Using Constraints to Model Disjunctions in Rule-Based Reasoning *

Bing Liu and Joxan Jaffar

Department of Information Systems and Computer Science
National University of Singapore
Lower Kent Ridge Road, Singapore 119260
{liub, joxan}@iscs.nus.sg

Abstract

Rule-based systems have long been widely used for building expert systems to perform practical knowledge intensive tasks. One important issue that has not been addressed satisfactorily is the disjunction, and this significantly limits their problem solving power. In this paper, we show that some important types of disjunction can be modeled with Constraint Satisfaction Problem (CSP) techniques, employing their simple representation schemes and efficient algorithms. A key idea is that disjunctions are represented as constraint variables, relations among disjunctions are represented as constraints, and rule chaining is integrated with constraint solving. In this integration, a constraint variable or a constraint is regarded as a special fact, and rules can be written with constraints and information about constraints. Chaining of rules may trigger constraint propagation, and constraint propagation may cause firing of rules. A prototype system based on this idea has been implemented.

1. Introduction

Rule-based systems are one of the great successes of AI [21, 22, 16]. They are widely used to build knowledge-based systems to perform the tasks that normally require human knowledge and intelligence. However, there are still some important issues that have not been handled satisfactorily in the current rules-based systems. One of them is the disjunction. This limits their problem solving power and expressiveness.

In the Constraint Satisfaction Problem (CSP) (which is characterized by finding values for a set of constraint variables subject to a set of constraints) research, many efficient constraint propagation algorithms have been produced [17, 11]. A number of languages or systems based on the model have also been developed and used for solving real-life problems [14].

In this paper, we show that some types of important disjunctions can be modeled with CSP. Thus, it is possible to use the simple representation scheme and efficient problem solving methods in CSP to handle these types of disjunctions. Specifically, the disjunctions can be represented as constraint variables and their domains. The relations among disjunctions can be represented as constraints. In this paradigm, constraint propagation and rule chaining are integrated. A constraint can be added as a special fact, and rules can be written with constraints and information about constraints. Chaining of rules may trigger constraint propagation, and constraint propagation may cause firing of rules. With the incorporation of CSP techniques, the power and the expressiveness of rule-based systems will be greatly increased. Based on this idea, a prototype system, called CFR, has also been implemented.

* A revised version of this paper will appear in *AAAI-96*.

The idea of incorporating CSP into a logic-based system is not new. Constraint solving has long been integrated with logic programming languages such as Prolog. This integration has resulted in a number of Constraint Logic Programming (CLP) languages [14], such as CLP(R) [13] and Chip [10]. These languages are primarily for modeling and solving real-life combinatorial optimization problems, such as scheduling and resource allocations. However, this work is different from that in CLP in a number of ways. The main difference is that CLP languages are all based on Horn clauses and backward chaining, while the proposed integration is based on forward chaining, which is suitable for solving a different class of reasoning problems. Integration of constraint solving and forward chaining has some specific problems that do not exist in CLP languages. The proposed integration is also mainly for improving reasoning capability of existing rule-based systems rather than for solving combinatorial search problems. Thus the types of constraints and their representations in the proposed approach are quite different from those in CLP languages.

We regard this work as the first step to a full integration of the CSP model with forward chaining rule-based systems. The current integration presented in this paper is still restrictive in the sense that it is mainly to help model and handle the problems with some disjunctions. A full integration could potentially change the way that people use rule-based systems and change the way that people solve practical reasoning problems, which are the main applications of the rule-based systems today. It may be just like that CLP languages have changed the way that people model and solve practical combinatorial search problems.

The paper is organized as follows. In the next section, we briefly review rule-based systems and the CSP model. Problems with some types of disjunctions are highlighted. Section 3 shows how CSP can be used to model these types of disjunctions in a rule-based system. Section 4 briefly describes the CFR system. Section 5 gives an example. Section 6 presents related work and Section 7 concludes the paper.

2. Rule-Based Systems and Constraint Satisfaction Problems

This section reviews rule-based systems and CSP. The focus is on highlighting the problems with disjunctions in current rule-based systems.

2.1. Rule-based systems

A rule-base system basically consists of the three main components.

1. A working memory: a set of facts representing the current state of the system.
2. A rule memory: a set of *IF-THEN* rules to test and to alter the working memory.
3. A rule interpreter: which applies the rules to the working memory.

The rule interpreter repeatedly looks for rules whose conditions match facts in the working memory. On each cycle, it picks a rule, and performs its actions. A rule is of the form:

IF <conditions> THEN <actions>

There are three common connectives in a rule-based system, i.e., *and, or* and *not*. We will only discuss *or* here as we are mainly interested in disjunctions in this paper. *or* in logic can be defined to be *inclusive* ($\vee$) or *exclusive* ($\oplus$). Let us first take a look at the inclusive *or*. For example, "if something is a block or a pyramid, then it is a pointy_object" (adapted from [4]) can be expressed as follows:

IF $isa(?x, block) \vee isa(?x, pyramid)$ THEN $add(isa(?x, pointy\_object))$

where $?x$ is a variable, and *add* adds a fact to the working memory. This rule, however, cannot be used in a typical rule-based system. Instead, it is usually replaced by two rules:

IF *isa*(?*x*, *block*) THEN *add*(*isa*(?*x*, *pointy_object*)), and
IF *isa*(?*x*, *pyramid*) THEN *add*(*isa*(?*x*, *pointy_object*)).

However, this does not say exactly the same thing as the $\vee$ version does, since there might be situations where we know that either ?*x* is a *block* or ?*x* is a *pyramid*, but do not know which. In this case, neither of these rules applies, but the original one that uses $\vee$ does.

Now, let us look at the exclusive *or*. For example, the following formula says that "either *NYC* or *albany* is the capital of *NY*, but not both".

capital(NY, NYC) $\oplus$ capital(NY, albany)

This can be rephrased as two rules: "*NYC* is the capital of *NY*, if *albany* is not", and "*albany* is the capital of *NY*, if NYC is not"

IF not(capital(NY, albany)) THEN add(capital(NY, NYC)), and
IF not(capital(NY, NYC)) THEN add(capital(NY, albany)).

Unfortunately, the *not* connective used in current rule-based systems is different from $\neg$ in logic. In a typical rule-based system, *not*(*P*) is satisfied if there is no fact in the working memory that matches the pattern *P*.

In general, disjunctions are difficult to handle in reasoning. In Section 3, we will show that CSP provides a convenient model to represent these situations.

2.2. Constraint satisfaction problem

A Constraint Satisfaction Problem (CSP) is characterized as finding values for variables subject to a set of constraints. The standard CSP has three components:

- Variables: A finite set $V = \{v_1, v_2, ..., v_n\}$ of n variables v_i, which are also referred to as constraint variables below.
- Values: Each variable v_i is associated with a finite domain D_i, which contains all the possible alternative values for v_i.
- Constraints: A set $C = \{C_1, C_2, ..., C_p\}$ of p constraints or relations on the variables.

The main approach used for solving CSPs is to embed constraint propagation (also known as consistency check) techniques in a backtrack search environment, where backtrack search performs the search for a solution and consistency check techniques prune the search space.

Consistency techniques are characterized by using constraints to remove inconsistent values from the domains of variables. Past research has produced many techniques for such a purpose. The main methods used in practice are arc consistency techniques, e.g., AC-3 [17], AC-5 [11], and AC-7 [1], and their generalizations and specializations [10, 11, 13]. For a complete treatment of these methods, please refer to [17, 8, 11, 1].

3. Modeling Disjunctions with CSP

This section shows how CSP can be used to model certain types of disjunction in a rule-based system. In this new paradigm with rules and constraints, the underlying techniques for reasoning are forward rule-chaining, constraint propagation and backtrack search.

3.1. The new paradigm

In the new paradigm, constraints are integrated into rule-based reasoning. It is described by:

1. A working memory: A set of facts representing the current state of the system. There are three types of facts:

- Simple facts: The traditional facts used in the existing rule-based systems.
- *csp*-disjunctions (inclusive and exclusive): The special types of disjunctions (defined below) represented by the CSP model.
- Constraints: The relations on the *csp*-disjunctions.

2. A rule memory: A set of *IF-THEN* rules to test and to alter the working memory.
3. A rule interpreter: It applies the rules to the working memory by using the traditional forward rule-chaining mechanism, and it is integrated with the constraint solver below.
4. A constraint solver: It uses consistency check and backtrack search for constraint satisfaction. It is integrated with the rule interpreter above.

The key difference between the new paradigm and the existing paradigm is that the CSP model and a constraint solver are included in the new paradigm. This results in the integration of forward rule-chaining and constraint solving.

3.2. Using constraint variables and domains to represent disjunctions

This sub-section describes how constraint variables and their domains can be used to represent disjunctions. Let us recall some basic definitions.

Definition 1: A term is defined inductively as follows:

(a) A variable is a term.
(b) A constant is a term.
(c) If f is a n-ary function symbol and $t_1, ..., t_n$ are terms, then $f(t_1, ..., t_n)$ is a term.

Definition 2: An atomic formula (or an atom) is an expression of the form $P(t_1, ..., t_n)$, where P is a n-ary predicate symbol, and $t_1, ..., t_n$ are terms.

Definition 3: A *ground term* is a term not containing variables. Similarly, a *ground atom* is an atom not containing variables.

We now define the two kinds of disjunctions that we will handle, the inclusive *csp*-disjunctions and exclusive *csp*-disjunctions. In what follows, we shall, for simplicity with respect to our examples later, restrict the terms in disjunctions to differ only in the last argument.

Definition 4: An exclusive *csp*-disjunction has the following form

$$\Delta(P(t_1, ..., t_{n-1}, t_{n1}), P(t_1, ..., t_{n-1}, t_{n2}), ..., P(t_1, ..., t_{n-1}, t_{nm}))$$

where $P(t_1, ..., t_{n-1}, t_{ni})$ is a ground atom with $n \geq 1$, t_{ni} is a constant, and $i \neq j$ implies $t_{ni} \neq t_{nj}$. The expression is TRUE *iff* exactly one of the ground atom is TRUE, otherwise it is FALSE. It is equivalent to the formula:

$$(P(t_1, ..., t_{n-1}, t_{n1}) \wedge \neg P(t_1, ..., t_{n-1}, t_{n2}) \wedge ... \wedge \neg P(t_1, ..., t_{n-1}, t_{nm})) \vee$$
$$(\neg P(t_1, ..., t_{n-1}, t_{n1}) \wedge P(t_1, ..., t_{n-1}, t_{n2}) \wedge ... \wedge \neg P(t_1, ..., t_{n-1}, t_{nm})) \vee \mathbf{...} \vee$$
$$(\neg P(t_1, ..., t_{n-1}, t_{n1}) \wedge \neg P(t_1, ..., t_{n-1}, t_{n2}) \wedge ... \wedge P(t_1, ..., t_{n-1}, t_{nm}))$$

Note that for all the atoms, the predicate symbols are the same, i.e., P, and so are the first n - 1 ground terms. Note also that t_{ni} may appear in any position as long as they are at the same position in each atom. We arbitrarily choose to put them at the end.

This exclusive *csp*-disjunction can be represented by $\Delta P(t_1, ..., t_{n-1}, D)$, where D is a set with the initial value $\{t_{n1}, t_{n2}, ..., t_{nm}\}$. During the reasoning process, some of the atoms (e.g., $P(t_1, ..., t_{n-1}, t_{nk})$) may be proven to be FALSE, then D will be modified to reflect the effect. Thus, D changes during the reasoning process, but it is always a subset of ($\subseteq$) of $\{t_{n1}, t_{n2}, ..., t_{nm}\}$.

When $|D| = 1$, we say D (= $\{t_{ni}\}$) is decided, which means that $P(t_1, ..., t_{n-1}, t_{ni})$ is TRUE. When D becomes $\emptyset$, it means that the exclusive *csp*-disjunction is proven to be FALSE.

According to this definition and the definition of CSP, we can see that $\Delta P(t_1, ..., t_{n-1}, D)$ can be represented with a constraint variable and its domain. D is actually the domain and $\Delta P(t_1, ..., t_{n-1}, \_)$ is the constraint variable, where "_" is just a place holder and has no significance.

Let us have an example. Suppose we wish to express that *john* is a *soldier*, or *john* is a *teacher*, or *john* is a *professor*, or *john* is a *doctor*, and that *john* is in only one of the professions, we can write $\Delta Isa(john, \{soldier, teacher, professor, doctor\})$.

The second type of *csp*-disjunction is defined below.

Definition 5: An inclusive *csp*-disjunction has the form

$$\vee(P(t_1, ..., t_{n-1}, t_{n1}), P(t_1, ..., t_{n-1}, t_{n2}), ..., P(t_1, ..., t_{n-1}, t_{nm}))$$

where $P(t_1, ..., t_{n-1}, t_{ni})$ is a ground atom with $n \geq 1$, t_{ni} is a constant, and $i \neq j$ implies $t_{ni} \neq t_{nj}$. This expression is equivalent to the following disjunctive formula:

$$P(t_1, ..., t_{n-1}, t_{n1}) \vee P(t_1, ..., t_{n-1}, t_{n2}) \vee ... \vee P(t_1, ..., t_{n-1}, t_{nm})$$

This formula is TRUE *iff* any non-empty subset of the m atoms are TRUE.

This inclusive *csp*-disjunction can be represented with $\vee P(t_1, ..., t_{n-1}, S)$. Let us see what S is. The initial value of S is the power set of $\{t_{n1}, t_{n2}, ..., t_{nm}\}$ excluding the empty set since any non-empty subset of $\{P(t_1, ..., t_{n-1}, t_{n1}), P(t_1, ..., t_{n-1}, t_{n2}), ..., P(t_1, ..., t_{n-1}, t_{nm})\}$ could be TRUE. Thus, during the reasoning process, S is always a set of sets. For convenience, we represent S by two sets (R, Q):

- A set of required elements R: the elements that have been proven to be true, i.e., whose associated atoms have been proven to be TRUE.
- A set of possible elements Q: the elements that belong to at least one possible value of S.

Then, R and Q satisfy these conditions: $R \cap Q = \emptyset$ and $R \cup Q \subseteq \{t_{n1}, t_{n2}, ..., t_{nm}\}$. The initial value of S may be $(\{\}, \{t_{n1}, t_{n2}, ..., t_{nm}\})$, and R will grow and Q will shrink in the reasoning process. When $Q = \emptyset$ and $|R| = 0$, we say the inclusive *csp*-disjunction is FALSE. When $Q = \emptyset$ and $|R| \neq 0$, we say S is decided, which means the following atoms are all TRUE:

$$P(t_1, ..., t_{n-1}, r_1), P(t_1, ..., t_{n-1}, r_2), ..., \text{and } P(t_1, ..., t_{n-1}, r_k)$$

where $R = \{r_1, r_2, ..., r_k\} \subseteq \{t_{n1}, t_{n2}, ..., t_{nm}\}$. According to this definition and the definition of CSP, we can see that $\vee P(t_1, ..., t_{n-1}, (R, Q))$ (or $\vee P(t_1, ..., t_{n-1}, S)$) can also be represented with a constraint variable and its domain. $\vee P(t_1, ..., t_{n-1}, \_)$ is the constraint variable and (R, Q) is the domain. Again, "_" is used as a place holder.

It seems that (R, Q) is quite different from a domain in CSP. But if we consider that (R, Q) represents a set of sets, then we can see that (R, Q) becomes a domain. Each value in the domain is simply a set. Because of different domain types, we distinguish two kinds of constraint variables. We call the type of constraint variables that uses (R, Q) as the domain the set constraint variables, and the other type the normal constraint variables.

Let us have an example. To express that *john* is a friend of *mike*, or *james* is a friend of *mike*, or *mary* is a friend of *mike*, or *steve* is a friend of *mike*, or *david* is a friend of *mike* (*or* is inclusive), we can write

$$\vee IsFriendOf(mike, (\{\},\{john, james, mary, steve, david\}))$$

If it is decided that *john* is definitely a friend of *mike*, then we have:

$\vee IsFriendOf(mike, (\{john\},\{james, mary, steve, david\}))$

3.3. Using constraints to represent relations among disjunctions

After introducing the two types of constraint variables to represent the two types of disjunctions, we now in the position to describe some of the constraints that can be used for representing relations among the disjunctions.

Constraint: cst_equal($\Delta P_1(t_{11}, ..., t_{1(n\text{-}1)}, \_)$, $\Delta P_2(t_{21}, ..., t_{2(m\text{-}1)}, \_)$)

where $t_{11}, ..., t_{1(n\text{-}1)}, t_{21}, ...,$ and $t_{2(m\text{-}1)}$ are all ground terms.

Assume D_1 and D_2 are the domains of the constraint variables $\Delta P_1(t_{11}, ..., t_{1(n\text{-}1)}, \_)$ and $\Delta P_2(t_{21}, ..., t_{2(m\text{-}1)}, \_)$) respectively, this constraint ensures that the sets D_1 and D_2 are equal at all time. It is defined by (this is not the real algorithm used for solving the constraint, but rather a definition):

- $D = D_1 \cap D_2$;
  if $D \neq \varnothing$ **then**
  begin
  if $D = \{v\}$ **then**
  begin
  add $P_1(t_{11}, ..., t_{1(n\text{-}1)}, v)$ to the working memory**;**
  add $P_2(t_{21}, ..., t_{2(m\text{-}1)}, v)$ to the working memory
  end
  $D_1 = D$; $D_2 = D$;
  return(TRUE);
  end
  else return(FALSE)

For example, we have

$\Delta Isa(john, \{soldier, teacher, professor, doctor\})$, and

$\Delta Isa(james, \{teacher, doctor, student\})$.

If we know that *john* and *james* have the same profession, we can express this with the constraint *cst_equal*($\Delta Isa(john, \_)$, $\Delta Isa(james, \_)$).

The system will automatically propagate the constraint by using the built-in consistency check algorithms to reduce both sets so that the following are obtained:

$\Delta Isa(john, \{teacher, doctor\})$, and $\Delta Isa(\text{james}, \{teacher, doctor\})$

If due to some other constraint (or information) it is decided that *john* is a *teacher*, then the following two elements will be added to the working memory:

Isa(*john*, *teacher*), and *Isa*(*james*, *teacher*)

If we have the following rule in the rule memory:

IF *Isa*(?*x*, *teacher*) THEN *add*(*has*(?*x*, *many_students*))

This rule will be fired to obtain two more facts:

has(*john*, *many_students*), and *has*(*james*, *many_students*)

This example shows that constraint propagation and rule chaining are integrated.

***Constraint*:** cst_not_equal($\Delta P_1(t_{11}, ..., t_{1(n\text{-}1)}, \_)$, $\Delta P_2(t_{21}, ..., t_{2(m\text{-}1)}, \_)$)

where $t_{11}, ..., t_{1(n\text{-}1)}, t_{21}, ...$, and $t_{2(m\text{-}1)}$ are all ground terms.

Assume D_1 and D_2 are the domains of the constraint variables $\Delta P_1(t_{11}, ..., t_{1(n\text{-}1)}, \_)$ and $\Delta P_2(t_{21}, ..., t_{2(m\text{-}1)}, \_)$) respectively, this constraint is defined by:

- **if** $|D_1| = 1$ and $|D_2| > 1$ **then**
  begin
  $D_2 = D_2 - D_1$;
  if $D_2 = \{v\}$ **then** add $P_2(t_{21}, ..., t_{2(m\text{-}1)}, v)$ to the working memory;
  return(TRUE)
  end
  elseif $|D_2| = 1$ and $|D_1| > 1$ **then** this case is similar to the above one;
  elseif $|D_1| = 1$ and $|D_2| = 1$ **then** **if** $D_1 \neq D_2$ **then** return(TRUE)
  else return(FALSE)
  else return(TRUE).

For instance, we have

$\Delta Isa(john, \{soldier, teacher, professor, doctor\})$, and
$\Delta Isa(james, \{teacher, doctor, student\})$.

The following constraint says that *john* and *james* have different professions:

cst_not_equal($\Delta Isa(john, \_)$, $\Delta Isa(james, \_)$)

Constraint: cst_not_in(v, $\Delta P(t_1, ..., t_{n\text{-}1}, \_)$)

where $t_1, ...$, and $t_{n\text{-}1}$ are all ground terms.

Assume D is the domain of the constraint variable $\Delta P(t_1, ..., t_{n\text{-}1}, \_)$, this constraint constrains that v is not a possible element in D, which also means that $P(t_1, ..., t_{(n\text{-}1)}, v)$ is FALSE. It is defined by:

- $D = D - \{v\}$; **if** $D = \varnothing$ **then** return(FALSE)
  else if $D = \{u\}$ (or $|D| = 1$) **then**
  add $P(t_1,..., t_{n\text{-}1}, u)$ to the working memory;
  return(TRUE)

For example, we have $\Delta Isa(john, \{soldier, teacher, professor, doctor\})$. The constraint

cst_not_in($teacher$, $\Delta Isa(john, \_)$)

will change the possible set to give $\Delta Isa(john, \{soldier, professor, doctor\})$.

Constraint: cst_set_equal($\vee P_1(t_{11}, ..., t_{1(n\text{-}1)}, \_)$, $\vee P_2(t_{21}, ..., t_{2(m\text{-}1)}, \_)$)

where $t_{11}, ..., t_{1(n\text{-}1)}, t_{21}, ...$, and $t_{2(m\text{-}1)}$ are all ground terms.

Assume (R_1, Q_1) and (R_2, Q_2) are the domains of $\vee P_1(t_{11}, ..., t_{1(n\text{-}1)},\_)$ and $\vee P_2(t_{21}, ..., t_{2(m\text{-}1)}, \_)$) respectively, this constraint is defined by:

- $R = R_1 \cup R_2$; $Q = \{r \mid r \in Q_1 \cap Q_2, r \notin R\}$;
  if $R \subseteq R_1 \cup Q_1$ and $R \subseteq R_2 \cup Q_2$ and ($R \neq \varnothing$ or $Q \neq \varnothing$) **then**
  begin
  $R_1 = R$; $R_2 = R$; $Q_1 = Q$; $Q_2 = Q$;
  for each $r \in R$ and $r \notin R_1$ **do** add $P_1(t_{11}, ..., t_{1(n\text{-}1)}, r)$ to the working memory;
  for each $r \in R$ and $r \notin R_2$ **do** add $P_2(t_{21}, ..., t_{2(m\text{-}1)}, r)$ to the working memory;
  return(TRUE);
  end
  else return(FALSE)

For example, we have

$\vee$*IsFriendOf*(*mike*, ({*john*}, {*james*, *mary*, *steve*, *david*})), and

$\vee$*IsFriendOf*(*andrew*, ({*steve*}, {*john*, *kate*, *david*}))

If we set the constraint *cst_set_equal*($\vee$*IsFriendOf*(*mike*, _), $\vee$*IsFriendOf*(*andrew*, _)), which says that *mike* and *andrew* have the same set of friends, we will obtain:

$\vee$*IsFriendOf*(*mike*, ({*john*, *steve*}, {*david*})), and

$\vee$*IsFriendOf*(*andrew*, ({*john*, *steve*}, {*david*})).

Two more facts will also be added in the working memory, i.e.,

IsFriendOf(*mike*, *steve*), and *IsFriendOf*(*andrew*, *john*).

Constraint: cst_set_not_in(v, $\vee P(t_1, ..., t_{n-1}, \_)$)

where t_1, ..., and t_{n-1} are all ground terms.

Assume (R, Q) is the domain of $\vee P(t_1, ..., t_{n-1}, \_)$, this constraint constrains that v is not a possible element in Q, which means that $P(t_1, ..., t_{n-1}, v)$ cannot be TRUE. It is defined by:

- **If** $v \notin R$ **then** **begin**
  $Q = Q - \{v\}$;
  return(TRUE)
  end
  else return(FALSE)

For instance, we have

$\vee$*IsFriendOf*(*mike*, ({*john*},{*james*, *mary*, *steve*, *david*})).

The constraint *cst_set_not_in*(*mary*, $\vee$*IsFriendOf*(*mike*, _)) will remove *mary* from the possible set to give $\vee$*IsFriendOf*(*mike*, ({*john*},{*james*, *steve*, *david*})).

3.4. Introducing choice making and backtracking

The consistency techniques used above for constraint solving are all based on arc consistency [1, 11, 15]. Arc consistency alone may not be sufficient to solve a CSP because arc consistency does not guarantee global consistency [17]. As mentioned, a combination of backtrack search and consistency check is required. This approach can be described as an iterative procedure of two steps: consistency check and choice making. If a choice is proved to be wrong (when the consistency check of a constraint returns FALSE), backtracking will be initiated. In the process, the previous state is restored, and an alternative is selected [10].

However, introducing backtrack search creates some complications because of their integration with forward chaining. This will be discussed in Section 3.6.

First, we define some choice making functions. Each of these functions also sets up a choice point for later backtracking. The choice making functions are actually also constraints because each value selection will trigger consistency check.

Choice function: cst_select($\Delta P(t_1, ..., t_{n-1}, \_)$, *func*)

where t_1, ..., and t_{n-1} are all ground terms, and *func* is a user defined procedure.

Assume D is the domain of $\Delta P(t_1, ..., t_{n-1}, \_)$, this function selects a value v from D using the procedure *func*. *func* allows the user to control the selection process in order to find the solution quickly. The choice function is defined by:

- **if** there is no more value to be selected in D **then** return(FALSE)
  else **begin**
  v is selected from D using *func*;
  $D = \{v\}$;
  add $P(t_1, ..., t_{n-1}, v)$ in the working memory;
  return(TRUE)
  end

For example, we have $\Delta Capital(NY, \{NYC, albany\})$, which says that the capital of *New York* (*NY*) is either *NYC* or *albany*, but not both. We can apply the selection by using

cst_select($\Delta Capital(NY, \_)$, *func*).

Suppose that *func* chooses the first possible value first, i.e., *NYC*. After it is selected, *Capital*(*NY*, *NYC*) will be automatically added in the working memory, and then constraint propagation will be carried out, etc. When backtracking occurs, the second value will be tried and so on.

Choice function: cst_set_select($\vee P(t_1, ..., t_{n-1}, \_)$, *func*)

where t_1, ..., and t_{n-1} are all ground terms, and *func* is a user defined procedure.

Assume (R, Q) is the domain of $\vee P(t_1, ..., t_{n-1}, \_)$, this function selects a value V (a set) from Q ($V \subseteq Q$) using the procedure *func*. It is defined by:

- **if** there is no more value (a set) to be selected from Q **then** return(FALSE)
  else **begin**
  V is selected from Q using *func*;
  $Q = \varnothing$; $R = R \cup V$;
  for each $r \in V$ **do** add $P(t_1, ..., t_{n-1}, r)$ to the working memory
  end

For instance, we have $\vee IsFriendOf(mike, (\{john\}, \{james, mary, steve, david\}))$ and we know that *mike* has only two friends. We can try the following:

cst_set_select($\vee IsFriendOf(mike, \_)$, *func*)

Suppose that *func* chooses the first possible value first, i.e., *james*, which effectively rules out the other three values, i.e., *mary*, *steve* and *david*. Then, *mike*'s friends are *john* and *james*. We obtain $\vee IsFriendOf(mike, (\{john, james\}, \{\}))$.

After that, other necessary operations are performed, e.g., adding *IsFriendOf*(*mike*, *james*) to the working memory and constraint propagation, etc. When a selection is proved to be wrong, backtracking will be performed. The second element, the third element, etc., will be tried and so on.

3.5. Some test functions on constraint variables

Here, we present some test functions on constraint variables. These test functions are used to exploit the partial information provided by disjunctions for various purposes.

Test function: test_in(T, $\Delta P(t_1, ..., t_{n-1}, \_)$)

where t_1, ..., and t_{n-1} are all ground terms, and T is a set of constants.

Assume D is the domain of $\Delta P(t_1, ..., t_{n-1}, \_)$, this test function is defined by:

- **if** $D \subseteq T$ **then** return(TRUE) **else** return(FALSE)

For example, we have Δ*Capital*(*NY*, {*NYC*, *albany*}), which says that the capital of *New York* (*NY*) is either *NYC* or *albany*, but not both, and the following rule:

IF *include*(?*tour*, *capitalOf*(*NY*)) and *test_in*({*NYC*, *albany*}, Δ*Capital*(*NY*, _))
THEN *add*(*join*(*I*, ?*tour*)))

This rule allows the system to act on the partial information, i.e., *test_in* does not have to find the fact *Capital*(*NY*, *NYC*) or *Capital*(*NY*, *albany*) in the working memory before firing. Instead, it only needs to check whether any one of these two cities or both are the only possible values for the capital of *NY*. It does not matter which.

If the working memory has the following two facts:

include(*tour*16, *capitalOf*(*NY*)), and Δ*Capital*(*NY*, {*NYC*, *albany*})

the rule will fire to add *join*(*I*, *tour*16)) to the working memory.

Test function: test_set_in(T, $\vee P(t_1, ..., t_{n-1}, \_)$)

where t_1, ..., and t_{n-1} are all ground terms, and T is a set of constants.

Assume (R, Q) is the domain of $\vee P(t_1, ..., t_{n-1}, \_)$, This test function is defined by:

- **if** $(T \cap R) \neq \varnothing$ or ($R = \varnothing$ and $Q \subseteq T$) **then** return(TRUE) **else** return(FALSE)

For example, we wish to express that "if something is a block or a pyramid, then it is a pointy_object" (*or* is inclusive) can be expressed as follows:

IF *test_set_in*({*block*, *pyramid*}, *isa*(?*x*, _)) THEN *add*(*isa*(?*x*, *pointy_object*))

3.6. Complications with the integration of choice making and rule chaining

Combining backtrack search and forward chaining creates some complications. The problem lies in the handling of inconsistency in reasoning. For our discussion, we classify two types of inconsistency. The first type is the normal inconsistency in logic (IL), e.g., both A and $\neg A$ are deduced, and the other is the inconsistency of constraints (IC). IC is easy to detect and to handle because when the domain of a constraint variable is empty, it is known that there is a inconsistency, and backtracking can be used to deal with it. However, IL is hard to detect as most rule-based systems are informal systems that have no mechanism for this purpose. This has some implications for our proposed integration.

- If a rule-based system is unable to detect IL, then (1) constraints cannot be conditions in a rule, (2) choice making and backtracking should not be allowed.

  Let us see why (1) is so. For example, we have the following rule

  IF *studyInSameCollege*(?*x*, ?*y*) and *cst_equal*(Δ*Isa*(*?x*, _), Δ*Isa*(*?y*, _))
  THEN *add*(*studyInSameDept*(?x, ?y))

  The conditional part of this rule has a constraint. When the constraint is first encountered, it may not be solved because its variable domains still have more than one value. For instance, assume we have the following in the working memory:

  Δ*Isa*(*john*, {*soldier*, *teacher*, *professor*, *doctor*})
  Δ*Isa*(*james*, {*teacher*, *doctor*, *student*})
  studyInSameCollege(*john*, *james*)

  After the two conditions of the above rule are matched and evaluated, both the domains of Δ*Isa*(*john*, _) and Δ*Isa*(*james*, _) become {*teacher*, *doctor*}. Since the consistency check will return TRUE, this rule will be fired to add the fact *studyInSameDept*(*john*, *james*) to the working memory. However, this inference may not be correct because the CSP involving the constraint may be inconsistent because arc consistency does not guarantee global consistency [17]. Since the inference cannot be guaranteed to be correct, then IL and/or IC, may be introduced.

Similarly, (2) is so because choice making is only a guess, which may proven to be wrong later if it introduces IL and/or IC.

In general, if a rule-based system is unable to detect IL, (1) and (2) should not be allowed. Then, constraints can only appear as consequents of rules, and there will be no backtrack search but only consistency check. Although this is incomplete, it is sound because a rule is basically an implication in Logic and arc consistency is sound (although not complete without choice making and backtracking) [10]. This situation may be somewhat restrictive, but it is still better than without constraints.

However, if an inconsistency checker is implemented in a rule-based system for detecting IL, then both (1) and (2) can be allowed, and both types of inconsistency will trigger backtracking. For (1) to work, we need to make a choice point with each rule that has constraints as its conditions. This simply means that backtracker can backtrack to the rule and retract its efects when a conditional constraint in a rule is proven to be wrong because of inconsistency.

Apart from the above two situations, a third one is also reasonable. We assume that only ICs may occur in an application, then we can also allow both (1) and (2) because IC is easily detected and handled. Our prototype system makes this assumption. This assumption is realistic because that is the case in most existing rule-based systems. They do not have mechanisms to detect IL. It is the user's responsibility not to introduce any or to check it if IL may be introduced. For a specific application, special rules can be programmed to check restricted types of IL.

4. An Implementation

We have implemented a prototype system (called CFR) in Lisp. Below are some implementation issues.

- Apart from the working memory and rule memory in a rule-based system, a constraint variable memory is introduced to store constraint variables. Constraints are attached to constraint variables in the same way as in a constraint solver [10, 12].
- For consistency check of constraints involving normal constraint variables, we used those algorithms in [10, 15] as they are the most efficient algorithms. For the set constraints, the algorithms are designed by us as there is no reported work on this type of constraints. Consistency check of *cst_equal*, *cst_not_equal*, and *cst_set_equal* can all be done in linear time to the size of the domain D or $|R \cup Q|$. *cst_not_in* and *cst_set_not_in* can be done in constant time.
- A choice stack is used to keep track of the choices that have been made and to remember the information necessary for restoring state upon backtracking. This is similar to CLP languages such as CHIP [10]. The difference is that each choice here has to remember the facts that have been added to the working memory after a choice is made. When backtracking comes to the choice, these facts must be removed.

- Finally, the pattern matching algorithm for rule-chaining needs to be modified to accommodate the constraint satisfaction facility. Due to the space limitation, we are unable to discuss this and many other issues in details.

Below, we briefly describe the syntax of rules, constraint variables, and constraints in CFR.

***IF-THEN* rules**: A rule is defined using the following construct:

(define-rule <rule name> <conditions> -> <actions>)

For example, the rule:

(define-rule edible_food (edible ?x) -> (add `(is_food ,x)))

says that if there is a fact in the working memory that matches (*edible ?x*), this rule will fire and add the evaluation result `(*is_food ,x*) to the working memory. `(*is_food ,x*) is in Lisp syntax ("`", "'", and "," are used according to their meanings in Lisp), and *x* here will be substituted to whatever value ?*x* has after matching with a fact in working memory.

Constraint variable declarations:

1). $\Delta P(t_1, ..., t_{n-1}, D)$ => (corresponding to) (cst_in '($P\ t_1 \ldots t_{n-1}\ D$))

e.g., $\Delta capital(NY, \{NYC, albany\})$ => (cst_in '(capital NY (NYC albany)))

2). $\vee P(t_1, ..., t_{n-1}, (R, Q))$ => (cst_set_in '($P\ t_1 \ldots t_{n-1}\ (R\ Q)$))

e.g., $\vee IsFriendOf(andrew, (\{steve\},\{john, kate, david\}))$ =>
(cst_set_in '(IsFriendOf andrew ((steve) (john, kate, david))))

Constraints:

1). $\textit{cst\_equal}(\Delta P_1(t_{11}, ..., t_{1(n-1)}, \_), \Delta P_2(t_{21}, ..., t_{2(m-1)}, \_))$ =>
(cst_equal '($P_1\ t_{11} \ldots t_{1(n-1)}\ \_$) '($P_2\ t_{21} \ldots t_{2(m-1)}\ \_$))

e.g., $\textit{cst\_equal}(\Delta Isa(john, \_), \Delta Isa(james, \_))$ =>
(cst_equal '(Isa john _) '(Isa james _))

2). $\textit{cst\_not\_equal}(\Delta P_1(t_{11}, ..., t_{1(n-1)}, \_), \Delta P_2(t_{21}, ..., t_{2(m-1)}, \_))$ =>
(cst_not_equal '($P_1\ t_{11} \ldots t_{1(n-1)}\ \_$) '($P_2\ t_{21} \ldots t_{2(m-1)}\ \_$))

e.g., $\textit{cst\_not\_equal}(\Delta Isa(john, \_), \Delta Isa(james, \_))$ =>
(cst_not_equal '(Isa john _) '(Isa james _))

3). $\textit{cst\_not\_in}(v, \Delta P(t_1, ..., t_{n-1}, \_))$ => (cst_not_in v '($P\ t_1 \ldots t_{n-1}\ \_$))

e.g., $\textit{cst\_not\_in}(teacher, \Delta Isa(john, \_))$ => (cst_not_in 'teacher '(isa john _))

4). $\textit{cst\_set\_equal}(\vee P_1(t_{11}, ..., t_{1(n-1)}, \_), \vee P_2(t_{21}, ..., t_{2(m-1)}, \_))$ =>
(cst_set_equal '($P_1\ t_{11} \ldots t_{1(n-1)}\ \_$) '($P_2\ t_{21} \ldots t_{2(m-1)}\ \_$))

e.g., $\textit{cst\_set\_equal}(\vee IsFriendOf(mike, \_), \vee IsFriendOf(andrew, \_))$ =>
(cst_set_equal '(IsFriendOf mike _) '(IsFriendOf andrew _))

5). $\textit{cst\_set\_not\_in}(v, \vee P(t_1, ..., t_{n-1}, \_))$ => (cst_set_not_in v '($P\ t_1 \ldots t_{n-1}\ \_$))

e.g., $\textit{cst\_set\_not\_in}(mary, \vee IsFriendOf(mike, \_))$ =>
(cst_set_not_in 'mary '(IsFriendOf mike _))

The choice and test functions also have their similar corresponding constructs in CFR. Due to the space limitation, we will not list them here.

5. An Example

We now present a simple example to illustrate how rules and constraints interact with each other in the reasoning process. The rule definitions here are self-explanatory.

```
(define-rule professor
      (isa ?x science_professor)
      ->     (add `(works_in_a ,x university))
             (cst_in `(teaches ,x (computer math physics chemistry biology))))
 (define-rule computer
      (isa ?x science_professor)
      (has_no ?x computer)
      ->     (cst_not_in 'computer `(,x teaches _)))
 (define-rule math
      (is_good_in ?x math)
      (isa ?x science_professor)
      ->     (cst_in `(teaches ,x (computer math physics))))
 (define-rule csp-test
      (test_in (physics math) (teaches ?x _))
      ->     (add `(gives_lecture_in ,x science_building)))
(define-rule lab
      (does_not_do ?x lab_work)
      ->     (cst_not_in 'chemistry `(teaches ,x _))
             (cst_not_in 'biology `(teaches ,x _)))
 (define-rule degree
      (teaches ?x ?y)
      ->     (add `(likes ,x ,y))
             (cst_set_in `(has ,x `(((PhD in ,y)) `((MSc in ,y))))))
```

Let us run the system with the following facts:

```
(add '(isa fred science_professor))
(add '(has_no fred computer))
(add '(isa john science_professor))
(add '(does_not_do john lab_work))
(cst_equal '(teaches john _) '(teaches fred _))
```

After all the rule chaining and constraint propagation, the working memory becomes:

```
1:  (isa fred science_professor)
2:  (works_in_a fred university)
3:  (cst_fact (teaches fred _) (math physics))
4:  (has_no fred computer)
5:  (isa john science_professor)
6:  (works_in_a  john university)
7:  (cst_fact (teaches john _) (math physics))
8:  (does_not_do john lab_work)
9:  (gives_lecture_in john science_building)
10: (gives_lecture_in fred science_building)
```

Fact 3 and 7 are special facts representing two constraint variables and their remaining domains. From them, we know that both *fred* and *john* teach either *math* or *physics*, but we still do not know which.

Let us say that we are not satisfied with the result. We would like to make a guess about what they teach. We can use the following selection function:

(cst_select '(teaches fred _) #'car)

This selects *math* as the subject that *fred* teaches. After constraint propagation and rule chaining, we obtain the fact that *john* also teaches *math*. The following facts are deduced:

11: (teaches fred math)
12: (teaches john math)
13: (likes fred math)
14: (likes john math)
15: (has fred (PhD in math))
16: (has john (PhD in math))
17: (cst_set_fact (has fred _) ((PhD in math)) ((MSc in math)))
18: (cst_set_fact (has john _) ((PhD in math)) ((MSc in math)))

The last two facts (17 and 18) say that *fred* and *john* have a *PhD* in *math* and may or may not have a *MSc* in *math*.

If later we have some more information saying that *fred* does not have a *PhD* degree in *math*, this can be expressed like this:

(cst_set_not_in '(PhD in math) '(has fred _))

It immediately causes a conflict with fact 17 because fact 17 says that *fred* has a *PhD* in *math*. Then, backtracking is performed. The facts from 11 to 18 are removed to restore the previous state. *physics* is selected this time as the subject that *fred* teaches, which in turn causes a number of facts to be produced:

11: (teaches fred physics)
12: (teaches john physics)
13: (likes fred physics)
14: (likes john physics)
15: (has fred (PhD in physics))
16: (has john (PhD in physics))
17: (cst_set_fact (has fred _) ((PhD in physics)) ((MSc in physics)))
18: (cst_set_fact (has john _) ((PhD in physics)) ((MSc in physics)))

Since *math* is eliminated as the possible course that *fred* and *john* teach. Fact 3 and 7 in the working memory become:

3: (cst_fact (teaches fred _) (physics))
7: (cst_fact (teaches john _) (physics))

The kind of reasoning illustrated here cannot be carried out in an existing rule-based system.

6. Related Work

The most closely related work to our research is constraint logic programming (CLP) [14] where a considerable amount of research has been done to integrate constraint satisfaction with Logic programming. A number of systems have been built, and many successful applications have also been reported [14]. Two representative CLP langauges are CLP(R) [13] and CHIP [10]. These languages are based on Horn clauses and backward chaining. Our work is different from CLP in a number of ways. The main differences are as follows.

1. Our proposed technique is based forward chaining rather backward chaining as in CLP languages. Forward chaining and backward chaining reason from different directions and are suitable for solving different types of problems. Forward chaining are mainly used for building expert systems for solving real-life knowledge intensive tasks. Since the CLP languages based on backward chaining have been very successful in practice for solving practical combinatorial search problems, it is only natural that forward chaining should also be integrated with constraint solving to provide a more powerful reasoning technique for solving practical reasoning problems. This work is the first step toward this direction.
2. In CLP languages, backtracking and choice making are provided by the host language Prolog. While in forward chaining, backtracking and choice making facilities have to be added, which creates some complications as discussed in Section 3.6.

To the best our knowledge, limited work has been done on combining constraint solving with forward chaining rule-based system. BABYLON [5] is one of the hybrid environments for developing expert systems that has attempted to include constraint solving in its rule-based system. BABYLON provides representation formalisms of objects, rules, Prolog and constraints. CONSAT is the constraint system of BABYLON, which is separated from others and cannot access rules. Although in the condition part of the rules, it is possible to verify whether a constraint is satisfied, the action part of a rule cannot access constraints. This is quite different from our system, within which constraint solving and rule-chaining are fully integrated. Rules can post and test constraints, and constraint satisfaction can also trigger off chaining of rules.

7. Conclusion

This paper shows how CSP can be used to model two types of important disjunctions in rule-based reasoning. These disjunctions have not been handled satisfactorily in the current rule-based systems, which limited their problem solving power and expressiveness. In the proposed scheme, the simple representation and efficient algorithms produced in the CSP research are used to deal with these types of disjunctions. This results in the integration of two important types of reasoning techniques, i.e., constraint propagation and (forward) rule-chaining. Hence. the power of rule-based systems is increased. From a cognitive science point of view, we feel that this integration is also attractive as human reasoning processes exploit constraint satisfaction mechanisms as well as exhibit rule-chaining like behaviors.

The current integration of CSP with rule-based system is still restricted, i.e., mainly for modeling the two types of disjunctions. Our next step is to integrate the full power of CSP with a forward chaining rule-based system.

References

[1] C. Bessiere, E. C. Freuder, and J-C. Regin, “Using inference to reduce arc consistency computation,” *IJCAI-95*, 1995, 592-598.

[2] W. Bibel, "Constraint satisfaction from a deductive viewpoint," *Artificial Intelligence* 35, 1988, 401-413.

[3] L. Brownston, R. Farrell, E. Kant and N. Martin, *Programming Expert Systems in OPS5* Addison-Wesley Publishing Company, 1985.

[4] E. Charniak, C.K. Riesbeck, D.V. McDermott and J.R. Meehan, *Artificial Intelligence Programming*, Lawrence Erlbaum Associates Inc., 1987.

[5] T. Christaller, F. di Primio, U. Schnepf and A. Voss, *The AI Workbench BABYLON* Academic Press Ltd., 1992.

[6] M. Dincbas, *et al*, "Solving large combinatiorial problem in logic programming," *J. Logic Prggramming* 8, 1990, 75-93.

[7] J. de Kleer, "A comparison of ATMS and CSP techniques," *IJCAI-89*, 1989, 290-296.

[8] E.C. Freuder, "Synthesizing constraint expression," *Communications of the ACM* 21 (11), 1978, 958-966.

[9] J. Giarratano, and G. Riley, *Expert Systems: principles and programming*, PWS Publishing Company, 1994.

[10] P.V. Hentenryck, *Constraint Satisfaction in Logic Programming*, MIT Press, 1989.

[11] P.V. Hentenryck, Y. Deville and C-M. Teng, "A generic arc consistency algorithm and its specializations," *Artificial Intelligence* 27, 2-3, 1992, 291-322.

[12] Ilog Solver, 1992, *Reference Manual*, ILOG, France.

[13] J. Jaffar, and J.L. Lassez, "Constraint logic programming," *Proceedings of the Fourteenth Annual ACM Symposium on Principle of Programming Language*, 1987.

[14] J. Jaffar, and M. Maher, “Constraint logic programming: a survey.” *J. Logic Programming* 19, 1994, 503-581.

[15] B. Liu, “Increasing functional constraints need to be checked only once,” *Proceedings of IJCAI-95*, 1995, 586-591.

[16] P. Lucas and L. Van Der Gaag, *Principles of Expert Systems*, Addison-Wesley Publishing Company, 1991.

[17] A.K. Mackworth, "Consistency in networks of relations," *Artificial Intelligence* 8, 1977, 99-118.

[18] A.K. Mackworth, "The logic of constraint satisfaction," *Artificial Intelligence* 58, 1992, 3-20.

[19] D.A. McAllester, "Truth maintenance," *AAAI-90*, 1990, 1109-1116.

[20] R. Mohr and T.C. Henderson, "Arc and path consistency revisited," *Artificial Intelligence* 28, 1986, 225-233.

[21] A. Newell, “Production systems: models for control structure,” In *Visual Information Processing*, W.G. Chase (Eds), Academic Press, 1973.

[22] A. Newell and H.A. Simon, *Human Problem Solving*, Englewood Cliffs, NJ: Prentice-Hall, 1972.

[23] L Wos, R Overbeek, E. Lusk, and J. Boyle, *Automated Reasoning: introduction and applications*, 2rd ed., McGraw-Hill, 1992.