

Constraint Reasoning over Strings

Keith Golden¹ and Wanlin Pang²

¹ Computational Science Division, NASA Ames Research Center, Moffett Field, CA 94035

² QSS Group Inc., NASA Ames Research Center, Moffett Field, CA 94035

Abstract. This paper discusses an approach to representing and reasoning about constraints over strings. We discuss how string domains can often be concisely represented using regular languages, and how constraints over strings, and domain operations on sets of strings, can be carried out using this representation.

1 Introduction

Constraint satisfaction problems (CSPs) involve finding values for variables subject to constraints that permit or exclude certain combinations of values. Since many tasks in computer science [13,5,22] and many real-world problems [23,14,16,20] can be formulated as CSPs, they have been attracting widespread research and commercial interests for the last two decades. Whereas much work has been done on constraints over finite discrete domains and numerical intervals, there has been little work on constraint reasoning over strings.

Strings appear everywhere, from databases to DNA, and the relationships between the strings and the real-world objects they represent can be formalized as constraints. For example, we are applying constraint-based planning to provide automation in software domains [9,8], domains in which the actions are operations in a software environment, such as moving files, searching for information on the Internet or image processing. One characteristic of nearly all software domains is the ubiquity of strings and constraints. File path names, URLs and the contents of text files and web pages are all represented as text, which often obey specific constraints. For instance, many programs have inputs or outputs in the form of files, whose names follow some canonical form:

- A Java compiler expects the pathname for the source code of “my.package.MyClass” to be “my/package/MyClass.java,” and it produces a file “my/package/MyClass.class.”
- The pathname of data down-linked from a spacecraft or planetary rover is often in a form like “phase2/sol29/my_instrument/seq0002.jpg,” where each component of the pathname refers to some meaningful aspect of the data.

A distinguishing characteristic of software domains and other domains involving strings is that the set of possible strings corresponding to a given name, input or file is either infinite or so large that listing them all would require unacceptable amounts of time and storage. The challenge of effectively representing and reasoning about constraints on strings is to represent infinite string sets without actually requiring infinite space and to enforce constraints over infinite string sets without exhaustively listing the consistent values. In this paper, we provide such a string representation, based on regular

languages, we discuss how common string constraints are defined and handled using this representation, and we show how string constraint problems can be solved.

The remainder of the paper is organized as follows. In Section 2, we review notations of constraint satisfaction problems. In Section 3, we discuss our string domain representation, namely, regular languages. In Section 4, we provide definitions of useful string constraints and describe how they are enforced using this domain representation. In Section 5, we discuss how standard domain operations, such as intersection and equality testing, are handled. In Section 6, we show how the string constraints can be applied to solving some interesting problems. And finally, in Section 7 we conclude by summarizing our contribution.

2 Constraint Satisfaction Problems

A **Constraint Satisfaction Problem (CSP)** is a representation and reasoning framework consisting of variables, domains, and constraints. Formally, it can be defined as a triple $\langle X, D, C \rangle$ where $X = \{x_1, x_2, \dots, x_n\}$ is a finite set of variables, $D = \{d(x_1), d(x_2), \dots, d(x_n)\}$ is a set of domains containing values the variables may take, and $C = \{C_1, C_2, \dots, C_m\}$ is a set of constraints. Each constraint C_i is defined as a relation R on a subset of variables $V = \{x_i, x_j, \dots, x_k\}$, called the constraint scope. R may be represented extensionally as a subset of the Cartesian product $d(x_i) \times d(x_j) \times \dots \times d(x_k)$. A constraint $C_i = (V_i, R_i)$ limits the values the variables in V can take simultaneously to those assignments that satisfy R . Let $V_K = \{x_{k_1}, \dots, x_{k_l}\}$ be a subset of X . An l -tuple $(x_{k_1}, \dots, x_{k_l})$ from $d(x_{k_1}) \times \dots \times d(x_{k_l})$ is called an *instantiation* of variables in V_K . An instantiation is said to be *consistent* if it satisfies all the constraints restricted in V_K . A consistent instantiation of all variables in X is a *solution*. The central reasoning task (or the task of solving a CSP) is to find one or more solutions.

A CSP can be solved by search using, e.g., standard backtracking algorithms [3,10]. However, for CSPs with infinite domains such as those of interest in this paper, it is not guaranteed that a solution can be found by search alone, because it is infeasible to enumerate all values of infinite variable domains. Instead, the CSPs with infinite domains need to be relaxed by consistency enforcement before or during the search. Enforcing local consistency eliminates inconsistent values from variable domains [15,2]. In theory, if a given CSP has only one solution, enforcing a certain level of consistency will eventually make every variable domain a singleton domain; if the CSP has more than one solution, or infinitely many solutions, every remaining value in the domain after consistency enforcement will be part of a solution. In practice, an effective constraint solving strategy enforces a certain level of consistency such as generalized arc consistency [17,18] at each node of the search tree. A key issue is the trade-off between time spent on propagation and the reduction in the search space needed to allow feasible and efficient search. Based on our experience dealing with constraint-based planning in software environments, much depends on how the variable domains are represented and how the constraints are evaluated or executed to enforce consistency. In the next three sections, we focus on our string domain representation and a definition of constraints over string domains. These string constraints are in the constraint library of the

constraint reasoning system we implemented and, together with other numerical and boolean constraints, are used to model planning problems.

3 String Domains

We use the same CSP representation both to represent the constraint problem and to search for a solution; the domain $d(x)$ of variable x , representing the set of values that x can take, will, in general, change during the course of search and constraint propagation. Typically, a variable's domain is represented as a list of values. For numeric domains, we can instead represent a domain as an interval, yielding substantial decreases in space and time requirements and making it possible to represent an infinite set of values [11].

In the domains of interest, we frequently want to represent infinite, or very large, sets of strings, such as all possible pathnames matching a given pattern. Representing this set as a list is clearly infeasible, since it is infinite. Intervals are equally inappropriate. While it is possible to represent some sets of strings as intervals, such as all names between "Jones" and "Smith" in the phone book, such intervals are far less useful in practice than are numeric intervals.

However, there is an alternative representation of sets of strings that is far more useful, as evidenced by its ubiquity: regular languages. Regular languages are sets of strings that are accepted by regular expressions or finite automata, which are widely used in string matching, lexical analysis and many other applications. Although there are many languages that are not regular, such as palindromes, regular languages provide a nice tradeoff between expressiveness and tractability. As we will discuss, not only can we enforce generalized arc consistency (GAC) [2] for a wide range of useful string constraints when the domains are represented as regular languages, but we can perform the domain operations necessary for constraint propagation and search.

Regular languages are a much more flexible representation than intervals, in that the set of regular languages is closed under intersection, union and negation, whereas the set of intervals is only closed under intersection. Note that moving to an even more expressive representation would not be an improvement. Neither context-free languages (CFLs) nor deterministic CFLs are closed under intersection, and determining whether a context-sensitive (or more expressive) language is empty is undecidable [12, p 281].

We use two different representations of regular languages: regular expressions and finite automata (FAs). Regular expressions are used as input and are converted to FAs, which are used computationally. A regular expression represents a regular language over an alphabet Σ . In our implementation, Σ is the set of Unicode characters. We use the notation described in Table 1 to describe regular expressions.

The purpose of the notation $\backslash c$ is to "quote" a symbol c that would otherwise be interpreted as a syntax character. For example, $\backslash[$ can be used to refer to the character "[" and $\backslash\backslash$ refers to the character "\\".

We represent regular languages internally using FAs, since they are easier to compute with than regular expressions. An FA is a pair $\langle \mathcal{S}, \mathcal{T} \rangle$, where \mathcal{S} is a set of states and \mathcal{T} is a set of labeled transitions between the states. Each transition in \mathcal{T} is a triple $\langle s_1, l, s_2 \rangle$, which we will write $\langle s_1 \xrightarrow{l} s_2 \rangle$, where s_1 is the starting state of the transition, s_2 is the ending state and $l \in \Sigma$ is the transition label. The input to the FA is

Expression	Accept
$[abc]$	one of the characters a, b, c
$[a - c]$	one of the characters in the range $a - c$
$\sim[abc]$	any character in Σ except a, b, c
$.$	any character in Σ
$\setminus c$	the literal character c
re_1re_2	re_1 followed by re_2
$re_1 re_2$	either re_1 or re_2
re^*	zero or more repetition of re
re^+	one or more repetitions of re
$re?$	zero or one occurrences of re
(re)	re (used to override precedence)

Table 1. Regular expression syntax

a sequence of symbols from Σ . Whenever there are symbols left to read, the FA reads the next symbol, c , and follows a transition from the current state whose label is c . If there are multiple transitions labeled c , one is chosen nondeterministically. If there are no transitions labeled c , the FA halts and returns failure. For efficiency, we allow transitions to have sets of labels, represented using the same notation as shown in the first five rows of Table 1 (i.e., one-character regular expressions). For example, we could have a transition $\langle s_1 \xrightarrow{[a-zA-Z]} s_2 \rangle$, meaning the transition will be taken if the symbol is any character from the English alphabet. This is logically equivalent to having a separate transition for each symbol. For notational convenience, we also refer to transitions labeled with ϵ . An ϵ -transition is always applicable and is followed without reading any characters. An FA has a single *start state*, which is always the first state, $\mathcal{S}[0]$, and zero or more *accept states*. To determine whether a string s is in the language accepted by an FA $\langle \mathcal{S}, \mathcal{T} \rangle$, we start the FA in $\mathcal{S}[0]$ and have it read s until there are no characters left to read. If, at that time, the FA is in an accept state, then s is in the language. Otherwise, it is not. In our visual depiction of FAs, we represent states, transitions, start states and accept states as follows:



A deterministic finite automaton (DFA) is an FA with no epsilon transitions and in which there is exactly one transition out of every state for each label $l \in \Sigma$. An FA that does not satisfy these conditions is a nondeterministic FA (NFA). The minimal DFA representation of a language is a unique, subject to renaming the states [12, p 57]. In the remainder of the paper, we will assume an FA is an NFA unless stated otherwise. NFAs and DFAs have equivalent expressive power, in that both accept the family of regular languages, but NFAs may be exponentially smaller. We call a domain represented using a regular expression or FA a *regular domain*.

Regular expressions and FAs have been used in many application domains involving strings, such as data mining from databases or the Web. For example, in [6], the

authors addressed the issue of mining frequent sequences from a database of sequences in the presence of regular expression constraints (see [1] for a detailed discussion on the issue of mining sequential patterns). Regular expression constraints are user-defined sequence patterns that are used to match strings in the database or web during query or search. Our work differs from past work in that we do not simply use regular languages to match fixed strings. Rather, we use them to propagate constraints among string variables, whose domains may be infinite. For example, `match` is indeed a common constraint in our library. However, the domain of the string to be matched need not be singleton. In addition to `match`, many other types of string constraints appearing in real-world problems need to be represented. We discuss some common ones in the next section.

There has been some work applying constraint reasoning to strings, but relying on less expressive representations of string domains. [4] reports on a language capable of specifying constraints for searching patterns in bio-sequences, such as the length of a string, the distance between two strings, and the position of a string where a character matches. The sequences (strings of symbols) are represented as lists, and the constraints and the constraint solver are implemented using CLP(FD). [19] discusses CLP(S), CLP extended to deal with strings, and its applications to natural language, images and genetic code processing. In contrast to the work presented here, strings are represented as concatenations of variables and constants, which are strictly less expressive than the regular language representation presented here.

4 Constraints

Constraints are usually defined as mathematical formulations of relationships to be held among objects. For example, $x + y = z$ is a constraint describing an equality relation that holds among three numeric variables x , y , and z . Similarly, for the string variables x , y , and z , we can define a string constraint as $x + y = z$ which represents a concatenation relation; that is, string z is the concatenation of strings x and y . We have implemented a number of string constraints in our constraint reasoning framework, which supports generalized arc consistency (GAC), even on infinite sets of strings. In the following, we give definitions of these constraints, illustrated by how they are enforced using FAs.

4.1 Matches

One of the constraints in the library tests whether a string matches a given regular expression:

```
matches(string  $x$ , regexp  $re$ )
```

Although `matches` takes two arguments, it is essentially a unary constraint, because it is not enforced unless the domain of re is a singleton, in which case it computes the FA corresponding to the regular expression represented by re and intersects it with the domain of x . `Matches` subsumes all possible unary constraints over strings expressible in our formalism, so other unary constraints, such as `allUpperCase` and `isAlphaNumeric`

need not be implemented. `matches` is used in type constraints to define the initial domains of variables of given subtypes of string. For example, we can define a Unix filename as any string of non-zero length that does not contain the character '/':

```
matches(filename, "~[/]+")
```

and we can define a time as a string of the form HH:MM:SS:

```
matches(time, "(([0-1][0-9])|(2[0-3])):[0-5][0-9]:[0-5][0-9]")
```

4.2 Concatenation

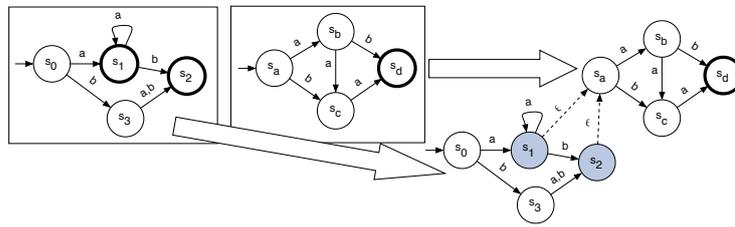


Fig. 1. Concatenation

One of the most obvious operations on strings is concatenation. The concatenation of two strings, x and y , yields another string, z , which consists of all the characters of x followed by all the characters of y :

```
concat(z, x, y)
```

This can be generalized to concatenation of three or more strings in the obvious way. If the domains of x and y are regular, the domain of z will simply be the result of concatenating the FA representations of x and y — that is, adding ϵ -transitions from the accept states of the FA for x to the start state of the FA for y , as shown in Figure 1, obviously a linear-time operation.

Less obviously, if the domains of x and z are regular, the domain of y is also regular. To construct an FA for y given FAs for x and z , we in effect traverse the FAs for z and x in parallel, exploring the cross-product of the nodes from the two FAs, starting with the pair of initial states and adding a transition $\{s_n, t_m\} \xrightarrow{lab} \{s_p, t_q\}$ from every node $\{s_n, t_m\}$ and every label lab such that the transitions $s_n \xrightarrow{lab} s_p$ and $t_m \xrightarrow{lab} t_q$ appear in the original FAs (see Figure 2). This is simply the operation that is performed when intersecting two FAs (Section 5.1). Whenever we reach a state $\{s, t\}$, such that s is an accept state in the FA for x , we mark state t . After the traversal is complete, the marked states in the FA for z represent all of the states that can be reached by reading a string accepted by x .

A new nondeterministic FA (NFA) for y is constructed by copying the FA for z , making the start node a non-start node and making all the marked nodes new start nodes. The complexity of the whole operation is dominated by generating the cross-product FA, so is the same as domain intersection (Section 5.1). A similar procedure can be used to construct an NFA for x , given FAs for y and z .

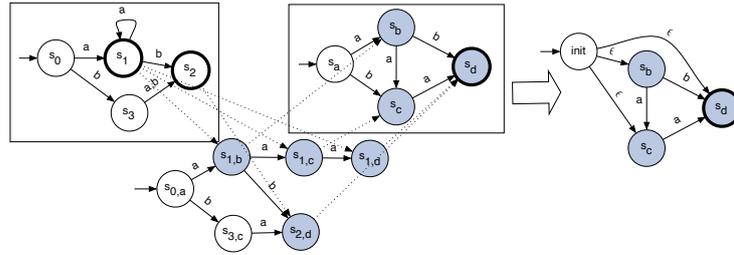


Fig. 2. Given FAs for x (left box) and z (right box), find an FA for y such that z is concatenation of x and y . First, traverse FAs for z and x in parallel, constructing cross-product FA (lower left). Then, identify states that are accept states for x and mark the corresponding states in the FA for z (shaded circles). Construct a new NFA (right) for y by copying the FA for z and making marked nodes start nodes.

4.3 Containment

The relation

$$\text{contains}(\text{String } a, \text{String } b)$$

means that string b is a substring of a . If the domain of b is a regular language r , then the domain of a is given simply by the regular expression $“. *r. *”$. Given an FA for r , we can create an FA for $“. *r. *”$ in linear time by concatenating the FAs for $“. *”$, r and $“. *”$. If we have some other FA representing the domain of a , we simply intersect that domain with the domain for $“. *r. *”$.

Less obviously, if the domain of a is regular, then so is the domain of b . Given an FA for a , we can construct an NFA for b in linear time by eliminating any dead-end nodes from a (that is, nodes from which it is impossible to reach an accept node), adding a new start state, with ϵ -transitions to all states, and then making all states in a accept states (Figure 3). Again, we simply intersect this domain with the original domain for b to enforce the constraint.

4.4 Length

Constraints on the length of a string can also be represented using FAs:

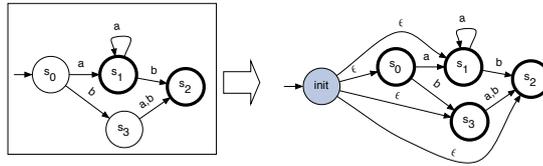
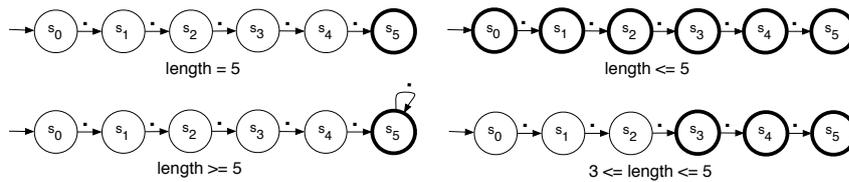


Fig. 3. Given an FA for a regular language r , construct a new FA for all substrings of strings in r .



As these examples show, intervals over the length are simple to represent; if we have a constraint of the form $\text{length}(s, n)$, and the domain of n is represented as a finite interval, we can enforce the constraint without waiting until n becomes singleton. We simply construct a linear FA whose size is one plus the upper bound of n , and label all of the states whose position exceeds the lower bound as accept states. Similarly, if $d(n) = [x, \infty)$, we construct a linear FA of size $x + 1$ and make the last state an accept state with a self-transition. The time to construct the FA is proportional to either the upper bound of n , or to the lower bound if there is no upper bound.

Conversely, if we have a regular domain representation of s , we can obtain lower and upper bounds for n by determining the shortest and longest paths from the start state to an accept state, a linear-time operation. If there is no upper limit on the size, there will be a loop along a path to an accept state.

4.5 Other constraints

Many other string constraints are straightforward to represent. To reverse all strings in a regular domain, we simply reverse the direction of all the transitions and reverse the status of start and accept states in the FA, a linear time operation. If doing so would result in multiple start states, we create a new, unique start state and add ϵ -transitions to all would-be start states. To substitute one character for another, we can perform the substitution on the labels of the transitions, also linear time. Subsequences of strings can be obtained using a combination of `concat` and `length`. For example, to specify the 5-character prefix p of string s , we can write $\text{length}(p, 5) \wedge \text{concat}(s, p, r)$, where r is an unconstrained string.

Another common operation on strings is to specify the character at a given location of the string: `characterAt(s, n, c)`, where c is the character at position n of string s . We will assume that n is a constant (The case where n is a variable can be handled in a similar fashion, but is more complex). We apply the same general idea as the `length`

constraint. In fact, for the character at position n in a string to have any value at all, the string must be at least n characters long, so the `characterAt` constraint looks like the constraint `length $\geq n$` , with the addition that the label of the transition leading to the accept state is restricted to the domain of c . Given the domain of s , we could similarly determine the domain of c in $O(n(|S| + |T|))$ time, by finding all states reachable in $n - 1$ transitions from the start state, then taking the union of the labels of transitions from which it is possible reach an accept state.

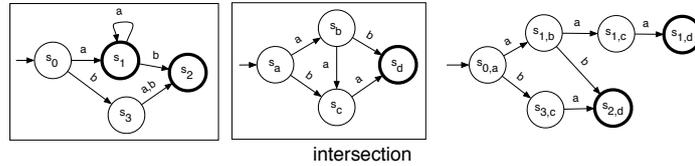
Of the constraints we discussed, `matches`, `concat`, `contains` and `reverse` are implemented in our constraint library. Implementation of the others is left as future work.

5 Domain operations

In order to effectively eliminate inconsistent values from regular domains during constraint propagation, we need to be able to perform set operations on the domains, including intersecting two domains, determining whether one is a subset of another and determining whether a domain is empty or singleton. We can perform these operations easily using FAs. It is well known that regular languages are closed under intersection, union and negation [12, p 58-60], and the algorithms for performing these operations on FAs are straightforward.

5.1 Intersection

Since intersection is such an important domain operation, we show the algorithm for intersection below.



```

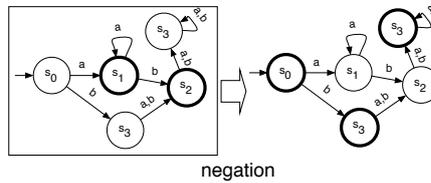
let init  $\leftarrow$   $\{S_1[0], S_2[0]\}$ 
push init
let  $S' \leftarrow \{\text{init}\}$ ,  $T' \leftarrow \{\}$ 
while(stack not empty)
   $\{s_1, s_2\} \leftarrow$  pop;
  isAccept( $\{s_1, s_2\}$ )  $\leftarrow$  isAccept( $s_1$ ) and isAccept( $s_2$ )
  foreach lab  $\in \Sigma$  such that  $\langle s_1 \xrightarrow{\text{lab}} s_x \rangle \in T_1$  and  $\langle s_2 \xrightarrow{\text{lab}} s_y \rangle \in T_2$ 
     $\langle \{s_1, s_2\} \xrightarrow{\text{lab}} \{s_x, s_y\} \rangle$  to  $T'$ 
    if( $\{s_x, s_y\} \notin S'$ )
       $\langle \{s_x, s_y\} \rangle$  add  $S'$ 
       $\langle \{s_x, s_y\} \rangle$  push
  return  $\langle S', T' \rangle$ 

```

The graph is built by exploring reachable states in $S_1 \times S_2$, starting from the pair of initial states. Because of the test in the innermost loop, no state in $S_1 \times S_2$ will be visited

more than once. In the idealized case of a DFA in which each transition is represented explicitly, the size of the new FA, and the time to build it, is thus $O(|\Sigma| |\mathcal{S}_1| |\mathcal{S}_2|)$, independent of the number of transitions in the input FAs. In reality, transitions have *sets* of labels, and the intersections of these sets can result in additional transitions. For example, given transitions in one FA on [a-g] and [h-z], and a transition in the other FA on [d-k], we may end up with transitions in the new FA on [a-c], [d-g], [h-k] and [l-z]. Additionally, an NFA may contain multiple transitions on the same label, so at worst we need to consider all pairs of transitions from the input FAs, giving a space and time complexity of $O(|\mathcal{T}_1| |\mathcal{T}_2| + |\mathcal{S}_1| |\mathcal{S}_2|)$.

5.2 Negation



Complementing the accept states of a DFA results in a DFA accepting the complement of the language [12, p 59]. Although complementing the accept states is clearly a linear time operation, converting an NFA to a DFA potentially generates the power set of the NFA, an exponential blowup. Although neither intersection nor negation result in NFAs, some of the constraints defined in Section 4 do.

Given intersection and negation, we can apply the following definitions to compute subset and equality relations between two domains:

$$\begin{aligned}
 (fa_1 \subseteq fa_2) &\equiv (\neg fa_2 \cap fa_1 = \emptyset) \\
 (fa_1 = fa_2) &\equiv (fa_2 \subseteq fa_1) \wedge (fa_1 \subseteq fa_2) \\
 (fa_1 - fa_2) &\equiv (fa_1 \cap \neg fa_2)
 \end{aligned}$$

5.3 Splitting domains

Using regular sets as a domain representation, we can propagate constraints very effectively, even when some of the variable domains are infinite. Searching over infinite domains, in contrast, runs the risk of infinite regress, but it can be done by successively splitting the domain into disjoint subsets. Any regular set r can be used to split a domain d , provided neither r nor its complement has an empty intersection with d . The new domains are $d \cap r$ and $d \cap \neg r$. In some applications, a natural choice for sets to split on may present itself. Otherwise, we can easily derive a set r from d by removing transitions from the FA for d . As long as $|d| > 1$, there will be at least one transition leading to an accept state that can be removed without making the language empty (although doing so may require partially unrolling a cycle). r is guaranteed to be a proper subset of d .

5.4 Domain Size

It is important to be able to determine the size of a domain. For example, if the size is 0 (empty), then the constraint network is inconsistent. If the size is 1, then a value for the corresponding variable is determined. Domain size is also useful for variable-ordering heuristics, and knowing whether a domain is finite or infinite is important to avoid searching over infinite domains. Determining the size of a regular domain is less straightforward than determining the size of a set or interval domain, but it can still be done fairly efficiently.

Given a DFA, we can determine the number of strings in the language as follows. We begin by removing all dead-end states from the FA, a linear-time operation. A dead-end state is a state from which it is impossible to reach an accept state. If the initial state is dead-end, then the domain is empty. Once the dead-end states are removed, if the FA contains any loops, then there are infinitely many solutions, because we can follow a loop any number of times and then follow a path to an accept state. We perform a topological sort of the FA, which is linear in the number of arcs. If the sort fails, then there is a loop and thus infinitely many solutions. Otherwise, we traverse the graph in the order dictated by the topological sort, keeping track of the number of paths there are from the initial state to the current state:

size($\langle \mathcal{S}, \mathcal{T} \rangle$)

```

[  $\mathcal{S} \leftarrow \text{topologicalSort}(\mathcal{S})$ 
  pathsFromInit[0] = 1
  for  $i = 0$  to  $|\mathcal{S}|$ 
    [ if isAccept( $s_i$ ) then numSolutions += pathsFromInit[ $i$ ]
      [ foreach transition  $\langle \mathcal{S}[i] \xrightarrow{l} \mathcal{S}[d] \rangle \in \mathcal{T}$ 
        [ pathsFromInit[ $d$ ] += pathsFromInit[ $i$ ]
      ]
    ]
  return numSolutions

```

Let $\mathcal{W}_i \subset \Sigma^*$ be the words that, when read from the initial state, lead to $\mathcal{S}[i]$. To show that the algorithm produces the correct result, we first show that, for all k , before the k th iteration of the **for** loop, $\text{pathsFromInit}[k] = |\mathcal{W}_k|$. The proof will be by induction on k .

Base case: Because \mathcal{S} is topologically sorted, $\mathcal{S}[0]$ is the initial state. Before the **for** loop is executed, $\text{pathsFromInit}[0] = 1 = |\epsilon| = |\mathcal{W}_0|$. Now assume, for all $j < k$, that $\text{pathsFromInit}[j] = |\mathcal{W}_j|$. Let \mathcal{S}_k be the set of states with transitions to $\mathcal{S}[k]$. Because \mathcal{S} is sorted topologically, $\forall \mathcal{S}[i] \in \mathcal{S}_k, i < k$, so all transitions to $\mathcal{S}[k]$ are visited (**foreach** loop) before the k th iteration of the **for** loop. Because the FA is deterministic, no word can be reached by multiple paths, so $|\mathcal{W}_k| = \sum_{\mathcal{S}[i] \in \mathcal{S}_k} |\mathcal{W}_i| \left| \left\{ c \mid \langle \mathcal{S}[i] \xrightarrow{c} \mathcal{S}[k] \rangle \in \mathcal{T} \right\} \right|$. By assumption, $|\mathcal{W}_i| = \text{pathsFromInit}[i]$, so this is precisely the value stored in $\text{pathsFromInit}[k]$ by the k th iteration.

Finally, it suffices to observe that $\text{numSolutions} = \sum_{\{i \mid \text{isAccept}(\mathcal{S}[i])\}} \text{pathsFromInit}[i] = \sum_{\{i \mid \text{isAccept}(\mathcal{S}[i])\}} |\mathcal{W}_i|$, i.e., the number of words accepted by the FA.

6.2 Crossword Puzzle

Another application of string constraints is the *crossword puzzle* problem. Solving crossword puzzles is a popular pastime and also a well-studied problem in computer science. The full problem of solving crossword puzzles, given only the puzzle layout and a list of clues, is a hard problem that involves many aspects of AI [21]. A more commonly addressed problem is generating crossword puzzles, given a fixed board and a list of possible words [7]. This problem becomes a classic constraint satisfaction problem, where the variables of the constraint problem are word slots on the puzzle board in which words can be written, the domains of variables are available words, and the binary constraints on variables enforce the agreement of letters at intersections between slots. Solving the problem reduces to finding a solution to the constraint problem: an assignment of values to the variables such that each variable is assigned a value in its domain and no constraint is violated.

We can use string constraints to formalize the crossword puzzle problem. There is a variable for each slot, each intersection point and each contiguous segment of text within a slot that does not cross an intersection. The variables for word slots take values from all available words, the variables for intersection points take values of letters from the alphabet, and the variables for segments take values of unknown strings of fixed length. Each word slot is constrained to be the concatenation of the segments and intersection points that it contains.

For example, suppose that we have the following crossword puzzle that is taken from <http://yoda.cis.temple.edu:8080/UGAIWWW/lectures95/search/puzzle.html>:

x1	b1	x2	b2	x3
		c1		c2
		b3		b4
	x4	x5		
	b5	c3	c4	c5
x6		x7		
b7		c6	c7	c8
x8				
c9	b9	c10	c11	c12
b8			b6	

The list of words:

AFT	LASER
ALE	LEE
EEL	LINE
HEEL	SAILS
HIKE	SHEET
HOSES	STEER
KEEL	TIE
KNOT	

To formalize this puzzle as a CSP with string constraints, we have

- 8 variables for the word slots as marked from x_1 to x_8
- 12 variables for those intersection points marked as c_i
- 9 variables for these segments marked as b_i

We have 8 constraints as follows:

1. $\text{concat}(x_1, b_1, c_1, b_2, c_2)$
2. $\text{concat}(x_2, c_1, b_3, c_3, c_6, c_{10})$
3. $\text{concat}(x_3, c_2, b_4, c_5, c_8, c_{12})$

4. $\text{concat}(x_4, b_5, c_3, c_4, c_5)$
5. $\text{concat}(x_5, c_4, c_7, c_1, b_6)$
6. $\text{concat}(x_6, b_7, c_9, b_8)$
7. $\text{concat}(x_7, c_6, c_7, c_8)$
8. $\text{concat}(x_8, c_9, b_9, c_{10}, c_{11}, c_{12})$

It is worth noting that, although we may have more variables than the traditional CSP formalization, only the x_i variables, that is, those variables representing word slots, need to be searched during the CSP solving. Other variables will be assigned values by propagation. In fact, with the constraint system we implemented to support a constraint-based planner, we can solve the above crossword puzzle example without backtracking.

7 Conclusions

We have discussed an approach to constraint reasoning over strings in which regular languages are used to represent and reason about infinite sets of strings. Regular languages have a number of qualities to recommend them as a domain representation:

- They are closed under intersection, union and negation.
- They can concisely represent infinite sets of strings.
- Many natural string constraints, such as concatenation, containment and length, can be represented in terms of operations on regular languages.
- They are widely used and well understood.

These advantages do come at a price; it can be substantially more costly to represent and reason about regular languages than, say intervals. All of the set operations and string constraints we have discussed are either linear or quadratic in the size of the FAs representing the string domains. However, as noted, converting an NFA to a DFA may result in an exponential blowup in the size of the FA. Furthermore, even when every operation on the FA results in a polynomially larger FA, the FA can still grow exponentially with the number of operations, i.e., the number of constraints that contain the variable whose domain is represented by the FA. Ultimately, how the FA grows will depend on the nature of the problem at hand. The FA representation can be viewed as a compression of the full sets of strings. It will tend to do well at compressing sets with a lot of symmetry and simple structure, but will not do so well at compressing arbitrary lists of strings, where there is little or no structure to exploit. In the latter cases, the representation will blow up, converging toward an explicit list of the members. The exponential blowup in the representation can be viewed as a failure in the exponential reduction that FAs are capable of providing.

We have implemented a constraint-based planner that uses many of the string constraints discussed here and demonstrated it in software planning domains. Our implementation is complete, but inefficient. Although there are many highly optimized FA packages freely available, they are tailored to string matching, not domain representation, so we wrote our own, with little regard for efficiency. For example, our algorithm for DFA minimization is $O(|T||S|^2)$, even though much faster algorithms are available. Improving the efficiency, and exploring other domains, such as bioinformatics, is the subject of future work.

References

1. R. Agrawal and R. Srikant. Mining sequential patterns. In *Proceedings of the 11th International Conference on Data Engineering*, 1995.
2. C. Bessiere and J. Ch. Arc-consistency for general constraint networks: Preliminary results. In *Proceedings of IJCAI-97*, pages 398–404, Nagoya, Japan, August 1997.
3. J. R. Bitner and E. M. Reingold. Backtrack programming techniques. *Communications of the ACM*, 18(11):651–656, 1975.
4. I. Eidhammer, I. Jonassen, and S. Grindhaug. A constraint based structure description language for biosequences. *Constraints*, 6:173–200, 2001.
5. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman and Co., 1979.
6. M. Garofalakis, R. Rastogi, and K. Shim. SPIRIT: sequential pattern mining with regular expression constraints. In *Proceedings of the 25th VLDB Conference*, 1999.
7. M. Ginsberg, M. Frank, M. Halpin, and M. Torrance. Search lessons learned from crossword puzzles. In *Proceedings AAAI-1990*, pages 210–215, 1990.
8. K. Golden. Automating the processing of earth observation data. In *7th International Symposium on Artificial Intelligence, Robotics and Automation for Space*, 2003.
9. K. Golden and J. Frank. Universal quantification in a constraint-based planner. In *AIPS02*, 2002.
10. S. W. Golomb and L. D. Baumert. Backtrack programming. *Journal of the ACM*, 12(4):516–524, 1965.
11. T. Hickey, M. van Emden, and H. Wu. A unified framework for interval constraints and interval arithmetic. In *Proceedings of CP-1998*, pages 250–264, 1998.
12. J. Hopcraft and J. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Philippines, 1979.
13. T. R. Jensen and B. Toft. *Graph Coloring Problems*. Wiley-Interscience, New York, 1995.
14. A. Jónsson and J. Frank. A framework for dynamic constraint reasoning using procedural constraints. In *Proceedings of ECAI-2000*, 2000.
15. A. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.
16. N. Muscettola. Computing the envelope for stepwise constant resource allocations. In *Proceedings of CP-2002*, 2002.
17. B. A. Nadel. Consistent satisfaction algorithms. *Computational Intelligence*, 5:188–224, 1989.
18. P. Prosser. Hybrid algorithms for the constrain satisfaction problem. *Computational Intelligence*, 9(3):268–299, 1993.
19. A. Rajasekar. Applications in constraint logical programming with strings. In *PPCP-1994*, 1994.
20. F. Rossi, A. Sperduti, K. Venable, L. Khatib, P. Morris, and R. Morris. Learning and solving soft temporal constraints: An experimental study. In *Proceedings of CP-2002*, 2002.
21. N. Shazeer, M. Littman, and G. Keim. Solving crossword puzzles as probabilistic constraint satisfaction. In *Proceedings of AAAI-1999*, 1999.
22. D. L. Waltz. Understanding line drawings of scenes with shadows. In P. H. Winston, editor, *The Psychology of Computer Vision*, pages 19–91. McGraw-Hill, 1975.
23. Monte Zweben and Mark S. Fox. *Intelligent Scheduling*. Morgan Kaufmann Publishers, San Francisco, California, 1994.