Constraint reasoning over strings

Keith Golden[1] and Wanlin Pang[2]

[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 many 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 [12,5,24] and many real-world problems [25,13,17,22] 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, constraint reasoning over strings, by and large, remains pretty much unexplored.

Strings appear everywhere. Like any other objects in the real-world, certain relationships exist among strings and between strings and other objects. In many real-world applications those relationships can be formalized as constraints over strings. For example, we are applying constraint-based planning to automate certain operations in software domains [8,9], 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 class "my.package.MyClass" to be "my/package/MyClass.java," and it produces a file "my/package/MyClass.class."
- The pathname of data downlinked 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.
- The contents of structured or semistructured text files can be described in terms of constraints between the text and what the text represents.

A distinguishing characteristic of software domains and others involving strings is that the set of strings corresponding to a variable representing 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 deal with constraints over infinite string sets without exhustively listing infinite string 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 the string constraint problems can be solved within the general-prupose constraint reasoning framework we have developed for an on-going constraint-based planning project.

The remainder of the paper is organized as follows. In Section 2, we review notations of constraint satisfaction problems. In Section 3, we discuss string domain representations, namely, as regular languages. In Section 4, we provide definitions of the constraints on strings and desribe how they are enforced using this domain representation. In Section 5 we discuss how standard domain operations, such as intersection and determining equality or cardinality, are handled. In Section 6, we analyze the computational complexity of all the operations involved in constraint reasoning using regular domains. In Section 7, we show how the string constraints can be applied to solving some interesting problems. And finally, in Section 8 we conclude by summerizing 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 $< X, D, C >$ where $X = \{x_1, x_2, \ldots, x_n\}$ is a finite set of variables, $D = \{d(x_1), d(x_2), \ldots, d(x_n)\}$ is a set of domains containing values the variables may take, and $C = \{C_1, C_2, \ldots, C_m\}$ is a set of constraints. Each constraint C_i is defined as a relation R on a subset of variables $V = \{x_i, x_j, \ldots, x_k\}$, called the constraint scope. R may be represented extensionally as a subset of Cartesian product $d(x_i) \times d(x_j) \times \ldots \times d(x_k)$. 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 $V_K = \{x_{k_1}, \ldots, x_{k_l}\}$ be a subset of X. An l-tuple $(x_{k_1}, \ldots, x_{k_l})$ from $d(x_{k_1}) \times \ldots \times d(x_{k_l})$ 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 algorithm [4,10]. However, for CSPs with infinite domains such as the ones we are interested 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 [16,3]. 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 [18,19] at each node of the search tree. A key issue is the trade-off between time spent on propagation and the reduction in search space needed to allow feasible and effecient search. Based on our experience dealing with constraint-based planning in software enviornment, 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 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 the planning problems.

3 String Domains

The domain $d(x)$ of variable x is the set of values that x can take. This set will, in general, change during the course of search and constraint propagation. Typically, a variable's domain is represented as a list of the values that the variable can take. 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 they 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) [3] 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.

We use two different representations of regular languages: regular expressions and finite automata. Regular expressions are used for input and are converted to FAs, which are used computationally. Since regular expressions and FAs are well known, we will not discuss them in depth, but we will briefly review for the sake of defining our terminology.

A regular expression represents a regular language over an alphabet Σ. In our implementation, Σ is the set of Unicode characters. We use the following notation to describe regular expressions.

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
$\backslash c$	the literal character c
$re_1 re_2$	re_1 followed by re_2
$re_1 \mid re_2$	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)

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

We represent regular languages internally using FAs, since the latter are easier to compute with that regular expressions. An FA is a pair $< \mathcal{S}, \mathcal{T} >$, 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 $< n_1, l, n_2 >$, which we will write $< n_1 \stackrel{l}{\rightarrow} n_2 >$, where $n_1$is the starting state of the transition, $n_2$is the ending state and $l \in \Sigma$ is the transition label. The input to the FA is a sequence of symbols from Σ. Whenever there are symbols left to read, the FA reads the next symbol and follows a transition from the current state whose label l is the symbol just read. If there are multiple transitions labeled l, one is chosen nondeterministically. If there are no transitions labeled l, the FA halts and returns failure. For efficiency, we allow transitions to have sets of labels, represented using the same notation as is used for regular expressions. For example, we could have a transition $< n_1 \stackrel{[a-zA-Z]}{\rightarrow} n_2 >$, 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 can be 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 $< \mathcal{S}, \mathcal{T} >$, 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, states, transitions, start states and accept states are represented as follows:

A deterministic finite automaton (DFA), is an FA with no epsilon transitions and in which there only 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). In the remainder of the paper, we will assume an FA is an NFA unless stated otherwise. As is well known, 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 [15] have been used in many application domains involving strings, such as data mining from databases or from web for discovering interesting data patterns and web structures. 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 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 string being matched need not be singleton. In addition to match, many other types of string constraints appearing in real-world problem need to be represented. We discuss some common ones in the next section.

4 Constraints

Constraints are usually defined as mathematical formulations of relationships to be held between objects. For example, $x+y=z$ is a constraint describing an equalitiy 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, string 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 possible constraints, such as allUpperCase in isAlphaNumeric are not 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(fn, "~[/]+")

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

$$\mathsf{matches}(d, \text{“}(([0-1][0-9])\,|\,(2[0-3])) : [0-5][0-9] : [0-5][0-9]\text{”})$$

4.2 Concatenation

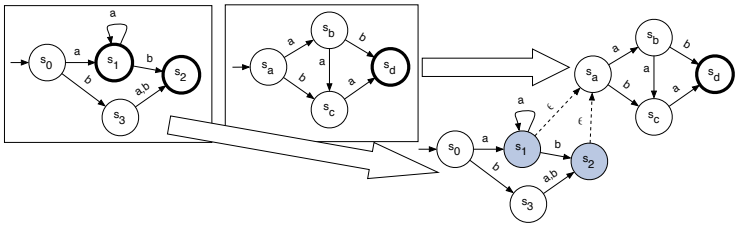


Fig. 1. Concatenation

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:

$$\mathsf{concat}(z, x, y)$$

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 from 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 $\{s_n, t_m\} \stackrel{lab}{\rightarrow} \{s_p, t_q\}$ from every node $\{s_n, t_m\}$ and every label lab such that the transitions $s_n \stackrel{lab}{\rightarrow} s_p$ and $t_m \stackrel{lab}{\rightarrow} t_q$ appear in the original FAs (see Figure 2). This is simply the operation that is performed when intersecting two FAs. Whenever we reach a node $\{s, t\}$, such that node s is an accept state in the FA for x, we mark node t. After the traversal is complete, the marked nodes 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 (O(mn), where m and n are the number of nodes in the FAs for x and z, respectively). A similar procedure can be used to construct an NFA for x, given FAs for y and z.

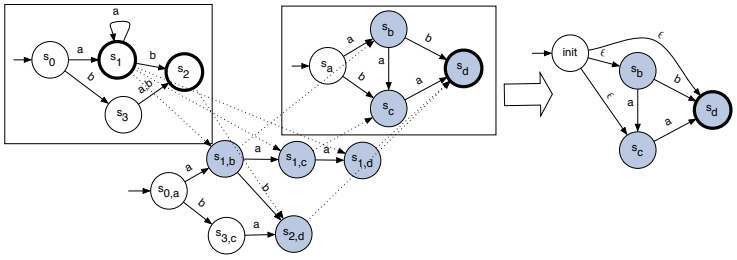


Fig. 2. Given FAs for *x*(upper left) and *z* (upper right), find an FA for *y* such that *z* is concatenation of *x* and *y*. First, traverse FAs for *z* and *x* in parallel, constructing cross-product FA (lower left). Then, identify states that are accept states for *x* and mark the corresponding states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

4.3 Containment

The relation

contains(String *a*, 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 simply the regular expression ".*r*.*". Given an FA for *r*, we can create an FA for ".*r.*" by adding new start and accept states that have self-loops on any string ("."), and connect them to the original start and accept states using ε-transitions (Figure 3). If we have some other FA representing the domain of *a*, we simply intersect that domain with the domain for ".*r.*".

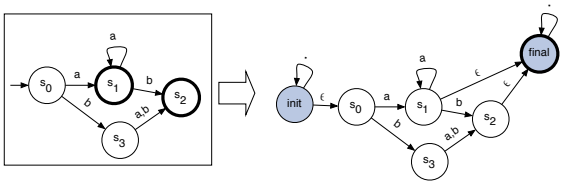


Fig. 3. Given an FA for a regular language *r*, construct a new FA for .**r*.*, strings that contain strings in *r.*

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* 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 states, with ε-transitions to all states, and then making all states in *a* accept states (Figure 4). Again, we simply intersect this domain with the original domain for *b* to enforce the constraint.

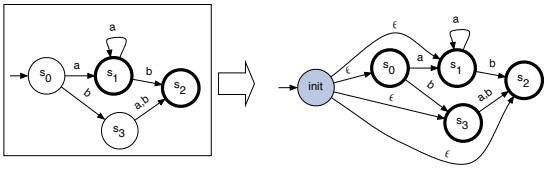
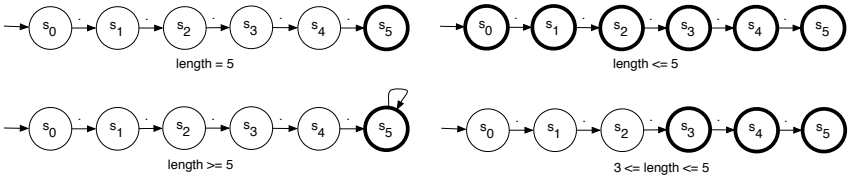


Fig. 4. Given an FA for a regular language r, construct a new FA for all substrings of strings in r.

4.4 Length

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



As the bottom two examples show, intervals over the length are simple to represent; if we have a constraint of the form 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.

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. To substitute one character for another, we could perform the substitution on the labels of the transitions. Subsequences of strings could be obtained using a combination of concat and length. For example, to specify the 5-character prefix p of string s, we can write length$(p,5)\wedge$concat(s,p,r), where r is an uncontrained 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 than 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

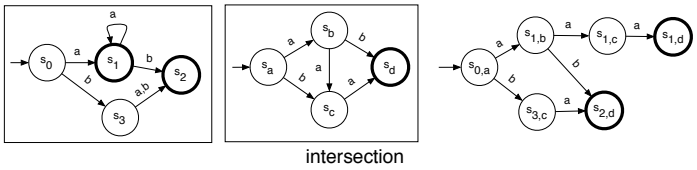
constraint. In fact, for the character at position n in a string to have any value at all, the string must be at least n characters long, so the characterAt constraint looks like the constraint length $\geq n$, with the addition that the label of the transition leading to the accept state is restricted to the domain of c.

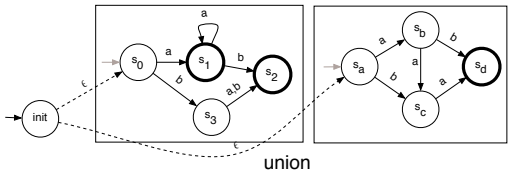
Given the domain of s, we could similarly determine the domain of c in $O(n(|\mathcal{S}| + |\mathcal{T}|))$, by finding all states reachable in $n-1$ transitions from the start state, then taking the union of the labels of transitions from which it is possible reach an accept state.

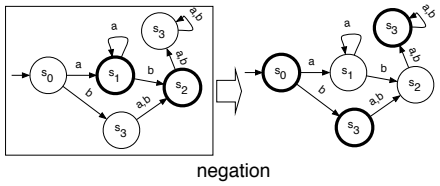
Of the constraints we have discussed, matches, concat, contains and reverse are implemented in our constraint library. Implementation of the others is left as future work.

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 intersected 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, and the algorithms for performing these operations on FAs are both straightforward and widely known, so we will not repeat them here, but we illustrate them graphically as a reminder.







Of these set operations, intersection is used frequently in constraint propagation and negation is useful for domain subtraction, subset tests and other operations, but union is not a common set operation for domains. Superficially, it may seem that intersection

is the most expensive operation, since it potentially generates the cross-product of its inputs, whereas union and negation take linear time in their inputs. However, union produces an NFA and negation requires a DFA. Converting an NFA to a DFA potentially generates the power set of the NFA, an exponential blowup.

Given these operations we can apply the following definitions to compute subset and equality relations between two domains:

$(fa_1 \subseteq fa_2) \equiv (\neg fa_2 \cap fa_1 = \emptyset)$

$(fa_1 = fa_2) \equiv (fa_2 \subseteq fa_1) \wedge (fa_1 \subseteq fa_2)$

5.1 Domain Size

It is important be 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. If the size is small and finite, then it may be appropriate to explicitly select a value in a search for a solution, but if the size is infinite, then such a search may never terminate. 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 an FA, 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. 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, an operation that 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($< \mathcal{S}, \mathcal{T} >$)

```
sort S topologically
pathsFromInit[0] = 1
numSolutions ← { 1 if isFinal(S[0])
               { 0 otherwise
for i = 1 to |S|
    foreach transition < n_i --l--> n_d > ∈ T starting from n_i
        if isFinal(n_d)
            numSolutions += pathsFromInit[i]
        pathsFromInit[d] += pathsFromInit[i]
return numSolutions
```

6 Complexity

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, many operations, such as union, produce NFAs as outputs, and some, such as negation, require

DFAs as inputs. As noted, converting an NFA to a DFA may result in exponential blowup in the size of the FA. Furthermore, even when every operation on the FA results in a polynomially larger FA, that can still mean exponential growth in 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.

Using regular domains is worth considering in problems in which one of the following holds:

1. There is a great amount of symmetry or the domain is highly under-constrained. In this case, the benefit of a precise domain representation should outweigh the negligible cost in time and storage.
2. It is necessary to explicitly consider all possible domain values or solutions to the CSP. In this case, the domain will have to be enumerated one way or the other. In the worst case, a minimized FA requires space that is linear in the size of the list of strings and could be arbitrarily better.
3. There are constraints over strings of unbounded length. In this case, the domain is infinite. The only alternative to regular languages that we know of for representing infinite sets of strings is to represent every infinite domain as the full domain (the set of all strings). With regular domains, we can enforce generalized arc consistency over infinite sets of strings, making it possible to solve problems that could not be solved otherwise.

7 Examples

7.1 Pathname

In Unix, sets of files are often represented using regular expressions on their pathnames. Correspondingly, regular domains are very useful for representing sets of files in a constraint-based planning problem. In addition to the ability to represent large sets concisely, we can also handle constraints that relate the file's pathname to other attributes of the file. For example, satellite images and other automatically generated data are typically stored in ordinary filesystems, with pathnames based on details of the data, such as the time, subject, source, file format, etc. Suppose we have a remote archive in which satellite images have pathnames of the form:

`/downlink/`< year >/<dayOfYear>/< sensor> <gridx><gridy>. <format>

We can represent this knowledge using a concatenation constraint:

rpn= concat("`/downlink/`", y, "/", d, "/", s, gx, gy, ".", fmt).

Given only this knowledge, all we know about *rpn* is that the set of files is characterized by the regular expression "`/downlink/.*/.*/.*/.*\..*`". However, most likely we know quite a bit about the other variables. We know how many years the satellite has been in operation, how many days are in a year, the sensors aboard the satellite, the grid coordinate system used to indicate the regions covered by the images, and the available formats. Assuming we are interested in just a subset of the data, we can impose additional constraints on these variables to specify just the files we are interested in. For example, if we want MOD17 data from January 27, 2002 in either HDF or binary format, then the domain of *rpn* is "`/downlink/2002/27/MOD17[0-9][0-9][0-9][0-9]\.(hdf|bin)`"

String constraints are not just useful for specifying sets of files, but also specifying the effects of file operations. Since the files are on a remote server, we can't access them directly, but we can copy them to a local disk. Suppose we executed the command `scp -r server:/downlink/2002 local02` to copy the contents of the directory `2002` to the directory `local02`. We can describe the effect on the pathnames of the resulting files using the pair of constraints:

1. concat(*rpn*, "`/downlink/2002/`", *ldir*)
2. concat(*lpn*, "`local02/`", *ldir*)

Since the concat constraint can be used to derive the domain of any variable, given the domains of the other two variables, and since we know that the domain of *rpn* (limited to the files we care about) is

`/downlink/2002/27/MOD17[0-9][0-9][0-9][0-9]\.(hdf|bin)`

we can enforce the first constraint to obtain the domain of *ldir*:

`27/MOD17[0-9][0-9][0-9][0-9]\.(hdf|bin)`

We can then apply the second constraint to obtain the domain of *lpn*:

`local02/27/MOD17[0-9][0-9][0-9][0-9]\.(hdf|bin)`

If, after copying the files, we discovered that there are only HDF files, we could apply the same constraints in the other direction to conclude that there were no binary files on the server.

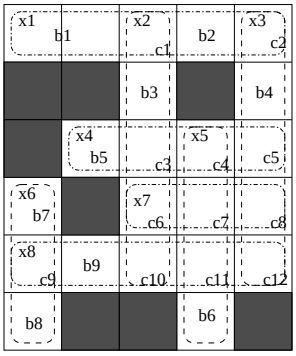
7.2 Crossword Puzzle

Another application of string constraints is to the *crossword puzzle* problem. Solving crossword puzzles is a very 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 [7,23]. A more commonly addressed simplification of the problem, in which a list of possible words is given instead of clues, is more akin to creating crossword puzzles than solving them. 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 a crossword puzzle 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, 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*



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 slot 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. concat$(x_1, b_1, c_1, b_2, c_2)$
2. concat$(x_2, c_1, b_3, c_3, c_6, c_{10})$
3. concat$(x_3, c_2, b_4, c_5, c_8, c_{12})$
4. concat$(x_4, b_5, c_3, c_4, c_5)$
5. concat$(x_5, c_4, c_7, c_1 1, b_6)$
6. concat(x_6, b_7, c_9, b_8)
7. concat(x_7, c_6, c_7, c_8)
8. concat$(x_8, c_9, b_9, c_{10}, c_{11}, c_{12})$

It is worth noting that, comparing to the traditional CSP formalization, we may have many additional variables introduced to the formalized crossword puzzle problem, but 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.

7.3 Bioinformatics

Constraint techniques have been applied to bioinformatics. For example, the authors in [14] reported their work on applying a constraint-based approach to determining protein structures. The problem of determining protein structures is modelled as a constraint problem where variables are the Cartesian coordinates of each atoms in the protein, and constraints are restrictions on these coordinates. In [20] an integer programming (IP) approach, which can be seen as a special case of constraint formulation, is applied to solving sequence alignment and protein threading problems in genetics. Numerical constraints are also applied to genome mapping [21] and protein structure prediction [2].

It is possible that string constraints could play an important role in applying constraint based approaches to bioinformatics. DNA, RNA and proteins can be represented as strings. In the case of DNA, the letters are the familiar nucleotides A, G, C and T. In the case of proteins, the letters are the 20 amino acids. Many problems in bioinformatics involve matching DNA sequences against a database, a classic textual search problem in which regular expressions are commonly used. Other problems, such as reconstructing chromosomes from short DNA fragments (or *clones*), can be formalized as constraint satisfaction problems or constrained optimization problems using string constraints, and could be solved using advanced constraint satisfaction algorithms. However, this is left as a future work.

8 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. On the other hand, the time and space complexity of constraint reasoning with regular languages can be literally infinitely less than that of reasoning over explicit sets of strings.

References

1. R. Agrawal and R. Srikant. Mining sequential patterns. In *Proceedings of the 11th International Conference on Data Engineering*, 1995.
2. R. Backofen. Constraint techniques for solving the protein structure prediction problem. In *Proceedings of CP-98*, pages 72–86, 1998.

3. 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.
4. J. R. Bitner and E. M. Reingold. Backtrack programming techniques. *Communications of the ACM*, 18(11):651–656, 1975.
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 and J. Frank. Universal quantification in a constraint-based planner. In *AIPS02*, 2002.
9. Keith Golden. Automating the processing of earth observation data. In *7th International Symposium on Artificial Intelligence Robotics and Automation for Space*, 2003.
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. T. R. Jensen and B. Toft. *Graph Coloring Problems*. Wiley-Interscience, New York, 1995.
13. A. J'onsson and J. Frank. A framework for dynamic constraint reasoning using procedural constraints. In *Proceedings of ECAI-2000*, 2000.
14. L. Krippahl and P. Barahona. Applying constraint programming to protein structure determination. In *Proceedings of CP-99*, pages 289–302, 1999.
15. H. Lewis and C. Papadimitriou. *Elements of the theory of computation.* Prentice Hall Inc., 1981.
16. A. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.
17. N. Muscettola. Computing the envolope for stepwise constant resource allocations. In *Proceedings of CP-2002*, 2002.
18. B. A. Nadel. Consistent satisfaction algorithms. *Computational Intelligence*, 5:188–224, 1989.
19. P. Prosser. Hybrid algorithms for the constrain satisfaction problem. *Computational Intelligence*, 9(3):268–299, 1993.
20. K. Reinert, H. Lenhof, P. Mutzel, and K. Melhorn andJ. Kececioglu. A branch-andcut algorithm for multiple sequence alignment. In *Proceedings of the 1st Annual Internatioanl Conferencs on Computational Molecular Biology*, pages 241–249, 1997.
21. P. Revesz. Refining restriction enzyme genome maps. *Constraints*, 2:361, 1997.
22. 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.
23. N. Shazeer, M. Littman, and G. Keim. Solving crossword puzzles as probabilistic constraint satisfaction. In *Proceedings of AAAI-1999*, 1999.
24. 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.
25. Monte Zweben and Mark S. Fox. *Intelligent Scheduling*. Morgan Kaufmann Publishers, San Francisco, California, 1994.