CS5202 Constraint-Based Programming

Paul Ahern

May 31, 2007

Abstract

Notes from lectures.

Contents

1 Key Facts to Know

1.1 Exam

1. Strong k-consistency and Freuder's Theorem.
2. Trees, unconnected components and restricted width.
3. Singleton Arc Consistency.
4. SAC-1 Algorithm.

2 Lecture 28 September 2006

The final exam will be in the summer examination period unless everybody agrees to do it in January.

As usual, you are expected to demonstrate that you know what the course is all about.

You are allowed to bring in an A4 size cribsheet with hand-written notes. One side of the sheet should remain empty and the sheet has to be signed by your lecturer.

Without a signature, your cribsheet will be invalid.

2.1 Constraints

Constraints are a declarative tool for specifying, representing, solving, and reasoning about many interesting problems occurring in computer science, mathematics, and the real world.

Constraints were first used by AI researchers as a special tool for polyhedral scene analysis.

Example applications:

- Graphics polyhedral scene analysis, GUI positioning, CAD Tool.
- Operations Research planning, scheduling, other optimization problems.
- Molecular Biology DNA Sequencing, phylogenetic trees, protein folding.
- . . .

2.2 Constraint Satisfaction Problem (CSP)

A CSP is made up of:

- n Variables $x_1 \ldots x_n$
- in Domains $D_{x_1} \ldots D_{x_n}$ (e.g. Boolean domain {true,false} or {1,0})
- Constraints that limit the possible values that can be taken by the variables (e.g. $(x_1 \land x_2) \land (\overline{x}_1 \lor x_3)$)

Example CSPs:

- Position n-Queens on a chessboard such that they do not threaten each other.
- Align sequences of DNA so as to minimize the number of mismatches.
- Scheduling problems such as assigning aircrew to planes.
- Solving the Zebra problem

3 Lecture 29 September 2006

Apt Book: pages 1-13.

3.1 Basic Concepts

Representing a given problem as a CSP is called *modeling*.

Note that $a < b \land b < c$ and $a < b \land b < c \land a < c$ are different but *equivalent* models of the same problem. In general there may be several models of the same problem.

CSPs can be classified according to the domain from which the variables take their values. Hence integer ($\mathbb{N}$), real ($\mathbb{R}$), Boolean ($\mathbb{B}$) and symbolic CSPs.

The notion of a variable in constraint satisfaction is similar to the notion of a variable in logic programming. It is *not* similar to the notion of a variable in an imperative programming language like C. Variables in constraint satisfaction are more like variables in mathematics. They are placeholders for values in expressions.

3.1.1 Formal Definitions

Let $Y = y_1, \ldots, y_k$ be a non-empty finite sequence of *variables* $(k > 0)$.

Let D_i be the domain of y_i, for $1 \leq i \leq k$. To say that D_i is the domain of y_i means that y_i takes its values from D_i.

A constraint C on Y is a subset of the Cartesian product, $D_1 \times \ldots \times D_k$, of the domains of Y .

Here Cartesian product of sequence $D_1 \times \ldots \times D_k$ is defined as follows:

$D_1 \times \ldots \times D_k = \{(v_1, \ldots, v_k) : v_1 \in D_1, \ldots v_k \in D_k\}$

C is called unary if $k = 1$ and binary if $k = 2$. If $k \geq 3$ then C is usually called a higher-order constraint.

By Constraint Satisfaction Problem (CSP) we mean:

1. A finite sequence X of variables $x_1, \ldots, x_k$;
2. A domain D_i for each variable x_i, $1 \leq i \leq k$;
3. A finite set C of constraints such that each constraint is on a *subsequence* of X.

Formally, we write $\langle C; \mathcal{DE} \rangle$ for this CSP, where $\mathcal{DE} := x_1 \in D_1, \ldots x_k \in D_k$.

4 Lecture 5 October 2006

Apt Book: pages 13-48.

Let $\langle C; \mathcal{DE} \rangle$ for this CSP, where $\mathcal{DE} := x_1 \in D_1, \ldots x_n \in D_n$. Furthermore, let $C \in \mathcal{C}$ be a constraint on $x_{i_1}, \ldots x_{i_k}$. Let $m \geq k$. We say that an m-tuple $(v_{i_1}, \ldots v_{i_k}) \in D(x_{j_1}) \times \ldots \times D(x_{j_m})$ *satisfies* C iff each of the following are true:

1. The sequence $x_{i_1}, \ldots x_{i_k}$ is a sub-sequence of $x_{i_1}, \ldots x_{i_m}$;
2. $(v_{i_1}, \ldots v_{i_k}) \in C$.

An m-tuple $(v_{i_1}, \ldots v_{i_k}) \in D(x_{j_1}) \times \ldots \times D(x_{j_m})$ is called a *solution* of $\langle C; \mathcal{DE} \rangle$ iff it satisfied each $C \in \mathcal{C}$.

4.1 Examples

- Integer CSPs - Send+More=Money; n-Queens.
- Real-Valued CSPs - Spreadsheets; Zeros of Polynomials.
- Boolean CSPs - special instance of Integer CSPs $\{0..1\}$.
- Symbolic CSPs - Crosswords; Qualitative Temporal Reasoning; Analysis of Polyhedral Scenes.
- Constrained Optimization Problems - Knapsack; Coins, Golomb Ruler (the distance between each pair of ticks should be different).

4.1.1 A Coins Problem

What is the minimum number of coins that lets you pay exactly any amount smaller than one euro.

Use integer variables i_{k_j}, for $1 \leq k \leq 99$ and $j \in \{1, 2, 5, 10, 20, 50\}$ such that $i_{k_j} \in D(x_j)$. The domains are:

$x_1 \in \{0..99\}, x_2 \in \{0..49\}, x_5 \in \{0..19\}$,

$x_{10} \in \{0..9\}, x_{20} \in \{0..4\}, x_{50} \in \{0..1\}$.

For each k we have the constraints:

$i_{k_1} \leq x_1, i_{k_2} \leq x_2, i_{k_5} \leq x_5, i_{k_{10}} \leq x_{10}, i_{k_{20}} \leq x_{20}, i_{k_{50}} \leq x_{50}$,

$1 \times i_{k_1} + 2 \times i_{k_2} + 5 \times i_{k_5} + 10 \times i_{k_{10}} + 20 \times i_{k_{20}} + 50 \times i_{k_{50}} = k$.

5 Lecture 6 October 2006

5.1 OPL Studio

ILOG product implementing the Optimization Programming Language.

5.2 n-Queens program

```
int n << "Number of queens: ";
range Position 1..n;
var Position pos[ Position ];
solve {
   forall( q1, q2 in Position : q1 < q2 ) {
      pos[ q1 ] <> pos[ q2 ];
      pos[ q1 ] + q1 <> pos[ q2 ] + q2;
      pos[ q1 ] - q1 <> pos[ q2 ] - q2;
   }
};
```

6 Lecture 12 October 2006

Apt Book: pages 54-74.

6.1 Projections

Given: variables $X := x_1, ..., x_n$ with the domains $D_1, ..., D_n$.

Consider

- $d := (d_1, ..., d_n) \in D_1 \times ... \times D_n$,
- subsequence $Y := x_{i_1}, ..., x_{i_l}$ of X.

Denote $(d_{i_1}, ..., d_{i_l})$ by $d[Y]$. $d[Y]$: *projection* of d on Y.

Note For a CSP $\mathcal{P} := \langle \mathcal{C}; x_1 \in D_1, ..., x_n \in D_n \rangle$
$(d_1, ..., d_n) \in D_1 \times ... \times D_n$ is a solution to $\mathcal{P}$ iff for each constraint C of $\mathcal{P}$ on a sequence of variables Y: $d[Y] \in C$.

6.2 Equivalence of CSPs

- $\mathcal{P}_1$ and $\mathcal{P}_2$ are equivalent if they have the same set of solutions.
- CSPs $\mathcal{P}_1$ and $\mathcal{P}_2$ are equivalent w.r.t. X iff
  $\{d[X] \mid d$ is a solution to $\mathcal{P}_1\} = \{d[X] \mid d$ is a solution to $\mathcal{P}_2\}$.
- Union of $\mathcal{P}_1, ..., \mathcal{P}_m$ is equivalent w.r.t. X to $\mathcal{P}_0$ if
  $\{d[X] \mid d$ is a solution to $\mathcal{P}_0\} = \{\cup_{i=1}^{m} d[X] \mid d$ is a solution to $\mathcal{P}_i\}$.

6.3 Solved and Failed CSPs

- C a constraint on variables $y_1, ..., y_k$ with domains $D_1, ..., D_k$, so $C \subseteq D_1 \times ... \times D_k$. C is solved if $C = D_1 \times ... \times D_k$.
- CSP is solved if
  - all its constraints are solved,
  - no domain of it is empty.
- CSP is failed if
  - it contains the false constraint $\perp$,
    or
  - some of its domains are empty.

6.4 Basic Framework for Constraint Programming

First the initial problem is formulated as a CSP (modeling). Then the generic procedure Solve:

```
Solve:
VAR continue: BOOLEAN;
continue:= TRUE;
WHILE continue AND NOT Happy DO
  Preprocess;
  Constraint Propagation;
  IF NOT Happy
  THEN
    IF Atomic
    THEN
      continue:= FALSE
    ELSE
      Split;
      Proceed by Cases
    END
  END
END
```

Solve contains the subsidiary procedures Preprocess, Constraint Propagation, Happy, Atomic, Split and Proceed by Cases.

6.4.1 Preprocess

Bring to desired syntactic form. Examples:

- Constraints on reals.
  Desired syntactic form: no repeated occurrences of a variable.

$$\frac{ax^7 + bx^5y + cy^{10} = 0}{ax^7 + z + cy^{10} = 0, bx^5y = z}$$

- Boolean Constraints.
  Desired syntactic form: conjunctive normal form.

$$(x \vee y \vee z) \wedge (\neg x \vee y) \wedge (x \vee \neg z)$$

6.4.2 Happy

- Found a solution,
- Found all solutions,
- Found a solved form from which one can generate all solutions,
- Determined that no solution exists (inconsistency),
- Found best solution,
- Found all best solutions.
- Reduced all interval domains to sizes smaller than the required precision.

6.4.3 Atomic

Check

- whether CSP is amenable for splitting, or
- whether search “under” this CSP is still needed.

Atomic tell us when we have to start splitting. The idea is that if a CSP is "atomic" then we can tell immediately if there is a solution or not. For example, this may be the case if all domains are singletons.

6.4.4 Split

Split a domain

- D finite (Enumeration)

$$\frac{x \in D}{x \in \{a\} \mid x \in D - \{a\}}$$

- D finite (Labeling)

$$\frac{x \in \{a_1, ..., a_k\}}{x \in \{a_1\} \mid ... \mid x \in \{a_k\}}$$

- D interval of reals (Bisection)

$$\frac{x \in [a..b]}{x \in [a..\frac{a+b}{2}] \mid x \in [\frac{a+b}{2}..b]}$$

- Split a constraint

$$\frac{\mid p(\bar{x}) \mid = a}{p(\bar{x}) = a \mid p(\bar{x}) = -a}$$

$$\frac{C_1 \vee C_2}{C_1 \mid C_2}$$

Each call to Split replaces current CSP $\mathcal{P}$ by CSPs $\mathcal{P}_1, ...\mathcal{P}_n$ such that the union of $\mathcal{P}_1, ...\mathcal{P}_n$ is equivalent to $\mathcal{P}$.

Split also determines which operation is to be applied next.

Some heuristics for Split, Which

- variable to choose,
- value to choose,
- constraint to split.

Examples:

- Select a variable that appears in the largest number of constraints (most constrained variable).
- For a domain being an integer interval: select the middle value.

6.4.5 Proceed by Cases

When given a finite number of CSPs, the task of the procedure ProceedByCases is to recursively solve these CSPs (using Solve) until - and this depends on the task that we're trying to accomplish - one solution, an optimal solution or all solutions have been found.

Usually, this is done with some form of backtracking algorithm. The algorithm regards a single non-splittable CSP as a leaf of a tree and the union of $k > 1$ splittable CSPs as a node of a tree with k branches, one for each of the k CSPs.

This defines a so-called search tree (whose solutions may be found by traversing it). Usually (almost always) the tree is not given in advance but computed on the fly.

Various search techniques.

- Backtracking,
  - Nodes generated "on the fly".
  - Nodes are CSPs.
  - Leaves are CSPs that are solved or failed.
- Branch and bound,
- Can be combined with Constraint Propagation,
- Intelligent backtracking.

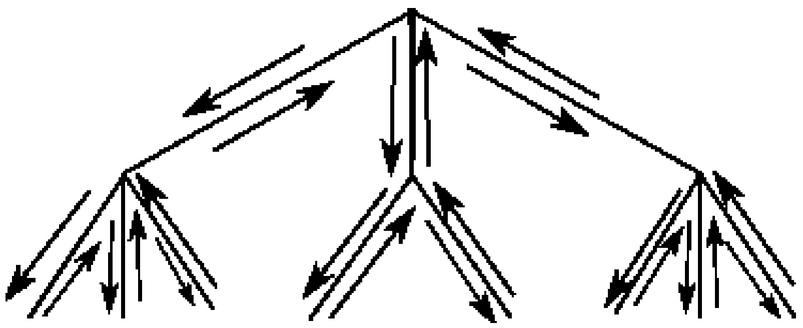

Figure 1: Backtracking

6.4.6 Constraint Propagation

Intuition: Replace a CSP by an equivalent one that is *simpler.*
Constraint propagation performed by repeatedly *reducing*

- domains

and/or

- constraints

while maintaining *equivalence.*

Projection Consider any constraint C and any domain D of variable x. The following is a typical example of constraint propagation: Remove from D all values for x that do not participate in any solution to C.
The idea is that if a value does not participate in any solution to C then it cannot participate in any solution of the CSP and can therefore be discarded. The act of removing from D all values for x that do not participate in any solution to C is called *projecting C onto x.*

Resolution Rule Is the cornerstone of automated theorem proving. It is an example of constraint propagation.
The idea is to derive new rules from a given Boolean formula in CNF. It is hoped that the new rules tighten the domains of the variables. Let L be any literal and $\bar{L}$ its negation. The following resolution rule derives the constraint $C_1 \cup C_2$which does not contain the literals:

$$\frac{\langle C_1 \cup \{L\}, C_2 \cup \{\bar{L}\}; \mathcal{DE}\rangle}{\langle C_1 \cup \{L\}, C_2 \cup \{\bar{L}\}, C_1 \cup C_2; \mathcal{DE}\rangle}$$

6.4.7 Constraint Propagation Algorithms

- Deal with scheduling of atomic atomic reduction steps.
- Try to avoid useless applications of atomic reduction steps
- Stopping criterion for general CSPs: a local consistency notion. For example: Projection rule - Take a constraint C. Choose a variable x of it with domain D. Remove from D all values for x that do not participate in a solution to C. Corresponding local consistency notion - Hyper-arc consistency - For every constraint C and every variable x with domain D, each value for x from D participates in a solution to C.

7 Lecture 13 October 2006

Apt Book: pages 82-107.

7.1 Proof Theoretic Framework

7.1.1 Proof Rules

Rules that transform CSPs will be written in the form

$$\frac{\langle \mathcal{C}; \mathcal{DE} \rangle}{\langle \mathcal{C}'; \mathcal{DE}' \rangle}$$

A rule $\frac{\phi}{\psi}$ is equivalence preserving if ϕ and ψ are equivalent.
All rules considered will be equivalence preserving.

7.1.2 Domain Reduction Rules

- $\mathcal{DE} := x_1 \in D_1, ..., x_n \in D_n$,
- $\mathcal{DE}' := x_1 \in D_1', ..., x_n \in D_n'$,
- for $i \in [1..n]$: $D_i' \subseteq D_i$,
- $\mathcal{C}'$: restriction of all constraints in $\mathcal{C}$ to the domains $D_1', ..., D_n'$.

7.1.3 Transformation Rules

- Not domain reduction rule,
- $\mathcal{C}' \neq \emptyset$,
- $\mathcal{DE}'$ extends $\mathcal{DE}$.

7.1.4 Introduction Rules

$$\frac{\langle \mathcal{C}; \mathcal{DE} \rangle}{\langle \mathcal{C}, C; \mathcal{DE} \rangle}$$

is an introduction rule if C does not occur in $\mathcal{C}$. Such rules introduce new constraints.
If the rule does not depend on $\mathcal{DE}$ then we write

$$\frac{\mathcal{C}}{\mathcal{C}, C}.$$

7.1.5 Derivations

Application of a rule (informally): replace in a CSP the part that matches the premise by the conclusion.
Relevant application of a rule (informally): the result differs from the initial CSP.
A CSP $\mathcal{P}$ is closed under the applications of R if

- R cannot be applied to $\mathcal{P}$

or

- no application of it to $\mathcal{P}$ is relevant.

Given: a finite set of proof rules.

- Derivation: a sequence of CSPs s.t. each is obtained from the previous one by an application of a proof rule.
- A finite derivation is called
  - successful: last element is the first solved CSP in this derivation,
  - failed: last element is the first failed CSP in this derivation,
  - stabilizing: last element is the first CSP closed under the applications of the proof rules.

7.1.6 Recap

- A constraint is solved if it equals the Cartesian product of the domains of its variables.
- CSP is solved if all its constraints are solved.
- CSP is failed if
  - it contains the false constraint $\perp$, or
  - some of its domains or constraints are empty.

7.2 Term Equations

The first complete solver we shall discuss deals with solving finite sets of term equations. This is known as the unification problem and is of paramount importance for automated theorem proving and for logic programming.

Some definitions:
The Alphabet consists of

- variables: $x, y, z, u, ...,$
- function symbols, each with a fixed arity: $f, g, h,$
- parentheses: '(' and ')',
- comma, that is: ',' .

7.2.1 Terms

Defined inductively as follows.
a variable is a term,
if f is an n-ary function symbol and $t_1, ..., t_n$ are terms, then $f(t_1, ..., t_n)$ is a term.
Note: Every constant is a term.

7.2.2 Substitutions

Finite mappings from variables to terms. To each variable x in its domain a term different from x is assigned.
Written as $\{x_1/t_1, ..., x_n/t_n\}$
where
$x_1, ..., x_n$ are different variables,
$t_1, ..., t_n$ are terms,
for $i \in [1, n], x_i \not\equiv t_i$.

- Given: term s, substitution θ.
  $s\theta$: result of applying θ to s.
  Replace simultaneously each variable in s by corresponding term from θ.
- Given substitutions θ and η.
  $\theta\eta$: is the composition of θ and η: $(\theta\eta)(x) := (x\theta)\eta$. i.e. first apply θ and only then apply η. Note that $\theta\eta$ is a substitution.
- θ is more general than τ if for substitution η, $\tau = \theta\eta$.

7.2.3 Unifiers and Most General Unifiers (MGUs)

Substitution θ is a unifier of s and t if $s\theta \equiv t\theta$.
θ is a most general unifier (mgu) of s and t if

- θ is a unifier of s and t.
- θ is more general than all other unifiers of s and t.

θ is a unifier of a set of term equations $\{s_1 = t_1, ..., s_n = t_n\}$ if θ is a unifier of s_i and t_i for $i \in [1..n]$.
θ is an *mgu* of E if

- θ is a unifier of E,
- θ is more general than all unifiers of E.

Two sets of equations are equivalent if they have the same set of unifiers.

Example: Both $\theta = \{x/c, y/g(c,a), z/b\}$ and $\eta = \{y/g(x,a), z/b\}$ are unifiers of $\{f(g(x,a),z) = f(y,b)\}$, however, η is more general than θ. Proof - $\theta = \{x/c, y/g(c,a), z/b\} = \{y/g(c,a), z/b\}\{x/c\} = \eta\{x/c\}$.

Deciding whether a given set of term equations has a unifier is called the *unification problem.* Unifiers may or may not exist. For example neither $\{a = f(a)\}$ nor $\{f(x) = g(x)\}$ has a unifier.

7.3 The UNIF Proof System

The UNIF proof system may be used to solve the unification problem. It consists of six rewrite rules:

1. Decomposition
$$\frac{f(s_1, ..., s_n) = f(t_1, ...t_n)}{s_1 = t_1, ..., s_n = t_n}.$$
2. Failure 1, if $f \not\equiv g$ then
$$\frac{f(s_1, ..., s_n) = g(t_1, ...t_n)}{\perp}.$$
3. Deletion
$$\frac{x = x}{}.$$
4. Orientation, if t is not a variable and x is a variable then
$$\frac{t = x}{x = t}.$$

5. Substitution, if $x \notin Var(t)$ and $x \in Var(E)$ then
$$\frac{x = t, E}{x = t, E\{x/t\}}.$$

6. Failure 2, if $x \in Var(t)$ and $x \neq t$ then
$$\frac{x = t}{\perp}.$$

Example using the UNIF Proof System to try to find if there's a unifier for the following set of equations:

$$E := \{k(z, f(x, b, z)) = k(h(x), f(g(a), y, z))\}$$

Using Decomposition rule to get rid of the outermost applications of the function symbol k we get
$$\{z = h(x), \underline{f(x, b, z) = f(g(a), y, z)}\}$$

Using Decomposition again we get
$$\{z = h(x), x = g(a), \underline{b = y}, z = z\}$$

Using the Transposition rule we get
$$\{z = h(x), x = g(a), y = b, \underline{z = z}\}$$

Using Deletion rule we get
$$\{z = h(x), \underline{x = g(a)}, y = b\}$$

Using Substitution rule we get
$$\{z = h(g(a)), x = g(a), y = b\}$$

No rule applies at this stage.
$\{z/h(g(a)), x/g(a), y/b\}$ is an *mgu* of E.

7.4 Martelli-Montanari Algorithm

There are cases for which UNIF does not terminate. A "fix" to this problem is called the Martelli-Montanari Algorithm, which always terminates. The only difference with UNIF is that it applies the Substitution rule globally (i.e. to all equations). It can be proved that the Martelli-Montanari Algorithm computes an *mgu* of a given system of term equations or proves that no unifier exists.

8 Lecture 20 October 2006

Apt Book: pages 107-131.

8.1 Linear Equations over the Reals

8.1.1 Basic Definitions

Alphabet

- each real number is a constant,

- for each real number r unary function symbol "$r\Delta$",
- binary function symbol "$+$", (written in the infix notation).

Linear expressions and equations

- Linear expression over reals: a term in this alphabet.
- Linear equation over reals: $s = t$, s, t linear expressions.

Normal Forms
Let $\prec$ (pronounced precedes) be a predefined ordering on the variables.

- Linear expression in normal form:
$$\Sigma_{i=1}^{n} a_i x_i + r$$
where $n \geq 0, x_1, ..., x_n$ are ordered w.r.t. $\prec$.
- Linear equation in normal form:
$$\Sigma_{i=1}^{n} a_i x_i = r$$
where $n \geq 0, x_1, ..., x_n$ are ordered w.r.t. $\prec$.
- Linear equation in pivot form:
$$x = t$$
if $x \notin Var(t)$ and t is in normal form.
- Each linear equation can be rewritten (normalizes) to a unique linear equation in normal form.
- *Substitution*: finite mapping from variables to linear expressions in normal form.
To each variable x in its domain a linear expression different from x is assigned.
- *Application* of a substitution to a linear expression: defined as before.
- Given: substitutions θ and γ.
$\theta\gamma$: composition of θ and γ.
Uniquely determined by
$$\eta(x) := norm((x\theta)\gamma).$$
- θ is a unifier of $s = t$ if $s\theta = t\theta$ normalizes to $0 = 0$.
- mgu: defined as before.

Three types of normal forms:

1. $0 = 0$,
2. $0 = r$ where r is a non-zero real,
3. $\Sigma_{i=1}^{n} a_i x_i = r$, where $n > 0$.

Pivot forms of linear equations

- Each linear equation e normalizes to a normal form.
- If it is type 1 or 2, then it has no *pivot form.*
- If it is type 3, then each equation

$$x_j = \Sigma_{i \in [1..j-1] \cup [j+1..n]} - \frac{a_i}{a_j} x_i + \frac{r}{a_j}$$

  is a *pivot form* of e.

8.2 LIN Proof System

- $norm(s)$: normal form of s,
- $stand(s = t) \equiv norm(s) = norm(t)$.

Deletion

$$\frac{s = v}{}$$

if $s = v$ normalizes to $0 = 0$,

Failure

$$\frac{s = v}{\perp}$$

if $s = v$ normalizes to $0 = r$,
  r is a non-zero real,

Substitution

$$\frac{s = v, E}{x = t, stand(E\{x/t\})}$$

where $x = t$ is a pivot form of $s = v$.

8.3 Gauss-Jordan Elimination Algorithm

If we use the rules from the Lin proof system and apply substitution globally (just like with the Martelli-Montanari system) then we have an algorithm for solving linear equations over the reals. The algorithm will always terminate. The algorithm will only derive $\perp$ iff the original system had no solutions. If the original system of equations has a solution then the algorithm will terminate with a system of equations which is a solved form and which is an *mgu* of the original system.

8.3.1 Example

Consider the following system of linear equations:

$$x + y = 3$$

$$2x + 3y = x + z + 4$$

$$3x + 2y + z = 10$$

Bringing it to normal form gives us:

$$x + y = 3$$

$$x + 3y - z = 4$$
$$3x + 2y + z = 10$$

Selecting x as our pivot, we rewrite the equation $x+y = 3$ to an equivalent form which is in pivot form:

$$x = -y + 3$$
$$x + 3y - z = 4$$
$$3x + 2y + z = 10$$

Substituting $-y + 3$ for x in the second equation we get:

$$x = -y + 3$$
$$2y - z = 1$$
$$3x + 2y + z = 10$$

Note that, as with term equations, this eliminates a variable from an equation. Substituting $-y + 3$ for x in the third equation we get:

$$x = -y + 3$$
$$2y - z = 1$$
$$-y + z = 1$$

Rewriting the third equation in pivot form we get:

$$x = -y + 3$$
$$2y - z = 1$$
$$y = z - 1$$

Substituting $z - 1$ for y in the first equation results in:

$$x = -z + 4$$
$$2y - z = 1$$
$$y = z - 1$$

Substituting $z - 1$ for y in the second equation results in:

$$x = -z + 4$$
$$z = 3$$
$$y = z - 1$$

We shouldn't rewrite an equation to pivot form that was already used for substitution. This leaves only one equation and it happens to be in pivot form already. Substituting 3 for z in the third equation we get:

$$x = -z + 4$$
$$z = 3$$
$$y = 2$$

Substituting 3 for z in the first equation we get:

$$x = 1$$
$$z = 3$$
$$y = 2$$

8.4 Linear Inequalities over the Reals

8.4.1 Fourier-Motzkin Algorithm

Fail if there is an equation of the form $0 \leq rhs$, where rhs (Right Hand Side) is a negative constant;

Succeed if there are no equations left or if all initial variables were eliminated; and otherwise

Eliminate a variable that was not eliminated before and recursively solve.

To *eliminate* the variable x from a set of equations E, do the following:

1. $E^0 := E^- := E^+ := \emptyset$;
2. For each equation e in E do:
   - If e is equivalent to $x \leq t^+$, s.t. $x \notin Vars(t^+)$ then add $x \leq t^+$ to E^+;
   - Otherwise if e is equivalent to $t^- \leq x$, s.t. $x \notin Vars(t^-)$ then add $t^- \leq x$ to E^-;
   - Otherwise add e to E^0;
3. Return $E^0 \cup \{t^- \leq t^+ : \exists(t^- \leq x, x \leq t^+) \in E^- \times E^+\}$.

9 Lecture 2 November 2006

Apt Book: pages 135-147.

9.1 Local Consistency

The main tool for solving a CSP is constraint propagation. This process must be done in such a way that no solutions are lost. These changes are to make a CSP more locally consistent.

Node Consistency A unary constraint C on variable x is node consistent is C is equal to the domain of x. A CSP is node consistent if all of its unary constraints are node consistent.

Arc Consistency Let x and y be variables and let D_x and D_y be their respective domains. A constraint C on the sequence x, y is arc consistent if the following holds:

- $\forall v \in D_x \exists w \in D_y \rightarrow (v, w) \in C$ and
- $\forall w \in D_y \exists v \in D_x \rightarrow (v, w) \in C$.

A CSP is arc consistent if all its binary constraints are arc consistent.

Hyper-Arc Consistency is a generalization of arc consistency for constraints of any arity. Let $x_1, ..., x_n$ be a sequence of n variables and let D_i be the domain of x_i for $1 \leq i \leq n$. A constraint C on $x_1, ..., x_n$ is called hyper-arc consistent if $D_i = C[x_i]$.
A CSP is called hyper-arc consistent if all its constraints are hyper-arc consistent.

Directional Arc Consistency Let $\prec$ be an order on the variables x_1 and $x_2$and let $D_1$and D_2 be their domains. A constraint C on x_1, x_2 is called directionally arc consistent with respect to $\prec$ iff $x_i \prec x_j \implies D_i = C[x_i]$, for $1 \leq i, j \leq 2$.

9.2 Proof Rules for Arc Consistency

Let $x_1, ..., x_n$ be a sequence of variables with respective domains $D_1, ..., D_n$. Let $C \subseteq D_1 \times ... \times D_n$, let $v = (v_1, ..., v_n)$ be any member of C, and let k be any integer such that $1 \leq k \leq n$. We denote the *projection* of v onto x_k by $v[x_k]$ and we define $v[x_k] = v_k$. We denote the projection of C onto x_k by $C[x_k]$ and we define $C[x_k] = \{v[x_k] : v \in C\}$.

The following two rules are called the Arc Consistency Proof Rules:

1.
$$\frac{\langle C; x \in D_x, y \in D_y \rangle}{\langle C; x \in C[x], y \in D_y \rangle}$$

   where C is a constraint on (the sequence) x and y. ($C[x]$ removes all values in domain of x not allowed by constraint C)

2.
$$\frac{\langle C; x \in D_x, y \in D_y \rangle}{\langle C; x \in D_x, y \in C[y] \rangle}$$

10 Lecture 3 November 2006

Apt Book: pages 147-156.

10.1 Normalised CSPs

A CSP $\mathcal{P}$ is normalized if for each pair x, y of its variables at most one constraint on x, y exists.

Denote by $C_{x,y}$ the unique constraint on x, y if it exists and otherwise the "universal" relation on x, y.

R and S: two binary relations.

- *transposition* of R:
$$R^T := \{(b, a) | (a, b) \in R\},$$
- *composition* of R and S by
$$R \cdot S := \{(a, b) | \exists c((a, c) \in R, (c, b) \in S)\}.$$

10.2 Path Consistency

Sometimes arc consistency is not sufficient to detect unsatisfiability of a binary CSP because examining the constraints one constraint at a time does not provide enough information.

In the following we will assume that if we have a CSP then there will be (exactly) one constraint between each *sequence* of variables. A CSP is called *path consistent* iff $C_{xz} \subseteq C_{xy} \cdot C_{yz}$ for any 3-variable subset $\{x, y, z\}$ of its variables such that:

- C_{xy} is the constraint between x and y,
- C_{xz} is the constraint between x and z,
- C_{yz} is the constraint between y and z.

10.3 Path Consistency Transformation Rules

A CSP is path consistent only if it is closed under application of the following three transformation rules.

$$\frac{C_{xy}, C_{xz}, C_{yx}}{C_{xy}, C'_{xz}, C_{yx}}$$

where $C'_{xz} = C_{xz} \cap (C_{xy} \cdot C_{yz})$.

$$\frac{C_{xy}, C_{xz}, C_{yx}}{C'_{xy}, C_{xz}, C_{yx}}$$

where $C'_{xy} = C_{xy} \cap (C_{xz} \cdot C^T_{yz})$.

$$\frac{C_{xy}, C_{xz}, C_{yx}}{C_{xy}, C_{xz}, C'_{yx}}$$

where $C'_{yz} = C_{yz} \cap (C^T_{xy} \cdot C_{xz})$.

10.4 Directional Path Consistency

Similar to the notion of directional arc consistency. A CSP is directionally path consistent with respect to an ordering $\prec$ on its variables iff $C_{xz} \subseteq C_{xy} \cdot C_{yz}$, if $x \prec y$ and $y \prec z$.

10.5 Gaussian Elimination

Apt Book: page 118.

The process of Gaussian elimination has two parts. The first part (Forward Elimination) reduces a given system to either triangular or echelon form, or results in a degenerate equation with no solution, indicating the system has no solution. This is accomplished through the use of elementary operations. The second step (Backward Elimination) uses back-substitution to find the solution of the system above.

Stated equivalently for matrices, the first part reduces a matrix to row echelon form using elementary operations while the second reduces it to reduced row echelon form, or row canonical form.

Another point of view, which turns out to be very useful to analyze the algorithm, is that Gaussian elimination computes a matrix decomposition. The three elementary operations used in the Gaussian elimination (multiplying rows, switching rows, and adding multiples of rows to other rows) amount to multiplying the original matrix with invertible matrices from the left. The first part of the algorithm computes an LU decomposition, while the second part writes the original matrix as the product of a uniquely determined invertible matrix and a uniquely determined reduced row-echelon matrix.

11 Lecture 9 November 2006

Apt Book: pages 157-164.

11.1 Instantiation

Consider a CSP $\mathcal{P} = \langle \mathcal{C}; \mathcal{DE} \rangle$ and a sequence $x_1, ...x_k$ of variables X.

- An *instantiation* on X is a function mapping each variable of X to a value from its domain. We write $\{(x_1, v_1), ..., (x_k, v_k)\}$ for the instantiation on X that maps x_i to v_i, for $1 \leq i \leq k$.
- Instantiation $\{(x_1, v_1), ..., (x_k, v_k)\}$ on X *satisfies* a constraint C on a subsequence $x_{i_1}, ..., x_{i_m}$ of X if $(v_{i_1}, ..., v_{i_m}) \in C$.
- Instantiation I on X is *consistent* with respect to a CSP if I satisfies each constraint of the CSP that is defined on a subsequence of X.
- A consistent instantiation is called *k-consistent* if it has k members.
- An instantiation is a *solution* of a CSP if it is consistent with respect to the CSP and is defined on all variables of the CSP.
- The *restriction* of instantiation $\{(x_1, v_1), ..., (x_k, v_k)\}$ to a subsequence $x_{i_1}, ..., x_{i_m}$ of $x_1, ...x_k$ is defined as $\{(x_{i_1}, v_{i_1}), ..., (x_{i_m}, v_{i_m})\}$. The restriction of I to subsequence W is denoted $I \mid W$.

11.2 *k*-consistency

CSP is 1-consistent if for every variable x with a domain D each unary constraint on x equals D.

CSP is k-consistent, $k > 1$, if every $(k-1)$-consistent instantiation can be extended to a k-consistent instantiation no matter which new variable is chosen.

1-consistency aka node consistency.

12 Lecture 10 November 2006

Apt Book: pages 162-166.

12.1 Projection and Joins

Let X be a sequence of variables, let Y be a subsequence of X and let C be a constraint on X.

The projection $\prod_Y(C)$ of C onto Y is defined as $\prod_Y(C) = \{t[Y] : t \in C\}$.

Let Y be a sequence of variables. Furthermore, let $X_1, ..., X_m$ be sub-sequences of Y . Finally, let C_i be a constraint on X_i, for $1 \leq i \leq m$.

The *join* $C_1 \bowtie ... \bowtie C_m$ of $C_1, ..., C_m$ is the unique constraint on Y that is defined by $C_1 \bowtie ... \bowtie C_m = \{t \in D : t[X_i] \in C_i, \text{ for } 1 \leq i \leq m\}$, where D is the Cartesian product of the domains of the variables in Y.

Let X be a sequence of variables and let $\mathcal{P} = \langle \mathcal{C}; \mathcal{DE} \rangle$ be a CSP. The X-join $\overline{C_X}$ of $\mathcal{P}$ is the unique constraint on X that is defined by $\overline{C_X} = \bowtie \{C_W \in \mathcal{C} : W$ is a subsequence of $X\}$, where the notation C_W means that C is defined on the sequence W.

12.2 Proof Rule for *k*-Consistency

Let X be a sequence of $k-1$ variables and let w be a variable not occurring in X, then the following proof rule (which is called k-Consistency) must hold

$$\frac{C_X}{C_X \cap \Pi_X(\overline{C_{X,w}})}.$$

12.3 Strong k-Consistency

A CSP with at least k variables is called *strongly* k-consistent if it is i-consistent for $1 \leq i \leq k$. The following is the application.

Theorem. Let $\mathcal{P}$ be a CSP with k variables, which is strongly k-consistent and whose domains are non-empty, then $\mathcal{P}$ is consistent.

13 Lecture 16 November 2006

13.1 4C Presentation

OPL is used to support industry.

14 Lecture 17 November 2006

14.1 Search Algorithms

1. Backtracking.
2. Forward Checking.
3. Maintain Arc Consistency (MAC) - k-way.
4. Maintain Arc Consistency - 2-way branching.

E.g. looking for solutions to 5-queens problem.

15 Lecture 23 November 2006

Apt Book: pages 166-175.

15.1 Relational Consistency

Let $\mathcal{P}$ be a CSP and $\mathcal{C}$ a sequence of its constraints.

Let $\mathcal{P} \mid \mathcal{C}$ denote the CSP that is obtained from $\mathcal{P}$ by removing all constraints not in $\mathcal{C}$ and all domain expressions involving variables that are not present in $\mathcal{C}$.

A CSP, $\mathcal{P} = \langle \mathcal{C}; \mathcal{DE} \rangle$, is called *relationally (i,m)-consistent* if for every subsequence $\mathcal{C}'$ of $\mathcal{C}$ of length m, every subset X of size i of the variables of $\mathcal{C}'$ and every consistent instantiation I with domain X can be extended to a solution to $\mathcal{P} \mid \mathcal{C}$.

15.1.1 Properties

- A node consistent binary CSP is arc consistent iff it is relationally (1,1)-consistent.
- A node consistent CSP is hyper-arc consistent iff it is relationally (1,1)-consistent.
- A CSP with m constraints is consistent iff it is relationally (0,m)-consistent.

15.1.2 Proof Rules

Let $C_1, ..., C_m$ be m constraints and let X be a subsequence of the (combined) variables of $C_1, ..., C_m$ of size i.

Then the following rule, called the Relational (i,m)-Consistency Rule holds:

$$\frac{C_X}{C_X \cap \Pi_X(C_1 \bowtie ... \bowtie C_m)}.$$

15.2 Backtrack Free Search

We will now use global properties of the constraint graph and local (consistency) properties to derive sufficient conditions for backtrack free search.

Graph is associated with a CSP $\mathcal{P}$.

Nodes: variables of $\mathcal{P}$.

Arcs: connect two variables if they appear jointly in some constraint.

Let $\mathcal{P}$ be a CSP, let $G = \langle V, E \rangle$ be its *associated graph* and let $\prec$ be an order on V.

- The $\prec$-width of a node w of G is $\mid \{(v, w) \in E : v \prec w\} \mid$.
- The $\prec$-width of G is the maximum of the $\prec$-widths of its nodes.
- The width of G is the minimum of its $\prec$-widths.

15.3 Freuder's Theorem

Consider a CSP with variables $x_1, ..., x_n$, such that

- $D_i \neq \emptyset$ is the domain for x_i;
- It is strongly k-consistent for some $k \leq n$; and
- Its width is $k - 1$,

then the CSP is consistent.

Proof. Let $x_1 \prec \ldots \prec x_n$ be any linear ordering on the CSP variables such that the $\prec$-width of the associated graph of the CSP is $k - 1$.

We shall prove that:

1. There exists a consistent instantiation with domain $\{x_1\}$;
2. For each $i \in \{1, \ldots, n-1\}$ each consistent instantiation with domain $\{x_1, \ldots, x_i\}$ can be extended to some consistent instantiation with domain $\{x_1, \ldots, x_{i+1}\}$.

(1) The CSP is *1*-consistent and the domain of x_1 is non-empty. Therefore, $\{(x_1, v_1)\}$ is a consistent instantiation with domain $\{x_1\}$, for any $v \in D_1$, where D_1 is the domain of x_1.

(2) Let $Y \subseteq \{x_1, \ldots, x_i\}$ be the neighbours of x_{i+1} in the associated graph of the CSP, then $s =\mid Y \mid< k$.

Let I be any consistent instantiation with domain $\{x_1, \ldots, x_i\}$.

Since the CSP is strongly $(s + 1)$-consistent we can extent $I \mid Y$ to a consistent instantiation $I \mid Y \cup \{(x_{i+1}, d_{i+1})\}$.

By construction $I \cup \{(x_{i+1}, d_{i+1})\}$ is consistent. The proof is completed by induction on i.

16 Lecture 24 November 2006

Presentation at 4C: Rostering Problem

17 Lecture 30 November 2006

Presentation at 4C: Scheduling Problem

18 Lecture 1 December 2006

18.1 ARC Consistency Algorithm AC-3

This is an algorithm for simplifying CSPs while maintaining arc-consistency.

Given a CSP with:

- n variables
- d maximum domain size
- e binary constraints

The worst-case time complexity is $O(ea)^3$.The space complexity is $O(e + nd)$.

G is the set of directed arcs (x, y) in the CSP.

```
$Q := G$;
WHILE $Q \neq \phi$ DO
    Select and remove any[1] $(x, y)$ from $Q$;
    IF Revise[2]$(x, y)$ THEN
        IF domain of $x$ has changed
            $Q := Q \cup \{z, x\} \in G : z \neq y$;
    ELSE
        RETURN Wipeout[3];
DONE
```

18.2 Worked Example

Iterations in the algorithm are reflected in the domains of the variables and contents of the Queue in the following table. On each row, the underlined arc is processed and any changes to the domains of the variables are reflected on that line.

[1]Best to pick one where the domain of x is small.

[2]Revise function removes all values of domain x which have no support in domain of y. Returns false if domain of x is now empty.

[3]i.e. an empty domain has been found so arc consistency cannot be maintained.

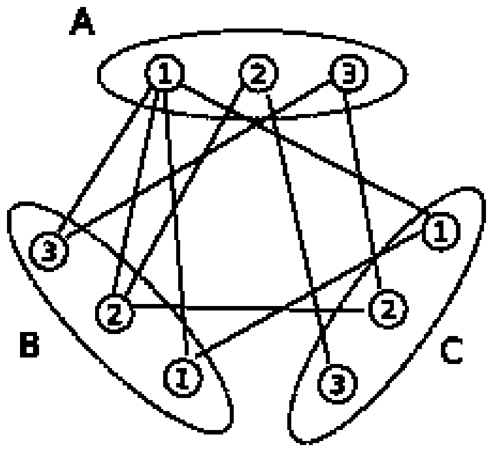

Figure 2: Example CSP

A	B	C	Q
1,2,3	1,2,3	1,2,3	$\underline{\text{AB}}$ AC BA BC CA CB
1,2,3	1,2,3	1,2,3	$\underline{\text{AC}}$ BA BC CA CB
1,2,3	1,2,3	1,2,3	$\underline{\text{BA}}$ BC CA CB
1,2,3	1,2	1,2,3	$\underline{\text{BC}}$ CA CB
1,2,3	1,2	1,2,3	$\underline{\text{CA}}$ CB AB
1,2,3	1,2	1,2	$\underline{\text{CB}}$ AB
1,2	1,2	1,2	$\underline{\text{AB}}$ AC
1	1,2	1,2	$\underline{\text{AC}}$ CA
1	1,2	1	$\underline{\text{CA}}$ BA
1	1,2	1	$\underline{\text{BA}}$ BC
1	1,2	1	$\underline{\text{BC}}$ AB
1	1	1	AB
1	1	1	ϕ

19 Lecture 7 December 2006

19.1 Binary and non-Binary Constraints

Remember that a k-ary constraint has a scope and a relation.

The *scope* is a sequence of k variables and the *relation* is a subset of the Cartesian product of their domains.

Throughout this section we shall write $\langle S, R\rangle$ for a constraint with scope S and relation R. We shall write $\langle V, D, C\rangle$ for the CSP with variables V , domain $D(v)$ for each variable $v \in V$, and constraints C.

There are two well known techniques for encoding a non-binary CSP as an equivalent binary CSP.

Hidden Encoding For each non-binary constraint, C, we add a dual variable, H, and for each variable, V , in the scope of C we add a constraint between V and H.

Dual Encoding The constraints become dual variables. We add a constraint between each pair of dual variables that share an original variable in their scope.

Both transformations may lead to CSPs with more variables and larger domains.

19.2 Hidden Transformation

Let $\mathcal{P} = \langle V, D, C\rangle$ be a CSP then the hidden transformation $\mathcal{P}^h = \left\langle V^h, D^h, C^h\right\rangle$ of $\mathcal{P}$ is defined as follows:

- $V^h = V \cup C$. The variables in V are called the ordinary variables of $\mathcal{P}^h$. The variables in C are called the dual variables of $\mathcal{P}^h$.
- $D^h(v) = D(v)$ if v is an ordinary variable and $D^h(v) = rel(v)$ if v is a dual variable.
- C^h is a set of binary constraints. For each dual variable c and for each variable v in $scope(c)$ there is a constraint allowing all $\langle w, t\rangle \in D(v) \times rel(c)$ such that $t[v] = w$.

19.3 Dual Transformation

Let $\mathcal{P} = \langle V, D, C\rangle$ be a CSP then the dual transformation $\mathcal{P}^d = \left\langle V^d, D^d, C^d\right\rangle$ of $\mathcal{P}$ is defined as follows:

- $V^d = C$. The variables in V^d are called *dual* variables.
- $D^d(v) = rel(v)$.
- C^d is a set of binary constraints. For each pair of dual variables c_1 and c_2 in V^d such that $scope(c_1) \cap scope(c_2) = S \neq \emptyset$ there is a constraint in C^d which allows all $\langle t_1, t_2\rangle \in rel(c_1) \times rel(c_2)$ such that $t_1[S] = t_2[S]$.

20 Local consistency

Let $\mathcal{P} = \langle V, D, C\rangle$ be a CSP and let f be some transformation of CSPs. Then we define $\mathcal{P}^f = f(\mathcal{P})$ and $\left\langle V^f, D^f, C^f\right\rangle = f(\mathcal{P})$.

Example: Let $ac(\mathcal{P})$ be the arc consistent equivalent of $\mathcal{P}$ and let

$$\mathcal{P} = \langle \{x, y\}, \lambda v. \{1, 2\}, \{\langle\langle x, y\rangle, \{1, 1\}\rangle\}\rangle,$$

then

$$\mathcal{P}^{ac} = \langle \{x, y\}, \lambda v. \{1\}, \{\langle\langle x, y\rangle, \{1, 1\}\rangle\}\rangle.$$

(λ is just an anonymous function)

A CSP is called *empty* if at least one of its domains is empty.

Let $\mathcal{LC}_1$ and $\mathcal{LC}_2$ be two local consistency transformations and let $\mathcal{T}_1$ and $\mathcal{T}_2$ be two transformations of CSPs.

Then we say that $\mathcal{LC}_1$ on $\mathcal{T}_1$ is *at least as tight* as $\mathcal{LC}_2$ on $\mathcal{T}_2$, written $\mathcal{LC}_1(\mathcal{T}_1) \succeq \mathcal{LC}_2(\mathcal{T}_2)$, iff for all CSPs $\mathcal{P}$ the following holds:

$$empty(\mathcal{LC}_2(\mathcal{T}_2(\mathcal{P}))) \Rightarrow empty(\mathcal{LC}_1(\mathcal{T}_1(\mathcal{P})))$$

We say that $\mathcal{LC}_1$ on $\mathcal{T}_1$ is *(strictly) tighter* as $\mathcal{LC}_2$ on $\mathcal{T}_2$, written $\mathcal{LC}_1(\mathcal{T}_1) \succ \mathcal{LC}_2(\mathcal{T}_2)$, iff $\mathcal{LC}_1(\mathcal{T}_1) \succeq \mathcal{LC}_2(\mathcal{T}_2)$ and not $\mathcal{LC}_2(\mathcal{T}_2) \succeq \mathcal{LC}_1(\mathcal{T}_1)$.

20.1 Arc Consistency on Hidden Transformation

Theorem. Given any CSP $\mathcal{P}$,

- $\mathcal{P}$ is arc consistent if and only if $\mathcal{P}^h$ is arc consistent.
- $\mathcal{P}^{acoh} = \mathcal{P}^{hoac}$.
- Arc consistency on $\mathcal{P}$ is equivalent to arc consistency on $h(\mathcal{P})$.

20.2 Arc Consistency on Dual Transformation

Theorem. Arc consistency on the dual representation is (strictly) tighter than arc consistency on the original representation.

21 Lecture 8 December 2006

From Pearson and Jeavons paper.

New CSP notation:

A constraint satisfaction problem is a tuple $\mathcal{P} = \langle V, D, R_1(S_1), \ldots, R_n(S_n)\rangle$, where

- V is a set of variables;
- D is a finite set of values called the domain of $\mathcal{P}$;
- $R_i(S_i)$ is constraint:
  - S_i is an ordered list of k_i variables called the scope of the constraint; and
  - R_i is a relation over D of arity k_i, called the constraint relation.

A solution to $\mathcal{P} = \langle V, D, R_1(S_1), \ldots, R_n(S_n)\rangle$ is a function $f : V \to D$, such that $f(S_i) \in R_i$, for $1 \le i \le n$.

Here we write $f(\langle v_1, \ldots, v_k\rangle)$ for $\langle f(v_1), \ldots, f(v_k)\rangle$, i.e. we apply the unary operator $f(\cdot)$ to the components of the single tuple $\langle v_1, \ldots, v_k\rangle$. The set of all solutions to $\mathcal{P}$ is denoted $Sol(\mathcal{P})$.

21.1 Tractability

The general constraint satisfaction problem is *NP*-complete. Throughout this and the next lecture it will be assumed that $P \neq NP$.

We say that a problem class is *tractable* if it is always possible to solve any problem instance from that class in polynomial time. A problem class is *intractable* if it is not tractable.

The following reasons have been identified as "making" problem classes tractable.

- *Restricted structure.* For example, if the constraint graphs of all instances are bounded-width trees then we can solve them without backtracking.
- *Restricted relations.* For example, if all relations of all problem instances contain a constant tuple $\langle v, \ldots, v\rangle$ then we can trivially solve each instance.
- *Restrictions to "local" properties.* For example, let $k = d(r-1)+1$, where d is the size of the domain and r is the size of the largest scope. If an instance is strongly k-consistent, then it is consistent.

21.2 Trees

A *chain* of length n in a hypergraph $\langle V, E\rangle$ is a sequence $x_1, E_1, x_2, \ldots, E_n, x_{n+1}$ such that

- $x_1, \ldots, x_n$ are distinct vertexes from V;
- $e_1, \ldots, e_n$ are distinct edges from E;
- $x_k, x_{k+1} \in E_k$, for $1 \leq k \leq n$.

A chain with length greater than 1 is called cyclic if $x_1 = x_{n+1}$.

A hypergraph is called a *tree* if it has no cyclic chains.

Remember:

Theorem. [Montanari/Freuder] Let C_{tree} be the class of all binary CSPs whose constraint hypergraphs are trees. Then C_{tree} is tractable.

21.3 Problems with Limited Interconnectivity

We can independently solve the problems induced by the connected components of a constraint hypergraph.

Similarly, we can decompose problems that have limited interconnectivity.

For example, if the edges are of the form $E_1 \cup E_2 \cup \{e\}$ where $\cup E_1 \cap \cup E_2 = \emptyset$ and $e \cap \cup E_1 \neq \emptyset \neq e \cap \cup E_2$ then it is probably better if we first solve $E_1(E_2)$, propagate from E_1 to E_2 via e and then solve $E_2(E_1)$. Here $\cup E$ denotes $\cup_{e \in E^e}$.

A *hinge* is a set of at least two edges which cuts the hypergraph into separate connected components such that each connected component intersects with the hinge within only one edge.

A hinge not properly containing other hinges is called a minimal hinge. Any hypergraph can be decomposed into a collection of minimal hinges, which overlap each other in a tree structure, which is called a *hinge-tree* of the hypergraph.

21.4 Hinge Tree Usage

It should be clear that the join of the problems given by the nodes of the hinge-tree of a CSP $\mathcal{P}$ is equal to $Sol(\mathcal{P})$.

If the nodes do not correspond to too large problems then we can solve $\mathcal{P}$ as follows:

- We solve the problems corresponding to the nodes in the hinge-tree and define a constraint for each of these problems.
- We define an acyclic CSP with these constraints.
- We solve the acyclic CSP.

21.5 Tractability due to Restricted Width

Let $\mathcal{P}$ be a CSP with hypergraph $\langle V, E\rangle$.

Let $\cdot \prec \cdot$ be an ordering on V with width $k - 1$.

If $\mathcal{P}$ is strongly k-consistent then a solution to $\mathcal{P}$ can be found by performing a backtrack-free search using the ordering $\cdot \prec \cdot$.

Unfortunately, it is difficult (*NP*-complete) in general to compute a minimum width ordering for a given hypergraph.

22 Lecture 14 December 2006

From Pearson and Jeavons paper.

22.1 Restrictions on the Constraint Language

Let R be a k-ary relation with domain D.

Let $\phi : D^n \mapsto D$ be a function.

Then we say that R is *closed* under ϕ if

$$\langle \phi(d_1^1, d_1^2, \ldots, d_1^n), \phi(d_2^1, d_2^2, \ldots, d_2^n), \ldots, \phi(d_k^1, d_k^2, \ldots, d_k^n) \rangle \in R$$

for all

$$\langle \langle d_1^1, d_2^1, \ldots, d_k^1 \rangle, \langle d_1^2, d_2^2, \ldots, d_k^2 \rangle, \ldots, \langle d_1^n, d_2^n, \ldots, d_k^n \rangle \rangle \in R^n$$

If R is closed under ϕ then we say that ϕ is a *polymorphism* of R. If ϕ is a polymorphism of R then we will write

$$\phi(\langle d_1^1, d_2^1, \ldots, d_k^1 \rangle, \langle d_1^2, d_2^2, \ldots, d_k^2 \rangle, \ldots, \langle d_1^n, d_2^n, \ldots, d_k^n \rangle)$$

for

$$\langle \phi(d_1^1, d_1^2, \ldots, d_1^n), \phi(d_2^1, d_2^2, \ldots, d_2^n), \ldots, \phi(d_k^1, d_k^2, \ldots, d_k^n) \rangle$$

If a relation is closed under an operation then all projections and joins of those relations are also closed under that operation. From now on we will assume that the domain of all relations is D and that D has at least two elements. Finally, we will assume that Γ is some set of relations over D.

The set of all CSPs in which the constraint relations are members of Γ will be denoted C_Γ. The set of all polymorphisms under which each member of Γ is closed will be denoted $Fun(\Gamma)$. The following theorem demonstrates that the polymorphism of Γ tell us something about the tractability of problems in C_Γ.

Theorem [Jeavons] The complexity of C_Γ is completely determined by $Fun(\Gamma)$.

22.2 Six Types of Functions

Constant function $\phi(v) = d$, for all $v \in D$, for some $d \in D$.

Idempotent binary function $\phi(d, d) = d$, for all $d \in D$.

Near-unanimity function $\phi(d, \ldots, d, d') = \phi(d, \ldots, d, d', d) = \phi(d', d, \ldots, d) = d$ for all $d, d' \in D$.

Affine function $\phi(d_1, d_2, d_3) = d_1 - d_2 + d_3$, for all $d_1, d_2, d_3 \in D$, where $\cdot + \cdot$ is a binary operator such that D is an Abelian group.

Semiprojection A function ϕ of arity 3 or more such that there exists an i, $1 \leq i \leq n$, such that $\phi(d_1, \ldots, d_n) = d_i$, for all $d_1, \ldots, d_n \in D$ such that $\mid \{d_1, \ldots, d_n\} \mid < n$.

Essentially unary function A function ϕ of arity n such that there exists an $1 \leq i \leq n$, and some function $f : D \mapsto D$, such that $\phi(d_1, \ldots, d_n) = f(d_i)$, for all $d_1, \ldots, d_n \in D$.

22.3 ACI Functions

Closure under a binary idempotent function alone does not guarantee tractability.

However, for the class of *ACI functions* (associative, commutative, and idempotent) we can guarantee tractability.

A binary function $\phi : D^2 \mapsto D$ is called an ACI function (or *semilatice function*) if it has each of the following three properties:

1. Associativity $\phi(x, \phi(y, z)) = \phi(\phi(x, y), z)$, for all $x, y, z \in D$.
2. Commutativity $\phi(x, y) = \phi(y, x)$, for all $x, y \in D$.
3. Idempotency $\phi(d, d) = d$, for all $d \in D$.

Theorem If Γ is closed under an ACI function then C_Γ is tractable.

22.4 Summary

C_Γ is tractable if any of the following is true:

- $Fun(\Gamma)$ contains a constant function.
- $Fun(\Gamma)$ contains a binary ACI function.
- $Fun(\Gamma)$ contains a near-unanimity function.
- $Fun(\Gamma)$ contains an affine function.

C_Γ is *NP*-complete if any of the following is true:

- $Fun(\Gamma)$ only contains semiprojections.
- $Fun(\Gamma)$ only contains essentially unary functions.

23 Lecture 15 December 2006

23.1 CSP Definition

A set of variables X. A domain $D(x)$ for each variable $x \in X$. A collection of constraints C among subsets of the variables in X.

Each constraint in C is a pair (S, R). $S = (x_1, \ldots, x_k)$ is a sequence of variables, which is called the scope of the constraint. $R \subseteq D(x_1) \times \ldots \times D(x_k)$ is called the relation of the constraint. The number k is called the arity of the constraint.

23.2 Shaving

Shaving is a commonly used technique to improve the pruning power of an existing consistency notion C. Operationally, shaving tries to make each value in the domain of each variable "singleton C consistent," removing singleton inconsistent values.

Here a value, v, in the domain of a variable, X, is singleton C consistent if it is possible to assign v to X and still make the problem C consistent.

23.3 Consistency

Two of the most interesting contributions of constraint satisfaction are the notions of *consistency* and *constraint propagation.*

Consistency tells us something about the existence of solutions. If search is used to solve a problem then consistency tells us something about how easy/difficult it is to make a mistake during search.

Constraint propagation uses the constraints and the domains of a CSP to actively enforce a certain level of consistency without losing/adding solutions.

Usually, enforcing consistency means removing nogoods from domains and recording nogood tuples. Effectively nogood recording results in the creation of new constraints.

23.4 SAC-1 Singleton Arc Consistency Algorithm

```
function SAC-1($X, D, C$) {
  Enforce arc consistency;
  if a domain is empty then
    return wipeout ;
  $Q := \{\langle x, v\rangle : x \in X, v \in D(x)\}$;
  while $Q \neq \emptyset$ do {
    Select and remove any $\langle x, v\rangle$ from $Q$;
    Assign $v$ to $x$;
    Enforce arc consistency;
    if a domain is empty then {
      Undo arc consistency;
      Remove $v$ from $D(x)$;
      Enforce arc consistency;
      if a domain is empty then
        return wipeout ;
      $Q := \{\langle x, v\rangle : x \in X, v \in D(x)\}$;
    }
    else
      Undo arc consistency and assignment;
  };
  return sac_consistent ;
};
```

23.5 *k*-Singleton Arc Consistency

Singleton arc consistency A CSP $\mathcal{P}$ with variables X is singleton arc consistent if and only if for any $x_1 \in X$ and any $v_1 \in D(x_1)$: $ac(P \wedge x_1 = v_1) \neq \bot$.

***k*-Singleton arc consistency** A CSP $\mathcal{P}$ with variables X is k-singleton arc consistent if and only if for any $x_1 \in X$ and any $v_1 \in D(x_1)$: $ac(P \wedge x_1 = v_1, \ldots, x_k = v_k) \neq \bot$,

for all $\{x_2, \ldots, x_k\} \subseteq X \setminus \{x_1\}$, and some $v_2 \in D(x_2), \ldots, v_k \in D(x_k)$.

***weak k*-Singleton arc consistency** A CSP $\mathcal{P}$ with variables X is *weakly* k-singleton arc consistent if and only if for any $x_1 \in X$ and any $v_1 \in D(x_1)$: $ac(P \wedge x_1 = v_1, \ldots, x_k = v_k) \neq \bot$,

for *some* $\{x_2, \ldots, x_k\} \subseteq X \setminus \{x_1\}$, and some $v_2 \in D(x_2), \ldots, v_k \in D(x_k)$.

24 Lecture 2 March 2007

24.1 Polymorphism

There is only one domain for all elements in the tuples.

$\phi(x, y)$ can return z so long as z is in the relation.

24.2 Singleton ARC-consistency

1. Assign any value to any variable (i.e. eliminate all other values from the domain of that variable)
2. Make CSP Arc-consistent.
3. If you get an empty domain then the original CSP is not SAC, and the value chosen cannot be part of a solution.

Make Arc-Consistent means remove values sans supports.

For a problem (CSP) to be SAC all values have to be SAC.