Notes on CSP

Will Guaraldi, et al

version 1.7 4/18/2007

Abstract

Original abstract

This document is a survey of the fundamentals of what we've covered in the course up to this point.

The information in this document was culled from a variety of sources: meetings with Professor Lieberherr, research papers we've been given to read in class (CSG260) [3] [2] [1], Wikipedia [4], ... See bibliography for details.

Recent versions of these notes can be found at:
http://www.ccs.neu.edu/home/guaraldi/csg260/

If you have any questions, comments, or find any issues, let Will know by email sent to *guaraldi at ccs dot neu dot edu*.

Abstract 4/18/2007

I wrote this document while taking CSG260: Advanced Software Engineering which was a course taught by Karl Lieberherr in Fall of 2006. We were studying adaptive programming and aspect-oriented programming techniques using constraint satisfaction problems as an application domain.

As such, this document is a collection of notes about the application domain in the context of the class. This is not an exhaustive survey of constraint satisfaction problems or solving of those problems.

I'm graduating in a few weeks (assuming all goes well) and it's not unlikely that my email address and web-site here at CCS will go away. If you email Professor Lieberherr, he should have a copy of things as well.

Contents

Revision History

version 1.7 (04/18/2007) - wbg (minor (but important) edits)
Added new abstract text. I learned bibtex and added citations to the papers that are cited and talked about.

version 1.6 (10/19/2006) - wbg (minor edits)
Made a bunch of corrections based on Professor Lieberherr's suggestions including a rewrite of the Precise algorithm which was wrong in previous versions.

version 1.5 (10/13/2006) - wbg (major edits)
I added a TOC, moved the revision history to a new page, and significantly expanded the sections on MAXMEAN, APPMEAN, and SAT Solvers.

version 1.1 (10/10/2006) - wbg (minor edits)
Fixed an exponent that was a 5 and should have been an s. Removed a line from one of the itemized lists that was incomplete and shouldn't have been there. Moved the $x = k/n$ to a better place in the APPMEAN explanation.

version 1.0 (10/10/2006) - wbg (major edits)
Made a bunch of changes based on Professor Lieberherr's comments.

version 0.5 (10/08/2006) - wbg (major edits)
First write-up, csp information, precise, maxmean, and appmean.

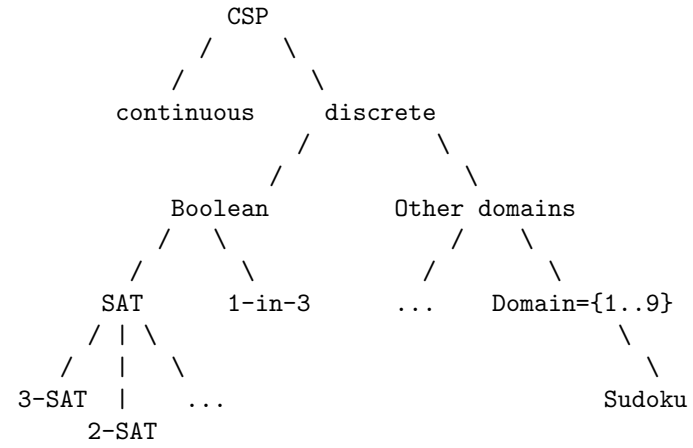

Figure 1: Sketch of CSP family

1 What is CSP?

Constraint Satisfaction Problems are problems that are formed by a series of constraints which must be satisfied by an assignment formed of variables set to values.

CSG260 has been concerned primarily with the section of the tree in *figure 1* where the values are discrete (i.e. all whole numbers), the domain of values is boolean, and the rank of the constraints is 3.

We talked briefly about CSP problems that are discrete where the domain of values is not boolean. One such example of this is Sudoku. We spent some time taking Sudoku and translating the problem so that it was discrete and the domain of values was boolean.

This paper covers the section of the tree we're primarily concerned with along with SAT Solving algorithms.

1.1 Boolean

Boolean CSP is any CSP problem where the domain of values is boolean. For any variable in the CSP, you can set it to either true (1) or false (0).

1.1.1 SAT

SAT is a subset of CSP problems where:

1. the constraints are formed by literals that are connected by logical *or*, and
2. the constraints are connected by logical *and*, and

3. the domain of values is boolean (i.e. $\{0, 1\}$)

SAT is important because many other NP problems can be converted to SAT problems and solved using SAT Solvers (Cook's Theorem). There are a variety of applications that benefit from this: program verification, electronic design, automation, Bayesian network evaluation, bioinformatics, ...

The following terms are used when talking about SAT:

- **formula** - A specific instance of a SAT problem. A formula consists of a series of clauses.
- **clause** - A formula S is formed of clauses that are connected by logical *and*. **clauses** refer specifically to SAT whereas **constraints** refer to any CSP. For SAT, these two terms are interchangeable.
- **literal** - Clauses are formed of literals. A literal has a variable and is either positive or negative.
- **interpretation** - An interpretation is a list of variable to value assignments.
- **rank** - Rank refers to the number of literals in a clause. For example:
  Or(x1 x2 x3) rank 3
  Or(x1) rank 1
  Or(x y) rank 2
- **weight** - Sometimes the clauses we're working with have a weight associated with them. For example:
  Or3(x1 x2 x3) : 10 weight 10
  Or1(x1) : 5 weight 5
  Or2(x y) : 2 weight 2

  Intuitively, a clause with a weight w is equivalent to having w of that clause in the formula of weight 1.
- **3-SAT** - A 3-SAT problem is a SAT problem where the clauses are restricted to Rank 3 or less.

  Some papers use the term "3-SAT" to refer to SAT problems where clauses are of Rank 3, though some papers refer to this as Exact-3-SAT.

SAT problems are usually shown using Conjunctive Normal Form (CNF). Examples of SAT in CNF:

```
X1  or X2  or !X13 or X4 or X10    AND
X21 or X22 or X23                  AND
!X32                               AND
X14 or !X33                        AND ...
```

Wikipedia adds that all of the following formulas are valid CNF:

```
A or B
!A or (B or C)
(A or B) or (!B or C or !D) or (D or E)
(!B or C)
```

The following are **NOT** valid CNF:

```
!(B or C)
(A and B) or C
A and (B or (D and E))
```

In class, we've been using syntax like this:

```
Or( x1 x2 !x3 ) and
Or(    x2     ) and
Or( x1    !x3 )
```

or syntax like this where we explicitly specify the type of the constraints:

```
Or3( x1 x2 !x3 ) and
Or1(    x2     ) and
Or2( x1    !x3 )
```

2-SAT and 3-SAT are SAT problems of rank 2 and rank 3 respectively.
Any problem can be reduced to a 3-SAT problem by adding new variables.

1.1.2 One-in-three

In the one-in-three CSP problem, only one literal in a clause can be set to 1:

```
x1 + x2 + x3 = 1
x1 + x5 + x6 = 1
x2 + x8 + x9 = 1
...
```

In order to maintain the equality, only one of the literals in each clause can be equal to 1—the others have to be equal to 0. For example, if $x1$ and $x2$ were equal to 1, then we would have this:

```
x1=x2=1, x3=x5=x6=x8=x9=0

x1 + x2 + x3 = 1    unsatisfied
x1 + x5 + x6 = 1    satisfied
x2 + x8 + x9 = 1    satisfied
...
```

In class, we've been using syntax for specifying one-in-three problems that looks like this:

```
OneInThree( x1 x2 x3 )
OneInThree( x1 x5 x6 )
OneInTwo(   x2 x8    )
```

where the + and the = 1 are implied. Also, the last one in that set has been reduced by $x9 = 0$.

2 SAT Solving

SAT is the prototypical NP-complete problem. However, we can find solutions that come "close" to satisfying the maximum number of clauses in polynomial time.

This section covers some of the terminology, but it's definitely a good idea to read The Quest for Efficient Satisfiability Solvers because it covers all of this in more depth.

SAT Solvers share various properties:

1. **complete** - A complete solver can find a solution to a SAT problem or prove that no such solution exists.
2. **incomplete** - An incomplete solver can find a solution to a SAT problem, but it can't distinguish between there being no solution and the solver's inability to find it.
3. **randomized algorithm** - A randomized algorithm has a random element. Running a solver that has a randomized algorithm on a solution twice may not produce the same results. This is also referred to as a **stochastic** algorithm.
4. **deterministic algorithm** - A deterministic algorithm has no random elements. Running a solver that has a deterministic algorithm on a solution multiple times will always produce the same result.

	complete	**incomplete**
randomized	biased coin-flipping with superresolution	MAXMEAN*
deterministic	superresolution, precise, MAXMEAN with superresolution	MAXMEAN

Terminology:

- **MAXSAT** - A MAXSAT algorithm produces an interpretation that satisfies the maximum number of clauses in a SAT formula.

  If the clauses have weight, then a MAX algorithm produces an interpretation that has the maximum weight of satisfied clauses in the SAT formula.

  Similarly, a MAXCSP algorithm produces an interpretation that satisfies the maximum number of constraints in a CSP formula.

- **MAX-3-SAT** - A MAX algorithm that operates on a 3-SAT problem.
- **free variable** - A free variable is a variable that is unassigned.
- **decision** - A decision is any time you assign a value to a free variable.
- **conflicting clause** - A clause that has all its literals assigned to 0.
- **resolvent** - A resolvent is a clause that's generated by a resolution step. For example if you had two clauses Or(a b) and Or(!b c) then the resolvent would be Or(a c).
- **variable ordering** - The algorithm uses a suitable ordering of the free variables to decide in which order to set them.
- **value ordering** - The algorithm looks at the values of a variable in order to decide which value to try first.

2.1 Homework 2: Precise

Precise is a recursive algorithm that finds the maximum interpretation (i.e. satisfying the maximum weight) of a given formula F.

Precise will return an interpretation I that satisfies the maximum weight of the clauses in f.

The intuition for Precise is:

```
P(f)
   if f has at least one unassigned variable x
      P(f[x=1])
      P(f[x=0])
```

However, this version of Precise will traverse the entire search tree of 2^n steps.

For Homework 2, we used a version of Precise that prunes sections of the tree that we discover aren't worth traversing.

Input: formula F, interpretation I
Output: interpretation I
Setup: Before calling Precise the first time, we need to create an interpretation I with a random assignment.

precise(F, I) **returns** *interpretation* :
    if not *empty*(F) :

        $WUC :=$ *weight*(F, I)
        $v :=$ next free variable in F

        $F_{true} := F[v = 1]$ (F reduced by $[v = 1]$)
        $I_{true} := I[v = 1]$ (I extended by $[v = 1]$)
        if *weight*$(F,\ I_{true}) < WUC$: (pruning check)
            $I' :=$ *precise*$(F_{true},\ I_{true})$
            if *weight*$(F_{true},\ I') < WUC$:
                $I := I'$
                $WUC :=$ *weight*$(F,\ I)$

        $F_{false} := F[v = 0]$ (F reduced by $[v = 0]$)
        $I_{false} := I[v = 0]$ (I extended by $[v = 0]$)
        if *weight*$(F,\ I_{false}) < WUC$: (pruning check)
            $I' :=$ *precise*$(F_{false},\ I_{false})$
            if *weight*$(F_{false},\ I') < WUC$:
                $I := I'$

    return I

where:

- $empty(F)$ - Returns true if there are no free variables in F.
- $weight(F, I)$ - Returns the total weight of the unsatisfied clauses in F with interpretation I.

2.2 Homework 2: MAXMEAN and MAXMEAN*

MAXMEAN and MAXMEAN* are defined in the paper Partial Satisfiability SAT II [3] and also in Algorithmic Extremal Problems in Combinatorial Optimization [2]. This paper covers large portions of it here, but you should refer to the original paper for specifics.

MAXMEAN is a deterministic algorithm that uses value ordering to determine which value to assign a variable. MAXMEAN works for CSP problems with any finite set of relations.

In class, we implemented MAXMEAN for CSP problems that are boolean, discrete, and of rank 3.

In this section, we use the following notation:

- **S** - S refers to a CSP formula. MAXMEAN works for SAT problems in CNF as well as OneInThree problems.
- **R** - R refers to a relation of a constraint in a CSP problem. If you're using CSP problems of rank 3 or less, then it's convenient to represent R as an 8-bit number (0 through 255) that specifies which rows of the truth table for x, y, and z are satisfied. For example:

```
row   x   y   z    R (!x or y)
---   ---------    --------------
1     0   0   0    1 <- satisfied
2     0   0   1    1 <- satisfied
3     0   1   0    1 <- satisfied
4     0   1   1    1 <- satisfied
5     1   0   0    0
6     1   0   1    0
7     1   1   0    1 <- satisfied
8     1   1   1    1 <- satisfied
```

  The R here is binary number 11001111 which is 207 in decimal. So we can refer to (!x or y) as R_{207}.

MAXMEAN uses a function $mean_k^n(S)$ which takes k, n, and S as input and returns the average fraction of satisfied clauses in S among all assignments having exactly k ones.

There are two invariants for $mean_k^n(S)$:

- $mean_{-1}^n(S) = mean_0^n(S)$, and
- $mean_{n+1}^n(S) = mean_n^n(S)$

Here's the algorithm for maxmean(S):

maxmean(S) **returns** *assignment* :
  $J :=$ empty assignment

  compute k such that:
  $mean_k^n(S) = \max 0 <= k' <= n$ of $mean_{k'}^n(S)$

  for all variables x **in** S **do** :
    if $mean_{k-1}^{n-1}(S[x=1]) > mean_k^{n-1}(S[x=0])$:
      $J[x] := 1, k := k-1, S := S[x=1]$
    else :
      $J[x] := 0, S := S[x=0]$

  return J

where:

- $S[x=0]$ - The formula S reduced by the assignment $x=0$.
- J - An assignment of variables to values.
- $J[x]$ - Variable x in assignment J.
- $mean_k^n(S)$ - The average fraction of satisfied clauses in S among all assignments having exactly k ones out of n free variables.

$mean_k^n(S)$ is defined as:

$$mean_k^n(S) = \sum_{i=0}^{255} t_{R_i}(S)\ SAT_k^n(R_i)$$

where:

- R_i - A relation.
- $t_{R_i}(S)$ - The fraction of the clauses in S that are of relation R. For example, if there are 10 claues in S and three of them are relation R_5, then t_{R_5} would be $\frac{3}{10}$.

One thing to notice here is that it's likely that many t_{R_i} will be 0 because there aren't any clauses of that relation. It makes sense to skip computing $SAT_k^n(R_i)$ for i where $t_{R_i} = 0$.

$SAT_k^N(R_i)$ is defined as:

$$SAT_k^n(R) = \sum_{s=0}^{r(R)} q_s(R) \frac{\binom{k}{s}\binom{(n-k)}{r(R)-s}}{\binom{n}{r(R)}}$$

where:

- $q_s(R)$ is the number of satisfying rows in the truth table of R which contain s ones.

  For example, if you had Or(!x y), then the truth table for the R for that clause is:

row	x	y	z	R (!x or y)
1	0	0	0	1
2	0	0	1	1
3	0	1	0	1
4	0	1	1	1
5	1	0	0	0
6	1	0	1	0
7	1	1	0	1
8	1	1	1	1

  R is satisfied in rows 1, 2, 3, 4, 7, and 8 because those rows have values for x and y that satisfy !x or y.

  If $s = 0$, then $q_0(R)$ for this truth table would be 1 because row 1 is the only row in the truth table that has zero 1s and is satisfied.

  If $s = 1$, then $q_1(R)$ for this truth table would be 2 because rows 2 and 3 both contain one 1 and are satisfied.

  If $s = 2$, then $q_2(R)$ for this truth table would be 2 because rows 4 and 7 both contain two 1s and are satisfied.

  We don't compute at $s = 3$ because R is of rank 2 (there are only two literals) and SAT_k^n is the summation from $s = 0$ to $r(R)$ which in this case is from 0 to 2.

- $r(R)$ is the rank of R.

The calculation for a binomial coefficient is:

$$\binom{n}{k} = \frac{n!}{k!\ (n-k)!}$$

MAXMEAN uses the lines:

$$\begin{array}{l}\text{if } mean_{k-1}^{n-1}(S[x=1]) > mean_k^{n-1}(S[x=0]): \\ \quad J[x] := 1, k := k-1, S := S[x=1] \\ \text{else:} \\ \quad J[x] := 0, S := S[x=0]\end{array}$$

to make the decision as to whether to set x to 0 or 1. This is called **value ordering**.

MAXMEAN* is defined as:

```
maxmean*(S) returns assignment :
  // max_satisfied holds the maximum number of satisfied
  // clauses we've found so far.
  max_satisfied := 0

  // max_J holds the assignment that satisfies the maximum
  // number of clauses
  max_J := empty assignment

  loop
    J := maxmean(S)
    h := satisfied(S, J)

    if h > max_satisfied :
      max_satisfied = h
      max_J = J
    else :
      exit loop

    rename all variables in S which are assigned 1 by max_J
  end
  return max_J
```

where:

- **satisfied(S, J)** - A function that takes a formula S and an assignment J and returns the number of satisfied clauses.

The Complexity of Partial Satisfaction II paper notes that "already after one iteration of the outermost loop of MAXMEAN* the fraction τ_ψ of the clauses is satisfied by assignment J".

2.3 Homework 3: MAXMEAN_APPMEAN

$mean_k^n(S)$ is computationally intensive. However, we can approximate $mean_k^n(S)$ (where $0 <= k <= n$) using $appmean_x(S)$ (where $0 <= x <= 1$ and $x = k/n$) [3].

$$
\begin{aligned}
appmean_x(S) &= \sum_{i=0}^{255} t_{R_i}(S)\ appSAT_x(R_i) \\
appSAT_x(R) &= \sum_{s=0}^{r(R)} q_s(R)\ x^s\ (1-x)^{r(R)-s}
\end{aligned}
$$

Intuitively, the computations for $mean_k^n(S)$ have the same structure as $appmean_x(S)$ where they're both computing a sum of operations on R_i and $t_{R_i}(S)$.

Plugging in $appmean_x(S)$ for $mean_k^n(S)$, we get:

maxmean_appmean(S) **returns** *assignment* :
  $J :=$ empty assignment

  if $maxappmean(S[x=1]) > maxappmean(S[x=0])$:
    $J[x] := 1, S := S[x=1]$
  else :
    $J[x] := 0, S := S[x=0]$
  return J

where:

- $maxappmean(S)$ - $\max 0 <= x <= 1\ appmean_x(S)$. To figure this out, you need to find the polynomial for $appmean_x(S)$, take the derivative of it which gives you a polynomial of degree 2. Then you use the quadratic formula to find the two points where that polynomial is equal to 0. Then:

$$
maxappmean(S) = \max \begin{cases} appmean_{point1}(S) \\ appmean_{point2}(S) \\ appmean_0(S) \\ appmean_1(S) \end{cases}
$$

- the rest of the notation is defined in the same way it is defined in the MAXMEAN section.

References

[1] K. Lieberherr. Complexity of superresolution. *Notices of the American Mathematical Society*, 24:A–433, 1977.

[2] K. J. Lieberherr. Algorithmic extremal problems in combinatorial optimization. *Journal of Algorithms*, 3(3):225–244, 1982.

[3] K. J. Lieberherr and E. Specker. Complexity of partial satisfaction ii. 1985.

[4] Wikipedia. Constraint satisfaction problem — wikipedia, the free encyclopedia, 2007. [Online; accessed 19-April-2007].