Quantum-Based SMT Solving for String Theory

Beatrice Casey
bcasey6@nd.edu
University of Notre Dame
Notre Dame, IN, USA

Joanna C. S. Santos
joannacss@nd.edu
University of Notre Dame
Notre Dame, IN, USA

Andrew Hennessee
ahenne3@nd.edu
University of Notre Dame
Notre Dame, IN, USA

ABSTRACT

Satisfiability Modulo Theory (SMT) solvers are a useful tool that can be applied to a variety of problems, such as configuring relationships in distributed systems, detecting race conditions, and program analysis. String constraints are particularly difficult for SMT solvers to navigate, as the search space is generally large. Often times, classical SMT solvers will have to quit generating a solution for string constraints because it takes too long to find the solution. Quantum computing offers the advantages of quantum mechanics (*e.g.*, superposition), which allows a system to explore a large search space much more efficiently. In this work, we explore creating a quantum-enabled SMT solver for string theory by using quantum annealing and Quadratic Unconstrained Binary Optimization (QUBO). Our preliminary results demonstrate that it is feasible to transform these string constraints to QUBO, and generate solutions for given constraints.

CCS CONCEPTS

• **Hardware** → **Quantum computation**; • **Theory of computation** → **Quantum computation theory**; *Logic and verification*; • **Mathematics of computing** → *Combinatorial optimization*; • **Software and its engineering** → *Formal software verification*; • **Computing methodologies** → Symbolic and algebraic algorithms.

KEYWORDS

Quantum Computing, Quantum Annealing, SMT Solving, QUBO

ACM Reference Format:

Beatrice Casey, Joanna C. S. Santos, and Andrew Hennessee. 2025. Quantum-Based SMT Solving for String Theory. In *Proceedings of July 20–23, 2025 (HPDC'25).* ACM, New York, NY, USA, 8 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Satisfiability Modulo Theory (SMT) is a decision problem concerned with determining whether a logical formula is *satisfiable*; that is, whether there exists an assignment to its variables that makes the formula true [1]. SMT generalizes the Boolean Satisfiability (SAT) problem by including constraints from various background theories such as arithmetic, fixed-size bit-vectors, arrays, uninterpreted functions, and strings [2]. Instead of just determining if a boolean formula has a satisfying assignment, an SMT solver checks if a formula involving variables of different *sorts* (*e.g.*, types such as integers, bit vectors, *etc*) and constraints from these theories has an assignment that satisfies both the boolean structure of the formula and the rules of the incorporated theories [3].

SMT is used in a variety of applications, such as symbolic execution [4], program verification [5], and problem partitioning [6, 7]. Consequently, the *reliability* of these applications depends heavily on the *correctness* of the answers provided by the solver [3, 8]. Fundamentally, SMT solving transforms a problem from one theory space into a boolean problem, and a SAT solver, which is responsible for the boolean structure and search, aims to find a solution to the formula. Once a potential solution is found, it is transformed back to the original theory, and checked for consistency (*i.e.*, checking whether the found solution actually satisfies the constraint). If the solver did not find a solution, it will go through the previous steps again until it finds an appropriate solution [9]. Herein lies one of the major struggles of SMT solving: as a search space becomes larger and larger, the complexity of finding a solution to a given formula also grows.

Within the landscape of SMT, the theory of strings introduces a unique set of challenges. String constraints are ubiquitous in software, particularly in applications dealing with input validation, and pattern matching [10, 11]. The SMT-LIB standard provides a specification for string operations, including deterministic semantics for operations such as `replace`, `indexOf`, `concat`, `substr`, and `length` [12]. Reasoning about these operations often needs the integration of linear integer arithmetic, further complicating the solving process [10, 11, 13]. Notably, the quantifier-free first-order string theory with regex constraints, string length arithmetic, and concatenation is undecidable. Even for decidable fragments, the computational complexity can be substantial, leading to performance bottlenecks and scalability issues in classical SMT solvers [10, 11, 13, 14].

Existing classical string solvers often rely on techniques such as automata-based methods and reductions to word equations. However, automata-based techniques can suffer from the high computational cost of operations like automata intersection, and reductions to word equations can lead to complex search spaces [13–16]. Empirical evaluations of state-of-the-art string solvers, such as Z3-seq [17], Z3str3 [18], and CVC4 [19], have shown instances of soundness bugs, completeness issues, and significant performance variability, including timeouts and non-deterministic behavior [8, 10]. This highlights the ongoing need for novel and more efficient approaches to tackle the challenges of SMT solving for strings.

Quantum Computing promises exciting opportunities across a variety of disciplines such as healthcare [20, 21], finance [22], and security [23–25]. Broadly, quantum computing is useful for solving

HPDC'25, South Bend, IN,

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/XXXXXXX.XXXXXXX

problems that are intractable or inefficient to solve using classical techniques, *i.e.*, classically hard problems. By leveraging principles of quantum mechanics, such as entanglement and superposition, quantum computers can offer speedups for certain problems compared to the best classical algorithms [26, 27].

In this work, we present a preliminary approach for a quantum-based SMT solver for string constraints using Quadratic Unconstrained Binary Optimization (QUBO) and quantum annealing. To do so, we first convert string operations into an optimization problem using QUBO form. We then pass our QUBO formulations into a quantum annealer [28], and parse the output. Our method currently supports the following string operations: string equality, substring matching, substring indexOf, string length, string replace and replace all, string reversal, string concatenation, string includes, palindrome generation, and regex matching.

The rest of the paper is organized as follows: Section 2 gives the necessary background of concepts to understand the work, Section 3 shares the related works, Section 4 describes the methodology for our solver, Section 5 shares the results of the work, and Section 6 concludes the work.

2 BACKGROUND

This section defines concepts and terminology that are key to understanding this work.

2.1 SMT Solvers

An **SMT solver** is a tool which decides the *satisfiability* of formulas within certain background theories, *e.g.*, integer arithmetic, arrays, *etc* [1]. An SMT solver can be considered as a generalization of the Boolean Satisfiability (SAT) problem. While SAT deals with the satisfiability of propositional formulas (*i.e.*, formulas created using boolean variables and logical connectives), SMT extends this by incorporating reasoning about other mathematical and logical theories [3]. For example, an SMT solver could be used to check that the formula $x + y > 10 \wedge x < 4$ is satisfiable and that possible integer values for x and y that satisfies it are 3 and 8, respectively.

The architecture of modern SMT solvers is often based on the **DPLL(T)** framework [29, 30]. This framework combines the Davis-Putnam-Logemann-Loveland (**DPLL**) algorithm that is used in SAT solvers with a specific theory solver (**T-solvers**) for all of the background theories being considered [3, 31]. The SAT solver manages the boolean structure of the formula by performing case splits and propagating truth assignments. The T-solvers are responsible for reasoning about the constraints which belong to the specific theories. They check whether the current boolean assignments provided by the SAT solver actually satisfy the theory constraints [3, 31].

2.1.1 SMT-Lib Format. The **SMT-Lib format** was introduced as a common format for all SMT problems in an effort to produce an online library of *benchmarks* for SMT solvers [12]. The main goal of the SMT-Lib initiative is to facilitate the evaluation and the comparison of SMT solvers and advance the state of the art in the field, similar to how the TPTP library has done for theorem proving and the SATLIB library for propositional satisfiability [12, 32].

In this context, a "benchmark" refers to a *logical formula that is to be checked for satisfiability for a combination of background theories.* Examples of such background theories include real and integer arithmetic, as well as theories of data structures like lists, arrays, and bit vectors [12]. The SMT-Lib format adopts an underlying logic, defines a number of background theories, and specifies a general syntax for various benchmarks, providing a way to indicate which class of formulas a particular benchmark belongs to [12, 32].

To illustrate the syntax for formulas in the SMT-lib format, consider the following example of a logic equation for linear integer arithmetic [32]:

$$(x1 + x2 > 0) \wedge (x1 + x2 < 3) \wedge (x1 = 3 * x3) \wedge (x2 = 6 * x4)$$

In SMT-Lib format, this formula is expressed as follows:

```
(and (> (+ x1 x2) 0)
     (< (+ x1 x2) 3)
     (= x1 (* 3 x3))
     (= x2 (* 6 x4)))
```

As shown above, the SMT-LIB format uses a LISP-like prefix notation, where each operator precedes its operands. Logical conjunctions (*e.g.*, `and`) group together multiple constraints, and arithmetic operations like addition are written in prefix form (*e.g.*, `(+ x1 x2)` instead of `x1 + x2`). This structured syntax allows SMT solvers to parse and reason about constraints efficiently.

2.1.2 SMT Solving Applications and Challenges. SMT solving has a variety of applications, particularly in distributed computing. For example, prior works [33] have used SMT solving to synthesize distributed systems as a solution to the exponential blowup in number of states that occurs when distributed systems are synthesized through automata transformations. Additionally, SMT solving has been used to monitor partially synchronous distributed systems in an effort to detect bugs caused by concurrency and race conditions among processes [34]. Given that the search space in distributed systems is incredibly large [35], SMT solvers can struggle to find solutions to the constraints presented to them. Thus, there is a need to have a more efficient method of solving these constraints.

Furthermore, although the DPLL(T) framework provides a theoretical foundation, the performance of SMT solvers rely heavily on various heuristics used within the SAT solver and theory solvers. In addition, formula pre-processing is another crucial component which is also often driven by heuristics [31].

2.2 Quantum Computing

Quantum Computing (QC) is a computing method that leverages the principles of quantum mechanics, such as *entanglement* and *superposition*. Unlike a classical bit, which can be either `0` or `1`, a ***qubit*** (quantum bit) can exist in a *superposition* of both `0` and `1` simultaneously, represented as a combination of both states with certain probabilities. Moreover, two or more qubits can be entangled, *i.e.*, their states become correlated such that the state of one instantly affects the state of the other, regardless of distance. Entanglement allows quantum computers to represent and process exponentially more states than classical ones with the same number of bits [36].

Quantum computers show the most promise in four broad problem categories: combinatorial optimization, problems based in linear algebra, problems involving differential equations, and factorization. These problem types underlie many potential quantum applications under exploration by industry, including drug discovery, logistics, cryptography, and machine learning. As such, they

represent areas where quantum computing may offer competitive advantages over classical approaches [36].

2.3 QUBO

Quadratic Unconstrained Binary Optimization (QUBO) is a versatile combinatorial optimization model with a wide range of applications [37]. *Combinatorial optimization* defines finding one or more optimal solutions to a problem. The solutions for these problems are searched for in a very large configuration space, with the set of possible solutions being defined with certain constraints. The goal is to optimize the objective function with the best solution.

QUBO problems are made up of three major components: *binary variables*, an *objective function*, and *penalty functions*. The **binary variables** represent decision points in the optimization problem. The **objective function** is the function that is designed to be minimized. The solution of the function will result in the lowest energy for the system, thus returning the most optimal solution for the set of variables provided. **Penalty functions** are an optional addition, which will guide the solution to the most optimal state by adding energy to the system when certain constraints are violated (*e.g.*, when the solution is not optimal) [37].

QUBO allows for flexibility in the way that problems are defined. It can be used for a variety of applications, such as scheduling, traffic management, supply chain optimization, optimizing public transportation schedules/routes, and even the map-coloring problem [36, 38, 39]. QUBO problems are particularly suited for quantum annealing due to its cost function being equivalent to an Ising model. This makes it such that its global optimum can be approximated by quantum annealing [38].

3 RELATED WORKS

Applications of quantum computing, particularly to the domain of software engineering, is still largely in its infancy. However, there have been multiple works which have begun exploring the use of quantum computing across a wide range of software engineering related tasks, as well as software engineering techniques applied to quantum computing.

3.1 Quantum Software Engineering

Existing research has looked into Quantum Software Engineering, working towards having usable and trustworthy quantum programs. Quantum Software Testing (QST) is a crucial area, with numerous approaches being explored to ensure the reliability of quantum programs. These include: statistical testing [40], assertion-based testing [40, 41], and mutation testing [42]. Statistical testing relies on sampling to verify program outputs. Existing sample methods can be costly in terms of measurements. Kang *et al.* use techniques such as quantum amplitude amplification to improve the testing of these programs. [40]. Assertion-based testing ensures that conditions are validated during execution [40], and mutation testing assesses the effectiveness of a test suite by introducing faults [42]. Other techniques, like metamorphic testing, property-based testing, coverage-guided test generation, search-based testing, and differential testing are also being investigated as possible methods for QST. [41, 42]. Another area in quantum software engineering is the mitigation of noise. Noise is an issue for quantum computers, which can greatly impact their reliability and effectiveness. Muqeet *et al.* investigate mitigation techniques for noise in quantum computing, as well as the challenges posed by this noise [42].

Further, the development of quantum programming languages and compilation tools has also been explored. In this context, ScaffCC is a notable compiler aiming for efficient compilation and analysis of quantum programs, focusing on modularity and resource analysis [43]. Javadi-Abhari *et al.* also consider the evolution of QASM as a potential quantum assembly language [43]. Finally, Jeon *et al.* have extended Probabilistic Model Checking (PMC) to the quantum domain by developing Quantum Probabilistic Model Checking (QPMC) using Quantum Amplitude Estimation to address the limitations of classical PMCs due to state explosion [44].

3.2 Quantum Algorithms

Recent works have investigated use and development of quantum algorithms in various applications. Lin *et al.* [9] create a quantum SMT solver for bit vector theory. Unlike our work, Lin *et al.* focus on *bit-vector* theory. Additionally, Lin *et al.* focus on quantum circuit execution and require universal quantum computers as their approach, whereas our work takes an optimization-focused and annealing-compatible approach, with a focus on complex string constraints. Another application of quantum algorithms is to the fundamental problem of string matching. Cantone *et al.* [45] provided a detailed analysis and extension of a quantum string matching algorithm by Niroula-Nam [46], including approximate matching with swaps. Faro *et al.* [47] explored a method to translate classical bit-parallel string matching techniques into quantum algorithms, achieving a quadratic speedup via Grover's search. Marino *et al.* [48] presented the first practical implementation of a quantum string matching circuit using Qiskit for binary strings. Furthermore, Faro *et al.* [49] introduced the quantum path parallelism (QPP) approach, a general strategy to adapt quantum computation to various nonstandard text searching problems by using automata-based string recognition. While all these works relate to string matching, our work focuses on a generalized approach to a variety of string operations. Additionally, these works use circuit-based methods in their approach, while we use quantum annealing and QUBO for our string operations.

The Quantum Approximate Optimization Algorithm (QAOA) is an area of research under quantum algorithms and applications. Studies focus on the ansatz design, parameter optimization, efficiency, and hardware implementations of QAOA [50, 51]. Grover's algorithm is another fundamental algorithm with applications in areas like quantum SMT solving [9, 52].

4 A QUANTUM APPROACH TO SMT SOLVING

In this paper, we describe an approach that leverages the power of quantum computing to better scale SMT solvers. Specifically, we focus on solving string constraints by formulating the SMT solving problem in QUBO form to be solved by a quantum annealear. As shown in Figure 1, our approach takes as input the operation performed (*e.g.*, string concatenation, *etc*), and a list of any other required arguments, such as the length of the output string in generation cases, substrings to include for substring matching or substring index of, characters to replace for string replace and

replaceAll, strings to concatenate for string concatenation, strings to reverse for string reversal, a regex pattern for regex matching, or two strings for string includes.

Subsequently, our approach generates out *binary variables*, which are the binary representations of our strings, and encode our *objective function* into a QUBO matrix. For some operations, we include *penalty functions* to further ensure that the annealer finds the optimal solution, and this penalty function is also encoded in the matrix. We pass this QUBO matrix to a quantum (or simulated) annealer, and we finally decode the output of the annealer to a string by transforming the binary strings to ASCII numbers, and then to their corresponding characters.

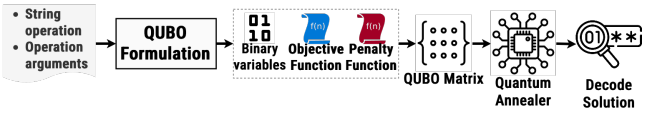

Figure 1: Overview of our approach.

The process for converting any problem to QUBO form follows the same conventions: first, we must define the **binary variables**. Next, the **objective** function must be defined. In other words, we need to define what exactly it is that we are trying to minimize by using our binary variables. Then, we need to define how the **QUBO matrix** will be formed. In our case, the formulations differ slightly for each code constraint our method covers. The details of the methods are described in the next subsections (§ 4.1–§ 4.12).

Unless otherwise specified, our binary variables remains the same throughout our formulations: each ASCII character in the target string is represented by 7 bits, resulting in $7n$ binary variables for a string of length n (7 bits per character in n). Each variable x_i represents one bit position in the final string, with the complete set of variables representing the entire target string in binary form. Formally, we define a function $\texttt{bin} : \Sigma \rightarrow \{0,1\}^7$, which maps each character from the alphabet Σ to a seven-bit binary vector. Further, we define a function $f : \Sigma^n \rightarrow \{0,1\}^{7n}$ that transforms a string of length n into a binary vector of length $7n$. Finally, we encode the string by applying the $\texttt{bin}$ function to each character, and then concatenating the result of each transformation to form a single binary vector: $f(s) = \texttt{bin}(s_1) \| \texttt{bin}(s_2) \| \cdots \| \texttt{bin}(s_n)$

Additionally, unless specified otherwise, our coefficients are $A = 1$ for all formulations. We find that this coefficient works best with our simulated annealer.

4.1 String Equality

Broadly, the goal of string equality is to verify whether a string S is the same as a string T. In our scenario, we are ***generating*** a string S to match a string T.

4.1.1 Objective Function: The objective function is expressed as: minimize the sum over all the bits m in a string S, where q_{ii} is set to $-A$ if the target bit should be 1, and $+A$ if the target bit should be 0. x_i is the value of the bit at position i (e.g. 0 or 1). The penalty strength A determines how strongly these constraints are enforced in the solution. For our use case, we set A to be 1.

$$Q(i) = \sum_{i=0}^{m} (q_{ii} x_i)$$

$$q_{ii} = \begin{cases} -A, & \text{if target bit should be 1,} \\ +A, & \text{if target bit should be 0.} \end{cases}$$

x_i = the character at position i in string S

4.1.2 QUBO Matrix Formulation: The matrix is a $(7n \times 7n)$ QUBO matrix, using only diagonal entries, which results in a sparse structure. Each diagonal entry corresponds to one bit position and contains either $-A$ or $+A$ depending on the desired bit value as described above. For example, generating the character "a" (ASCII 97 = 1100001) requires a 7×7 QUBO matrix with diagonal entries [-A, -A, +A, +A, +A, +A, -A].

4.2 String Concatenation

String concatenation involves taking a string $s1$ and appending a string $s2$ to the end of it. One example would be is string $s1$ ="hello" and string $s2$ =" world", then $s1 + s2$ would be "hello world". We approach this constraint in the same way as string equality. Essentially, we wish to generate a string which combines two given or existing strings. Thus, our binary variables, objective function, and QUBO matrix generation remains the same in that we encode the desired concatenated string into the QUBO matrix.

4.3 Substring Matching

Substring matching is defined as finding a string T that contains a substring S. This is very similar to string equality, in that we are generating a string T which contains a substring S.

4.3.1 Objective Function: Our objective function remains the same as string equality: we minimize the sum over all the bits m in a string S, where q_{ii} is set to $-A$ if the target bit should be 1, and $+A$ if the target bit should be 0.

4.3.2 QUBO Matrix Formulation: When generating our string, we give as input the length of the larger string that we are generating, and the substring which must be included in the larger string. When we are encoding the values in the matrix, we encode the substring at every possible starting position within the matrix. For example, if we are generating a 4-character string that must contain the substring "cat", then the possible starting position for our substring is indexes 0 and 1 (assuming a zero-based indexing system). When there are conflicting entries in the matrix, we overwrite the previous entries, thus resulting in our substring being encoded in the last possible starting position in the matrix. Again in the example of the 4-character string containing the substring "cat", our formulation would result in "ccat" being encoded into the matrix. This is because when we initially encode "cat" as starting at position 0, we encode "cat?", where the "?" is unconstrained (*i.e.*, no entries are given in our matrix). When we encode "cat" as starting at position 1, we retain the "c" at position 0, but *overwrite* the encoding for "a" and "t" in the matrix with "c" and "a". Thus, instead of "cat?", we are left with the encoding for "ccat".

4.4 String Includes

This operation involves identifying whether a string exists within a larger string. Our approach is determining where, in a larger string T, does the substring S begin.

4.4.1 Binary Variables: we define the binary variables to be x_i for $i = 0, 1, ..., n - m$ where n is the length of T and m is the length of S. Our variable $x_i = 1$ if the substring S starts at position i, otherwise $x_i = 0$.

4.4.2 Objective Function: This approach defines that the objective function rewards matches between S and T at every possible starting position in T. The function is defined as:

$$Q(i, j) = -A \sum_{i=0}^{n-m} \sum_{j=0}^{m-1} \delta(t_{i+j}, s_j) * x_i$$

Where t_{i+j} is the character at position $i + j$ in string T, s_j is the character at position j in string S, and

$$\delta(t_{i+j}, s_j) = \begin{cases} 1, & \text{if } t_{i+j} = s_j, \\ 0, & \text{if } t_{i+j} \neq s_j. \end{cases}$$

4.4.3 Penalty Function. In the case of string includes, we need to enforce that only one starting position is chosen (meaning, only one $x_i = 1$). We do so by defining the quadratic penalty function:

$$Q(i, j) = B \sum_{i=0}^{n-m} \sum_{j=i+1}^{n-m} x_i x_j$$

Our penalty ensures that if more than one $x_i = 1$, then the energy of the solution will be penalized (*e.g.*, increased), thus encouraging the optimization algorithm to select only one starting position.

Further, we define another penalty term which ensures that only the first valid starting position is chosen:

$$Q(i, i) = \sum_{i=0}^{n-m} C_i * \delta(T[i : i + m], S) * x_i$$

$$\delta(T[i : i + m], S) = \begin{cases} 1, & \text{if } T[i : i + m] = S, \\ 0, & \text{else.} \end{cases}$$

$$\text{and } C_i = \begin{cases} 0, & \text{if } i = 0 \\ C_{i-1} + D, & \text{if } T[i : i + m] = S \\ C_{i-1}, & \text{otherwise} \end{cases}$$

In other words, C_i is updated (incremented by a constant D) only when there is a match. If there's no match, C_i remains the same for the next iteration.

4.4.4 QUBO Matrix Formulation: To create the QUBO matrix, we create a matrix of the size of the number of possible starting positions. For example, if we are looking for substring of length three in a string of length four, we have two starting positions (positions 0 and 1), thus we have a matrix of size 2×2. We enforce our objective and penalty functions to fill the matrix.

4.5 Substring IndexOf

In this constraint, we generate the string which contains a substring S at a position x within a larger string T.

4.5.1 Objective Function: Our approach maintains the same objective function as for string equality and substring matching, given that we are once again generating the string. However, in this instance, we are only enforcing the positions where the substring should be; we therefore have softer constraints for the rest of the string, where any ASCII character can appear. In other words, wherever we require a specific string to appear, we encode a stronger or higher penalty/value (for example 2* the penalty strength A), and the rest of the string, which could be any character, we encode a softer constraint (for example, 0.1* the penalty strength A).

4.5.2 QUBO Matrix Formulation: To create the QUBO matrix, we create a matrix of the size of $7t \times 7t$, where t is the length of the larger string T. We enforce our *strong* constraints where the substring should appear, and add softer constraints in the remaining positions such that other valid ascii characters can be generated at those positions.

4.6 String Length

This operation is used to determine is a string S matches a desired length L.

4.6.1 Binary Variables: We have a binary string x (*e.g.*, a string, where each character is transformed into the binary representation of the ascii number representing that character) of length n. We determine whether the string is of length L by ensuring that the first L bits for the string $(x_1, x_2, ..., x_L)$ are equal to 1, while the rest of the bits $(x_{L+1}, ..., x_n)$ are equal to 0.

4.6.2 Objective Function: Our objective function minimizes the following function:

$$Q = \sum_{i=1}^{L} (-x_i) + \sum_{i=L+1}^{n} x_i$$

This function is divided into two parts. The first summation ensures, or encodes, that we want the first L bits to be 1; that is, for each i to L, we want $x_i = 1$.

The second summation encodes that we want the remaining bits to be 0. In other words, for each i from L to $n - L$ where n is the length of the given string, $x_i = 0$. We combine these two summations to get the resultant objective function.

4.6.3 QUBO Matrix Formulation: We create a $7n \times 7n$ QUBO matrix for a string of length n, and encode our objective function. That is, for each position in the matrix along the diagonal that is less than our desired length L, we encode $-A$, encouraging those bits to be used. Otherwise, for the positions that are greater than our desired length, we encode A to encourage that those bits are 0.

4.7 String replaceAll

While seemingly a simple constraint, the z3 library does not currently support this string operation. String replaceAll is defined as, given a string S, replace all instances of the character x within S with the character y. This results in a new string T, which contains no instances of x within the string, and only instances of y (along with whatever other characters were originally in S). We thus treat this operation similarly to our string equality operation, in that we generate our desired string.

4.7.1 Objective Function: Our objective function is to minimize the sum over all of the bits in a string S where q_{ii} is equal to $-A$ if the bit should be one and A if the bit should be zero:

$$Q(i) = \sum_{i=0}^{7n-1} q_{ii} x_i$$

$$q_{ii} = \begin{cases} -A, & \text{if target bit should be 1,} \\ +A, & \text{if target bit should be 0.} \end{cases}$$

x_i = the character at position i in string S

Additionally, if the character position j in the input string S ($S[j]$) is the character to be replaced x, we use the bit pattern of the replacement character y. Otherwise, we maintain the same bit pattern throughout the string.

4.7.2 QUBO Matrix Formulation: Just as in our string equality formulation, the matrix is a $(7n \times 7n)$ matrix that uses only diagonal entries. Each diagonal entry corresponds to one bit position and contains either $-A$ or $+A$ depending on the desired bit value. For this operation, we take an additional step, in that when building the matrix, we check at each character position j whether $S[j]$ is the character x that we wish to replace. If it is, we then enforce the encoding for character y in the matrix for those seven bits.

4.8 String Replace

This operation is a variation of the previously described operation, and thus the set up is exactly the same. The only difference is that when building the matrix, we only replace the *first* instance of x. We modify our check at each character position to reflect this change.

4.9 String Reversal

This operation involves taking in a string, and reversing each element in it such that the output is the reverse of the input. For example, an input string "`hello`" would be output from a string reversal as "`olleh`".

To achieve this goal, we perform a similar operation to our **string replace** method. We encode our string backwards (*e.g.*, the reverse of the string) into the QUBO matrix. Our matrix is $(7n \times 7n)$, with the diagonal encoding each desired or target bit ($+A$ for 0 and $(-A)$ for 1).

4.10 Palindrome Generation

A palindrome is a string which reads the same way forwards as it does backwards, for example "abba" or "gobog". This is a more complex constraint than the ones we have dealt with earlier, and is not supported by the z3 library.

4.10.1 Objective Function: Our objective function is looking to minimize the energy of the system when the mirrored bits are the same. For example, in a string with length five, we want the energy of the system to be at a minimum when the character at position 0 matches the character at position 4, and so on. As a reminder, we are performing these operations at the bit level. At a high level, our objective function is as follows:

$$Q(i, j) = A(x_i + x_j - 2x_i x_j)$$

Where x_i is the position at the front of the string, and x_j is the position at the end of the string, and A is our penalty strength. When x_i and x_j are equal, this results in the energy of the system being equal to 0. When x_i and x_j are *not* equal, it results in the energy of the system being A. Thus, when our characters/bits are not mirrored, we do not have the minimum energy of the system. Because we are performing these operations at a bit level, we expand this objective function further to become:

$$\sum_{j=0}^{\lfloor N/2 \rfloor - 1} \sum_{i=0}^{6} A \cdot \left(x_{7j+i} + x_{7(N-1-j)+i} - 2 \cdot x_{7j+i} \cdot x_{7(N-1-j)+i}\right)$$

Here, we sum over character positions from 0 to just before the middle. We only go to the middle because both sides are mirrored. The second summation goes across each bit within the ASCII representation of a character. x_{7j+i} is the bit i of the character at j. $x_{7(N-1-j)+i}$ is the bit i of the the mirrored position of j (at position $N - 1 - j$).

4.10.2 QUBO Matrix Formulation: To build the matrix, we initially enforce our penalty of A along the diagonal. We then assign in our matrix at position $Q(i, j) = -2A$. This helps ensure that the energy of our system is minimized when $x_i == x_j$.

4.11 Regex Matching

This operation involves generating a string which will match a given regular expression (regex) pattern. Due to its complexity, our method supports matching a subset of regex operations, namely *literal characters*, *character classes*, and *plus*. Literal characters define specific characters such as '`a`', '`b`', or '`c`'. Character classes involve matching a character from a given set. For example, a regex with the character class `[bc]` will match any string that contains the characters '`b`' *or* '`c`'. Finally, the plus operation defines that there must be one or more of the given character. An example of a regex would be `a[tyz]+b`. Some valid solutions to this regex would be: '`atytyzb`', '`azb`', or '`atyzb`'.

4.11.1 Objective Function: Due to the nature of this constraint (where we are trying to optimize different operations), we define two objective functions, which are used at different positions depending on the regex constraint at that position.

Our objective function for literals and the plus operation is minimizing the following:

$$Q_{\text{pos}}(x) = \sum_{i=0}^{6} q_i \cdot x_{\text{pos}\cdot 7+i}$$

Where

$$q_i = \begin{cases} -A, & \text{if target bit should be 1,} \\ +A, & \text{if target bit should be 0.} \end{cases}$$

$x_{pos\cdot 7+i}$ defines the bit i at position x_{pos} within the string. This enforces the specific character we want at that position.

Our objective function for character classes varies slightly. We minimize the following:

$$Q_{\text{pos}}(x) = \sum_{i \in \text{chars}} \sum_{j=0}^{6} \frac{q_{i,j}}{|\text{chars}|} \cdot x_{\text{pos}\cdot 7+j}$$

Our previous variable definitions hold here as well. In this objective function, we divide the strength of our penalty coefficient by the number of characters in our character class to give equal and shared preference, and encode all the characters within the character class.

4.11.2 QUBO Matrix Formulation: To create our matrix, we first identify which patterns we are encoding. That is, if we have the regex a[bc]+, and we are generating a string of length 3 to match this pattern, then we know we are encoding a literal, a character class, and another character class. Once we gather this information, we are able to appropriately encode our matrix. We consider the plus constraint as a literal when it appears after a literal, and a character class when it appears after a character class.

We then encode the appropriate objective function into the matrix depending on if the position in the string is a literal or a character class.

4.12 Combining Constraints

Often times, SMT solvers will have to find a solution to satisfy multiple constraints at the same time. To achieve this in our QUBO formulations, we perform each operation sequentially. That is, if we first want to reverse the string "hello", we will run that through

Table 1: Results from our approach to sample string constraints. The matrices are abbreviated due to space limitations.

Constraint	Matrix	Output
Reverse 'hello' and replace 'e' with 'a'	$\begin{bmatrix} -1 & 0 & 0 & \cdots & 0 \\ 0 & -1 & 0 & \cdots & 0 \\ 0 & 0 & -1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & -1 \end{bmatrix}$	ollah
Generate a palindrome with length 6	$\begin{bmatrix} 1.00 & 0.00 & 0.00 & \cdots & -2.00 & 0.00 & 0.00 \\ 0.00 & 1.00 & 0.00 & \cdots & 0.00 & -2.00 & 0.00 \\ 0.00 & 0.00 & 1.00 & \cdots & 0.00 & 0.00 & -2.00 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0.00 & 0.00 & 0.00 & \cdots & 1.00 & 0.00 & 0.00 \\ -2.00 & 0.00 & 0.00 & \cdots & 0.00 & 1.00 & 0.00 \\ 0.00 & -2.00 & 0.00 & \cdots & 0.00 & 0.00 & 1.00 \end{bmatrix}$	OnFFnO
Generate the regex `a[bc]+` with length 5	$\begin{bmatrix} -2.00 & 0.00 & 0.00 & \cdots & 0.00 \\ 0.00 & 2.00 & 0.00 & \cdots & 0.00 \\ 0.00 & 0.00 & 2.00 & \cdots & 0.00 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0.00 & 0.00 & 0.00 & \cdots & -1.00 \end{bmatrix}$	abcbb
Concatenate 'hello' and ' world', and replace all 'l' with 'x'	$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & -1 \end{bmatrix}$	hexxo worxd
Generate a string of length 6 that contains the substring 'hi' at index 2	$\begin{bmatrix} 0 & 0 & 0 & \cdots & 0 \\ 0 & 2 & 0 & \cdots & 0 \\ 0 & 0 & 6 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & -42 \end{bmatrix}$	qphiqp

our solver first. Then, if we want to replace "e" with "a", we then will take the output solution of the first iteration of our solver, and pass it through as the input to the second solver.

5 RESULTS

Table 1 demonstrates some sample constraints. The matrices demonstrate how the constraints are encoded into the QUBO. Note that for space considerations, we do not show the full matrix.

When running our experiments, we use DWave's Simulated Annealer [28]. However, our QUBO formulations are compatible with a real quantum annealer, thus we would see similar outputs from our results, with the additional speedup benefit that quantum offers. We expect, however, that our palindrome or regex generation, for example, would produce a different string every time, while still obeying the given constraints.

The key observations are that our method successfully encodes each constraint problem into the QUBO matrix, and generates the expected output. We see that we can transform strings accurately, and generate strings with structural constraints in the instances of palindromes and regular expressions. We also are able to generate 'flexible' examples, even when enforcing substring positions. This means that our method can generate a unique string, while still enforcing the necessary constraints.

Our preliminary results demonstrate that our QUBO formulations are a viable approach for encoding a variety of string constraints, which offers a pathway to leverage quantum annealing hardware for problems involving string manipulations that are challenging for classical solvers.

6 CONCLUSION

Quantum computing offers exciting advantages for problems which are classically complex. In this work, we presented a novel method for SMT solving for string theory using quantum annealing. This preliminary work demonstrates how we can go about defining string constraint problems in the form of an optimization problem in QUBO form. We extend the current capabilities of solvers such as z3 by defining QUBOs for replace all constraints and palindrome generation.

Future works include testing these formulations on a real quantum computer, as well as using these formulas in applications such as symbolic execution and program testing. Further, we can create more formulations based on this preliminary work for other string constraints.

7 ACKNOWLEDGEMENT

This work was funded by the Center for Quantum Technologies (CQT) under NSF's I/UCRC program.

REFERENCES

[1] Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. *Handbook of model checking*, pages 305–343, 2018.
[2] Roberto Sebastiani. Lazy satisfiability modulo theories. *Journal on Satisfiability, Boolean Modelling and Computation*, 3(3-4):141–224, 2007.
[3] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction*

and Analysis of Systems, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[4] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

[5] Shuvendu Lahiri and Shaz Qadeer. Back to the future: revisiting precise program verification using smt solvers. *ACM SIGPLAN Notices*, 43(1):171–182, 2008.

[6] Alessandro B. Trindade and Lucas C. Cordeiro. Applying smt-based verification to hardware/software partitioning in embedded systems. *Design Automation for Embedded Systems*, 20(1):1–19, Mar 2016.

[7] Mingxuan Yuan, Xiuqiang He, and Zonghua Gu. Hardware/software partitioning and static task scheduling on runtime reconfigurable fpgas using a smt solver. In *2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 295–304, 2008.

[8] Alexandra Bugariu and Peter Müller. Automatically testing string solvers. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1459–1470, New York, NY, USA, 2020. Association for Computing Machinery.

[9] Shang-Wei Lin, Si-Han Chen, Tzu-Fan Wang, and Yean-Ru Chen. A quantum smt solver for bit-vector theory, 2023.

[10] Murphy Berzish, Mitja Kulczynski, Federico Mora, Florin Manea, Joel D. Day, Dirk Nowotka, and Vijay Ganesh. An smt solver for regular expressions and linear arithmetic over string length. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification*, pages 289–312, Cham, 2021. Springer International Publishing.

[11] Sanu Subramanian, Murphy Berzish, Omer Tripp, and Vijay Ganesh. A solver for a theory of strings and bit-vectors. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 124–126, 2017.

[12] Cesare Tinelli. The smt-lib format: an initial proposal. 01 2003.

[13] Tianyi Liang, Andrew Reynolds, Nestan Tsiskaridze, Cesare Tinelli, Clark Barrett, and Morgan Deters. An efficient smt solver for string constraints. *Form. Methods Syst. Des.*, 48(3):206–234, June 2016.

[14] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Lukáš Holík, Ahmed Rezine, and Philipp Rümmer. Trau: Smt solver for string constraints. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–5, 2018.

[15] G. Karakostas, R.J. Lipton, and A. Viglas. On the complexity of intersecting finite state automata. In *Proceedings 15th Annual IEEE Conference on Computational Complexity*, pages 229–234, 2000.

[16] František Blahoudek, Yu-Fang Chen, David Chocholatý, Vojtěch Havlena, Lukáš Holík, Ondřej Lengál, and Juraj Síč. Word equations in synergy with regular constraints (technical report), 2022.

[17] Z3.seq. https://z3prover.github.io/api/html/ml/Z3.Seq.html.

[18] Murphy Berzish, Vijay Ganesh, and Yunhui Zheng. Z3str3: A string solver with theory-aware heuristics. In *2017 Formal Methods in Computer Aided Design (FMCAD)*, pages 55–59. IEEE, 2017.

[19] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. cvc4. In *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*, pages 171–177. Springer, 2011.

[20] Raihan Ur Rasool, Hafiz Farooq Ahmad, Wajid Rafique, Adnan Qayyum, Junaid Qadir, and Zahid Anwar. Quantum computing for healthcare: A review. *Future Internet*, 15(3):94, 2023.

[21] Nick S Blunt, Joan Camps, Ophelia Crawford, Róbert Izsák, Sebastian Leontica, Arjun Mirani, Alexandra E Moylett, Sam A Scivier, Christoph Sunderhauf, Patrick Schopf, et al. Perspective on the current state-of-the-art of quantum computing for drug discovery applications. *Journal of Chemical Theory and Computation*, 18(12):7001–7023, 2022.

[22] Dylan Herman, Cody Googin, Xiaoyuan Liu, Yue Sun, Alexey Galda, Ilya Safro, Marco Pistoia, and Yuri Alexeev. Quantum computing for finance. *Nature Reviews Physics*, 5(8):450–465, 2023.

[23] Geeta N Brijwani, Prafulla E Ajmire, and Pragati V Thawani. Future of quantum computing in cyber security. In *Handbook of research on quantum computing for smart environments*, pages 267–298. IGI Global, 2023.

[24] Rozhin Eskandarpour, Pranav Gokhale, Amin Khodaei, Frederic T Chong, Aleksi Passo, and Shay Bahramirad. Quantum computing for enhancing grid security. *IEEE Transactions on Power Systems*, 35(5):4135–4137, 2020.

[25] Olakunle Abayomi Ajala, Chuka Anthony Arinze, Onyeka Chrisanctus Ofodile, Chinwe Chinazo Okoye, and Andrew Ifesinachi Daraojimba. Exploring and reviewing the potential of quantum computing in enhancing cybersecurity encryption methods. *Magna Sci. Adv. Res. Rev*, 10(1):321–329, 2024.

[26] Richard Jozsa. Entanglement and quantum computation, 1997.

[27] Ashley Montanaro. Quantum algorithms: an overview. *npj Quantum Information*, 2(1):15023, Jan 2016.

[28] D-Wave Systems. Quantum computing | d-wave, 2025. Accessed: 2025-03-14.

[29] Tianyi Liang, Andrew Reynolds, Cesare Tinelli, Clark Barrett, and Morgan Deters. A dpll (t) theory solver for a theory of strings and regular expressions. In *International Conference on Computer Aided Verification*, pages 646–662. Springer, 2014.

[30] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Dpll (t): Fast decision procedures. In *Computer Aided Verification: 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004. Proceedings 16*, pages 175–188. Springer, 2004.

[31] Leonardo de Moura and Grant Olney Passmore. *The Strategy Challenge in SMT Solving*, pages 15–44. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[32] Clark Barrett, Leonardo Moura, and Aaron Stump. Design and results of the first satisfiability modulo theories competition (smt-comp 2005). *J. Autom. Reason.*, 35(4):373–390, November 2005.

[33] Bernd Finkbeiner and Sven Schewe. Smt-based synthesis of distributed systems. In *Proceedings of the Second Workshop on Automated Formal Methods*, AFM '07, page 69–76, New York, NY, USA, 2007. Association for Computing Machinery.

[34] Vidhya Tekken Valapil, Sorrachai Yingchareonthawornchai, Sandeep Kulkarni, Eric Torng, and Murat Demirbas. Monitoring partially synchronous distributed systems using smt solvers. In Shuvendu Lahiri and Giles Reger, editors, *Runtime Verification*, pages 277–293, Cham, 2017. Springer International Publishing.

[35] Pranav Tendulkar. *Mapping and Scheduling on Multi-core Processors using SMT Solvers*. Theses, Universite de Grenoble I - Joseph Fourier, October 2014.

[36] Vikas Hassija, Vinay Chamola, Vikas Saxena, Vaibhav Chanana, Prakhar Parashari, Shahid Mumtaz, and Mohsen Guizani. Present landscape of quantum computing. *IET Quantum Communication*, 1(2):42–48, 2020.

[37] Mark Lewis and Fred Glover. Quadratic unconstrained binary optimization problem preprocessing: Theory and empirical analysis. *Networks*, 70(2):79–97, 2017.

[38] Zsolt Tabi, Kareem H. El-Safty, Zsofia Kallus, Peter Haga, Tamas Kozsik, Adam Glos, and Zoltan Zimboras. Quantum optimization for the graph coloring problem with space-efficient embedding. In *2020 IEEE International Conference on Quantum Computing and Engineering (QCE)*, page 56–62. IEEE, October 2020.

[39] R. Rietsche, C. Dremel, and C. et al. Bosch. Quantum computing. *Electron Markets*, 2022.

[40] Chan Gu Kang, Joonghoon Lee, and Hakjoo Oh. Statistical testing of quantum programs via fixed-point amplitude amplification. *Proc. ACM Program. Lang.*, 8(OOPSLA2), October 2024.

[41] Noah H. Oldfield, Christoph Laaber, Tao Yue, and Shaukat Ali. Faster and better quantum software testing through specification reduction and projective measurements, 2024.

[42] Asmar Muqeet, Tao Yue, Shaukat Ali, and Paolo Arcaini. Mitigating noise in quantum software testing using machine learning, 2024.

[43] Ali JavadiAbhari, Shruti Patil, Daniel Kudrow, Jeff Heckey, Alexey Lvov, Frederic T. Chong, and Margaret Martonosi. Scaffcc: a framework for compilation and analysis of quantum computing programs. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, CF '14, New York, NY, USA, 2014. Association for Computing Machinery.

[44] Seungmin Jeon, Kyeongmin Cho, Chan Gu Kang, Janggun Lee, Hakjoo Oh, and Jeehoon Kang. Quantum probabilistic model checking for time-bounded properties. *Proc. ACM Program. Lang.*, 8(OOPSLA2), October 2024.

[45] Domenico Cantone, Simone Faro, and Arianna Pavone. Quantum string matching unfolded and extended. In Martin Kutrib and Uwe Meyer, editors, *Reversible Computation*, pages 117–133, Cham, 2023. Springer Nature Switzerland.

[46] Pradeep Niroula and Yunseong Nam. A quantum algorithm for string matching. *npj Quantum Information*, 7(1):37, Feb 2021.

[47] Simone Faro, Arianna Pavone, and Caterina Viola. Bridging classical and quantum string matching: A computational reformulation of bit-parallelism, 2025.

[48] Francesco Pio Marino, Simone Faro, and Antonio Scardace. Practical implementation of a quantum string matching algorithm. In *Proceedings of the 2024 Workshop on Quantum Search and Information Retrieval*, QUASAR '24, page 17–24, New York, NY, USA, 2024. Association for Computing Machinery.

[49] Simone Faro, Arianna Pavone, and Caterina Viola. Quantum path parallelism: A circuit-based approach to text searching. In Xujin Chen and Bo Li, editors, *Theory and Applications of Models of Computation*, pages 247–259, Singapore, 2024. Springer Nature Singapore.

[50] Kostas Blekos, Dean Brand, Andrea Ceschini, Chiao-Hui Chou, Rui-Hao Li, Komal Pandya, and Alessandro Summer. A review on quantum approximate optimization algorithm and its variants. *Physics Reports*, 1068:1–66, June 2024.

[51] Sami Boulebnane and Ashley Montanaro. Solving boolean satisfiability problems with the quantum approximate optimization algorithm, 2022.

[52] Andriy Miranskyy. Using quantum computers to speed up dynamic testing of software. In *Proceedings of the 1st International Workshop on Quantum Programming for Software Engineering*, QP4SE 2022, page 26–31, New York, NY, USA, 2022. Association for Computing Machinery.