Notes on CSP

Will Guaraldi (et al)

version 1.1 10/10/2006

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, Wikipedia, ...

VERSIONS

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.

1 What is CSP?

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

Our class so far has been concerned 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.

I'll go through the section of the tree we're primarily concerned with.

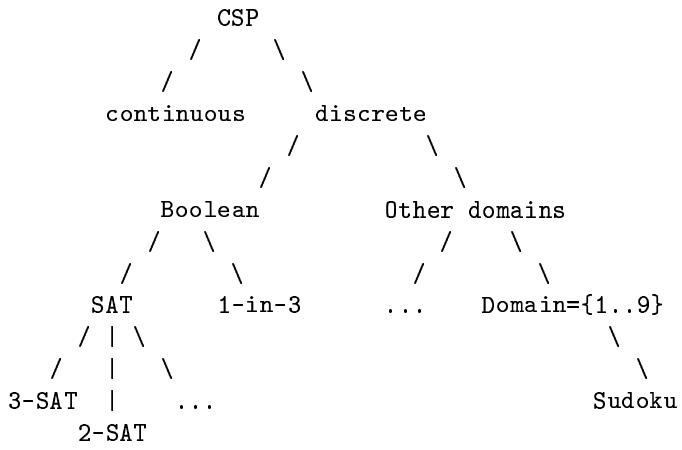

Figure 1: Sketch of CSP family

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 $0, 1$

The following terms are used when talking about SAT:

- **formula** - A specific instance of a CSP 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 that solves the CSP.

- **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      = 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=0

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

2 SAT Solving

SAT is the prototypical NP-complete problem. However, we can approximate solutions that satisfy the maximum number of clauses in polynomial time.

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.

- **variable ordering** - For each variable, we pick an assignment that maximizes the potential solution.

Now we'll go through the algorithms we've used so far.

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, weight of unsatisfied clauses WUC
Output: interpretation I and weight of unsatisfied clauses WUC

Before calling Precise the first time, do these setup steps:
   I = random assignment for all variables
   WUC = total weight of unsatisfied clauses in F using I

Then call Precise:

   I, WUC = Precise(F, I, WUC)


function Precise(F, I, WUC) returns I, WUC:
   if F is empty (everything is satisfied or unsatisfied):
      return I, WUC

   v = variable in F

   F' = F[v=1]
   I' = I[v=1]
   WUC' = total weight of unsatisfied clauses in F'
   if WUC' < WUC:
      I', WUC' = Precise(F', I', WUC')
      I = I'
      WUC = WUC'

   F' = F[v=0]
   I' = I[v=0]
   WUC' = number of unsatisfied clauses in F'
```

```
if WUC' < WUC:
   I', WUC' = Precise(F', I', WUC')
   I = I'
   WUC = WUC'

return I, WUC:
```

2.2 Homework 2: MAXMEAN

MAXMEAN is defined in the paper Partial Satisfiability SAT II. I reprint it here, but you should refer to the original paper for specifics.

Note the invariants $mean_{-1}(S) = mean_0(S)$ and $mean_{n+1}(S) = mean_n(S)$.

```
max_assignment := 0
loop
   compute k such that:
   mean_k(S) = max (0 <= k' <= n) of mean_k'(S)

   for all variables x in S do:
      if mean_k-1(S[x=1]) > mean_k(S[x=0]):
         J[x] := 1,  k := k-1,  S := S[x=1]
      else:
         J[x] := 0,  S := S[x=0]

   h := SATISFIED(S, J)

   if h > max_assignment:
      max_assignment = h
   else:
      exit loop
end;
```

where:

- **max_assignment** is the best assignment we've found so far of variables to values.
- **J[x]** is variable x in assignment J.
- **SATISFIED(S, J)** is the number of satisfied clauses in formula S using assignment J.
- $mean_k(S)$ is the average fraction of satisfied clauses in S among all assignments having exactly k ones.

  Formally, $mean_k(S)$ is calculated as:

$$mean_k^n(\overrightarrow{t}) = \sum_{i=1}^{m} t_{R_i} SAT_k^N(R_i)$$

  where:

  - R_i is a relation R
  - t_{R_i} is the fraction of the clauses in S that are of relation R.
  - $\overrightarrow{t}$ is the vector of t_{R_i}.

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

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

MAXMEAN uses the lines:

```
if mean[k-1](S[x=1]) > mean[k](S[x=0]):
   x = 1
else:
   x = 0
```

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

2.3 Homework 3: APPMEAN

We can approximate $mean_k^n$ (where $0 <= k <= n$) using $appmean_x$ (where $0 <= x <= 1$ and $x = k/n$).

$$appmean_k(\overrightarrow{t}) = \sum_{i=1}^{m} t_{R_i} appSAT_x(R_i) \quad (1)$$

$$appSAT_x(R) = \sum_{s=0}^{r(R)} q_s(R) x^s (1-x)^{r(R)-s} \quad (2)$$

The parts are similar to $mean_k(S)$ and $appmean_k(S)$.

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