern synthesis techniques rely on pruning unpromising paths, guiding the search with heuristics, and, increasingly, using probabilistic or neural methods to guess good candidates.

Currently, the literature breaks down program synthesis into three conceptual pillars [4]:

1. **Intention**: How does the user convey what they want? This may be done via examples, natural language, sketches, or constraints.

2. **Invention**: What algorithms and techniques generate the program? This encompasses symbolic search, machine learning, logic-based reasoning, and hybrid models.

3. **Adaptation**: How does the program integrate with its environment? Adaptation includes type inference, refactoring, and context-awareness.

In this thesis, we primarily focus on the first two pillars: *intention* and *invention*. Our goal is to examine how different synthesis methodologies interpret user-provided constraints and how well they generate minimal, correct, and expressive solutions. Later sections will also explore how humans might better express their intent through improved test design or specification writing.

Program synthesis emphasizes not just correctness, but also alignment with user expectations. A solution such as a verbose switch-case statement may meet all functional requirements but fail to generalize or be useful in real-world development. Thus, elegance, minimalism, and maintainability are essential qualities. A well-synthesized program is one that satisfies the constraints while maintaining simplicity and avoiding overfitting.

This thesis specifically examines the synthesis of regular expressions (regex). Regular expressions are a core tool in text processing, validation, and extraction, widely used across many domains. They are highly expressive yet notoriously difficult to write, read, and debug, especially for non-experts. This complexity—paired with their concise formal definition—makes regex a strong test bed for studying program synthesis. Automating regex generation from examples not

only advances our understanding of synthesis algorithms but also offers practical value by alleviating a common pain point in software development.

Throughout the thesis, we will explore different synthesis approaches for generating regex from constraints, compare their performance, and investigate their limitations. We begin with an in-depth review of current techniques in program synthesis and proceed to evaluate three methods—AlphaRegex (heuristic search), the L* algorithm (procedural learning), and LLMs (neural generation)—through empirical and theoretical lenses. In doing so, we aim to uncover how synthesis tools can be made more efficient, interpretable, and user-friendly, ultimately bringing us closer to bridging the gap between human intent and machine-generated code.

1.1 Scope of Research

Regular expressions offer an ideal domain for exploring program synthesis due to their constrained grammar, well-defined semantics, and broad applicability in software development tasks. Unlike general-purpose programming languages, regex has a relatively limited set of operations, making the search space tractable for many synthesis algorithms while still presenting significant challenges in correctness, efficiency, and usability. Furthermore, regex is frequently used by developers across different experience levels, often leading to errors or inefficiencies that automation can help mitigate.

Despite their ubiquity, regular expressions are widely known in the programming community as being difficult to write and understand. In fact, their complexity and opacity have become the subject of a long-standing joke among developers: "Some people, when confronted with a problem, think 'I know, I'll use regular expressions.' Now they have two problems." This humor reflects a deeper truth—regex operates under a different computational model than most developers are used to. Rather than resembling imperative control flow or object-oriented abstractions, regex corresponds to a more mathematical model rooted in formal language theory.

To appreciate the relevance of regex to program synthesis, it helps to understand its theoretical foundations. Regular expressions describe the class of extitregular languages, which are precisely those recognized by deterministic or nondeterministic finite automata (DFAs or NFAs).

A deterministic finite automaton (DFA) is an abstract machine that processes input strings symbol by symbol, transitioning through a finite set of states according to a fixed set of rules. If the input string leads the machine into an accepting state, the string is considered part of the language the DFA recognizes. This model of computation is simple but powerful—it enables fast, predictable evaluation of strings and forms the backbone of many practical tools in programming languages and compilers. In the context of the theory of computation, regular languages represent the simplest class of problems that machines can solve—forming the lowest level in the Chomsky hierarchy of formal languages [3]. These are languages that can be recognized without the need for memory or a stack, meaning they are computationally limited compared to full-fledged programming languages, but they are still powerful enough to encode many practical string-processing tasks.

The class of regular languages is a strict subset of those problems solvable by a Turing machine, the abstract model underlying modern programming languages [7]. By studying synthesis in this restricted space, we can test our ideas and algorithms in a mathematically rigorous and computationally tractable setting. It is a way of starting small, while still engaging with real-world problems. If synthesis methods cannot handle the generation of regex—simple, constrained programs—they will certainly struggle with more general programming tasks.

In practical applications, regular expressions are widely used for tasks such as text validation, pattern matching, and data extraction. Modern regex engines, such as those found in programming languages like Python, Java, and JavaScript, support a rich set of features that extend beyond the theoretical foundations of regular languages. These include advanced constructs like lookaheads, lookbehinds, and backreferences, which enable more expressive matching but also increase complexity and computational cost.

In this thesis, we focus on the core operations of regular expressions: concatenation, union, and the Kleene star. These operations form the theoretical basis of regex and correspond directly to the operations used in formal language theory to define regular languages. By limiting our scope to these foundational constructs, we ensure that our synthesis methods remain grounded in the well-defined and computationally tractable domain of regular languages. We will also allow the ?

operator, which matches zero or one occurrences of a preceding element, as it is a common and useful construct in practical regex applications. This is a shorthand for the union of the empty string and the preceding element, which is a regular operation.

However, it is important to compare this theoretical rule set to the practical features supported by modern regex engines. While theoretical regex operates within the constraints of deterministic or nondeterministic finite automata, practical regex engines often implement additional features that go beyond the regular language class. For example, backreferences allow matching repeated patterns, which introduces non-regular behavior and makes the matching problem computationally harder.

By focusing on the theoretical core of regex, we aim to develop synthesis methods that are both rigorous and extensible. Future work could explore how these methods might be adapted to handle the more complex constructs found in practical regex engines, bridging the gap between theoretical elegance and real-world applicability.

This thesis compares three distinct approaches to regex synthesis, each reflecting a unique methodology in program synthesis: Alpha Regex (heuristic search), L* (procedural learning), and large language models (neural generation). Alpha Regex performs an explicit search over regex structures, guided by heuristics and weight functions that help prioritize promising candidates. It aligns closely with traditional search-based methods in the literature. L*, by contrast, incrementally builds a DFA from a teacher's feedback, producing equivalent regex representations through procedural refinement. While it guarantees correctness under ideal feedback, it heavily depends on the human teacher's accuracy. LLMs such as ChatGPT represent a different paradigm entirely—they generate regex from natural language or test descriptions but do so probabilistically, with no formal correctness guarantees.

By studying these methodologies side by side, we aim to uncover insights not just about regex generation, but about the strengths and limitations of broader synthesis paradigms. Although our focus is on regex, the comparative framework and evaluation metrics developed here can be extended to other constrained domains, such as parser combinators, grammar induction,

or domain-specific languages. The synthesis strategies, especially their integration into compositional or hybrid frameworks, may also offer insights into systems that combine multiple paradigms to improve performance, usability, and correctness.

1.2 Central Research Questions

This thesis is structured around three core research questions that aim to evaluate and improve the synthesis of regular expressions:

1. How do the Alpha Regex, L* algorithm, and LLMs compare in terms of correctness, efficiency, and complexity when generating regular expressions?

2. Can these systems build off of each other to produce a better result overall?

3. How can the weights in AlphaRegex change how we prioritize the answers that it generates?

The three different systems in generating regex differ not only in implementation but also in their underlying assumptions, guarantees, and interaction with user input. Correctness refers to whether the synthesized regex satisfies the user's constraints. Efficiency pertains to how quickly and resource-efficiently a method can produce a correct solution. Complexity, meanwhile, refers to the readability and generalization of the resulting regex.

In the integration approach, the output of one system informs or constrains another presents a compelling strategy for synthesizing more accurate and generalizable regex. For example, a language model might propose candidate expressions that Alpha Regex could refine or verify, or L* might bootstrap the initial search space of AlphaRegex.

The second research question explores a more internal mechanism of Alpha Regex: Alpha Regex relies on weighted heuristics to navigate its search space. These weights significantly affect which candidates are evaluated first and how quickly optimal solutions are found. Tuning these weights involves aligning them with implicit human notions of simplicity, coverage, or interpretability, making this an opportunity to investigate how algorithmic biases shape synthesis outcomes.

1.3 Contributions

First, we provide an in-depth experimental comparison of Alpha Regex, L*, and large language models, evaluating them on correctness, efficiency, and complexity. This comparison helps illuminate the trade-offs inherent to heuristic, procedural, and neural methods of synthesis.

Second, we explore the role of weighting in Alpha Regex and how tuning these weights influences the prioritization and quality of generated regex. Through case studies and parameter tuning experiments, we show how different heuristics affect the structure and performance of synthesized expressions.

Third, we investigate the potential for integration synthesis methods, presenting preliminary strategies for combining the strengths of multiple approaches. These hybrid models offer a promising path toward more robust and adaptable synthesis systems.

2. LITERATURE REVIEW

2.1 Historical Perspective on Program Synthesis

The desire to automate programming tasks has existed nearly as long as programming itself. In fact, program synthesis can be seen as a natural extension of compilation, with the goal of narrowing the semantic gap between human-level intent and machine-level execution. Where compilers translate precise, complete source code into machine instructions, synthesis aspires to generate that source code from higher-level, often incomplete, specifications. Computer science, at its core, has always aimed to build tools and models that convert abstract logic into executable behavior, and program synthesis is one of its purest manifestations [9].

2.2 Program Synthesis Approaches

Research in program synthesis has produced a wide range of approaches, each grounded in different assumptions about how users communicate intent and how synthesis engines evaluate correctness. Two major families of synthesis techniques are *inductive synthesis* and *deductive synthesis*. Inductive synthesis focuses on learning programs from a set of concrete examples, often called Programming by Example (PBE). This approach tries to generalize a program that matches provided input-output pairs, but must do so under uncertainty due to underspecification [10][5]. Deductive synthesis, on the other hand, constructs programs from logical specifications using theorem proving or constraint solving. These methods offer strong guarantees but often require formal specifications that are difficult for non-experts to write.

Another axis of classification relates to the direction and structure of the search process. *Explicit search* methods, which are most often bottom-up or top-down, systematically enumerate and evaluate potential program candidates by applying grammar rules to generate expressions. These methods do not rely on learned models or probabilistic inference; instead, they explore the program space in a structured and interpretable way. Explicit search ensures that no candidate is proposed without a formal justification and often integrates pruning strategies to eliminate unpromising or

redundant paths [9].

Bottom-up search in this context starts from simple atomic expressions—like constants or minimal regex patterns—and composes them into more complex structures. Although conceptually straightforward, this can quickly generate a large number of syntactically valid but semantically irrelevant candidates. Therefore, bottom-up search often includes observational equivalence checks or cost-based heuristics to prioritize promising paths.

Top-down search, on the other hand, begins with a high-level incomplete program—typically containing holes or placeholders—and attempts to refine it by recursively filling in those parts. This allows the synthesizer to preserve global structural constraints while focusing search efforts on the most relevant sub-expressions. The use of partial evaluation, type information, or semantic filtering often makes top-down explicit search more efficient and targeted than its bottom-up counterpart.

2.3 User Intention Guidance

One of the persistent challenges in program synthesis is inferring the user's true intent from sparse or ambiguous inputs. To mitigate this, systems have introduced ways for users to explicitly or implicitly guide the synthesis process. A notable innovation in this space is *program sketching*, where users write partial programs containing holes or placeholders, leaving the synthesizer to fill in the gaps. Sketching reduces the need for formal specification expertise and allows users to encode high-level structural intent directly [8]. This technique has shown promise in both educational settings and industrial applications, offering a powerful human-in-the-loop approach to synthesis.

2.4 Regular Expressions and Their Challenges

Regular expressions (regex) are widely used for pattern matching, lexical analysis, and data validation tasks. Despite their ubiquity and formal simplicity, regex is often perceived as unintuitive and error-prone. The syntax is compact and symbolic, making it powerful but difficult to read, write, and debug—especially for those without formal training. These usability challenges have led to increased interest in automatically generating regex from examples, specifications, or natural language descriptions.

Formally, regular expressions describe the class of regular languages, which correspond

to patterns recognized by deterministic finite automata (DFAs). A DFA is a finite-state machine that consumes input strings one symbol at a time and transitions through states according to fixed rules. If the machine ends in an accepting state, the string is accepted by the language. DFAs are equivalent in expressive power to regular expressions, and this equivalence can be proven constructively using algorithms such as Brzozowski's derivative-based method, which incrementally builds a DFA from a regex specification.

3. METHODS

3.1 The AlphaRegex Algorithm

AlphaRegex is a heuristic-driven algorithm for synthesizing regular expressions from user-provided examples. It models the synthesis task as a state-space search, beginning with an incomplete regex (a "hole") and iteratively expanding it into complete, valid expressions. The algorithm emphasizes simplicity, correctness, and efficiency by prioritizing low-cost states and eliminating unproductive ones using formal approximations. This section outlines the core mechanisms of AlphaRegex, including its mutation strategy, pruning heuristics, cost-based prioritization, and state representation. Readers seeking implementation-level details should refer to the original paper by Lee et al. [6].

Algorithm 1 AlphaRegex Search

```
1: w ← □
2: while w ≠ ∅ do
3:     s ← w.pop()
4:     if s satisfies examples then
5:         return s
6:     end if
7:     w ← w ∪ next states
8: end while
9: return no solution
```

3.1.1 Algorithm

The AlphaRegex algorithm leverages a greedy search mechanism to synthesize regex from positive and negative examples. As seen in Algorithm 1, AlphaRegex begins with an initial state consisting of a single placeholder (or "hole") representing an empty regex. From this minimal starting point, it enters a loop, expanding and evaluating candidate regex expressions. A candidate is considered valid if it matches all positive examples and rejects all negative examples. If a valid

solution is found, the algorithm terminates; otherwise, it continues exploring new states derived by filling holes with regex constructs.

As seen in Figure 1, the process begins with a starting state represented by an empty regex tree. From this initial state, the algorithm enters a loop where it continually evaluates states pulled from the queue. For each state, it checks whether it meets the requirements of being a solution. A state is considered a solution if it matches all the positive examples and none of the negative examples. This validation ensures that the regex pattern adheres to the constraints of the problem.

If a state is not a solution, the algorithm generates potential next states. This function fills in placeholders ($\square$) in the regex pattern with all possible values, such as constants (`a`, `b`), operators ($\star$, $\cdot$ concatenation, $\cup$ union), or other elements like the empty string (ε) or empty language ($\emptyset$). The generation process explores all logical expansions of the current state, effectively branching out the search tree and allow the space of all possible regex to be searched.

$$\frac{s_1 \to s_1'}{s_1 + s_2 \to s_1' + s_2} \quad \frac{s_2 \to s_2'}{s_1 + s_2 \to s_1 + s_2'}$$

$$\frac{s_1 \to s_1'}{s_1 \cdot s_2 \to s_1' \cdot s_2} \quad \frac{s_2 \to s_2'}{s_1 \cdot s_2 \to s_1 \cdot s_2'}$$

$$\frac{s \to s'}{s^* \to s'^*}$$

$$\frac{}{\square \to a} \quad a \in \Sigma \quad \frac{}{\square \to \epsilon} \quad \frac{}{\square \to \emptyset}$$

$$\frac{}{\square \to \square + \square} \quad \frac{}{\square \to \square \cdot \square} \quad \frac{}{\square \to \square^*}$$

Figure 1: This is the formal system for the next state function that is responsible for expanding the potential states to explore.

In the exploration of states, the algorithm prioritizing states with the least cost at each step. This cost-based heuristic ensures that the algorithm iteratively narrows its search to states that are most likely to satisfy the given examples. If a state satisfies all positive examples while rejecting all negative examples, the algorithm concludes the search, identifying the synthesized regex. However, if the current state fails to meet these criteria, the algorithm expands its scope by exploring new states derived from the current representation, continuing the search in a systematic and cost-efficient manner. This greedy approach allows the algorithm to explore a theoretically infinite space of potential regex expressions while focusing computational resources on the most promising paths.

To optimize the search and reduce computational time, AlphaRegex incorporates a set of powerful heuristics. One key optimization is dead state elimination, which ensures that states incapable of yielding a valid solution are pruned early. Dead states are identified using two complementary techniques: over-approximation and under-approximation. In over-approximation, all placeholders in a state are replaced with wildcard expressions (e.g., (.*) which represents any character for any length). If this generalized state fails to match any positive example, it is deemed a dead state. Conversely, in under-approximation, placeholders are replaced with the empty language, producing a highly restrictive form of the regex. If this restricted state matches any negative example, it is similarly classified as a dead state. By eliminating such states, the algorithm avoids unnecessary exploration of unproductive paths.

Additionally, AlphaRegex employs heuristics to detect and remove redundant states, further streamlining the search process. The redundant states heuristic involves unrolling the stars in the regex states. As we go through and unroll various stars that exist in the state, many different variations that are future potential states can be measured now once again using the overestimate technique. If any of the generated varying states do not pass all of the positive examples, then this state is thus redundant and should also be eliminated. These simplifications reduce the complexity of the state space while preserving the integrity of the search.

At its core, the algorithm employs a priority queue to manage the states (regex patterns)

under consideration. The queue ensures that states with the highest likelihood of success—those with lower costs—are explored first. This prioritization helps streamline the search by focusing on the most promising options, minimizing unnecessary computation.

3.1.2 Role of Weights in Prioritizing Solutions

The cost function in AlphaRegex plays a pivotal role in determining the priority of states during the search process. By assigning weights to individual regex elements such as union ($\cup$), concatenation ($\cdot$), star ($*$), symbols, and holes ($\square$), the algorithm calculates the total cost of a state by traversing its structure and summing these weights. This traversal ensures that the cost reflects the complexity of the regex, with simpler states generally having lower costs.

The weights assigned to each element influence the order in which states are expanded. For instance, states with fewer operations or simpler structures are prioritized, as they are more likely to yield concise solutions. This prioritization aligns with the algorithm's goal of finding the shortest valid regex first, ensuring efficiency in the search process.

By dynamically adjusting the weights, the algorithm can adapt its exploration strategy to emphasize certain types of solutions or balance between exploration and exploitation. This flexibility makes the cost function a critical component in guiding the search towards optimal solutions while avoiding unnecessary computation.

3.1.3 Implementation Notes

The implementation of AlphaRegex relies on a binary-tree-like structure to represent regular expressions. Each node in the tree corresponds to a regex element, with leaf nodes representing constants (symbols) or holes, and internal nodes representing operators such as concatenation, union, or star. For example, the regex "aabb" could be represented as a tree where the root is a concatenation node with two children as concatenations with their corresponding children: the symbols "a" twice, "b" twice.

We decided on an implementation that allows unions and concatenations to have an arbitrary number of children, rather than being restricted to binary operators for a better time simplify-

ing equivalent regex to something more human readable. This flexibility enables the representation of complex regex patterns while maintaining a clear hierarchical structure. The tree structure facilitates efficient manipulation and traversal during the search process, allowing for easy expansion and pruning of states.

When the regex is needed in its standard form, the tree is flattened into a string representation. This flattening process traverses the tree in a depth-first manner, converting the hierarchical structure into a linear sequence of regex elements. This approach ensures that the tree representation remains efficient for manipulation during the search process, while still allowing for easy conversion to a usable regex format when required. For further details and to actually use the prototype, refer to the exact source code provided in Appendix A.

3.2 L* Learner Algorithm

The L* learner algorithm, introduced by Dana Angluin in 1987, is a foundational method in the field of formal language learning. It is designed to infer a deterministic finite automaton (DFA) that recognizes an unknown regular language through a series of structured interactions with a minimally adequate teacher (MAT). The MAT provides answers to membership queries, indicating whether a given string belongs to the language, and supplies counterexamples when the learner's conjectured DFA is incorrect. By iteratively refining its hypothesis based on this feedback, the L* algorithm converges on a correct DFA that accurately represents the target language. [1]

Unlike exhaustive search-based approaches like AlphaRegex, the L* algorithm is procedural and always yields a proposal DFA in poynomial time, leveraging the structure of regular languages and the power of queries to minimize the number of interactions required. The following sections delve into the design, implementation, and challenges of the L* algorithm, highlighting its role as a cornerstone in the study of formal methods and automated learning.

3.2.1 Algorithm

As seen in Algorithm 2, at the core of L* is the construction and refinement of an observation table, which serves as the foundation for building a deterministic finite automaton (DFA). The MAT facilitates this process by providing answers to membership queries and supplying coun-

terexamples when the learner's conjectures are incorrect. These interactions allow the observation table to capture detailed information about the regular set being learned, classifying strings over the alphabet based on their membership in the set. As the learner proposes hypotheses about the nature of the regular set, the MAT tests these hypotheses against the actual set. Counterexamples are particularly valuable as they highlight specific discrepancies between the learner's current understanding and the true regular set, guiding the learner to adjust its hypothesis in a direction that systematically reduces these discrepancies. This iterative refinement continues until the learner's model accurately reflects the regular set, as verified by the absence of further counterexamples from the MAT.

Algorithm 2 L* Observation Table Construction

$S \leftarrow \{\lambda\}$
$E \leftarrow \{\lambda\}$
$A \leftarrow$ alphabet
$T \leftarrow$ membership in language
$O \leftarrow (S, E, T)$
repeat
    while O is not closed or consistent **do**
        if O is not closed **then**
            find $s_1 \in S$ and $a \in A$
            such that $row(s_1 \cdot a) \neq row(s)$ for all $s \in S$
            add $s_1 \cdot a$ to S
        end if
        if O is not consistent **then**
            find s_1 and $s_2 \in S$, $a \in A$, and $e \in E$
            such that $row(s_1) = row(s_2)$ and $T(s_1 \cdot a \cdot e) \neq T(s_2 \cdot a \cdot e)$
        end if
    end while
until Teacher agrees with conjecture M
return M

An example of the L* algorithm in action can be found in Appendix B. The algorithm begins with an empty observation table, which is iteratively filled with strings and their corresponding membership statuses. The MAT provides feedback on the membership of these strings, allowing

the learner to refine its understanding of the language. As the observation table evolves, it becomes closed and consistent, indicating that it accurately represents the language being learned.

3.2.2 Implementation Notes

To convert the DFA generated by the L* algorithm into a regular expression, we employ Brzozowski's algebraic method. This method systematically eliminates states from the DFA while preserving its language, ultimately reducing the DFA to a single regular expression that represents the language. The process involves constructing equations for each state in the DFA, where the equations describe the transitions and accepting conditions of the states. By iteratively solving these equations and eliminating states, we derive a regular expression that captures the language of the DFA.

The implementation of this method begins by representing the DFA as a system of equations, where each equation corresponds to a state and expresses its behavior in terms of transitions to other states. For a state q_i, the equation takes the form:

$$R_i = \bigcup_{q_j \in Q} \sigma_{ij} \cdot R_j \cup \varepsilon \text{ (if } q_i \text{ is accepting)},$$

where σ_{ij} represents the transition label from q_i to q_j, R_j is the regular expression for state q_j, and ε accounts for the possibility of q_i being an accepting state.

Algorithm 3 Brzozowski's State Elimination Method

```
Initialize equations for all states in the DFA
while more than one state remains do
    Select a state q to eliminate
    for each pair of states (q_i, q_j) with transitions through q do
        Update the equation for q_i to bypass q
    end for
    Remove q and its equation from the system
end while
return the regular expression for the remaining state
```

States are eliminated, as shown in Algorithm 3, one by one by substituting their equations into the remaining equations, effectively removing the state while preserving the language. This process continues until only the start state and a single accepting state remain, at which point the resulting equation directly represents the desired regular expression.

This method ensures that the resulting regular expression is equivalent to the language recognized by the original DFA. By integrating this conversion step, we standardize the output of the L* algorithm, enabling direct comparison with the results of the AlphaRegex algorithm. [2]

3.3 LLM-Generated Regular Expressions

With the advent of large language models (LLMs) such as OpenAI's GPT-4o, generating regular expressions has become more accessible to users without extensive knowledge of formal languages. To assess the performance of GPT-4o, we provided it with a series of natural language prompts describing simple regular expression tasks. These prompts were designed to elicit basic regex patterns, such as matching specific strings, character classes, or simple repetitions. For consistency, the prompts avoided complex constructs or ambiguous descriptions, ensuring that the generated regex could be directly compared to those produced by algorithmic methods.

To engineer a prompt that achieves the objectives of accepting the same input format as AlphaRegex and constraining the output to a standard set of regex operations, we adopt a structured and systematic approach. The prompt is designed to first provide a natural language description of the desired regex, followed by a list of positive and negative examples. This ensures that the input format aligns with the requirements of AlphaRegex, facilitating direct comparison between the outputs of the LLM and the algorithm.

To enforce the use of a limited set of regex operations, the prompt explicitly instructs the model to generate regex patterns using only concatenation, union, and the Kleene star. By clearly specifying these constraints, the prompt ensures that the generated regex adheres to the standard formalism, enabling rigorous evaluation of the LLM's performance in synthesizing regex patterns within these boundaries.

The structured format of the prompt, combined with explicit constraints on the output,

ensures consistency and comparability across different methods of regex synthesis. This approach not only facilitates the evaluation of LLM-generated regex but also highlights the potential of natural language interfaces in formal language tasks.

4. EVALUATION METRICS

To evaluate the performance of the synthesis methods explored in this thesis—AlphaRegex, the L* algorithm, and large language models—we adapted a benchmark suite of 25 regular expression problems originally proposed* in the AlphaRegex paper [6]. These tasks include string containment, structural alternation, and sequence matching. As shown in Figure 2, each benchmark begins with a natural language description of the regex problem, followed by a set of positive and negative examples. The benchmarks are loosely organized by increasing difficulty.

Example of a benchmark file used in the evaluation process.

```
w contains the substring abab
++
abab
XababX
XXababXX
--
a
b
XX
XXX
XXXa
XaXX
bXXX
XXbX
```

Figure 2: Example of a benchmark task to find a string that contains "ab". The specificiation starts with a natural language description, followed by positive and negative examples. The "X" in the individual examples represent that this example exists with any character in the alphabet.

*While working with the original benchmarks, we identified an issue in the benchmark examples for task 2 ("Ends with ab"). Specifically, the negative examples did not account for the case where the string consists solely of the character 'b', which does not end with "ab". This oversight could lead to incorrect evaluations of synthesis methods. We updated the benchmark to include this missing negative example, ensuring a more robust evaluation.

Table 1: Summary of the 25 benchmarks, including the number of positive and negative examples, along with the expected human-generated regex solutions.

No	Description	Pos	Neg	Human-written
1	Starts with a	3	3	$a.*$
2	Ends with ab	3	7	$.*ab$
3	Contains the substring abab	3	8	$.*abab.*$
4	Begins with b and ends with a	3	5	$b.*a$
5	Length is at least 3 and the 3rd symbol is a	3	4	$..a.*$
6	Length is a multiple of 3	2	3	$(...)*$
7	Number of a's is divisible by 3	8	7	$(b\|(ab*ab*a))*$
8	Even number of a's	7	7	$(b\|ab*a)*$
9	Fifth symbol from the right is b	3	3	$.*b....$
10	Alternating a and b	9	8	$b?(ab)*a?$
11	Each a in w is followed by at least one b	7	6	$(b\|ab)*$
12	$a^n b^m$ where $n \geq 3$ and m is even.	6	5	$aaaa*(bb)*$
13	Have at most two a's	7	5	$b*a?b*a?b*$
14	Starting a and odd length or b and even length	5	5	$a.(..)*\|b(..)*$
15	Length greater than 1	3	2	$...*$
16	Does not end with ab	8	3	$.\|.*(a\|ba\|bb)$
17	Contain at least one a and at most one b	6	9	$a*(ab?\|b?a)a*$
18	At least two b between any two a	7	7	$(b*\|abbb*a)*$
19	Does not contain baa	8	4	$a*(b\|ba)*$
20	Every odd position is b	6	9	$(b.)*b?$
21	Consecutive pair aa appears exactly once	4	4	$(a?b)*aa(ba?)*$
22	Length ≥ 2 and does not end with ba	9	4	$.*(aa\|bb\|ab)$
23	Even a's, each a is followed by at least one b	6	9	$(b\|abb*ab)*$
24	All adjacent a's appear before any adjacent b's	9	9	$(b?a)*(a?b)*$
25	At most one pair of consecutive b's	5	5	$(b?a)*bb(ab?)*$

As the different algorithms require different types of input, the evaluation process was tailored accordingly. ChatGPT and AlphaRegex were provided with the benchmark file in its entirety, including the natural language description and examples. For the L* algorithm, the natural language description was interpreted by a human to manually construct the required membership and equivalence queries.

4.1 Metrics

To comprehensively compare synthesis approaches, we defined three categories of evaluation metrics: Correctness was evaluated by determining whether a solution satisfied all positive

examples and rejected all negative examples. Final validation was performed via human inspection to account for edge cases or oversimplified expressions. Efficiency metrics included the number of states explored for AlphaRegex, the number of membership and equivalence queries made for L*, and the wall-clock time required to generate results for all three methods. Complexity was assessed in three ways: total character count, the number of operators, and a subjective readability score ranging from 1 (unreadable) to 5 (clean and intuitive).

4.2 Experimental Setup

We implemented AlphaRegex and L* in Python and the code is readily available as seen in Appendix A. LLM queries were issued through the OpenAI GPT-4o API (March 2025 snapshot) with a defined system prompt detailing the upcoming inputs.

5. RESULTS AND DISCUSSION

This chapter presents the results of our empirical evaluation and synthesizes key insights from the behavior of the three synthesis methods studied—AlphaRegex, the L* algorithm, and LLMs. We evaluate these methods on a diverse benchmark suite of 25 regex generation tasks, analyzing their correctness, performance, and complexity across both quantitative and qualitative dimensions.

5.1 AlphaRegex Results

In the original configuration, weights, and heuristics as described in the original paper, the findings indicate that the heuristics do not consistently improve wall-clock time performance as seen in Table 2. This is likely due to the overhead associated with verifying solutions against all examples. Specifically, the redundancy elimination heuristic requires significant computational effort, as it involves copying and branching out potential equivalent states, which must then be validated against the entire set of examples.

Table 2: Comparison of regex synthesis methods based on wall-clock time. Skipped entries indicate that the algorithm failed to find a solution within the 10,000-second time limit. "Full" refers to the complete process with all heuristics enabled, "No Apr" excludes over-approximation, and "No Rd" disables redundancy elimination. Times are reported in seconds.

No	Description	Full	No Apr	No Rd	Output
1	Starts with a	0.011	0.008	0.002	$a.*$
2	Ends with ab	36.06	50.96	16.99	$.*ab$
4	Begins with b and ends with a	0.077	0.101	0.044	$b.*a$
5	Length is at least 3 and the 3rd symbol is a	0.587	0.585	0.284	$..a.*$
6	Length is a multiple of 3	1.39	1.51	0.254	$(...)*$
8	Even number of a's	55.25	84.86	11.59	$(b*(ab*a)*)*$
11	Each a in w is followed by at least one b	9.63	9.71	1.58	$(b*(.b)*)*$
15	Length greater than 1	0.058	0.0833	0.018	$..*.$

Additionally, the implementation appears to encounter issues in scenarios requiring union operations. In the cases where the synthesis succeeds, the generated solutions exclusively utilize concatenation and Kleene star operations, with no unions present. This suggests a potential flaw in the implementation when handling union operations.

Table 3: Performance comparison of regex synthesis methods using the number of states explored/expanded and considered as potential candidate solutions, from the same setup as Table 2.

No	Description	Full	No Apr	No Rd
1	Starts with a	18	23	12
2	Ends with ab	20389	75483	31235
4	Begins with b and ends with a	95	238	151
5	Length is at least 3 and the 3rd symbol is a	453	1195	597
6	Length is a multiple of 3	1143	3870	487
8	Even number of a's	17124	62688	17638
11	Each a in w is followed by at least one b	3510	17638	3105
15	Length greater than 1	78	202	3105

Despite these challenges, the over-approximation and under-approximation heuristics occasionally provide substantial time savings, making their inclusion worthwhile in certain cases. However, when examining the number of states explored in Table 3, the heuristics consistently demonstrate their effectiveness in reducing the search space. They reliably prune unnecessary states and guide the search process in a more efficient direction. This aligns with the findings of the original paper, which reported that while the heuristics were highly effective at limiting the search space, they did not always reduce the time required to find a solution. The overhead introduced by the heuristics could potentially be optimized further in the current implementation to improve overall performance.

AlphaRegex tends to produce regex expressions that are relatively readable, primarily due to its greedy nature in selecting solutions. This approach often prioritizes simplicity and clarity, making the generated regex easier to interpret and understand. However, this same greedy strategy

can lead to challenges when dealing with tasks that require deeper nesting or involve ambiguous patterns. In such cases, the algorithm may fail to explore alternative solutions that could better capture the desired behavior. Additionally, the performance and output of AlphaRegex are highly sensitive to the tuning of its weights. Small adjustments to these weights can significantly influence the synthesis process, potentially leading to either improved results or suboptimal solutions. This sensitivity underscores the importance of careful parameter selection to achieve the desired balance between efficiency and correctness.

5.1.1 Weight Analysis

We experimented with three different weight configurations in AlphaRegex to investigate how heuristic prioritization affects the quality and structure of the synthesized regex. Each configuration modifies the internal cost assigned to various regex components—such as holes ($\square$), union ($\cup$), concatenation ($\cdot$), Kleene star ($*$), and constants (e.g., a, b, or wildcards)—with the goal of steering the search process toward expressions with desired properties.

In the official release of the AlphaRegex source code, the default weights are as follows:

- Hole: 100
- Union: 30
- Concatenation: 5
- Star: 20
- Constants: 20

To calculate the total cost of a given regex, we implemented a function that parses the regex and counts the occurrences of each component: holes ($\square$), union ($\cup$), concatenation ($\cdot$), Kleene star ($*$), and constants (e.g., a, b, or wildcards). The total cost is then computed as the weighted sum of these counts, using the weights defined for each component. This approach ensures that the cost reflects the structural complexity of the regex, aligning with the heuristic priorities set in AlphaRegex.

These values were selected to favor expressions with fewer holes (encouraging completion), lower branching (via limited union), and moderately complex structure. However, the effect of these weights on real-world benchmark performance remained unclear without experimentation.

To explore this, we varied the weight assigned to the hole (□) relative to the other operators, simulating how the algorithm would behave under different cost-driven priorities. By lowering the hole weight to the same level as other operators (or even lower), the search process became more eager to expand unfinished expressions. Conversely, increasing the hole cost biased the algorithm toward deeper search within fewer partially completed candidates.

Across multiple benchmarks, we observed that modifying hole weight significantly impacted runtime and regex complexity. In configurations with lower hole penalties, AlphaRegex often generated more complete but verbose expressions quickly. However, these expressions were not always optimal in length or readability. In contrast, higher hole weights forced more cautious exploration, sometimes resulting in simpler and more elegant regex, albeit with longer execution time or failure to converge.

These results highlight the delicate balance between search breadth and depth in cost-based synthesis. While AlphaRegex does not guarantee optimal solutions, adjusting its weight configuration provides a powerful lever for navigating the trade-offs between correctness, simplicity, and efficiency. Further work could include dynamic weight tuning based on benchmark characteristics or integrating learned heuristics from past runs.

Table 4 summarizes the effect of different hole weight settings on runtime and regex length for a sample of benchmarks. Lower hole weights result in faster expansion but tend to produce longer and occasionally incorrect expressions, while higher weights offer greater precision at the cost of performance.

By not prioritizing the hole in reducing its value, we observed a significant increase in the number of states explored before arriving at a solution—or, in some cases, failing to converge entirely. This behavior underscores the critical role of the hole weight in guiding the search process. Lower hole weights encourage the algorithm to expand incomplete expressions more aggressively,

Table 4: Impact of Hole Weight on States Explored and Regex Output (Example Benchmarks)

Benchmark	**Hole Weight**	**States Explored**	**Regex Output**
Begins with b and ends with a	100	94	$b.*a$
Begins with b and ends with a	30	257	$b.*a$
Begins with b and ends with a	10	3609	$b.*a$
Even number of as	100	17425	$(b*(ab*a)*)*$
Even number of as	30	>20000	–
Even number of as	10	>13000	–
Len $\geq$ 3 and 3rd symbol is a	100	416	$..a.*$
Len $\geq$ 3 and 3rd symbol is a	30	1747	$..a.*$
Len $\geq$ 3 and 3rd symbol is a	10	>8000	–

leading to a broader but less focused exploration of the search space. As a result, the synthesis process becomes less efficient, often requiring substantially more computational resources to identify a valid solution.

For example, in the benchmark task "Begins with b and ends with a," reducing the hole weight from 100 to 10 increased the number of states explored from 94 to 3609, as shown in Table 4. While the final regex output remained correct, the additional computational effort highlights the inefficiency introduced by deprioritizing the hole. In more complex benchmarks, such as "Even number of as," the algorithm failed to converge within a reasonable time frame when the hole weight was set too low, further emphasizing the importance of this parameter in maintaining a manageable search space.

These findings suggest that the hole weight should be carefully tuned to balance exploration and efficiency. While lower weights may occasionally yield more complete solutions faster, they often do so at the cost of increased computational overhead and reduced likelihood of convergence in complex tasks. Conversely, higher hole weights promote a more focused and deliberate search, improving the algorithm's ability to synthesize concise and correct regex expressions within a reasonable timeframe.

Finally, we experimented with significantly increasing the cost of specific operators, such as setting their values to an extremely high number like 10,000. When any operator is assigned

such a disproportionately high cost, the algorithm avoids using it entirely, leading to solutions dominated by other operators repeatedly chained together. This behavior prevents the algorithm from exploring solutions that involve the expensive operator, even when it is necessary for correctness.

Based on these observations, we recommend assigning the highest cost to holes to prioritize their resolution, while keeping the costs of all other operators within a reasonable range, such as between 1 and 100, and notably, within two orders of magnitude of each other, to achieve optimal performance.

5.2 L* Algorithm Results

For the initial examples, the majority of proposals tend to default to the empty language. This occurs because the observation table often becomes satisfied and complete once the empty string, single 'a', and single 'b' are all rejected, leading to the conclusion that no strings are accepted.

Table 5: The results of the benchmarks synthesized by the L algorithm, demonstrating its performance in generating regular expressions. The Q column refers to the number of membership queries and the P column refers to the number of equivalence queries verifying proposals.*

No	Description	Q	P	Output
1	Starts with a	14	2	$a.*$
2	Ends with ab	21	2	$b*a(a\|bbb*a\|ba)*b$
3	Contains the substring abab	29	2	$b*aa*b(aaa*b\|bb*aa*b)*ab.*$
4	Begins with b and ends with a	27	3	$bb*a(bb*a\|a)*$
5	Len ≥ 3 and 3rd symbol is a	44	3	$..a.*$
6	Len $\%3 = 0$	17	2	$(...)*$
7	Number of a's is divisible by 3	14	2	$(b\|ab*ab*a)*$
8	Even number of a's	5	1	$(b*a(ab*a\|b)*ab*\|b*)$

The L* algorithm generates minimal DFAs, though these do not always correspond to minimal regex, especially when guided by specific counterexamples. Its effectiveness is highly dependent on the quality of the teacher, as the teacher provides membership and equivalence queries to refine the DFA. This approach is best suited for simpler patterns or cases where the teacher can provide

precise and consistent feedback.

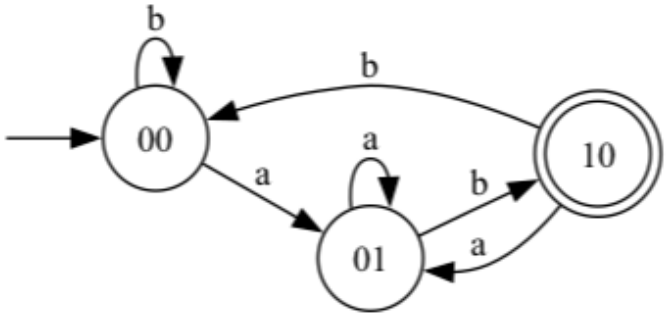

Figure 3: The solution DFA machine to capture a string that ends with ab.

The second example, as shown in Figure 3, demonstrates a straightforward DFA. However, the resulting regex is unexpectedly long and complex. This is due to the collapsing process of the DFA and the methodology used to derive the regex, as discussed earlier. Similarly, Figure 4 illustrates another simple DFA, yet the derived regex exhibits similar verbosity, highlighting the challenges in minimizing expressions during the conversion process.

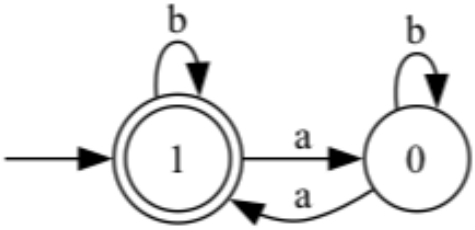

Figure 4: The solution DFA machine to capture a string that has an even number of a's.

The L* algorithm's reliance on human-provided membership and equivalence queries can lead to a significant number of interactions before arriving at a correct solution. While the algorithm is guaranteed to converge to the correct answer eventually, the process can be tedious and time-consuming, especially for complex benchmarks. This highlights the importance of optimizing

the query generation process or integrating automated tools to reduce the burden on human participants. Despite these challenges, the L* algorithm remains a powerful method for synthesizing minimal DFAs and corresponding regex solutions, particularly when paired with a well-designed teacher or automated query system.

5.3 LLM Results

Developing an effective prompt for LLMs to generate regex using only the standard set of operators proved to be an iterative and somewhat delicate process. One of the persistent challenges was conveying that the input alphabet consisted solely of the characters a and b. Despite multiple prompt refinements, the model frequently interpreted this constraint as a directive to use the explicit disjunction $(a|b)$, rather than the shorthand . to represent any character within the defined alphabet. This behavior reflects a broader limitation of LLMs: they lack a formal understanding of semantic equivalence. In this case, they fail to recognize that . and $(a|b)$ are functionally identical under a fixed alphabet.

Another issue surfaced as the prompt history grew longer. Initially, the system prompt reliably constrained the model to use only standard regex operators. However, as additional benchmark prompts were introduced in sequence, the model began to drift from the original specification. Around benchmark 12, responses increasingly violated earlier constraints, suggesting that the LLM was "forgetting" the original instructions. This degradation highlights a fundamental limitation in prompt-based control: when engaging with LLMs interactively over extended sequences, it becomes difficult to maintain consistency, especially when relying solely on static system prompts.

The model's reasoning style evolved over two sets of conversations. For the initial set of benchmarks (1-11), it generated one-shot answers—regex expressions without any accompanying explanation. These outputs were all correct and closely matched (or even exactly replicated) the manually written regex solutions, suggesting the model was likely retrieving or reconstructing familiar patterns from its training data. However, starting with benchmark 12 in a new conversation––—without system-level reminders or reissued constraints——the model shifted its approach. It began breaking down the prompt into logical steps, reasoning through the requirements before

proposing a solution. This reasoning-based approach yielded correct results for benchmarks 13, 17, and 22. However, these solutions often relied on enumerating all possible cases that satisfied the requirements, resembling a verbose switch-case structure in program synthesis. While functional, this approach lacks elegance and can hinder readability and maintainability.

One hypothesis is that the increased complexity of benchmark 12 prompted this behavioral shift. Once the model encountered a problem that didn't map cleanly onto a memorized pattern, it defaulted to reasoning. While sometimes helpful, this strategy proved less effective for simpler problems, where direct pattern recognition would have sufficed. As the benchmarks continued to grow in difficulty—particularly those involving negation or constraints requiring implicit counting—the model increasingly produced incorrect regex. Notably, it struggled with counting constructs, often failing to distinguish between mandatory and optional repetitions. For example, in attempting to describe "at least three a's," it returned $aaa*$ instead of the correct $aaaa*$, mistakenly treating the third a as optional.

Overall, the LLM's early success with one-shot outputs correlated strongly with benchmark tasks that were simple and structurally familiar. As task complexity increased, the model's shift toward reasoning introduced more nuanced errors, often stemming from overgeneralization or misinterpretation of constraints. Nonetheless, even when incorrect, the outputs frequently resembled the structure of a valid solution—indicating that the model had some abstract grasp of the form, even if the details were flawed. These results reinforce the broader theme that LLMs excel at recalling surface-level patterns from training but falter under compositional reasoning, especially in domains that demand formal correctness.

> This GPT takes a structured prompt consisting of a natural language description, a list of positive string examples, and a list of negative string examples. The format is: a one-line description, followed by '+++', then one or more positive examples, followed by '—', then one or more negative examples. Based on this, it generates a minimal theoretical regular expression using only the operations of concatenation, union, Kleene star, and the wildcard symbol '.'. It ensures that the generated expression accepts all

positive examples and rejects all negative examples. The '?' symbol may be used to denote either a term or the empty language, when necessary. The alphabet is limited to 'a' and 'b', and the character 'X' in any input stands for a wildcard representing either 'a' or 'b'. When the output regex needs to match either 'a' or 'b', it uses '.' instead of '(a|b)' for minimality.

Table 6: The results of the benchmarks synthesized by GPT-4o, demonstrating the performance of the LLM in generating regular expressions.

No	Description	Output	Correct?
1	Starts with a	$a.*$	✓
2	Ends with ab	$.*ab$	✓
3	Contains the substring abab	$.*abab.*$	✓
4	Begins with b and ends with a	$b.*a$	✓
5	Length is at least 3 and the 3rd symbol is a	$..a.*$	✓
6	Length is a multiple of 3	$(...)*$	✓
7	Number of a's is divisible by 3	$(b*ab*ab*ab*)*b*$	✓
8	Even number of a's	$(b*ab*ab*)*b*$	✓
9	Fifth symbol from the right is b	$.*b....$	✓
10	Alternating a and b	$(ab)*(a)?\|(ba)*(b)?$	✓
11	Each a in w is followed by at least one b	$(b\|ba)*$	✓
12	$a^n b^m$ where $n \geq 3$ and m is even.	$aaa.*(bb)*$	✗
13	Have at most two a's	$b*\|b*ab*\|b*ab*ab*$	✓
14	Starting a and odd length or b and even length	$a(..)*\|b(..)*$	✗
15	Length greater than 1	$..*$	✗
16	Does not end with ab	$?\|.\|.*(aa\|ba\|bb)$	✗
17	Contain at least one a and at most one b	$aa*\|aa*ba*\|a*baa*$	✓
18	At least two b between any two a	$b*a(bb*a)*b*$	✗
19	Does not contain baa	$(aa\|ab\|ba?\|bb)*a?$	✗
20	Every odd position is b	$(b.)*b?$	✓
21	Consecutive pair aa appears exactly once	$(b\|ab)*aa(b\|ab)*$	✗
22	Length ≥ 2 and does not end with ba	$.*(aa\|bb\|ab)$	✓
23	Even a's, each a is followed by at least one b	$b*(abb*abb*)*b*$	✓
24	All adjacent a's appear before any adjacent b's	$(a\|ba)*(bb(a\|b)*a?b*)?$	✗
25	At most one pair of consecutive b's	$(a\|ba)*\|(a\|ba)*bb(a\|ba)*$	✗

Upon reviewing the results in Table 6, we observe that for benchmarks 1 through 6, as well as benchmark 9, the LLM-generated regex expressions exactly match the manually authored

ground truth. These examples represent problems where the target expression is relatively simple and where only a single minimal and semantically clean regex is likely. The convergence between the LLM output and the human-written regex in these cases suggests that the model is not just generating plausible-looking patterns, but also capturing the underlying structure when it aligns well with common training examples.

For benchmarks 7 and 8, the model produced correct and valid solutions, such as $(b * ab * ab * ab*) * b*$ for benchmark 7. While semantically accurate, these expressions are longer than necessary and could be simplified. However, the verbosity in these outputs arguably improves readability by making the intended repetitions more explicit. In contrast, the ground truth prioritizes minimal string length, often at the cost of visual clarity. This discrepancy raises an interesting point: readability and brevity do not always align, and models trained on human-written code may favor explicit enumeration over abstraction.

Benchmark 10 highlights a situation where the LLM unnecessarily introduces a union to distinguish between inputs beginning with a or b. Although the result is functionally correct, the added complexity neither enhances correctness nor readability, and it contrasts with the simpler expected solution. In benchmark 11, despite its slightly increased complexity, the model again matched the correct regex exactly.

From benchmark 12 onward, the correctness rate declines. Benchmark 12 is incorrect because it allows for an odd number of b characters, violating the requirement that bs appear in even counts. Benchmarks 13 and 17, while correct in behavior, exhibit syntactic redundancy that could be removed with minor simplification. For example, patterns with nested repetitions or excessive alternations can often be flattened without affecting correctness.

Benchmarks 14, 15, and 18 all display a common off-by-one counting error—a recurring failure mode for LLMs in tasks that involve precise cardinality constraints. These errors reveal the model's difficulty in reasoning about exact quantities, a known limitation in sequence modeling.

Benchmark 16 contains invalid regex syntax, misusing the ? operator without a preceding token. The intended goal may have been to include optional content, but the structure used violates

standard regex grammar. Similarly, benchmark 19 is logically flawed: the use of $(aa|ab|ba?|bb)*$ followed by a suffix expression can incorrectly accept strings like bbaa, which should be rejected. This occurs because of incorrect assumptions about how the starred term interacts with the rest of the expression.

In contrast, benchmark 20 is entirely correct, demonstrating that the model can occasionally capture subtle constraints when the structure of the problem is well-aligned with common representations. However, benchmark 21 fails due to overpermissiveness. The pattern $aa(b|ab)*$ accepts strings like aaab, which includes two pairs of consecutive as—violating the task specification.

Benchmark 22 is nearly correct, but misses a boundary case: it fails to handle strings with fewer than two characters, which should automatically satisfy the negated condition of "not ending in ba." Benchmark 23 is both correct and highly human-readable, though there are opportunities for syntactic reduction, such as eliminating unnecessary $b*$ terms in the suffix. This speaks again to the tension between brevity and expressiveness.

Finally, benchmarks 24 and 25 are incorrect due to subtle logic flaws. In 24, the use of $(a|b)*$ is too permissive and allows the occurrence of double a after bb, violating the restriction. In 25, the model allows bbba via $bb(a|ba)*$, a construction that does not correctly constrain repetition patterns involving pairs of bs.

Overall, the model's strongest performance appears in problems that are well-aligned with familiar symbolic patterns, especially those involving prefixes, suffixes, or fixed-length structural constraints. As tasks begin to involve negation, exact counting, or nested disjunctions, correctness degrades. Notably, even incorrect outputs often resemble the correct structure, suggesting that the model maintains a surface-level understanding of the domain—but without the formal rigor needed for correctness in all cases.

5.4 Integration Synthesis

5.4.1 LLM and AlphaRegex Integration

We explored the potential of integrating LLMs with AlphaRegex to enhance regex synthesis. The idea was to leverage the strengths of both approaches: the LLM's ability to generate human-readable regex and AlphaRegex's efficiency in synthesizing correct solutions.

Though not fully implemented, we explored ideas for combining models. One early prototype used ChatGPT to generate candidate regex, which were validated and refined by AlphaRegex. This semi-automated pipeline showed promise: even when the LLM failed to produce correct output directly, its regex often seeded AlphaRegex with promising starting points. The particular regex generated by ChatGPT can be incorrect, we can construct the regex in the internal representation of AlphaRegex, and then run the synthesis algorithm to find a solution. This approach can be seen as a hybrid model that leverages the strengths of both LLMs and traditional synthesis methods.

5.4.2 LLM and L Integration*

Another promising approach involves leveraging ChatGPT to generate Python test cases for verifying the correctness of regex solutions. By providing the natural language description and examples as input, ChatGPT can output Python code that automates the validation process. This approach significantly reduces the manual effort required for constructing membership and equivalence queries in the L* algorithm. For instance, for benchmark 14, which is a challenging example, we can use an LLM to automate the membership queries. The task involves checking if the first character is 'a' and the length is odd, or if the first character is 'b' and the length is even. ChatGPT can generate Python code like the following:

```
if w[0] == 'a' and len(w) % 2 == 1:
    return True
if w[0] == 'b' and len(w) % 2 == 0:
    return True
return False
```

This Python code can then be used to act as a large art of the MAT answering the membership queries generated by the L* algorithm. By automating the verification process, this approach streamlines the synthesis pipeline and ensures consistency in evaluating correctness.

5.4.3 AlphaRegex and L Integration*

The integration of AlphaRegex and L* is another promising avenue for future work. By combining the strengths of both methods, we can potentially improve the synthesis process. For example, L* could serve as a guide to generate high-quality examples that AlphaRegex can use to refine its search. This hybrid approach could leverage L*'s ability to systematically explore the space of possible solutions while benefiting from AlphaRegex's heuristics to efficiently navigate the search space.

The integration of these two methods could also address some of the limitations observed in their individual performances. For instance, AlphaRegex's challenges with union operations could be mitigated by L*'s structured generation of counterexamples, which can help AlphaRegex focus on critical areas of the search space. Conversely, AlphaRegex's heuristics could enhance L*'s exploration by providing initial candidates or pruning less promising paths, thereby reducing the number of membership and equivalence queries required.

5.5 Discussion

5.5.1 Lessons for Broader Synthesis

The findings from our results provide several insights that can generalize to other program synthesis tasks. The effectiveness of heuristics in reducing the search space, as demonstrated with AlphaRegex, underscores the importance of carefully designed heuristics in program synthesis. However, it is crucial to balance the trade-off between computational overhead and the reduction of the search space. The performance of LLMs highlights the significance of structured and well-defined input representations. By providing clear examples and constraints, synthesis models can be guided to generate more accurate outputs. The integration of LLMs with traditional synthesis methods suggests that hybrid approaches can leverage the strengths of different paradigms. For

example, LLMs can offer initial guesses or templates, while formal methods refine and validate the solutions.

The multi-faceted evaluation metrics used in this study—correctness, efficiency, and complexity—are broadly applicable to other synthesis tasks, ensuring a holistic assessment of synthesized programs. Observing failure cases, such as the inability of AlphaRegex to handle unions effectively, emphasizes the importance of analyzing and addressing specific limitations in synthesis algorithms. The reliance on human interpretation for L* algorithm inputs and the qualitative analysis of regex readability highlight the value of human-in-the-loop systems for complex synthesis tasks. Furthermore, the time and state exploration results emphasize the need for scalable algorithms, especially as problem complexity increases. Techniques like pruning and approximation can be adapted to address scalability challenges in other domains.

The tendency of LLMs to overfit to examples illustrates the broader challenge of balancing generalization and specificity in program synthesis. These lessons suggest that combining domain-specific insights, robust evaluation frameworks, and hybrid methodologies can enhance the applicability and effectiveness of program synthesis across diverse domains.

These lessons suggest that combining domain-specific insights, robust evaluation frameworks, and hybrid methodologies can enhance the applicability and effectiveness of program synthesis across diverse domains.

6. CONCLUSION AND FUTURE WORK

6.1 Conclusion

This thesis explored how different synthesis methodologies perform in the constrained task of regex generation. By comparing AlphaRegex, the L* algorithm, and LLMs, we have answered the central research question: how do these approaches compare in correctness, efficiency, and complexity—and could they be combined for better outcomes?

AlphaRegex demonstrates the strength of heuristic search, particularly when weights are carefully tuned. Its flexibility lies in its cost function, which allows for prioritizing minimal length or computation time. However, this comes at the cost of unpredictable performance across benchmarks—too low a hole penalty may produce verbose or invalid solutions, while too high a penalty can lead to excessive runtimes or failure to converge.

The L* algorithm offers a formally grounded approach with guaranteed correctness under a proper teacher. Its output, derived from DFAs, is provably equivalent to the target language, but often lacks elegance or conciseness. Additionally, it is sensitive to the formulation of counterexamples and can be inefficient in practice on complex languages.

LLMs represent a fundamentally different paradigm: they do not search or construct programs in a structured way but instead rely on pattern recognition and learned syntax. Their performance is impressive on benchmarks resembling seen examples, yet fragile when compositional logic or negation is required. We observed degradation in performance as benchmark complexity increased, along with drift from user constraints in long prompting sessions.

Together, these findings suggest that no single method dominates; rather, each has a niche. Chaining methods—for instance, using LLMs for initial guesses, refined by AlphaRegex or verified through L*—offers a promising avenue. Our experiments with heuristic adjustments in AlphaRegex showed that even modest changes in search prioritization can lead to trade-offs in correctness and output readability, highlighting the importance of interpretability and controllability

in synthesis.

From a broader perspective, this work underscores the importance of test-driven synthesis. The quality of input-output examples or formal constraints fundamentally shapes the success of all three methods. We advocate for further research into human-in-the-loop tools that help users craft effective specifications, guided by intelligent feedback.

6.2 Future Work

From here, there are many potential directions for future work. The following list outlines some of the most promising avenues:

- **Human-in-the-loop synthesis:** Building tools that help users generate, evaluate, and refine test cases could greatly improve synthesis outcomes. Future systems might include example-suggestion engines, test coverage visualizations, or even learning-based test generators.
- **Enhanced chaining strategies:** This work offers only preliminary results in chaining LLMs with traditional synthesizers. A more robust pipeline could involve using LLMs to draft regex, AlphaRegex to refine structure, and L* to verify equivalence.
- **Generalization to other DSLs:** Although this thesis focused on regex, many insights apply to broader program synthesis contexts, including configuration synthesis, data transformation scripts, or query generators.
- **Weight learning and adaptation:** In AlphaRegex, we relied on static weight configurations. Future work could explore meta-learning or reinforcement learning to dynamically tune weights based on benchmark structure.
- **Prompt stability in LLMs:** The degradation observed across long sessions raises the need for tooling or architecture-level improvements to preserve intent and context across multiple invocations.

Ultimately, this thesis contributes both a comparative framework and practical implementations of three representative synthesis strategies. It lays the groundwork for building future sys-

tems that combine the controllability of symbolic methods with the flexibility of neural networks, all while centering the human user's intent through better test design and feedback tools.

REFERENCES

[1] Dana Angluin. 1987. Learning regular sets from queries and counterexamples. *Information and Computation* 75 (11 1987), 87–106. doi:10.1016/0890-5401(87)90052-6

[2] Janusz A Brzozowski. 1964. Derivatives of Regular Expressions. *J. ACM* 11 (10 1964), 481–494. doi:10.1145/321239.321249

[3] N. Chomsky. 1956. Three models for the description of language. *IEEE Transactions on Information Theory* 2 (09 1956), 113–124. doi:10.1109/tit.1956.1056813

[4] Justin Gottschlich, Armando Solar-Lezama, Nesime Tatbul, Michael Carbin, Martin Rinard, Regina Barzilay, Saman Amarasinghe, Joshua B Tenenbaum, and Tim Mattson. 2018. The three pillars of machine programming. *DSpace@MIT (Massachusetts Institute of Technology)* (06 2018). doi:10.1145/3211346.3211355

[5] Emanuel Kitzelmann. 2010. Inductive Programming: a Survey of Program Synthesis Techniques. *Lecture Notes in Computer Science* 5812 (01 2010), 50–73. doi:10.1007/978-3-642-11931-6_3

[6] Mina Lee, Sunbeom So, and Hakjoo Oh. 2016. Synthesizing regular expressions from examples for introductory automata assignments. *GPCE 2016: Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (10 2016). doi:10.1145/2993236.2993244

[7] Michael Sipser. 1996. Introduction to the Theory of Computation. *ACM SIGACT News* 27 (03 1996), 27–29. doi:10.1145/230514.571645

[8] Armando Solar-Lezama. 2012. Program sketching. *International Journal on Software Tools for Technology Transfer* 15 (08 2012), 475–495. doi:10.1007/s10009-012-0249-7

[9] A Solar-Lezama. 2023. Introduction to Program Synthesis. https://people.csail.mit.edu/asolar/SynthesisCourse/index.htm

[10] Patrick H. Winston. 1970. *Learning Structural Descriptions From Examples*. Technical Report. Cambridge, MA, USA.

APPENDIX A: SOURCE CODE

This appendix contains links to our open-source implementations of the algorithms discussed in this paper. Detailed instructions for usage and experimentation can be found in the README files of the respective repositories.

- **Replicating AlphaRegex Algorithm**

  Visit: `https://github.com/lilacstella/replicate_alpha_regex`

  A reimplementation of the AlphaRegex algorithm, featuring a greedy search over regex syntax trees guided by cost heuristics. Includes benchmark evaluation and custom weight tuning options.

- **Replicating L* Algorithm**

  Visit: `https://github.com/lilacstella/replicate_l_star`

  An implementation of the L* learning algorithm that synthesizes DFAs from membership and equivalence queries, with conversion to regex via Brzozowski's method.

APPENDIX B: L* REGEX GENERATION EXAMPLE

The following is the output of an example run of my implementation of the L* algorithm. The goal is to achieve the regex `.*ab.*`, but the algorithm yielded an equivalent regex. The L* application interacts with the MAT by continuously prompting for membership queries, DFA proposals, and counterexamples until the process is complete.

```
Is "" in the language? no
Is "a" in the language? no
Is "b" in the language? no
Is this regex ∅ correct? no
```

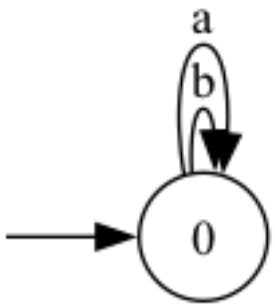

Figure B.1: The first DFA generated by L according to the answers so far, which is the empty language. This DFA is generated before asking the question, is this regex correct? The user then has the chance to view it and provide a counter example.*

```
What is a counter example? abab
Is "ab" in the language? yes
Is "aba" in the language? yes
Is "abab" in the language? yes
```

```
Is "aa" in the language? no
Is "abb" in the language? yes
Is "abaa" in the language? yes
Is "ababa" in the language? yes
Is "ababb" in the language? yes
Is "bb" in the language? no
Is "aab" in the language? yes
Is "abbb" in the language? yes
Is "abaab" in the language? yes
Is "ababab" in the language? yes
Is "ababbb" in the language? yes
Is this regex b*aa*b.* correct? yes
```

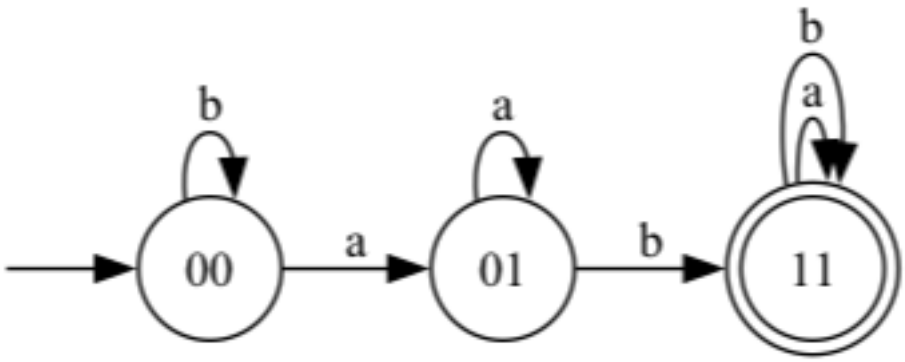

Figure B.2: This is the second DFA generated by L according to the current state of the observation table. This DFA does accurately capture that the language contains the string "ab". We accept this proposal and the algorithm concludes the regex* `b*aa*b.*` *as the final output.*