String Variables for Constraint-Based Local Search

Gustav Björdal
Abstract

String Variables for Constraint-Based Local Search

Gustav Björdal

String variables occur as a natural part of many computationally challenging problems. Usually, such problems are solved using problem-specific algorithms implemented from first principles, which can be a time-consuming and error-prone task.

A constraint solver is a framework that can be used to solve computationally challenging problems by first declaratively defining the problem and then solving it using specialised off-the-shelf algorithms, which can cut down development time significantly and result in faster solution times and higher solution quality.

There are many constraint solving technologies, one of which is constraint-based local search (CBLS). However, very few constraint solvers have native support for solving problems with string variables.

The goal of this thesis is to add string variables as a native type to the CBLS solver OscaR/CBLS.

The implementation was experimentally evaluated on the Closest String Problem and the Word Equation System problem. The evaluation shows that string variables for CBLS can be a viable option for solving string problems. However, further work is required to obtain even more competitive performance.
Acknowledgements

I would like to thank my supervisor Prof. Pierre Flener for all the help and guidance I have received during this thesis, and for continuing to present me with both challenging and interesting problems.

I would also like to thank my co-supervisor Dr Jean-Noël Monette for all of our technical discussions of the implementation, as they saved me a lot of headaches.

Finally, I would like to thank Dr Joseph Scott for helping me, and sharing the frustration of finding suitable string problems for evaluating a string solver.
1 Introduction

String variables occur as a natural part of many computationally challenging problems from different areas, such as computational biology (e.g., protein folding, sequence alignment), interactive configuration, signal analysis (e.g., pattern matching), and computer science (e.g., verification, test case generation, software analysis).

Constraint solving is the technique of declaratively modelling problems in terms of their variables and constraints, and solving the model using an off-the-shelf solver. There are many solvers, based on different underlying technologies such as constraint programming (CP) [33], linear programming [30], constraint-based local search (CBLS) [25], SAT solving [7], or a hybrid thereof (e.g., iZplus [19]).

CBLS is a fairly recent technology, which iteratively improves an assignment of all variables, until a solution of good enough quality is found. Unlike most other technologies, CBLS is based on local search [26], which means that it can usually find close to optimal solutions very quickly. However, this comes at the cost of being incomplete, which means that it cannot prove a solution optimal or a problem unsatisfiable, except for in a few extraordinary cases.

Only a few solvers, based on CP or SAT modulo theories (SMT), currently have a native support for string variables and constraints [1, 18, 21, 35]. This forces users to emulate strings by other means, e.g., by using arrays of integers, resulting in computational overhead, limitation in expressiveness, and incomprehensible models.

The goal of this thesis is to implement string variables and constraints in CBLS for the first time and evaluate their usefulness.

The implementation is done in OscaR/CBLS. OscaR [32] is an open-source toolkit for solving operations research problems in the Scala language. Existing NP-hard string problems are identified and modelled using the implemented features to evaluate both the performance and the expressiveness of the modelling language.

2 Background

As a running example, we will here use the $n$-queens problem [29]. The problem is to place $n$ queens on an $n \times n$ chessboard, such that no two queens can attack each other. Figure 1 shows a non-solution and a solution to the 4-queens problem.

2.1 Constraint Problems

A constraint satisfaction problem is a set of variables $\mathcal{X}$ and a set of constraints $\mathcal{C}$, where each variable $x_i \in \mathcal{X}$ is associated with a domain $\mathcal{D}(x_i)$ and
each constraint $c(x_i, \ldots, x_j) \in \mathcal{C}$ expresses some relation between variables. A solution to a constraint satisfaction problem is an assignment, $A(\mathcal{X})$, of each variable to a value in its domain such that each constraint is satisfied.

Constrained optimisation problems, extend constraint satisfaction problems by associating an objective value with each solution or, more generally, with each assignment $A(\mathcal{X})$. The goal of a constrained optimisation problem is to minimise or maximise the objective by finding a solution with the lowest or highest cost. For the sake of brevity only minimisation problems will be discussed in this thesis, as a maximisation problem can, by minimising the negated objective, be transformed into a minimisation problem.

The $n$-queens problem can be expressed as a constraint satisfaction problem by associating a variable $x_i$ with the queen placed in column $i$; note that there can be at most one queen in each column or row in any solution. The value of $x_i$ represents the row that the queen in column $i$ is placed in. The constraints of the problem are then that all queens are placed on different rows, which can be expressed by the constraint $\text{ALLDIFFERENT}(x_1, \ldots, x_n)$, and placed on different diagonals, which can be efficiently expressed by $\text{ALLDIFFERENT}(x_1 + 1, \ldots, x_n + n)$ and $\text{ALLDIFFERENT}(x_1 - 1, \ldots, x_n - n)$. For the 4-queens problem, where $n = 4$, an assignment that satisfies all constraints is:

$$x_1 \leftarrow 3, x_2 \leftarrow 1, x_3 \leftarrow 4, x_4 \leftarrow 2$$

2.2 Constraint Solvers

A constraint solver take a definition of a constraint satisfaction or constrained optimisation problem as input and solve it using some form of systematic or stochastic search.

There are many solvers, based on technologies such as linear programming, constraint programming, SAT solving, constraint-based local search,
or hybrids. Each solver has its own benefits and drawbacks and no technology is generally better than all others, unless $P = NP$.

Each solver usually provides a high-level modelling language that allows the user to declaratively model the problem in terms of its variables and constraints. There are also some solver-independent languages that are supported by several different solvers and technologies, e.g., MiniZinc and Essence \cite{31, 17}.

2.3 Constraint-Based Local Search

Local search is a family of search methods that start from a possibly arbitrary initial assignment of all variables and iteratively perform a small modification to the current assignment, until some stopping condition is met. The modification from the current to a new assignment is called a move and is drawn from a set of possible moves, called a neighbourhood. The search is driven by some heuristic for selecting moves. A meta-heuristic, such as tabu search \cite{22}, is usually employed in order to escape local minima. The stopping condition is usually finding an assignment of high enough quality or the depletion of some limited resource, such as a time limit or maximum number of iterations.

Constraint-based local search (CBLS), a fairly recent constraint solving technology, aims to combine local search with the modelling paradigm of constraint programming. In CBLS, a constraint problem can be declaratively modelled by its variables, invariants, constraints, and objective.

Each variable in a model is defined by an initial value and a domain. Note that a variable’s domain in CBLS is primarily used to build internal data structures for some constraints and invariants. Associating a domain with each variable also allows moves to be defined in an automatic fashion; however moves can be defined explicitly without the need of an (associated) domain. Currently, only variables of integer or set-of-integer types are supported in existing CBLS frameworks.

Invariants are expressions that functionally define one or more variables and automatically maintain the value of the defined variables. For example, $x \leftarrow \text{Plus}(y, z)$ is an invariant, corresponding to the constraint $x = y + z$, that tells the CBLS system to maintain the value of the variable $x$ to always be equal to $y + z$. Invariants are usually maintained incrementally to make neighbourhood evaluation fast during search.

To help guide the search, each constraint in a CBLS model maintains a degree of violation that is incrementally updated at each iteration. The degree of violation is a representation of how close the constraint is to being satisfied. For example, for the $\text{Less}(x, y)$ constraint, which constrains $x$ to be less than $y$, if $x \geq y$, then the violation is (commonly) $x - y$. Otherwise, if the constraint is satisfied, the violation is 0.

The constraints in a model are added to a constraint system, which
maintains the (weighted) sum of the violations of all constraints, the global violation. During search, each candidate move can be evaluated by probing the constraint system with the move for the resulting global violation.

A CBLS model for the \(n\)-queens problem written in Scala using the OscaR/CBLS framework is shown in Listing 1. The variables are defined on line 16, where each variable is given a different value between 1 and \(N\). The constraint that all values are different is not needed as each variable is initially given a unique value and the neighbourhood on line 24 only performs moves that swap the value of two variables, which will maintain that they are different. The constraints that queens are placed on different diagonals are stated on lines 19 and 20. Note that \(\text{queens}(q) + q\) is shorthand for creating the invariant Plus(\(\text{queens}(q)\),q). On line 24 the neighbourhood is defined to perform moves that swap the value of any two queens. The actual search is performed on line 25 by doing all moves for \(N\) iterations or until the global violation is 0, which corresponds to finding a solution.

2.4 String Variables and Constraints

Given an alphabet, \(\Sigma\), a string is a sequence of characters from \(\Sigma\). The length of a string \(s\) is denoted \(|s|\), while \(s[i]\) denotes the character at position \(i\), counting from 1, and \(s[i..j]\) denotes the substring of \(s\) from index \(i\) until index \(j\) inclusive.

An unknown string over an alphabet is known as a string variable. For some constant \(\ell\), a string variable \(v\) is said to be of fixed length if \(|v| = \ell\), bounded length if \(|v| \leq \ell\), or unbounded length if no explicit length requirement exists.

There are many string constraints; some that can be found in the literature are:

-EQ\((v, w)\) holds if \(|v| = |w|\) and \(\forall i : v[i] = w[i]\).

-REVERSE\((v, x)\) holds if \(x\) is the reverse of \(v\).

-CONCAT\((v, w, x)\) holds if \(x\) is the concatenation of \(v\) and \(w\).

-LENGTH\((v, \ell)\) holds if \(|v| = \ell\).

-SUBSTRING\((v, w, i)\) holds if \(w\) is a substring of \(v\) starting at index \(i\), i.e., \(|w| + i - 1 \leq |v|\) and \(\forall j \in \{0, \ldots , |w| - 1\} : w[j] = v[j + i]\).

-CHARACTERAT\((v, c, i)\) holds if \(c\) is the character at index \(i\) in \(v\), i.e., \(v[i] = c\).

-REPLACEALL\((v, c_1, c_2, x)\) holds if \(x\) is equal to \(v\) but with all occurrences of character \(c_1\) replaced by \(c_2\).
Listing 1: A CBLS model for the $n$-queens problem where $n = 1000$, written in Scala using the OscaR/CBLS framework. The model is taken from the OscaR/CBLS examples and modified to not include a meta-heuristic for readability.

```scala
package oscar.examples.cbls.queens

import oscar.cbls.invariants.core.computation.CBLSIntVar
import oscar.cbls.modeling._
import scala.util.Random

/** Local Search for NQueens
 * Moves are operated by swapping variables, using a standard neighborhood
 */
object NQueensEasy1 extends CBLSModel with App{

  val N = 1000
  val range:Range = Range(0,N)
  val init = Random.shuffle(range.toList).iterator
  // Variables
  val queens = Array.tabulate(N)((q: Int) => CBLSIntVar(init.next(),0 to N-1, "queen" + q))
  // Constraints
  //add(AllDiff(Queens)) //enforced because we swap queens and they are always allDiff
  add(allDiff(for (q <- range) yield (queens(q) + q)))
  add(allDiff(for (q <- range) yield (queens(q) - q)))
  // Close the model
  close()
  // Search
  val neighborhood = swapsNeighborhood(queens, "SwapQueens")
  val it = neighborhood.doAllMoves(_ => N || c.violation.value == 0, c)

  println("it: " + it)
  println(queens.mkString(","))
}
```
\text{COUNT}(v, [c_1, \ldots, c_n], [i_1, \ldots, i_n]) \text{ holds if the number of occurrences of}
\text{character } c_k \text{ in } v \text{ is equal to } i_k, \forall k \in \{1, \ldots, n\}.

\text{REGULAR}(v, \mathcal{L}) \text{ holds if } v \text{ is a member of the regular language } \mathcal{L}.

Note that some constraints, such as \textsc{Reverse}, \textsc{Concat}, \textsc{Length}, and \textsc{ReplaceAll} can be implemented as invariants in a CBLS system.

3 Related Work

String solving is becoming an increasingly popular topic of research. In recent years, several string solvers, based on different technologies, have been developed, such as \textsc{norn} \cite{1}, \textsc{kaluza} \cite{24}, \textsc{sushi} \cite{18}, \textsc{hampi} \cite{21}, and \textsc{gecode+S} \cite{35}. Most of these string solvers are developed with a specific application in mind, most commonly software verification. For this reason, most string solvers tend to be limited in the types of constraints supported and even put restrictions on the usage of some constraints. Currently, \textsc{gecode+S} is the only general-purpose string solver, which combines string variables and constraints with multiple types of variables and a large library of constraints.

String solvers are generally categorised based on the supported length of string variables, i.e., fixed, bounded, or unbounded length. Of the previously mentioned solvers, \textsc{hampi}, \textsc{gecode+S}, and \textsc{kaluza} are bounded-length string solvers, while \textsc{norn} and \textsc{sushi} are unbounded-length string solvers.

Since the string variables implemented as part of this thesis are of unbounded length, it should be noted that the unbounded-length variables of, e.g., \textsc{norn} and \textsc{sushi} are fundamentally different from the unbounded-length variables implemented in this thesis. Unbounded-length string solvers generally represent a string variable as an automaton, corresponding to the language of the string variable, and then perform automaton intersections to eventually get the automata corresponding to all solutions for the variable. This allows the solver not only to get all solutions, but also to reason on the set of all possible strings. Conversely, the string variables considered in this thesis are, due to the nature of CBLS, always assigned to a string from the variable’s language. This means that, in this case, only one solution can be found at a time and the constraints can only reason on a string rather than a set of strings. For a more extensive overview of string solvers, see \cite{35} Chapter 7.3.

Even though only a few solvers support string variables as a native type, a bounded-length string variable can be emulated in most solvers using an \textit{aggregate implementation of bounded-length string variables}, as described in \cite{35} Chapter 9. Essentially, a string variable \( s \) of bounded length \( \ell \leq |s| \leq u \) with alphabet \( \Sigma \) can be represented by introducing a padding character \( \lambda \) and using an array of length \( u \) of integer variables with domain \( \mathcal{N} \subseteq \mathbb{N} \),
where $\mathcal{N}$ is a one-to-one mapping of $\Sigma \cup \{\lambda\}$. The rules are then that the integer corresponding to the padding character can only occur at the last $u - \ell$ indices of the array, and that for any index whose element is assigned to the padding character, all following indices are also assigned to the padding character. In the description of [35, Chapter 9] a scalar variable, corresponding to the length of the variables, is also included with the aggregate representation. It can be noted that this variable is actually optional, as its value and domain can always be computed given the array of integer variables. However, it can be considered bad practice not to include this length variable, as it would introduce an unnecessary time overhead.

4 Design

The goal of this thesis is to implement string variables, constraints, and invariants as native constructs in CBLS. The implementation will be done in the OscaR/CBLS framework [32], by mainly extending and reworking existing components.

The focus of the implementation is to get working interfaces and classes in place, such that adding string invariants and constraints is no harder than adding integer or set invariants and constraints. An initial set of invariants and constraints will be implemented in order to evaluate the implementation, both from a performance and a modelling point of view.

4.1 Propagation in OscaR/CBLS

A core concept of CBLS solvers is the propagation graph. The propagation graph is a directed acyclic graph that determines in which order the propagation of variable changes occurs. The nodes in the propagation graph are called propagation elements and represent either invariants or variables. The directed edges in the graph represent dependencies, e.g., the edge from node $a$ to node $b$ states that the value of $b$ is dependent on the value of $a$. Note that in the context of the propagation graph, a constraint is considered to be an invariant that maintains its violation as the output variable, which is also a node of the graph.

Each invariant registers a dependency of its input variable, from which the propagation graph is built. When the value of a variable is changed, the change will not be directly propagated to nodes that depend on it. Instead the variable will register itself for propagation and store the change.

A propagation wave is triggered whenever a variable is queried for its value, and a change, caused either by the search heuristic or by propagation of an invariant, has occurred along any path from a variable in the model to the queried variable in the propagation graph. During propagation, in topological order, each propagation element that is registered for propagation and connected to the queried variable receives a propagation
signal. Upon receiving the signal, the propagation element will commit to its stored change and send a change notification to each dependent propagation element. Note that change notifications are used to compute the new value directly after they are sent. If a change notification results in the change of a propagation element, then that propagation element will register itself for propagation and therefore receive a propagation signal during the same propagation wave.

4.2 String Variables

The essential components of a variable in OscaR/CBLS are: a value containing the current assignment, a domain, methods for propagating changes to the variable, and methods for changing the value of the variable.

4.2.1 Value

Each variable holds a value that contains the current assignment of the variable.

In the case of an integer variable, this is an Int. For a set variable, this is a Set of Int.

The value of a string variable is represented with the native Scala class StringBuilder, which is a mutable string representation that allows fast string operations such as insert, remove, and replace.

The StringBuilder object does however have an overhead for reading the value of a variable, as it will build a String object based on its current value. However, directly using a String as the value would result in an even greater overhead, as they are immutable objects that can only be modified by implicitly creating a StringBuilder that performs the change and creates a new string object.

By using the StringBuilder object for the value representation, there is no explicit restriction on the length of the string. A string variable with this representation is thus of unbounded length.

4.2.2 Domain

In the context of OscaR/CBLS, a variable’s domain does not always refer to the actual domain of the variable, i.e., the set of values that the variable can take. Instead, a variable’s domain is a collection of elements that is used by modification methods to change the value of the variable, such that it is assigned to a value within its actual domain. Thus, a variable’s domain and its modification methods implicitly define the actual domain.

\[\text{http://www.scala-lang.org/api/current/#scala.Int}\]
\[\text{http://www.scala-lang.org/api/current/#scala.collection.Set}\]
\[\text{http://www.scala-lang.org/api/current/#scala.collection.mutable.StringBuilder}\]
The domain of a variable mainly has two uses: it is used by some invariants and constraints to build internal data structures, and it allows the CBLS solver to provide generic moves on variables based on the given domain.

For an integer variable, the domain is the set of integers that the variable can be assigned to. In this case, the domain corresponds to the actual domain. A generic move on an integer variable is to reassign it to another element in its domain.

For a set variable, the domain is the set of integers that the set variable can contain. There is currently no way in OscaR/CBLS to specify the set of integers that a set variable must contain, this must be enforced by the heuristic instead. Two generic moves on a set variable are: insert or remove an element of the domain into or from the set’s value respectively.

The domain of a string variable is the alphabet of the string. Due to the ambiguous terminology, the CBLS domain of a string variable will from here on be referred to as its alphabet. Some generic moves on a string variable are: insert or replace a character at some index of the string with a character from the alphabet, and remove a character at some index of the string. Note that a more complex collection than the alphabet can be used as the domain, by for example allowing strings as well as characters in the collection. This would allow for more types of generic moves, such as inserting or replacing a substring at some index of the string. However, for a first implementation, such moves are deemed unnecessary and it is left as future work to investigate the possible benefits of having such a domain.

4.2.3 Modification Methods

Each variable type provides methods for modifying the value of a variable. However, since a variable does not commit to its new value until it receives a propagation signal, the modification methods must perform internal bookkeeping to keep track of uncommitted changes as well as the old value, as the old value is a required part of a change notification.

An integer variable only provides one modification method, namely reassignment. The first reassignment after a propagation signal will save the new value of the variable. Any subsequent reassignment before receiving a propagation signal will overwrite the new value of the variable. Any subsequent reassignment before receiving a propagation signal will overwrite the new value of the variable.

A set variable provides three modification methods: insert an element, remove an element, and reassign to another set. Between any two propagation signals, each insertion and removal will modify the new set value appropriately and the modification is kept track of in a change queue, which is emptied during propagation. If an element that is or is not already in the new set value is inserted or removed respectively, then the modification is not performed. However, if the set variable is reassigned, then the change queue is discarded and the new set value is saved. Any insertion or removal
that occurs after a reassignment, but before a propagation signal, is not saved in the change queue and will only update the new set value. When a propagation signal is received and if a reassignment has occurred, then the change queue that corresponds to transforming the old set value, using only insertions and removals, into the new set value is computed.

A string variable provides four modification methods: insert a character, remove a character, replace a character, and reassign to another string. The bookkeeping of modification methods of a string variable functions essentially the same way as for a set variable, with the addition of the replace modification method. Each insertion, removal and replacement updates the new value and is kept track of in a change queue. If the string variable is reassigned, then the change queue is discarded and subsequent insertions, removals, or replacements will only affect the new value. The only difference from a set variable is that if a reassignment has occurred when a propagation signal is received, then the change queue that would transform the old value into the new value is not computed. Instead the new value is, in this case, used directly during propagation and the change queue is completely ignored.

4.2.4 Propagation Methods

A variable is said to be modified if its value has changed since last receiving a propagation signal. If a modified variable receives a propagation signal, then it notifies all dependent invariants of the change and commit to its current value, discarding the old value.

Invariants are notified by calling a notification method, providing information about: the type of change that was made, the variable that was changed, the old value, and details specific to the change. For example, if the integer variable \(i\) has been reassigned to \(v\) and receives a propagation signal, then it calls the \(\text{notifyIntChanged}(i, v)\) method of each dependent invariant, as defined by the propagation graph. It is assumed that \(i.\text{oldValue}\) contains the old value of \(i\). However, for the sake of brevity, change notifications are from here on discussed as notifications being sent and received with the following notation: for the same example, a \(\text{Reassign}[i](v)\) notification is sent to each dependent invariant.

If a modified integer variable \(i\) that has been reassigned to \(v\) receives a propagation signal, then it sends the mentioned reassignment notification \(\text{Reassign}[i](v)\) to each dependent invariant.

If a modified set variable \(s\) receives a propagation signal, then, for each change in its change queue and for each dependent invariant, it sends the corresponding \(\text{Insert}[s](v)\) or \(\text{Remove}[s](v)\) notification, where \(v\) is the inserted or removed element.

If a modified string variable \(s\) receives a propagation signal and it has not been reassigned to a new string, then, for each change in its change queue
and for each dependent invariant, it sends the corresponding notification of one of the following types:

- **Insert**\([s](i, c)\) stating that \(c\) has been inserted at index \(i\) in \(s\).
- **Remove**\([s](i)\) stating that the character at index \(i\) has been removed.
- **Replace**\([s](i, c)\) stating that the character at index \(i\) in \(s\) has been replaced by \(c\).

If a string variable has been reassigned, then it instead sends a **Reassign**\([s](v)\) notification, where \(v\) is the new value of \(s\).

### 4.3 String Invariants

An invariant, or one-way constraint, takes a number of variables or constants as input and maintains an output value that the invariant defines. The invariant will register a dependency of the input variables and will thus be notified whenever they are changed. Some internal data structure is usually maintained such that, upon receiving a change notification, the invariant can recompute its output value efficiently.

For example, the **Length**\([s]\) invariant, which maintains the length of string \(s\) as its output value, can store the current length of \(s\) and increase or decrease this value upon receiving an **Insert**\([s](i, c)\) or **Remove**\([s](i)\) notification, as opposed to recomputing the length each time.

The output value is usually represented with a variable of the appropriate type. This allow the invariant to make use of the variable’s notification and propagation functionally, by calling the variable’s modification methods when changing the output value. However, finding ways of updating the output value using the available modification methods is not always trivial.

For example, consider the **Concat**\([ab, ba]\) invariant where the output value is the concatenation of the input values, i.e., \(abba\) in this case. If the input value \(ba\) is changed to \(c\), then one might want to update the output value by replacing the substring \(ba\) of \(abba\) with \(c\). However, no such modification method is available in the current implementation. Instead the output value must be updated by either performing a series of inserts, removals, or replacements or reassigning the entire output string; the former is faster for short strings and the latter for large strings.

Some invariants can have multiple output values, such as the **Sort**\([arr]\) invariant, however such invariants will not be discussed here, as no such invariants will be considered in this thesis.

The following string-related invariants are implemented as part of this thesis. Unless stated otherwise, all invariants currently recompute their output value from scratch upon receiving any change notification. This is because implementing incremental updating for each invariant can be both a difficult and a time consuming task. Since not all implemented invariants
are used in the experimental evaluation, implementing incremental updating for all invariants is left as future work, however proposed methods can be found in Section 7.

\[ \text{int} \leftarrow \text{Length}(s) \text{ maintains as its output value the length of string } s. \]

\[ \text{int} \leftarrow \text{Hamming}(s_1, s_2) \text{ maintains as its output value the Hamming distance between strings } s_1 \text{ and } s_2, \text{ where the Hamming distance is defined as the difference in their length plus the number of indices with different characters.} \]

The Hamming distance is only maintained incrementally for Replace\[s](i, c) \text{ notifications, where } s \text{ is either } s_1 \text{ or } s_2. \text{ If a Replace}[s_1](i, c) \text{ notification is received, then the following constant-time comparisons can determine if the distance increased, decreased, or is unaffected:} \]

1. If \( c = s_1[i] \), then the distance is unaffected.
2. Otherwise, if \( s_1[i] = s_2[i] \), then the distance is increased by 1.
3. Otherwise, if \( s_1[i] \neq s_2[i] \land c = s_2[i] \), then the distance is decreased by 1.
4. Otherwise, the distance is unaffected.

The same rule is also used for Insert\[s_2](i, c) \text{ notifications but with } s_1 \text{ and } s_2 \text{ swapped in the rules.}

For any Insert\[s](i, c), Remove\[s](i), \text{ or Reassign}[s](v) \text{ notification, where } s \in \{s_1, s_2\}, \text{ the distance is currently recomputed from scratch.}

\[ \text{int} \leftarrow \text{Levenshtein}(s_1, s_2) \text{ maintains as its output value the Levenshtein distance, also known as the edit distance, between strings } s_1 \text{ and } s_2. \]

The Levenshtein distance between string \( s_1 \) and \( s_2 \) is the smallest number of edits (insertions, removals, or replacements) on \( s_1 \), or \( s_2 \), or both that are required for \( s_1 \) to be equal to \( s_2 \).

The Levenshtein distance is currently not incrementally maintained. Instead it is recomputed for each change to any of the input strings.

\[ \text{int} \leftarrow \text{LevenshteinSubString}(s_1, s_2) \text{ maintains as its output value the Levenshtein substring distance between strings } s_1 \text{ and } s_2. \]

The Levenshtein substring distance between \( s_1 \) and \( s_2 \) is the smallest number of edits (insertions, removals, or replacements) on \( s_1 \), or \( s_2 \), or both that are required for \( s_2 \) to be a substring of \( s_1 \). It is used to compute the violation of the IsSubString\[s_1, s_2\] constraint (see Section 4.4).

The Levenshtein substring distance is currently not incrementally maintained.
int ← IndexedLevenshteinSubString(s_1, s_2, i) maintains as its output value the indexed Levenshtein substring distance, between strings s_1 and s_2, given index i. The indexed Levenshtein substring distance is the smallest number of edits (insertions, removals, or replacements) and shifts of index i, where a shift is equal to changing i by 1, required for s_2 to be a substring of s_1 starting at index i. It is used to compute the violation of the SUBSTRING(s_1, s_2, i) constraint (see Section 4.4). The indexed Levenshtein substring distance is currently not incrementally maintained.

string ← Concat(s_1, ..., s_n) maintains as its output value the concatenation of strings s_1 until s_n.

The invariant keeps track of the length of each string s_1, ..., s_n. When it receives an Insert[s_j][i, c], Remove[s_j][i], or Replace[s_j][i, c] notification for some string string s_j, the same modification is performed for the output value but with the index shifted based on the lengths of s_1 until s_j-1. When a Reassign[s_j](v) notification is received, the output value is recomputed based on the value of all input strings.

string ← SubString(s, i, ℓ) maintains as its output value the substring of s starting at index i (starting at 0) of length ℓ. Indices outside of the bound of s contain the empty character, i.e., the output value of SubString(abc,2,5) is cd. The reason for this is that if i, or ℓ, or both are the output of an invariant, then it can be difficult to write a heuristic that ensures that i + ℓ ≤ |s| without possible disconnecting the search space. Instead it is easier to post a constraint that i + ℓ ≤ |s| and allow the heuristic to explore these out-of-bounds assignments. Therefore, it is more helpful to define this behaviour instead of aborting with an index out-of-bounds exception.

string ← CharacterAt(s, i) maintains as its output value the character at index i in s.

string ← ReplaceAll(s, [r_1, ..., r_n], [t_1, ..., t_n]) maintains s with all occurrences of character r_i replaced by t_i as its output value, ∀i : 1 ≤ i ≤ n.

string ← Reverse(s) maintains s reversed as its output.

4.4 String Constraints

A constraint expresses a relationship between a number of input variables or constants. Each constraint maintains a degree of violation, representing how close the constraint is to being satisfied.
In OscaR/CBLS the violation of a constraint is represented by an integer variable, which means that other constraints and invariants can register a dependency on the violation and receive notifications when it is changed.

The violation is calculated differently for each constraint, however the violation of a satisfied constraint is always 0. For example, the violation of \textsc{Less}(a, b) is \(a - b + 1\) if \(a \geq b\) and otherwise 0. For \textsc{Eq}(a, b) the violation is given by \(|a - b|\). For some constraints, giving an exact value for the violation can be computationally intensive. In this case, an approximation can be used instead. However, it has been observed that under-approximating the violation can significantly decrease the performance.

The violation of a constraint is maintained incrementally, i.e., each constraint will register a dependency for each of its arguments and receive notifications when any of those variables are modified and then update the violation accordingly. However, it is common practice to express the violation as an invariant whenever possible, such that the violation will be maintained incrementally by some already implemented invariants. For some constraints, when the violation is expressed conditionally, as in the case of \textsc{Less}(a, b) where the condition is \(a \geq b\), the notifications that the constraint receives are used to check if the condition has changed and select one of two or more invariants as the violation value.

The following string-related constraints are implemented with the following method for calculating the violation:

\textsc{Eq}(s_1, s_2) holds if \(|s_1| = |s_2| \land \forall i : s_1[i] = s_2[i]\).

The violation is given by the output of the invariant \textbf{Hamming}(s_1, s_2), which is 0 if and only if they are equal and otherwise the number of indices where characters differ plus the difference in the strings length. Note that the Levenshtein distance could be used instead of the Hamming distance, but since the \textbf{Levenshtein}(s_1, s_2) invariant is currently not incrementally updated, the \textbf{Hamming}(s_1, s_2) invariant is preferable for performance reasons.

\textsc{IsSubString}(s_1, s_2) holds if \(s_2\) is a substring of \(s_1\).

The violation is given by the output of the invariant \textbf{LevenshteinSubString}(s_1, s_2), which is 0 if and only if \(s_2\) is a substring of \(s_1\) and otherwise equal to the minimum number of edits that must be performed on \(s_1\), or \(s_2\), or both in order for \(s_2\) to be a substring of \(s_1\).

\textsc{SubString}(s_1, s_2, i) holds if \(s_2\) is a substring of \(s_1\) starting at index \(i\), i.e., \(s_1[i..(i + |s_2| - 1)] = s_2\).

The violation is given by the output of the invariant \textbf{IndexedLevenshteinSubString}(s_1, s_2, i), which is 0 iff \(s_2\) is a substring of \(s_1\) starting at index \(i\). Otherwise the violation is equal to the
minimum number of edits to any of $s_1$, $s_2$, and $i$ required in order for $s_2$ to be a substring of $s_1$ starting at index $i$.

In CBLS, many constraints are expressed using invariants and other basic constraints. For example, the constraint for the relation $a + b = c$, called $\text{Plus}(a,b,c)$, can also be expressed as $\text{Eq}(\text{Plus}(a,b),c)$, i.e., $c$ is equal to the output value of the invariant $\text{Plus}(a,b)$, which maintains $a + b$ as its output value.

Therefore, it is not necessary to implement all string constraints that are found in the literature. For example, the following constraints can be implemented as indicated:

- $\text{Reverse}(s,x) \iff \text{Eq}(\text{Reverse}(s),x)$.
- $\text{Concat}(s_1,s_2,x) \iff \text{Eq}(\text{Concat}([s_1, s_2]),x)$
- $\text{Length}(s,\ell) \iff \text{Eq}(\text{Length}(s),\ell)$
- $\text{CharacterAt}(s,i,c) \iff \text{Eq}(\text{CharacterAt}(s,i,c))$
- $\text{ReplaceAll}(s,c_1,c_2,x) \iff \text{Eq}(\text{ReplaceAll}(s,c_1,c_2),x)$

### 4.5 String Neighbourhoods

A neighbourhood is a set of moves that are defined by some type of modification of some number of variables, such as reassigning any variable to a value within its alphabet or swapping the value of any two variables.

When designing a search procedure, it is usually helpful to implement multiple neighbourhoods, one for each type of move that the search requires, and then to draw moves from a combined neighbourhood that is the union of all neighbourhoods.

There are several benefits of having multiple, more basic neighbourhoods: the code becomes more readable, manageable, and it encourages experimentation, such as adding additional neighbourhoods. Furthermore, it also allows for a more complex behaviour of the search procedure. For example, all neighbourhoods do not need to be included in the combined neighbourhood at each iteration, instead neighbourhoods can be evaluated based on some strategy, such as a round-robin order over several iterations or only evaluating large, and thus time consuming, neighbourhoods when the smaller ones do not contain a good enough move.

In fact, a recent addition to OscaR/CBLS is the concept of `neighbourhood combinators` [12], which introduce a declarative way of defining search procedures by stating and combining neighbourhoods as well as the rules for when and how neighbourhoods are evaluated. The resulting combined neighbourhood can then be queried for a move by calling `getMove(obj, acceptanceCriterion)`, where `obj` is an objective function and `acceptanceCriterion` is a function.
that takes the old and the new objective value as arguments and returns true if a move that results in the new objective is allowed, otherwise false. The acceptance criterion is used by the neighbourhood when evaluating each move to determine if the move is allowed.

Standard domain-independent neighbourhoods are provided as part of the neighbourhood combinator framework in OscaR/CBLS. For example, the AssignNeighborhood neighbourhood takes an array of integer variables as input and changes the value of a single variable in the array as its move. A rule is created by a combinator, such as the Best(a, b) or the Random(a, b) combinator, which take two neighbourhoods a and b as input and, when queried for a move, returns respectively the best move of both a and b and the best move of either a or b, chosen randomly. A combinator is also considered to be a neighbourhood, which means that a combinator can be combined with other neighbourhoods or combinators. For example, Random(c, Best(a, b)) will, randomly, either return the best move of c or the best move of a and b. OscaR/CBLS provides a large number of combinators, some of which are described in [12]. New neighbourhoods and combinators that can be used with the existing framework can be added by extending existing interfaces.

The following general purpose neighbourhoods for string variables are implemented using the neighbourhood combinator framework:

ReplaceCharNeighbourhood([vars, tabuFunction, best, hotRestart]) takes an array vars of string variables as input. Moves are created by, for some variable in vars, replacing a character at some index with another character in the variable’s alphabet.

The optional function tabuFunction, which takes two arguments var and i, is used to implement tabu search. It returns true if a move involving variable var and a character at index i is allowed, otherwise false. If tabuFunction is not provided, then all moves are allowed, however note that moves returned by this neighbourhood must still satisfy the acceptance criterion that is provided when it is queried.

If the optional flag best is set to true, then all moves that the neighbourhood can create are evaluated when it is queried, and the move resulting in the lowest objective, where the objective function is provided with the query, is returned. Otherwise, the first move that satisfies the acceptance criterion is returned. By default best is set to true.

If the optional flag hotRestart is set to true and best is set to false, then the neighbourhood keeps its internal state when returning a move and, upon the next query, continues with the next move it would have evaluated in the previous query.
InsertCharNeighbourhood\((\text{vars}, [\text{tabuFunction}, \text{best}, \text{hotRestart}])\) takes an array \text{vars} of string variables as input. Moves are created by, for some variable in \text{vars}, inserting a character from the variable’s alphabet at some index.

The optional arguments \text{tabuFunction}, \text{best}, and \text{hotRestart} are used in the same way as in ReplaceCharNeighbourhood.

RemoveCharNeighbourhood\((\text{vars}, [\text{tabuFunction}, \text{best}, \text{hotRestart}])\) takes an array \text{vars} of string variables as input. Moves are created by, for some variable in \text{vars}, removing a character at some index.

The optional arguments \text{tabuFunction}, \text{best}, and \text{hotRestart} are used in the same way as in ReplaceCharNeighbourhood.

RandomReplaceCharsNeighbourhood\((\text{vars}, \text{numToReplace})\) takes an array \text{vars} of string variables and a natural number \text{numToReplace} as input. Moves are created by, for some variable in \text{vars}, randomly choosing \text{numToReplace} distinct indices and, for each chosen index, replacing the character at that index with a random character from the variable’s alphabet. If \text{numToReplace} is greater than the length of the variable, then all indices of the variable are replaced. The neighbourhood always returns the first move that satisfies the given acceptance criterion, as this neighbourhood is primarily used to escape local minima during search; by quickly randomising the assignment of a variable without taking the quality of the assignment into consideration.

RandomAssignStringNeighbourhood\((\text{vars}, \text{minLength}, \text{maxLength})\) takes an array \text{vars} of string variables and two integers \text{minLength} and \text{maxLength} as input. Moves are created by, for some variable in \text{vars}, reassigning it to some string of a random length, between \text{minLength} and \text{maxLength}, where each character of the string is chosen randomly from the alphabet of the variable.

The neighbourhood always returns the first move that satisfies the given acceptance criterion, as this neighbourhood is primarily used to escape local minima during search; by quickly randomising the assignment of a variable without taking the quality of the assignment into consideration.

A combination of ReplaceCharNeighbourhood, InsertCharNeighbourhood, and RemoveCharNeighbourhood is enough for a string variable during search to take potentially the value of any string consisting of characters from its alphabet. Furthermore, adding restrictions to when these neighbourhoods are queried based on the current length of a string, by for example only inserting characters when a string is under a certain length, the string variable can be limited to either a bounded length or a fixed length. Such restrictions on
when neighbourhoods are queried can be implemented using neighbourhood combinators.

5 Experimental Evaluation

The implementation is experimentally evaluated to assess its quality from both a modelling and a performance point of view. That is, we evaluate how expressive and readable the model is when using native string variables and compare the runtime of existing benchmarks with published results.

Two problems, the closest string problem and word equation systems, are considered in the evaluation. There exist many other problems that contain string variables. However, the closest string problem and word equations are chosen for the evaluation as they can easily be understood, there exist publicly available benchmarks for them, and their definition is already stated as a constraint problem. Note that translating a problem that is not commonly stated in terms of its variables and constraints into a constraint problem can be an extensive task.

All benchmarks for OscaR/CBLS were run on a machine with the following specification:

- Operating System: Windows 10
- Processor: 2.69 GHz Intel Core i7-4500U
- Memory: 8 Gb
- Cache: 4MB L3 cache
- Scala: Scala 2.11.7
- Java: Java 1.7

5.1 Closest String Problem

The closest string problem (CSP) is, given a set $S$ of strings of length $\ell$, to find a string $s$ of length $\ell$ and an integer $d$ such that: $\forall s' \in S: \delta(s, s') \leq d$ and $d$ is minimal, where $\delta(s, s')$ is the distance between $s$ and $s'$ given some metric. The distance metric is traditionally the Hamming distance, however there are some attempts of solving CSP with either the edit distance or the rank distance as metric [27] [13].

The CSP with Hamming distance as its metric is NP-complete [16] and has applications in computational biology [38] and coding theory [16].
5.1.1 CBLS Program

Listing 2 shows the CSP problem modelled in OscaR/CBLS using the new string variables, invariants, and neighbourhoods. The model takes as input the alphabet as a set of strings, which are assumed to be of length 1, an array of input strings that corresponds to the set $S$ in the problem definition, and the maximum runtime in milliseconds.

The problem is modelled between line 14 and 33. The initial value of the closest string is created on line 17 using the `generateInitialValue` method shown in Listing 3. The `generateInitialValue` method outputs a string where the character at each index is one of the most frequent characters at the same index of the strings in the input array. Essentially, `generateInitialValue` is a naïve approximation of the closest string. The variables corresponding to $s$ and $d$ in the problem definition are `closestString` and `minDistance`, created on lines 20 and 21 respectively. Line 24 creates an array of `HammingDistance` invariants for each input string and the closest string. On line 22 a constraint is posted for each distance invariant, which states that the distance is less than or equal to the `minDistance` variable. Finally, two objective variables are created on lines 29 and 32 that are used in different parts of the search heuristic.

The search procedure is implemented between lines 35 and 77. The search tries to find the optimal solution by solving a series of satisfaction problems, each with a lower value of `minDistance`. That is, the value of `minDistance` is initially set to the maximum value of a distance invariant minus 1 on line 37, and the search procedure will then search for a value of `closestString` that satisfies the constraints. Upon satisfying the model, `minDistance` is decremented by 1 on line 72 and the search procedure then tries to satisfy the constraints for the new assignment. This continues until the maximum runtime is exceeded.

Tabu search is used as a meta-heuristic and is implemented by passing a tabu function, as defined between lines 38 and 44, to `ReplaceCharNeighbourhood` on line 50 and by updating the tabu value after each move on line 58. The tabu list keeps a tabu value for each index of the string. The tenure is set to 3 as this was found to be a sufficiently low value to help the search procedure escape local minima.

The neighbourhood is defined between lines 49 and 61 using neighbourhood combinators. On line 50 `replaceNeighbourhood` is defined to replace a character in `closestString` as its move and to perform at most 200 moves without improving `viol`, after which it will be exhausted and return `NoMoveFound`. On line 51 `replaceAndRestoreOnFailNeighbourhood` is defined to return moves from `replaceNeighbourhood`, and save the best assignment found. It restores the best assignment when `replaceNeighbourhood` is exhausted. On line 53 `neighbourhood` is defined to return moves from `replaceAndRestoreOnFailNeighbourhood`. The `maxFails` combinator states
that `replaceAndRestoreOnFailNeighbourhood` is reset, and re-queried for a new move, when it returns `NoMoveFound`. After failing for the second time, `replaceAndRestoreOnFailNeighbourhood` is considered exhausted. The `onFirstMove` and `retry` combinators are used to reset the neighbourhood, reset the tabu list, and set `closestString` to its initial value, when `replaceAndRestoreOnFailNeighbourhood` is exhausted. This acts like a re-start to help escape local minima. Experiments on the benchmark indicated during development that restarting from the initial assignment is preferable to restarting from a random assignment. The `afterMoveOnMove` combinator attaches code that is run after each move, in this case the attached code updates the tabu value of the index affected by the move.

Finally, the actual search takes place between line [63 and 77] where, until the maximum runtime is reached, the neighbourhood is queried for a move that results in the lowest objective value, which may increase the objective value depending on which moves are tabu. The found move, if any, is committed to and if a new solution is found, then `minDistance` is decreased by 1, `replaceNeighbourhood` is reset such that the internal counter of `maxMovesWithoutImprovement` is reset, and the tabu list is reset.

Listing 2: Model for the Closest String Problem written in Scala using the OscaR/CBLS framework.

```scala
import oscar.cbls.constraints.core._
import oscar.cbls.constraints.lib.basic._
import oscar.cbls.constraints.lib.number._
import oscar.cbls.constraints.lib.string._
import oscar.cbls.objective._
import oscar.cbls.search._
import oscar.cbls.search.core._
import oscar.cbls.search.move._

object CSPSolver extends SearchEngine with StopWatch {
  def solve(alphabet: Set[String], inputStrings: Array[String],
             maxTime: Long) = {
    startWatch()
    // Model
    val m: Store = new Store(false, None, true)
    // Values
    val initialVal = generateInitialValue(alphabet, inputStrings)
    val len = initialVal.length
    // Variables
    val closestString = CBLSStringVar(m, initialVal, alphabet)
    val minDistance = CBLSIntVar(m, len, 0 to len, "minDistance")
    // Constraints
    val c = ConstraintSystem(m)
    val distances = for (s <- inputStrings) yield
                       HammingDistance(s, closestString)
    // ...
distances.foreach {
  d: HammingDistance => c.add(LE(d, minDistance))
}

// Violation of the constraint system
val viol = Objective(c, m)
c.close()

// The minDistance plus the violation
val objectiveAndViolation = Objective(Sum2(c.violation, minDistance))
m.close()

// Search
// Set initial minDistance
minDistance := (distances.maxBy(_.value).value - 1)

// Tabu
var currentIteration = 0
val tenure = 3
val tabulList: Array[Int] = Array.fill(len)(0)
def tabuFunction(str: CBLSStringVar, idx: Int): Boolean = {
  return tabulList(idx) <= currentIteration
}
def restartFromInitial(): Unit = {
  closestString := initialVal
  tabulList.transform(_ => 0) // Reset tabu list
}

// Neighbourhoods
val replaceNeighbourhood =
  ReplaceCharNeighbourhood(Array(closestString), tabuFunction)
val maxMovesWithoutImprovement = 200
val replaceAndRestoreOnFailNeighbourhood = replaceNeighbourhood
val saveBestAndRestoreOnExhaust = objectiveAndViolation

// Final neighbourhood
val neighbourhood = (replaceAndRestoreOnFailNeighbourhood maxFails 2
  onFirstMove(restartFromInitial) retry (_ => true))
  afterMoveOnMove{
    // Code to run after each move to update tabulist
    (move: Move) =>
    stripInstrumentedMove(move) match {
      case m: ReplaceCharMove =>
        tabulList(m.idx) = currentIteration + tenure
      case _ =>
    }
  }

// Perform the search
while (getWatch < maxTime) {
  neighbourhood.getMove(obj = viol,
    acceptanceCriteria = (oldObj, newObj) => true) match {
    case move: MoveFound => move.commit()
    case _ =>
  }
}
Listing 3: Method for generating the initial assignment of the closest string.

def generateInitialValue(alphabet: Set[String],
                        inputStrings: Array[String]): String = {
  var characterCount: Map[Char, Int] = Map.empty
  for(c <- alphabet){
    characterCount = characterCount + (c.charAt(0) -> 0)
  }
  var initialValue = ""
  for( i <- 0 until inputStrings(0).length){
    var currentCounter = characterCount
    for(j <- inputStrings.indices){
      currentCounter = currentCounter + (inputStrings(j).charAt(i) ->
                                               (currentCounter(inputStrings(j).charAt(i)) + 1))
    }
    initialValue = initialValue + currentCounter.maxBy(_._2)._1.toString()
  }
  initialValue
}

5.1.2 Model Evaluation

The CBLS model is in this case essentially a one-to-one mapping of the problem’s definition, having the same variables and constraints, albeit with different names. This is not the case for the linear model or pseudo-code of the methods used in [10] and [15], where the former translates the problem into a number of linear equations on a 2D-array of integer variables with domain \{0, 1\}, and the latter translates the problem into a path-finding problem.

Furthermore, the declarative nature of the model and the encapsulation of notions, such as the distance metric within the concept of a constraint,
make it easy to modify or extend. For example, to change the distance metric, one only needs to replace the `HammingDistance` invariant with another distance invariant, such as `LevenshteinDistance`. For the linear model of [10], changing the distance metric to Levenshtein distance would require reformulating most, if not all, of the model, as it is built specifically for the Hamming distance.

The search heuristic is not as readable as the model, requiring more than twice as many lines of code and expressing a fairly complex behaviour that is not necessarily evident at first glance. However, from a design point of view, the concept of neighbourhood combinators makes the search heuristic easy to work with and develop. In this case, the heuristic was designed by starting from `ReplaceCharNeighbourhood` and incrementally adding or removing combinators while experimentally evaluating the effect.

### 5.1.3 Benchmarks

The performance of the CBLS model is evaluated on the benchmark, of randomly generated instances, used in [10]. Each instance is generated by first creating a random master string \( m \) of length \( n \) over alphabet \( \Sigma \) and then making \( k \) copies of \( m \). Each copy is then randomly altered at \( \alpha \) random positions. The \( k \) strings generated this way as well as \( \Sigma \) are then the input strings and alphabet.

Note that there are more recent papers on the CSP, such as [36, 39, 11], that improve on the results of [10]. Unfortunately, these were initially overlooked during the evaluation and, due to time limitations, they will not be a part of the evaluation. However, it should be noted that [11] includes a larger variety of instances in terms of the parameters, such as instances for \( k = n \) and \( k \gg n \) as opposed to only having instances where \( k << n \) as in [10]. Comparing against these other types of instances would be very interesting as they may reveal some further benefits of using CBLS, but this is left as future work. Note also that the instances of [11] are not publicly available, so an exact comparison would be impossible to make, as the minimum distance can be different for different random instances generated with the same parameters.

Table 1 shows a comparison between the runtime and distance found by OscaR/CBLS using the model of Listing 2, reported under the `CBLS` column, and the results, obtained on different hardware, reported in Table 1 of [10].

The algorithms `ANT`, `H`, and `HP` come from [15], [28], and [23] respectively and are incomplete methods. The algorithms `RT` and `BB` are introduced in [10], where `BB` is a complete method that uses an off-the-shelf branch-and-bound algorithm, while `RT` is an incomplete method that uses an off-the-shelf heuristic. The different algorithms were run on very different

---

4Instances available at http://www-lehre.informatik.uni-osnabrueck.de/theoinf/index/research/csp_csp
Table 1: Comparison of the results presented in [10] and the performance of the CBLS model in Listing 2. The runtime and distance $d$ are the average over 10 different instances generated for the same $k$ and $n$, all with $|\Sigma|=4$. The %BB column shows the percent of instances where the branch-and-bound algorithm (BB) was able to prove a distance optimal. Values in bold are the best reported values while - denotes that the result is not available, as the instances were considered either too small or too large.

<table>
<thead>
<tr>
<th>$k$</th>
<th>$n$</th>
<th>Runtime (sec)</th>
<th>Distance $d$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>ANT</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>500</td>
<td>2.11</td>
<td>1.00</td>
</tr>
<tr>
<td>10</td>
<td>1000</td>
<td>7.85</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>2000</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>5000</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>20</td>
<td>500</td>
<td>3.51</td>
<td>2.00</td>
</tr>
<tr>
<td>20</td>
<td>1000</td>
<td>11.80</td>
<td>-</td>
</tr>
<tr>
<td>20</td>
<td>2000</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>20</td>
<td>5000</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>30</td>
<td>500</td>
<td>6.76</td>
<td>3.00</td>
</tr>
<tr>
<td>30</td>
<td>1000</td>
<td>10.70</td>
<td>-</td>
</tr>
<tr>
<td>30</td>
<td>2000</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>30</td>
<td>5000</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

hardware, which makes it impossible to do a true runtime comparison. The ant algorithm (ANT) [13] was run on an Intel Pentium M750 1.86 GHz and 1 GB RAM, the heuristic (H) [28] on 2.8 GHz and 512 MB RAM, the parallel heuristic (HP) [23] on 28x Intel Xeon dual processors with 1 GB RAM each, and the root heuristic (RT) and branch-and-bound (BB) [10] on an Intel Xeon E5520, 2.27 GHz, and 8 GM RAM.

The result shown in Table 1 for each pair of $k$ and $n$ is the average over 10 randomly generated instances for these values of $k$ and $n$ with $|\Sigma|=4$ and $\alpha=n$. Note that [13, 28, 23, 10] all use different randomly generated instances, but generated with the same parameters. As noted in [10], comparing the average over different instances can only show a tendency in performance as different instances will have a different minimal distance. However, the same instances that are used in [10] are used by CBLS.

Due to time limitations, each instance was run for 5 minutes, as opposed to the 30-minute-timeout of [10]. The reported runtime for CBLS is the average time it took to find the best solution. Since, in this case, CBLS cannot prove a solution optimal, the solver always runs for the full 5 minutes.
5.1.4 Results

Table 1 shows that CBLS is outperformed by the root heuristic of [10], in terms of both runtime and quality of solutions. Furthermore, for all instances where \( k = 10 \) and for two instances where \( k = 20 \), the branch-and-bound algorithm, which proves its solutions optimal, runs faster than CBLS. However, CBLS outperforms, by a significant margin, the results of ANT, H, and HP in the quality of the solutions.

Comparing the runtimes of ANT, H, and HP with CBLS, we can see that CBLS appears to be slower. However, since CBLS finds solutions of better quality, the runtime difference can be attributed to the difference in the quality. In fact, during the CBLS evaluation it was observed that the runtime required to find the best and second-best solutions often differed by an order of magnitude.

5.2 Word Equation Systems

Given an alphabet \( A \) and a set \( \Omega \) of string variables, a word expression over the theory of concatenation is a sequence of concatenations of variables from \( \Omega \) and strings of characters from \( A \). For example, if \( A = \{0, 1\} \) and \( \Omega = \{x_1, x_2, x_3\} \), then \( 1x_10110x_2x_1 \) is a word expression, assuming an implicit concatenation following each variable and string.

A word equation, over the theory of concatenation, is an equality between two word expressions, \( L = R \). The satisfiability problem of word equations is known to be NP-hard [5]. If a word equation is satisfiable, then a solution is an assignment of each variable to a string, such that both word expressions are equal. Continuing the previous example, \( 1x_10110x_2x_1 = x_1x_310x_21 \) is a word equation with an infinite number of solutions, one being: \( x_1 \leftarrow 1, x_2 \leftarrow 01, x_3 \leftarrow 101 \).

Given an alphabet \( A \), a set \( \Omega \) of string variables, and a set of word equations \( S = \{L_1 = R_1, \ldots, L_n = R_n\} \), the word equation system (WES) problem, over the theory of concatenation, is to find a solution if one exists or otherwise determine its non-existence. A solution is an assignment of each variable in \( \Omega \) to a string over alphabet \( A \), such that all word equations in \( S \) are satisfied. Solving a bounded version of a WES, where the maximum length of each variable is bounded by a natural number \( d \), called the \( d \)-WES problem, is known to be NP-complete [4].

5.2.1 CBLS Program

Listing 4 shows the WES problem modelled in OscaR/CBLS using the implemented string variables, invariants, and neighbourhoods. The model takes as input the number of variables, i.e., \( |\Omega| \), the equations of the WES, and the maximum runtime. The model is built to follow the specifications of the benchmarks of [4] [3] [2]. As a result, the alphabet is always
binary, \( A = \{0, 1\} \), and each equality in the WES is represented by a pair of arrays of integers, where each array of integers corresponds to a string expression. Each string expression is created by mapping the integers 0 and 1 to the corresponding character 0 and 1, the integer 2 to the empty string, and integers \( i > 2 \) to variable \( x_{i-2} \). The word equation \( 1x_10110x_2x_1 = x_1x_310x_31 \) is thus expressed as the following pair of arrays of integers: \([1, 3, 0, 1, 1, 0, 4, 3], [3, 5, 1, 0, 4, 1] \). Note that the mapping of the integer 2 to the empty string is used by \([4] \), however it is not used in the CBLS model.

Between lines 13 and 27 the string variables are created, as is the mapping from integers to constants and string variables. Each string variable is given the empty string as its initial value.

For each equation in the input, the left-hand and right-hand sides of the equation are built from their integer array representations on lines 31 and 32 respectively, by applying the mapping and concatenating the resulting arrays. An equality constraint is posted on each left-hand and right-hand side pair on line 33.

Tabu search, defined between lines 41 and 52, is used as a meta-heuristic and its tabu function is passed to neighbourhoods on line 54 and updated on line 61. The tabu map, essentially equivalent to a tabu list, maps a variable and an index to a tabu value, indicating if modifying a variable at an index is allowed. The tenure is set to the number of variables, as this was found to be a good value. When updating the tabu value of a variable and index pair, the value is set to a random value between \( \text{tenure} \) and \( 2 \cdot \text{tenure} \).

The neighbourhood is defined between lines 53 and 64. On line 54, \( \text{charMoveNeighbourhood} \) is defined to return the best non-tabu move of inserting, replacing, or deleting a character in a string variable. On line 55, \( \text{charMoveNeighbourhood} \) is defined to return moves from \( \text{charMoveNeighbourhood} \) but taking at most 500 moves without improving the best value of \( \text{viol} \). It also saves the best found solution and restores it on exhaust. On line 56, \( \text{resetNeighbourhood} \) is defined to exhaust \( \text{restoreNeighbourhood} \), allowing it to fail two times before it is exhausted, and then randomly assigns up to one fourth of the variables to a random string of length between 1 and 5. It then resets and goes back to querying \( \text{restoreNeighbourhood} \). Finally, on line 57, the \( \text{afterMoveOnMove} \) combinator is attached to update the tabu value after each move.

The actual search, defined between lines 65 and 73, iterates until a solution is found or until it has run for the maximum runtime. In each iteration, it queries \( \text{neighbourhood} \) for its best move and commits to it.

```scala
import oscar.cbls.constraints.core.ConstraintSystem
import oscar.cbls.constraints.lib.string.StrEQ
import oscar.cbls.invariants.core.computation.{CBLSStringVar, Store, StringValue}
import oscar.cbls.invariants.lib.string.Concat
import oscar.cbls.objective.Objective
import oscar.cbls.search._
import oscar.cbls.search.core.MoveFound
import oscar.cbls.search.move._
import scala.collection.mutable.Map => MutableMap
object WordEquationSolver extends SearchEngine with Stopwatch {
  def solve(numVars: Int, equations: Array[(Array[Int], Array[Int])],
            maxTime: Long) = {
    startWatch()
    // model
    val model: Store = new Store(false, None, true)
    val alphabet = Set("0", "1")
    // Mapping from int to variables
    var variableMap: MutableMap[Int, StringValue] = MutableMap.empty
    // Map 0 and 1 to constant strings
    variableMap += (0 -> "0")
    variableMap += (1 -> "1")
    // Create variables and add to mapping
    val variables = for (i <- 0 until numVars) yield {
      val id = i + 3
      val variable = CBLSStringVar(model, ",", alphabet, "Var" + id)
      variableMap += (id -> variable)
      variable
    }
    // Constraints
    val c = ConstraintSystem(model)
    for ((leftSide, rightSide) <- equations) {
      val left = Concat(leftSide.map(variableMap(_)))
      val right = Concat(rightSide.map(variableMap(_)))
      c.add(StrEQ(left, right))
    }
    c.close()
    // Violation of the constraint system
    val viol = Objective(c.violation)
    model.close()
    // Search
    // Tabu
    var tabuMap: MutableMap[(CBLSStringVar, Int), Int] = MutableMap.empty
    var currentIteration = 0
    var tenure = numVars
    def tabuFunction(str: CBLSStringVar, idx: Int): Boolean = {
      true
    }
  }
}
```
```scala
if (tabuMap.contains((str, idx))) {
    return tabuMap((str, idx)) <= currentIteration
} else {
    tabuMap += (str, idx) => 0
    return true
}

// Neighbourhoods
val charMoveNeighbourhood = InsertCharNeighbourhood(variables.toArray, move: tabuFunction)
val best ReplaceCharNeighbourhood = variables.toArray, move: tabuFunction)
val best DeleteCharNeighbourhood = variables.toArray, move: tabuFunction)

val restoreNeighbourhood = charMoveNeighbourhood
val restoreBest = maxMovesWithoutImprovement (500, viol)
val saveBestAndRestoreOnExhaust = viol
val resetNeighbourhood = (restoreNeighbourhood maxFails 2) exhaustBack
val RandomStringAssignNeighbourhood = variables.toArray, 1, 5
val maxMoves = numVars/4

val neighbourhood = resetNeighbourhood afterMoveOnMove {
    (move: Move) =>
    stripInstrumentedMove(move) match {
        case mv: CharMove =>
            tabuMap += ((mv.variable, mv.idx) => (currentIteration + tenure + RandomGenerator.nextInt(tenure)))
        case _ =>
    }
}

// Perform the search
while(getWatch < maxTime && viol.value > 0) {
    neighbourhood.getMove(viol, (_, _) => true) match {
        case move: MoveFound =>
            move.commit
            currentIteration = currentIteration + 1
        case _ =>
    }
}

// Restore best solution
restoreNeighbourhood.restoreBest()

// Print results
if(viol.value == 0){
    println("Solution Found")
    println("Variables = " + variables.mkString(" "))
    println(s"Current iteration: $currentIteration, current time: $getWatch")
} else{
    println("No Solution Found")
}
```
5.2.2 Model Evaluation

Overall, the CBLS model is very similar to the problem description: given just the model, it is easy to understand what is being solved. The search heuristic is fairly complex but is both self-descriptive and easy to modify. In comparison, the pseudo-code and descriptions presented in [4, 3, 2] spend a lot of effort on describing fitness functions and genetic operators.

For the CBLS model, the equivalent of a fitness function is given by the violation of the constraint system. This comes with several benefits compared to tailoring a fitness function for the problem. For example, if we were to add additional constraints to the model, then the violation would by default include these constraints as well, as opposed to manually adapting a fitness function. Furthermore, the equality constraint currently uses the Hamming distance invariant to compute the violation. If we want to use the Levenshtein distance as the metric, then it is just a matter of changing one line of code in the constraint, namely from `HammingDistance` to `LevenshteinDistance`, without modifying the model, as opposed to rewriting the entire fitness function as done in [2].

There exist more general formulations of word equation systems, including more string operators, such as morphisms, antimorphisms, and reversal [9]. Another strength of the CBLS model is that, in its current form, it can easily be extended to include such operators, as it is just a matter of using, or implementing, the appropriate invariants when building the equations. Such extensions can be made without changing the search heuristic, although some tuning would probably be required.

5.2.3 Benchmarks

The performance of the CBLS model is evaluated on the benchmark used in [4, 3, 2]. Each instance is generated by some unspecified algorithm that ensures that each instance has at least one solution, in which the length of each variable is lower than or equal to some value $q \in \mathbb{N}$.

Each instance is denoted by $p_n-m-q$, where $n$ is the number of equations, $m$ is the number of variables, and $q$ is the maximum variable length in at least one solution of the instance. It is important to note that the solver of [4, 3, 2] is solving the $d$-WES problem for different values of $d \geq q$, which means that it puts an upper bound $d$ on the length of any variable. However, the CBLS model puts no such upper bound. This means that, in theory, the CBLS model is solving a harder problem, as no restriction is put on the size of the search space. However, during the evaluation, it was observed that, for the benchmark instances, the explored candidate solutions tend to be of a short length. This is not surprising as the random restart, which

\footnote{Instances available at \url{http://www.aic.uniovi.es/Tc/spanish/repository.htm#testproblems}}
Table 2: The result of LSG1 [2] compared with running Listing 4 using Hamming distance or Levenshtein distance as the violation metric for the equality constraint, shown under CBLS (Hamming) and CBLS (Levenshtein) respectively. The success rate SR, runtime, shown in seconds, and iterations are the average over 50 independent runs with a timeout of 5 minutes. The U.B column shows the upper bound on the length of any string variable used in LSG1. No such upper bound was used in CBLS.

<table>
<thead>
<tr>
<th>P.instance</th>
<th>LSG1</th>
<th>CBLS (Hamming)</th>
<th>CBLS (Levenshtein)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>U.B</td>
<td>SR</td>
<td>AES</td>
</tr>
<tr>
<td>p5-15-3</td>
<td>10</td>
<td>80%</td>
<td>299770.87</td>
</tr>
<tr>
<td>p10-8-3</td>
<td>10</td>
<td>100%</td>
<td>24892.70</td>
</tr>
<tr>
<td>p10-15-3</td>
<td>5</td>
<td>100%</td>
<td>14851.88</td>
</tr>
<tr>
<td>p10-15-5</td>
<td>5</td>
<td>100%</td>
<td>15498.80</td>
</tr>
<tr>
<td>p15-12-4</td>
<td>4</td>
<td>100%</td>
<td>1690.00</td>
</tr>
<tr>
<td>p15-25-5</td>
<td>5</td>
<td>90%</td>
<td>84708.33</td>
</tr>
<tr>
<td>p25-8-3</td>
<td>20</td>
<td>100%</td>
<td>19377.40</td>
</tr>
<tr>
<td>p25-23-4</td>
<td>5</td>
<td>100%</td>
<td>10210.25</td>
</tr>
</tbody>
</table>

randomly assigns variables to random strings of length 1 to 5, encourages the search heuristic to explore solutions where variables have a short length.

As part of the evaluation, we will also explore how different ways of calculating the violation for the equality constraint affect the performance. Specifically, the Levenshtein distance invariant, which is not incrementally maintained and has a high time complexity, will be compared with the Hamming distance invariant, which is incrementally maintained for replace character moves and has a low time complexity.

The success rate, runtime, and number of iterations of the CBLS model are reported in Table 2, where they are compared with the results, obtained on different hardware, of [2]. For each instance, only the result from [2] where the highest upper bound is used is reported. Note that the results of [1, 3] are not reported as [2] improves on these results in every aspect.

Unfortunately, the used hardware is not reported in [2]. It is reasonable to assume that it is significantly slower than the hardware used here, as there is a 9 year difference between now and then. However, the hardware can be assumed to be at least as good as the hardware used in [3], which is AMD Athlon XP 1900+; 1.6 GHz and 512 MB RAM.

Each reported value is the average over 50 independent runs. For CBLS a timeout of 5 minutes is used; the timeout or runtime are not reported in [2], but runtimes ranging from two seconds to five minutes are mentioned in [3].
5.2.4 Results

Table 2 shows that the CBLS model is able to solve almost all instances with a 100% success rate in less than a few seconds.

Looking at the impact of using Levenshtein distance instead of Hamming distance to calculate the violation, we can see that the number of iterations is significantly reduced for all instances. If we look at the runtime and consider the number of iterations per seconds, then we can see that using Hamming distance is much faster, in terms of iterations per seconds, than using Levenshtein distance. However, in this case, it appears that the slowdown of using Levenshtein distance is not significant thanks to its ability to help guide the search.

While it is surprising that using the Levenshtein distance helps guide the search to such an extent, it was not unexpected. After all, the Levenshtein distance measures the distance assuming that insertion, replacement, and removal of characters can be made, which is true for the used search heuristic. The Hamming distance however measures the distance assuming only replacement of characters can be made.

Comparing the success rate between CBLS (Hamming) and CBLS (Levenshtein) we can see two things. First, for CBLS (Levenshtein), instance $p_{10-15-5}$ is solved with a 100% success rate and in one order of magnitude fewer iterations than CBLS (Hamming). Secondly, instance $p_{15-25-5}$ and $p_{25-23-4}$ are solved with a higher success rate than for CBLS (Hamming). It could be that these instances are solved by chance rather than by strategy and thus the higher number of iterations causes these instances to be solved with a higher success rate. However, while this is plausible for $p_{25-23-4}$, the success rate of 98% for $p_{15-25-5}$ indicates that using the Hamming distance may be desirable for this particular instance.

When comparing the result of LSG1 and CBLS, it needs to be emphasised that the results of LSG1 are very old and were run on significantly slower hardware. Furthermore, CBLS is solving a harder version of the problem, with no upper bound on the length of any variable. Therefore, no claim can be made about CBLS outperforming LSG1, instead the comparison is made to give a context for the benchmark. With this in mind, if we do a comparison with both of the Hamming and Levenshtein results, then we can note a couple of things. To begin with, there is unfortunately no conversion between the average number of evaluations until solution (AES) and the number of iterations. However, we can use the AES metric to see how difficult an instance was for LSG1. Based on the highest AES, it would appear that $p_{5-15-3}$ is the hardest instance for LSG1. Indeed, in [1, 3] this instance is mentioned as a particularly hard instance. However, this is an easy instances for CBLS for both violation measurements. On the other hand, LSG1 has a 100% success rate for instance $p_{25-23-4}$, which is the hardest instance for CBLS. This indicates that LSG1 and CBLS solve the instances
with somewhat orthogonal methods and that there may be more difficult instances where one outperforms the other.

5.3 Discussion

Considering that this is a first attempt at implementing string variables in CBLS, the results in Section 5 are fairly promising.

The model evaluation in Sections 5.1.2 and 5.2.2 showed that the implemented constraints and invariants were very expressive and easy to understand, as opposed to the specialised algorithms for the same problems. However, this is very much expected, as having expressive models is the purpose of constraint modelling. The performance evaluation in Sections 5.1.4 and 5.2.4 showed that using string variables, invariants, and constraints can yield good results. While, in this case, the performance of either model did not improve the state-of-the-art, they were still better than some of the methods compared with. Seeing how only a moderate effort was put into experimenting with different heuristics and parameters, it stands to reason that further experimentation with the heuristic could improve the performance.

Several choices were made when implementing the string variable type, such as the value representation and the domain representation. Using the StringBuilder class as the value representation and the alphabet as the domain representation did not provide any obstacles during the experimental evaluation. However, it may be worth considering other representations. If the value was instead represented as a sequence of character objects, then it could be possible to attach more fine-grained information to each character. For example, with the current representation, the violation is given for the entire string. However with a sequence of character objects representation, each character object could have an associated violation, which could be useful in a heuristic that does not explore moves for indices with a violation of 0. Furthermore, each character object could have an associated alphabet that can be different from the alphabet of other characters in the same string. This could have been useful for the closest string problem, where characters that do not occur at an index in the input strings can be removed for the alphabet at that index as a preprocessing step. This would significantly increase the performance for instances were the alphabet is larger than the number of input strings, as it would remove unnecessary moves. However, such alternative representations were not evaluated as part of this thesis, as it is unclear how they could be implemented efficiently and in accordance with some restrictions that are imposed by the OscaR/CBLS framework.
6  Conclusion

The goal of this thesis was, for the first time, to implement native string variables, invariants, and constraints in a CBLS framework and evaluate how useful they are when solving string problems.

The implementation was done in the OscaR/CBLS framework, by extending and reworking existing classes. The experimental evaluation in Section 5 on the CSP and WES problems showed that using string variables, invariants, and constraints resulted in expressive models that are easy to understand.

The performance of the CBLS models was evaluated on a set of benchmark instances. The model for the WES problem produced what appears to be competitive results. However, due to a significant difference in the used hardware and a lack of reported runtimes in the literature, it is not possible to say which method is better. For the CSP problem, the model produced results that were good, but not competitive with current specialised algorithms. However, both CBLS models only required a short development time, with most of the time spent on experimenting with different heuristics. This suggest that a benefit of using CBLS with string variables is that good-quality results can be achieved after only a short development time.

In conclusion, CBLS with string variables is off to a promising start, as CBLS seems to be be a viable option for modelling and solving string problems. However, to produce results that are competitive with specialised algorithms, more effort needs to be put into investigating (meta-)heuristics and the performance of invariants and constraints needs to be improved.

7  Future Work

There is plenty of future work to be done, ranging from implementing more invariants, constraints, and neighbourhoods to improving the algorithms for incrementally maintaining existing invariants and constraints. Furthermore, the domain representation as well as the types of moves on string variables should be explored further. For example, the use of constraint-specific neighbourhoods is not explored in this thesis.

Improving Invariants

Currently, only Hamming($s_1, s_2$) and Concat($s_1, \ldots, s_n$) are incrementally maintained, as only these were used in the evaluation, and implementing efficient algorithms for incrementally maintaining invariants can be very time consuming. Note that Hamming($s_1, s_2$) is only incrementally maintained for Replace[$s$](i, c) notifications.

Since the Eq($s_1, s_2$) constraint uses an invariant, Hamming($s_1, s_2$) by
default, to maintain its violation, it is as incrementally updated as the invariant it uses.

However, the following methods for incrementally maintaining invariants were considered, but not implemented, during the thesis:

```
int ← Length(s) Upon receiving an Insert[s](i, c) or Remove[s](i) notification, the current value of the invariant is increased or decreased by 1 respectively. When receiving a Replace[s](i, c) notification, no action is taken, as the length has not been affected.
```

```
int ← Hamming(s₁, s₂) The Hamming distance is already maintained incrementally for Replace[s](i, c) notifications.
For Insert[s](i, c) and Remove[s](i) notification, the side of the string to the right of the insertion or removal point i is changed in its alignment to the other string. Therefore, the delta of the distance for the right side of the string is computed and added to the invariants current value.
```

```
string ← SubString(s, i, ℓ) For an Insert[s](k, c) notification where k < i, the last character in the substring is removed and the character at index i in the new version of s is added to the start of the string. If i ≤ k < i + ℓ, then the last character of the substring is removed and c is inserted at index k − i in the substring.

For a Remove[s](k) notification where k < i, the first character of the substring is removed and the character at index i + ℓ in the old version of s is added to the end of the substring. If i ≤ k < i + ℓ, then the character at index k − i of the substring is removed and the character at index i + ℓ in the old version of s is added to the end of the substring.

For either an Insert[s](k, c) or Remove[s](k) notification for an index k ≥ i + ℓ, the substring is not affected.
```

```
string ← CharacterAt(s, i) Each Replace[s](k, c) notification, where k ≠ i, the change does not affect the output value. For all other notifications, the constant-time lookup of s[i] is made on the new value of s[i]. If s[i] differs from the current output value, the output value
is changed. Otherwise, no action is performed so as not to trigger any
unnecessary notifications, as reassigning a variable to its current value
would do that.

\[ \text{string } \leftarrow \text{ReplaceAll}(s, [r_1, \ldots, r_n], [t_1, \ldots, t_n]) \]
For an Insert\([s](i, c)\) or \(\text{Replace}[s](i, c)\)
notification where \(c = r_k\) for some \(k\), the character \(t_k\)
is used instead of \(c\) to insert or replace the character at index \(i\)
in the output string. For all other \(\text{Insert}[s](i, c)\), \(\text{Remove}[s](i)\),
and \(\text{Replace}[s](i, c)\) notifications, the modification that corresponds to the
notification is performed on the output string.

\[ \text{string } \leftarrow \text{Reverse}(s) \]
For an Insert\([s](i, c)\), Remove\([s](i)\), or Replace\([s](i, c)\)
notification, the same modification is made to the output string but
at index \(|s| - i|\).

\[ \text{int } \leftarrow \text{Levenshtein}(s_1, s_2) \]
is currently implemented as a dynamic pro-
gramming algorithm with a time complexity of \(O(|s_1| \cdot |s_2|)\). Modifying
this algorithm to efficiently maintain the distance incrementally, could
prove to be very difficult.

However, as observed in \cite{24}, Section 5.5, the Levenshtein distance,
also known as the edit distance, can be computed more efficiently us-
ing the algorithm of Ukkonen \cite{37}. This algorithm takes as input two
strings \(s_1\) and \(s_2\) as well as an upper bound \(w\) of the Levenshtein
distance. The algorithm then computes in \(O(w \cdot \min(|s_1|, |s_2|))\) time a
distance \(d\), such that if \(d \leq w\), then the Levenshtein distance is \(d\).
Otherwise the Levenshtein distance is some value greater than \(w\). In the
original algorithm of Ukkonen, the algorithm starts with \(w = 1\) and it-
eratively doubles the value of \(w\) until the Levenshtein distance is found.
However, in the case of incrementally maintaining the distance, this
can be done more efficiently. When a \(\text{Levenshtein}(s_1, s_2)\) invariant
that outputs a distance \(d'\) receives an \(\text{Insert}[s](i, c)\), a \(\text{Remove}[s](i)\), or
a \(\text{Replace}[s](i, c)\) notification, then the distance can at most increase
or decrease by 1, by the definition of Levenshtein distance. Therefore,
the new distance can be computed by running the algorithm only one
more time, with the upper bound \(w' = d' + 1\).

**Adding Invariants**

There are of course many more string invariants that could be added to
OscaR/CBLS, as only a few general ones are implemented as part of this
thesis. However, two invariants that, if time permitted, would have been
implemented as part of this thesis are:

\(\text{RankDistance}(s_1, s_2)\) The rank distance is a distance metric with applica-
tions in biology \cite{14} that places somewhere between the Hamming
and Levenshtein distance in terms of both time complexity and the type of edits they take into account. There already exists work on solving the closest string problem with rank distance [13], so doing a comparison using this distance would be interesting. Furthermore, as shown in Section 5.2.4 using the Levensthein distance as the violation metric for the equality constraint can help guide the search, but the higher time complexity can also be an obstacle when doing so. It is possible that the rank distance, with its lower time complexity, is a good violation metric.

**HammingSubString** $(s_1, s_2)$ Currently, an invariant for computing the substring distance based on the Levenshtein distance is implemented, as this is easily implemented with minor modifications of the dynamic programming algorithm. However, some problems such as the closest substring problem (CSSP) [10] require a substring distance based on the Hamming distance. A good starting point for implementing this would be Boyer-Moore’s string search algorithm, which Galil improved to a linear worst-case running time [20]. Another interesting algorithm that is used in bioinformatics is the algorithm of [6], which uses finite-field fast Fourier transforms to approximate the minimum Hamming distance for any alignment of two strings in $O(|\Sigma| n \log(n))$ time, where $n$ is the length of the larger string and $|\Sigma|$ the size of the alphabet.

**Adding Constraints**

Currently, many constraints that are not implemented can be expressed with the constraints and invariants that are implemented, as explained in Section 4.4. However, the **Regular** $(s, \mathcal{L})$ constraint, which holds if word $s$ is a member of the regular language $\mathcal{L}$, cannot be expressed with the implemented constraints and invariants unless $\mathcal{L}$ can be unrolled into a finite set of finite strings, for instance when $\mathcal{L}$ is infinitely large. Since **Regular** is a very common constraint in string problems, this is probably the most important constraint to implement. The only reason that it was not implemented as part of this thesis is that it was deemed to be too time-consuming, and no good benchmark that contains the **Regular** constraint could be found.

**Constraint-Specific Neighbourhoods**

A constraint-specific neighbourhood is a neighbourhood that performs moves such that some constraint is always satisfied. For example, consider the closest-string variable in the CSP model of Listing 2. The length of the closest-string variable is implicitly constrained to be fixed. This implicit length constraint is enforced by the initial value of the closest-string variable
and the replace-character neighbourhood, which guarantees that no move can change its length.

More complex constraint-specific neighbourhoods for integer constraints, such as ALLDIFFERENT and CIRCUIT, are implemented as part of [5] and have proved to be useful, as they can remove the overhead of maintaining the constraint violation and they can provide more complex moves during search. Therefore, it could be useful to implement constraint-specific neighbourhoods for some of the more complicated string constraints. For example, it might be possible to implement the REGULAR\(s, \mathcal{L}\) constraint as a constraint-specific neighbourhood by generating moves based on \(\mathcal{L}\) and the current assignment of \(s\).

References


[38] Ying Wang, Wei Chen, Xu Li, and Bing Cheng. Degenerated primer design to amplify the heavy chain variable region from immunoglobulin cDNA. BMC bioinformatics, 7 Suppl 4(Suppl 4):S9–S9, 2006.