Model Checking for String Problems

Milka Hutagalung and Martin Lange*

School of Electr. Eng. and Computer Science, University of Kassel, Germany

Abstract. Model checking is a successful technique for automatic program verification. We show that it also has the power to yield competitive solutions for other problems. We consider three computation problems on strings and show how the polyadic modal μ-calculus can define their solutions. We use partial evaluation on a model checking algorithm in order to obtain an efficient algorithm for the longest common substring problem. It shows good performance in practice comparable to the well-known suffix tree algorithm. Moreover, it has the conceptual advantage that it can be interrupted at any time and still deliver long common substrings.

1 Introduction

Model checking is the process of automatically evaluating a logical formula on a given interpretation. This logical decision problem has proved to be extremely useful in the area of systems verification where dynamic systems are modelled as transition systems and formulas of temporal logics are being used to formalise behavioural properties [6, 20]. Model checking is used to answer the question of whether or not such systems are correct with respect to some specification, namely whether or not they possess the formalised properties. The logics that are typically used in program verification have been designed in order to express typical correctness properties of dynamic systems: LTL [19], CTL [7], CTL* [8], etc. The name "model checking" is derived from the process of checking whether some interpretation in the form of a mathematical structure has the property defined by the formula, i.e. is a model of the formula in logical terms.

The impact that model checking has had for program verification has led to a common understanding of model checking as a program verification method. Still, the applicability of model checking is not limited to that area. Model checking can in principle be used to solve all kinds of decision problems, provided that the used specification language is strong enough to express that problem in the usual sense of a word problem: given a representation of an instance x of a problem, decide whether or not x belongs to some set P. For instance, x could be a directed graph, and P may consist of all graphs having a Hamiltonian path. Take for example Monadic Second-Order Logic (MSO) interpreted over graphs.

* The European Research Council has provided financial support under the European Community's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement no 259267.

It is not too difficult to construct a formula φ_{ham} which is satisfied by a graph G iff it has a Hamiltonian path. In fact, Fagin's Theorem [10] and the fact that the Hamiltonian path problem is easily seen to belong to NP give such a formula straight-away. Thus, φ_{ham} *expresses* the Hamiltonian path problem, and any model checking algorithm for MSO on graphs can be used to *solve* the Hamiltonian path problem. Likewise, any problem that is definable in a logic with a decidable model checking problem can therefore be solved by model checking.

The logics used in program verification mentioned above are not capable of expressing more complex properties like the Hamiltonian path for instance, unless NLOGSPACE=NP. This is a simple consequence of the fact that the complexity of model checking a fixed formula – known as *data complexity* – in either of these logics is only NLOGSPACE whereas the Hamiltonian path problem is NP-hard.

MSO is not necessarily a good logic for model checking. In order to be useful, such logics must provide a good balance between expressive power on one hand and efficient decision procedures on the other. Clearly, these two goals may be in conflict. It is commonly understood that the use of fixpoint quantifiers provide such a good balance because fixpoints can be computed using iteration methods for instance, and often they increase expressive power.

Fixpoints are implicitly present in the temporal logics mentioned above. Another prominent logic in model checking for program verification is the modal μ-calculus $\mathcal{L}_\mu$ [15] which explicitly adds fixpoint quantifiers to a basic modal logic. Its data complexity is in P, and it is known that it can express more properties or problems than temporal logics like LTL, CTL, etc. Still, its expressive power is limited by other facts, for instance it can only express properties of single nodes in a directed graph. This weakness has been overcome with the introduction of the polyadic or higher-dimensional μ-calculus [1, 18] which behaves very much like the ordinary modal μ-calculus when it comes to model checking.

In this paper we show how model checking can be used in order to solve problems in an area that is quite different from program verification: string problems. We consider three of the most prominent examples of such problems: given a set of strings over some finite alphabet, find the *longest common substring*, the *longest common subsequence*, resp. the *shortest common superstring*. Such problems have important applications in bio-informatics as in sequence and genome analysis [12, 22], in linguistic information retrieval [24]; plagiarism detection, for instance in publications [11] or source code [17, 5]; data compression [21] and so on.

We use the polyadic μ-calculus in order to express these problems on simple graphs encoding string inputs and derive algorithms for these problems from a generic model checking algorithm for this logic. This realises more than a Turing or Cook reduction in two ways. First of all, model checking is a decision problem whereas the string problems mentioned above are computation problems. It is not obvious how model checking could be used to solve them. We do not implement standard tricks like binary search or others to find a solution. Instead, we use fixpoint quantifiers and iteration in the polyadic μ-calculus in

order to compute solutions of these strings problems. Second, we want to show that model checking can be used in order to derive competitive algorithms in areas other than program verification. A vital step towards this goal is *partial evaluation*. Model checking takes two inputs: a formula and an interpretation. Algorithms for particular decision problems can be derived from model checking algorithms by fixing the formula input to the one that expresses the problem. Partial evaluation is the process of optimising the generic algorithm to one that operates on a fixed formula and a variable (encoding of) its interpretation.

The fact that highly expressive modal fixpoint logics can be used in order to solve problems that are more complex than the reachability problems arising in automatic program verification has been observed before. For instance, [2] develops a model checking algorithm for an extension of the modal μ-calculus with first-order predicate transformers. It is shown that this can be used to solve problems of higher complexity like NFA universality, QBF, or the shortest common supersequence problem (SCS). The principals are applicable to the other mentioned string problems as well. However, that work only presents the principal applicability of model checking for SCS. The work presented here improves and extends this in the following ways: the use of first-order predicate transformers turned out to be an overkill; here we show that modal fixpoint logics without higher-order features suffice for these string problems. Moreover, while [2] only presents the principal definability of these problems, here we take the work to a further level and show how the generic model checking algorithms can be optimised in order to arrive at practical algorithms for string problems.

The rest of the paper is organised as follows. Sect. 2 recalls the polyadic μ-calculus including the question of how model checking can be done for this logic. Sect. 3 shows how the three aformentioned string problems can be represented as model checking problems in this logic. It also contains a discussion on how model checking logics with fixpoint quantifiers can be used to solve computation problems. In order to evaluate the viability of this approach we concentrate on one particular problem in Sect. 4 where we show how partial evaluation is being used to turn the model checking algorithm for the polyadic μ-calculus on a fixed formula into an algorithm for the longest common substring problem. Sect. 5 compares the obtained algorithm against existing approaches. Sect. 6 finishes the paper with some concluding remarks.

2 The Polyadic μ-Calculus

Syntax and Semantics. The polyadic μ-calculus is interpreted over *transition systems*. Let Σ and $\mathcal{P}$ be finite sets of labels. A transition system is a labeled directed graph $\mathcal{T} = (\mathcal{S}, \to, \lambda)$ with $\mathcal{S}$ being a set of nodes, $\to \subseteq \mathcal{S} \times \Sigma \times \mathcal{S}$ the edge relation, and $\lambda : \mathcal{S} \to 2^{\mathcal{P}}$ a function assigning a set of labels from $\mathcal{P}$ to every node. We write $s \xrightarrow{a} t$ for $(s, a, t) \in \to$.

Formulas of the polyadic μ-calculus $\mathcal{L}_\mu^\omega$ are given by

$$\varphi \ ::= \ p_i \mid \neg\varphi \mid \varphi \wedge \varphi \mid \langle a \rangle_i \varphi \mid X \mid \mu X.\varphi$$

where $p \in \mathcal{P}$, $a \in \Sigma$, $i \in \mathbb{N}$, and $X \in \mathcal{V}$ for some countably infinite set $\mathcal{V}$ of variables. We additionally assume that any free occurrence of a variable X is under the scope of an even number of negation symbols in $\mu X.\varphi$.

Apart from the usual Boolean operators that can be expressed using $\wedge$ and $\neg$ we introduce the following abbreviations: $[a]_i\varphi := \neg\langle a\rangle_i\neg\varphi$ and $\nu X.\varphi := \neg\mu X.\neg\varphi[\neg X/X]$.

The *dimension* of a formula is the number of different indices occurring in operators of the form p_i, $\langle a\rangle_i$ or $[a]_i$ in it. The fragment $\mathcal{L}_\mu^k$ consists of all formulas of dimension at most k.

A formula of dimension k is interpreted in a k-tuple of nodes in a transition system $\mathcal{T} = (\mathcal{S}, \to, \lambda)$. The indices an atomic propositions and modal operators refer to a particular dimension, i.e. p_i is to be read as "the i-th component (of the k-tuple) under consideration) satisfies p". Likewise, $\langle a\rangle_i\varphi$ formalises that the tuple can be changed in its i-th component to some successor state such that φ holds.

The semantics assigns to every formula of dimension k the set of all k-tuples that satisfy it as follows. In order to handle free variables we use a variable interpretation $\rho : \mathcal{V} \to 2^{\mathcal{S}^k}$ assigning to each variable a set of k-tuples of nodes. Then $\rho[X \mapsto S]$ denotes the function that maps X to the set S and agrees with ρ on all other arguments.

$$
\begin{aligned}
[\![p_i]\!]_\rho^{\mathcal{T}} &:= \{(s_1,\ldots,s_k) \mid p \in \lambda(s_i)\}\\
[\![\neg\varphi]\!]_\rho^{\mathcal{T}} &:= 2^{\mathcal{S}^k} \setminus [\![\varphi]\!]_\rho^{\mathcal{T}}\\
[\![\varphi \wedge \psi]\!]_\rho^{\mathcal{T}} &:= [\![\varphi]\!]_\rho^{\mathcal{T}} \cap [\![\psi]\!]_\rho^{\mathcal{T}}\\
[\![\langle a\rangle_i\varphi]\!]_\rho^{\mathcal{T}} &:= \{(s_1,\ldots,s_k) \mid \exists t \in \mathcal{S} \text{ s.t. } s_i \xrightarrow{a} t \text{ and}\\
&\qquad\qquad (s_1,\ldots,s_{i-1},t,s_{i+1},\ldots,s_k) \in [\![\varphi]\!]_\rho^{\mathcal{T}}\}\\
[\![X]\!]_\rho^{\mathcal{T}} &:= \rho(X)\\
[\![\mu X.\varphi]\!]_\rho^{\mathcal{T}} &:= \bigcap\{S \subseteq \mathcal{S}^k \mid [\![\varphi]\!]_{\rho[X\mapsto S]}^{\mathcal{T}} \subseteq S\}
\end{aligned}
$$

Thus, $\mu X.\varphi$ defines the least fixpoint of the function that takes a set of k-tuples of nodes S and returns the set of all k-tuples satisfying φ assuming that X is interpreted as S [14, 23]. We write $\boldsymbol{s} \models_\rho \varphi$ if $\boldsymbol{s} \in [\![\varphi]\!]_\rho^{\mathcal{T}}$ for a k-tuple $\boldsymbol{s}$ of nodes in $\mathcal{T}$, denoting the fact that $\boldsymbol{s}$ satisfies the property formalised by φ. If φ does not contain any free variables we may also drop the interpretation ρ.

A prominent example of a $\mathcal{L}_\mu^2$ formula is

$$\varphi_{\mathsf{bis}} := \nu X.(\bigwedge_{p\in\mathcal{P}} p_1 \leftrightarrow p_2) \wedge \bigwedge_{a\in\Sigma} [a]_1\langle a\rangle_2 X \wedge [a]_2\langle a\rangle_1 X \ .$$

It expresses bisimilarity in the sense that for all pairs of two nodes (s,t) we have $(s,t) \models \varphi_{\mathsf{bis}}$ iff s and t are bisimilar.

Model Checking $\mathcal{L}_\mu^k$. A simple model checking algorithm for $\mathcal{L}_\mu^\omega$ is implicitly given in the semantics of that logic. Given a finite transition system $\mathcal{T}$ and a

closed $\mathcal{L}_\mu^k$ formula φ, one can compute the set $[\![\varphi]\!]^{\mathcal{T}}$ by induction on the structure of φ. Fixpoint subformulas of the form $\mu X.\psi$ or $\nu X.\psi$ can be handled using Knaster-Tarski fixpoint iteration: for least fixpoint formulas one binds X to the empty set and computes the value of ψ on $\mathcal{T}$. Then X is bound to this set of k-tuples and so on until a fixpoint is reached. For greatest fixpoints one starts the iteration with $\mathcal{S}^k$ instead.

The model checking problem for the polyadic μ-calculus has been investigated before [1, 16]. Essentially, there is no conceptual difference to model checking the ordinary μ-calculus [9] which consists of all formulas of arity 1. In fact, there is a simple reduction from model checking formulas of arity k to formulas of arity 1 on the k-fold product of a transition system. Thus, one of the major parameters in its complexity – besides the formula's arity – is its *alternation depth*. Intuitively, it measures the nesting depth of fixpoints of different type. For formulas with no such nestings we set it to 1. Since alternating fixpoint quantifiers do not play any role in tackling string problems in the next section we omit a formal definition of alternation depth here and refer to the literature instead [4].

The next proposition summarises the findings on the complexity of model checking $\mathcal{L}_\mu^\omega$.

Proposition 1 ([1, 16]). *Given a transition system $\mathcal{T}$ with n nodes and a closed $\mathcal{L}_\mu^k$ formula φ of alternation depth d, the set of all k-tuples of nodes in $\mathcal{T}$ satisfying φ can be computed in time $\mathcal{O}((|\varphi| \cdot n^k)^{\lceil d/2 \rceil})$.*

3 Defining String Problems

The three problems under consideration – longest common substring, resp. sequence, and shortest common superstring – all get as input a set $W = \{w_1, \ldots, w_m\}$ of strings over some finite alphabet Σ. For ease of presentation we assume them all to have length n. The theory and procedures to follow are easily adapted to handle strings of varying lengths. First we consider straight-forward representations of such inputs by transition systems. Distinguishing the two cases of finding longest substructures or shortest superstructures turns out to be beneficial.

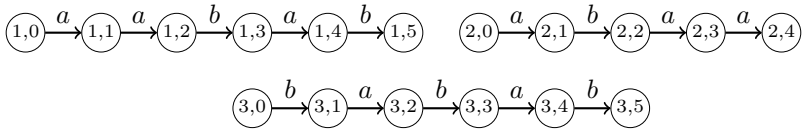
Longest Common Substring and -Sequence. For these two problems we represent the input strings $W = \{w_1, \ldots, w_m\}$ by a transition system containing a single path for each such string. We also use the symbols $1, \ldots, m$ as propositions on the nodes in order to assess which string a node is in. Let $w_i = a_{i,1} \ldots a_{i,n}$ for $i = 1, \ldots, m$. Then $\mathcal{T}_W$ is given as

Now consider the $\mathcal{L}_\mu^m$-formula $\varphi_{\mathsf{lcst}} := \nu X.(\bigwedge_{i=1}^m i_i) \wedge \bigvee_{a\in\Sigma}\langle a\rangle_1 \ldots \langle a\rangle_m X$ interpreted over transition systems of the form $\mathcal{T}_W$. In order to explain its meaning we consider how the naïve fixpoint iteration algorithm computes $[\![\varphi_{\mathsf{lcst}}]\!]^{\mathcal{T}_W}$. First we observe that $\bigwedge_{i=1}^m i_i$ satisfies exactly those m-tuples for which each i-th component belongs to the i-th input string. Thus, fixpoint iteration only yields tuples of positions with exactly one for each input string. Let us call these normal.

The greatest fixpoint iteration starts with the set of all tuples, and we can restrict our attention to all normal tuples only. This set can be seen as a representation of all the position at which the string ε occurs. The next fixpoint iteration forms the union of all sets of normal tuples which represent positions such that some a-edge is possible from all of them, and the resulting tuple represents occurrences of the substring a. Thus, it computes all positions of common substrings of length 1. In general, the j-th fixpoint iteration computes all positions (as normal m-tuples) of a common substring of length j. Clearly, this process is monotonically decreasing and there is some j – at most $n+1$ – such that the j-th iteration returns the empty set.

Indeed we have $\mathcal{T}_W \not\models \varphi_{\mathsf{lcst}}$ for any set W of strings. Nevertheless, model checking via fixpoint iteration computes all common substrings of W before finding out that the formula is not satisfied. This is the basis for an algorithm computing the longest common substring using model checking as described in detail in the next section.

Example 1. Consider the input $W = \{aabab, abaa, babab\}$, represented by the following transition system. For convenience, we have given the nodes names. We also omit the nodes' labels since they are just the same as the names' first components.



A greatest fixpoint iteration for φ_{lcst} on $\mathcal{T}_W$ starts with X^0 as the set of all positions. In order to compute the next iteration, note that $[\![(\bigwedge_{i=1}^3 i_i)]\!]^{\mathcal{T}_W}$ is the set of all tuples of the form $((1,j_1),(2,j_2),(3,j_3))$ for appropriate j_1, j_2, j_3. Every further iteration intersects some set obtained by evaluating the modal terms with this set. We therefore disregard all other tuples. Under this assumption, $[\![\bigvee_{c\in\{a,b\}}\langle c\rangle_1\langle c\rangle_2\langle c\rangle_3 X]\!]^{\mathcal{T}_W}_{[X\mapsto X^0]}$ then evaluates to

$$\begin{aligned} X^1 \;:=\; & \{(1,0),(1,1),(1,3)\}\times\{(2,0),(2,2),(2,3)\}\times\{(3,1),(3,3)\} \\ \cup\; & \{(1,2),(1,4)\}\times\{(2,1)\}\times\{(3,0),(3,2),(3,4)\} \end{aligned}$$

which is exactly the set of node tuples from which all components can do an a-edge or all components can do a b-edge.

The next iteration for the evaluation of the greatest fixpoint is obtained by evaluating the fixpoint body again, this time under the variable interpretation

$[X \mapsto X^1]$, and it yields

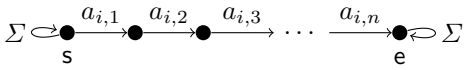
$$\begin{aligned} X^2 \;:=\; & \{(1,1),(1,3)\} \times \{(2,0)\} \times \{(3,1),(3,3)\} \\ \cup\; & \{(1,2)\} \times \{(2,1)\} \times \{(3,0),(3,2)\} \end{aligned}$$

which is the set of positions of *ab*, resp. *ba*, in the input strings. Note that *aa* for instance is no common substring, and this is reflected by the fact that there is no p such that $((1,0),(2,2),p)$ belongs to X^2.

The next iteration yields $X^3 := \{((1,1),(2,0),(3,1))\}$ which denotes the positions of the common substring *aba*. Finally, we get $X^4 = \emptyset$ at which point the fixpoint is reached, and the solution to this longest common substring instance is obtained as the value of the last iteration beforehand, namely *aba* at positions 1, 0, and 1 in the three input strings.

$\mathcal{L}_\mu^2$ is also capable of expressing the longest common subsequence problem in the same sense. Let $\langle * \rangle_i \psi$ abbreviate $\mu Y.\psi \vee \bigvee_{a \in \Sigma} \langle a \rangle_i Y$. Informally, it denotes the set of all tuples such that the i-th component can make an arbitrary number of steps along any edge and some resulting tuple satisfies ψ. Now consider $\varphi_{\mathsf{lcsq}} := \nu X.(\bigwedge_{i=1}^m i_i) \wedge \bigvee_{a \in \Sigma} \langle a \rangle_1 \langle * \rangle_1 \ldots \langle a \rangle_m \langle * \rangle_m X$. Evaluating this formula on a transition system of the form $\mathcal{T}_W$ will compute the longest common subsequence of the input strings in W in the same way as above. Note that, again, we have $\mathcal{T}_W \not\models \varphi_{\mathsf{lcsq}}$ for any W but all the solutions to this instance are being found in the last iteration of the greatest fixpoint evaluation.

Shortest Common Superstring. In order to model the shortest common superstring problem we change the representation of words by simple paths in a transition system. Given a finite set W of words over Σ, the transition system $\mathcal{T}'_W$ consists of one component for each word $w_i = a_{i,1} \ldots a_{i,n}$ which has the form



Two additional atomic propositions s, e are being used to mark the start node and the end node of each component. They, together with the special structure of these transition systems, can also be used to enforce tuples to contain exactly one node from each input string. Thus, propositions $1, \ldots, m$ are not needed anymore.

Now consider $\varphi_{\mathsf{scs}} := (\bigwedge_{i=1}^m \mathsf{s}_i) \;\wedge\; \mu X.(\bigwedge_{i=1}^m \mathsf{e}_i) \vee \bigvee_{a \in \Sigma} \langle a \rangle_1 \ldots \langle a \rangle_m X$. Intuitively, it denotes the set of all tuples such that

1. each component is labeled with s, and
2. there is a sequence of edge labels w such that each component has a path with these labels and the nodes of the tuple at the end of all these paths are all labeled with e.

It should be clear from this description that we have $\mathcal{T}'_W \models \varphi_{\mathsf{scs}}$ for any W. However, evaluating φ_{scs} on $\mathcal{T}'_W$ by a least fixpoint iteration will ultimately

construct a shortest common superstring for all the strings in W. This iteration starts with $X^0 := \emptyset$ and – when restricted to tuples with one component in each string – gradually finds tuples $(p_1, \ldots, p_m)$ of positions in the i-th iteration such that there is a word w of length i with a path labeled with w from each of their components. By the structure of $\mathcal{T}'_W$, the iteration grows monotonically "to the left", i.e. it only ever adds tuples with positions further left in the input words. Eventually – after no more than $(n+1)m$ iterations in the worst case – the tuple $((1,0), \ldots, (m,0))$ is being found and the least fixpoint is being reached. The number of iterations done to achieve this equals the length of a shortest common superstring, and this string can easily be computed by annotating the found tuples of positions successively.

4 Partial Evaluation and Optimisation

The algorithms sketched above are rather naïve and do not exploit any optimisation potential at all. The descriptions above are only meant to show how model checking with fixpoint logics can in principle be used in order to solve such computation problems. Here we focus on one particular problem, namely finding longest common substrings, and show how partial evaluation of model checking algorithms can be used to obtain an efficient procedure. Also note that a naïve estimation of the worst-case time complexity of these algorithms according to Prop. 1 yields a horrendous overapproximation: in general, model checking $\mathcal{L}^k_\mu$ is exponential in the arity k, here equalling the number of strings m in the input. This, however, ignores the special structure of the transition systems used here and that of the fixed formula.

Consider the algorithm that has been described in Example 1. It basically works on a set X of common substrings, and in each iteration it extends all elements of X to a longer common substring by considering one more letter to the left. For m input strings of length n, the set X is represented by a set of m-tuples, which initially contains n^m tuples that represent the positions of the empty string.

A straightforward optimisation changes the representation of the set X. Instead of using a set of m-tuples, we can represent a single substring w with a set $t(w)$ of pairs such that $(i,j) \in t(w)$ iff w occurs in w_i at position j. By using this representation, initially we have nm positions instead of n^m positions for the empty string. Moreover, it is easy to check whether w is a common substring which is true iff for every $i = 1, \ldots, m$ there is some j with $(i,j) \in t(w)$.

Applying these straight-forward optimisations to the procedure described in Ex. 1 yields Algorithm 1. It collects all non-extendable common substrings in a set Y, and uses that for a return value.

In the following we describe further optimisations for Algorithm 1, so that it can find longest common substrings faster and more efficiently.

Extension restriction. To extend $w \in X$ in each iteration, it is not necessary to consider all letters from Σ. Suppose $w = w'a$, where $a \in \Sigma$ and $w' \in \Sigma^*$,

Algorithm 1 Finding the longest common substring

```
1: X ← Σ, Y ← ∅
2: while X ≠ ∅ do                                   ▷ extend some w ∈ X
3:     take w ∈ X
4:     if Ext(w) ≠ ∅ then          ▷ Ext(w) := {aw | a ∈ Σ, aw common substring }
5:         X ← Ext(w) ∪ X\w                  ▷ replace w with its extension Ext(w)
6:     else
7:         Y ← {w} ∪ Y                ▷ have w as non-extendable common substring
8:     end if
9: end while
10: return the longest w ∈ Y
```

then to extend w it is enough to consider letters that have successfully extended w', since for every $a' \in \Sigma$ if $a'w'$ is not a common substring, then $a'w$ is not either. We can get the letters that extend w' in constant time by always keeping a pointer from w to w' for each $w \in X$, and from w to all of its extensions for each $w \in X$ that have been extended. Moreover, we should always take the shortest $w \in X$ in each iteration, to make sure that the extension of w' is already computed in the previous iteration.

Multiple substrings extension. Under some conditions, extending a single $w \in X$ may imply extensions of some other substrings $u \in X$. For any $w \in X$ let $S(w) = \{u \in X \mid u = wv, v \in \Sigma^+\}$. If w is extendable to aw, in general we cannot conclude that $u \in S(w)$ is also extendable to au. However it is the case if $t(aw)$ is equal to $\{(i, j-1) | (i, j) \in t(w)\}$, since this means that all occurrences of w in the input strings are always preceded by a. In this case, we can extend w to aw, and also every $u \in S(w)$ to au. Likewise if w is not extendable to any longer common substring, then every $u \in S(w)$ is also not extendable. In this case we can move w and all $u \in S(w)$ to Y. Extending all $u \in S(w)$ (resp. moving all $u \in S(w)$ to Y) can be done in constant time by exploiting the pointers defined before, i.e. a pointer from $w = w'a$ to w' since every u are successively linked by the pointer to w.

Multiple letters extension. It is also possible to extend $w \in X$ with a sequence of letters $a_n a_{n-1} \dots a_1 \in \Sigma^n$ instead of only one single letter. Suppose $w = w'a$, and the string w' was extended to a common substring $a_n a_{n-1} \dots a_1 w'$ because of the previous extension policy, i.e. because $t(a_1 w') = \{(i, j-1) | (i, j) \in t(w')\}$, $\dots$, $t(a_n \dots a_1 w') = \{(i, j-1) | (i, j) \in t(a_{n-1} \dots a_1 w')\}$. Then if we can extend w to $a_1 w$, we can immediately conclude that w can be extended to $a_n a_{n-1} \dots a_1 w$.

All of these optimisations will not make the extension of a single substring $w \in X$ harder since we store more information on each common substring, to accommodate the optimisations. The extension policies derived from these optimisations can cut down the number of iterations needed in Algorithm 1.

Example 2. Let $W = \{$`cgtacgag, aacgtag, agcgtacg`$\}$ be the input strings. We illustrate the computation of the longest common substring using Algo-

i	X	Y
1	g,t,c,a	-
2	ag,cg,t,c,a	-
3	ag,cg,gt,c,a	-
4	ag,cg,gt,ac,a	-
5	ag,cg,gt,ac,ta	-
6	cg,gt,ac,ta	ag
7	acg,gt,ac,ta	ag
8	acg,cgt,ac,ta	ag
9	acg,cgt,ta	ag,ac
⋮	⋮	⋮

(a) without optimisation

i	X	Y
1	g,t,c,a	-
2	ag,cg,t,c,a	-
3	ag,cg,gt,c,a	-
4	ag,cg,gt,ac,a	-
5	ag,cg,gt,ac,gta	-
6	cg,gt,ac,gta	ag
7	acg,gt,ac,gta	ag
8	acg,cgt,ac,cgta	ag
9	acg,cgt,cgta	ag,ac
10	cgt,cgta	ag,ac,acg
11	-	ag,ac,acg,cgt,cgta

(b) with optimisation

Fig. 1: Computation for $W = \{\texttt{cgtacgag, aacgtag, agcgtacg}\}$

rithm 1 in Fig. 1, both with and without optimisation. In each iteration we pick the shortest common substring to be extended. Figure (a) shows the first 9 iterations (of 14 altogether) without any optimisation. Figure (b) shows the computation with optimisation which finds the longest common substring `cgta` after 11 iterations.

Note that in Fig.1 (b), we apply the optimisation in the 4th, 7th, and 10th iteration. In the 4th iteration it is found that `a` can be extended to `ta` and we have $t(\texttt{gt}) = \{(i, j-1) \mid (i,j) \in t(\texttt{t})\}$ from the previous iteration, so `a` can be extended directly to `gta`. In the 7th iteration, by extending `gt` to `cgt` we also extend `gta` to `cgta` since $\texttt{gta} \in S(\texttt{gt})$. In the 10th iteration `cgt` is not extendable so we conclude that `cgta` are not either.

Theorem 1. *For input strings $w_1, \ldots, w_m$ each of length n, the number of iterations needed by the optimised algorithm is at most $n + n + m(n-1)$.*

Proof. In each iteration i, we pick the shortest common substring $w \in X^i$ to be extended, which satisfies one of these properties, either:

1. w cannot be extended to the left anymore, or
2. w can be extended to aw and $t(aw) = \{(i, j-1) | (i,j) \in t(w)\}$, i.e. w allows multiple substring extension as described previously, or
3. none of these two conditions apply to w.

Let L_1, L_2, L_3 be the set of common substring of $w_1, \ldots, w_n$, such that $w \in L_i$ iff w satisfies the i-th property.

$|L_1| \leq n$, since we have a one-to-one mapping from L_1 to the set of prefixes of w_1. Note that each $v \in L_1$ is a substring of w_1 (resp. $w_2, \ldots, w_n$), and it can be mapped to a prefix uv of w_1. Every two different $v_1, v_2 \in L_1$ are mapped to two different prefixes of w_1, for otherwise one of them would be a suffix of the other, and thus could be extended to the left which would contradict $v_1, v_2 \in L_1$.

$|L_3| \leq m(n-1)$, since if $v \in L_3$ then v occurs on all input strings, and there exists an input string w_i such that v occurs more than once on w_i. If $|L_3| = k$, then $|w_1| + \ldots + |w_m|$ is at least $m + k$. However, the added lengths of all input strings is bounded by mn, so $|L_3| \leq mn - m$.

In general, $|L_2| \leq n^2$. But consider $L_2' \subseteq L_2$ where $v \in L_2'$ iff its longest proper prefix v' does not belong to L_2. Suppose on the i-th iteration, we pick $v \in L_2$ to be extended. If $v \notin L_2'$ then the longest proper prefix v' of v is not yet extended on any j-th iteration, $j < i$. Otherwise either the optimisation rule: multiple substring extension or multiple letters extension have been applied to obtain the extension av of v, and implies $v \notin X^i$ for every $i > j$. So $v \notin L_2'$ contradicts to v being the current shortest common substring found.

$|L_2'| \leq n$, because we have a one-to-one mapping from L_2' to the set of suffixes of w_1. Each $v \in L_2'$ can be mapped to a suffix vu of w_1. Every two different $v_1, v_2 \in L_2'$ are mapped into two different suffixes of w_1, for otherwise one of them would be a prefix of the other. Suppose w.l.o.g. that v_1 was a prefix of v_2. Then v_1 is also a prefix of the longest proper prefix v_2' of v_2. Hence $v_2' \in L_2$ which contradicts that $v_2 \in L_2'$. □

5 A Comparison Against the Suffix-Tree Approach

The literature describes two algorithms for solving the longest common substring problem: *dynamic programming* [13] and the *suffix tree algorithm* [12]. However, it is known that the speed and versatility of the suffix tree algorithm is better than dynamic programming. It is therefore the state-of-the-art and standard algorithm used for the longest common substring problem.

A suffix tree of W is a tree storing all suffixes of strings in W. It has many applications in biological sequence data analysis [3], especially for searching patterns in DNA or protein sequences. For a more detailed explanation of suffix trees see [12]. We compare the optimised Algorithm 1 with the suffix tree algorithm empirically on a biological data set, and also conceptually.

Empirical Comparison. An interesting application for the longest common string problem comes from bioinformatics area for identification of common features in genome analysis [12]. We compare the performance of the optimised Algorithm 1 and the suffix tree algorithm on a data set of complete genomes. We consider the 11 species that are named in Fig. 2. All of these complete genomes can be obtained on a public GenBank database of NCBI[1].

We choose mostly virus and bacteria genomes since they usually have only one or two chromosomes, and the size of their complete genome is approximately 3 megabytes, which is still suitable for the experiment. For more complex species such as humans the size of their complete genome is approximately 800 megabytes[1]. The bacteria *Vibrio cholerae* and *Agrobacterium tumefaciens* have

[1] see http://www.ncbi.nlm.nih.gov/genbank

Species	Size	Suffix Tree Alg.	Opt. Alg. 1
E. coli, M. tuberculosis	9.8 mb	70 min 33 sec	54 min 26 sec
A. tumefaciens (I), V. cholerae (I)	7.1 mb	42 min 46 sec	38 min 17 sec
S. enterica, B. subtilis (I)	9.2 mb	64 min 2 sec	52 min 13 sec
A. fugildus, N. gonorrhoeae, A. tumefaciens (II)	7.1 mb	35 min 10 sec	33 min 50 sec
C. trachomatis, A. aeolicus, H. influenzae, V. cholerae (II)	5.7 mb	21 min 33 sec	24 min 34 sec
E. coli, V. cholerae (I)	8.2 mb	60 min 53 sec	46 min 21 sec
M. tuberculosis, V. cholerae (I)	7.5 mb	43 min 36 sec	40 min 4 sec

Fig. 2: Comparison with suffix tree algorithm

two chromosomes, and we treat each chromosome separately. We take two to four species for each experiment and try to find their longest common substring.

Fig. 2 compares the running time of the optimised Algorithm 1 and the suffix tree algorithm on some benchmarks. Both algorithms were implemented in OCaml. The suffix tree algorithm uses Sébastien Ferré's implementation of the suffix tree data structure[2]. The experiments have been run on a machine with 16 Intel Xeon cores running at 1.87GHz and 256GB of memory.

The result suggests that the optimised Algorithm 1 is comparable to the suffix tree algorithm. The time needed is often even less than the suffix tree algorithm, except on the data set with 5.7 mb, where the optimised Algorithm 1 was 3min slower than the suffix tree algorithm. However, in general we can conclude that the optimised Algorithm 1 performed well compared to thesuffix tree algorithm.

Conceptual Comparison. The suffix tree algorithm builds the tree first and then searches for the deepest node that represents the longest common substring of all input strings. The usual approaches to building the tree are incremental with respect to the number of the input strings [12]. For example, to build a suffix tree of $W = \{w_1, \ldots, w_n\}$, one starts with a suffix tree of w_1, then gradually modifies the tree to include the suffixes of w_2, and so on. This has the disadvantage of not being able to see the common substrings of all $w_1, \ldots, w_m$ during the tree construction. The whole tree has to be constructed first before searching for any common substring of $w_1, \ldots, w_m$. What can be recorded during the tree construction is only the common substring for the first m-input strings.

Now suppose that the input data is large such that despite the linear time complexity an entire run of the algorithm would still take, say, days to terminate. In such a case it would be great if the algorithm was able to report the finding of long common substrings on-the-fly, i.e. incrementally produce longer and longer common substrings. The suffix tree algorithm is not able to do this, because it needs to process all input strings before finding even the shortest common substring. However it is not the case for Algorithm 1. We have seen that the algorithm always maintains the currently longest common substring

[2] see http://www.irisa.fr/LIS/ferre/software.en.html

found in each iteration, and it is able to incrementally report longer and longer common substrings rather than finding them only at the very end of the entire computation.

6 Conclusion and Further Work

We have shown that certain string problems can be expressed as model checking problems for modal fixpoint logics. Fixpoint computation can be used to find optimal solutions for these problems. We have assumed straight-forward encoding of input strings as transition systems of disjoint paths. However, the formulas of the poyadic μ-calculus that were used to define these string problems also work on more compact graph encodings when common parts of input strings are being shared.

We have focused on the longest common substring problem and shown that it is possible to derive a new competitive algorithm by partial evaluation of a generic model checking algorithm for the polyadic μ-calculus. It turned out to have the conceptual advantage of being interruptable: roughly speaking, at half of the running time it has computed the half-longest common substrings of all inputs. The suffix tree algorithm, as a standard for this problem, on the other hand has computed, at half of the running time, the longest common substrings of half of the input strings.

Further work on the longest common substring algorithms includes a broader practical evaluation and a thorough study of possibilities to combine features from the suffix tree algorithm and the model checking approach. It also remains to execute the partial evaluation work for the two other string problems considered here, and possibly others as well in order to create hopefully competitive and usable algorithms for these problems, too.

Finally, we believe that model checking technology can contribute to algorithmic solutions for all sorts of other problems as well. This, of course, has to be studied for every possible decision or computation problem of interest separately.

References

1. H. R. Andersen. A polyadic modal μ-calculus. Technical Report ID-TR: 1994-195, Dept. of Computer Science, Technical University of Denmark, Copenhagen, 1994.
2. R. Axelsson and M. Lange. Model checking the first-order fragment of higher-order fixpoint logic. In *Proc. 14th Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR'07*, volume 4790 of *LNCS*, pages 62–76. Springer, 2007.
3. P. Bieganski, J. Riedl, J.V. Cartis, and E.F. Retzel. Generalized suffix trees for biological sequence data: applications and implementation. In *System Sciences, 1994. Proceedings of the Twenty-Seventh Hawaii International Conference on*, volume 5, pages 35 –44, jan. 1994.
4. J. Bradfield and C. Stirling. Modal mu-calculi. In P. Blackburn, J. van Benthem, and F. Wolter, editors, *Handbook of Modal Logic: Studies in Logic and Practical Reasoning Volume 3*, pages 721–756. Elsevier, 2007.

5. R. A. C. Campos and F. J. Z. Martínez. Batch source-code plagiarism detection using an algorithm for the bounded longest common subsequence problem. In *Proc. 9th Int. IEEE Conf. on Electrical Engineering, Computing Science and Automatic Control, CCE'12*, pages 1–4. IEEE, 2012.
6. E. A. Emerson and E. M. Clarke. Characterizing correctness properties of parallel programs as fixpoints. In *Proc. 7th Int. Coll. on Automata, Languages and Programming, ICALP'80*, volume 85 of *LNCS*, pages 169–181. Springer, 1981.
7. E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.
8. E. A. Emerson and J. Y. Halpern. "Sometimes" and "not never" revisited: On branching versus linear time temporal logic. *Journal of the ACM*, 33(1):151–178, 1986.
9. E. A. Emerson, C. S. Jutla, and A. P. Sistla. On model checking for the μ-calculus and its fragments. *TCS*, 258(1–3):491–522, 2001.
10. R. Fagin. Generalized first-order spectra and polynomial-time recognizable sets. *Complexity and Computation*, 7:43–73, 1974.
11. B. Gipp and N. Meuschke. Citation pattern matching algorithms for citation-based plagiarism detection: greedy citation tiling, citation chunking and longest common citation sequence. In *Proc. 2011 ACM Symp. on Document Engineering*, pages 249–258. ACM, 2011.
12. D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University, 1997.
13. D. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18(6):341–343, 1975.
14. B. Knaster. Un théorèm sur les fonctions d'ensembles. *Annals Soc. Pol. Math*, 6:133–134, 1928.
15. D. Kozen. Results on the propositional μ-calculus. *TCS*, 27:333–354, December 1983.
16. M. Lange and E. Lozes. Model checking the higher-dimensional modal μ-calculus. In *Proc. 8th Workshop on Fixpoints in Computer Science, FICS'12*, volume 77 of *Electr. Proc. in Theor. Comp. Sc.*, pages 39–46, 2012.
17. J. Oetsch, J. Pührer, M. Schwengerer, and H. Tompits. The system Kato: Detecting cases of plagiarism for answer-set programs. *Theory and Practice of Logic Programming*, 10(4–6):759–775, 2010.
18. M. Otto. Bisimulation-invariant PTIME and higher-dimensional μ-calculus. *Theor. Comput. Sci.*, 224(1–2):237–265, 1999.
19. A. Pnueli. The temporal logic of programs. In *Proc. 18th Symp. on Foundations of Computer Science, FOCS'77*, pages 46–57, Providence, RI, USA, 1977. IEEE.
20. J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. 5th Symp. on Programming*, volume 137 of *LNCS*, pages 337–371. Springer, 1982.
21. J. A. Storer. *Data Compression: Methods and Theory.* Comp. Sci. Press, 1988.
22. W.-K. Sung. *Algorithms in Bioinformatics: A Practical Approach.* CRC Press, 2009.
23. A. Tarski. A lattice-theoretical fixpoint theorem and its application. *Pacific Journal of Mathematics*, 5:285–309, 1955.
24. Y. Xiao, R.W.P. Luk, K.F.Wong, and K.L. Kwok. Using longest common subsequence matching for chinese information retrieval. *Journal of Chinese Language and Computing*, 15(1):45–51, 2010.