Consistency and Inference

The binary Constraint Satisfaction Problem

Given three sets:

variables $V = \{X_1, X_2, ..., X_n\}$,

domains $D = \{D_1, D_2, ..., D_n\}$ allowable values for each variable, and

constraints $C = \{C_1, C_2, ..., C_m\}$ restricting the values that groups of variables can take simultaneously,

find an assignment for each variable X_i of a value v_i from its domain D_i, so that all constraints are satisfied.

A constraint C_{ij} acts on a pair of variables $\{X_i, X_j\} \subseteq V$ called its *scope*, and specifies a subset of the cartesian product of its domains

$$C_{ij} \subseteq D_i \times D_j$$

An assignment (v_i, v_j) to the variables in the scope of C_{ij} *satisfies* C_{ij} if and only if $(v_i, v_j) \in C_i$

Complexity of CSP?

There are *n* variables.

Suppose each of them has a domain of size *d*.

Therefore there are *d*d*... *d*   possible complete assignments.

In the worst case, we might have to examine them all to find a solution.

Simple nested for loops will generate every assignment.

So CSP complexity could be as high as $O(d^n)$

Exercise:

(Make sure Java is on your system)
(Make sure Eclipse is on your system)
Download and install Choco as an Eclipse project

Model and solve the following problem:

$V = \{V1,V2,V3,V4\}$
$D = \{1,2,3,4,5\}$ for each var
$C = \{V1 \leq V4-1,$
$V1 < V2,$
$V2 + V3 > 6,$
$V2 + V4 = 5,$
$V4 < V3\}$

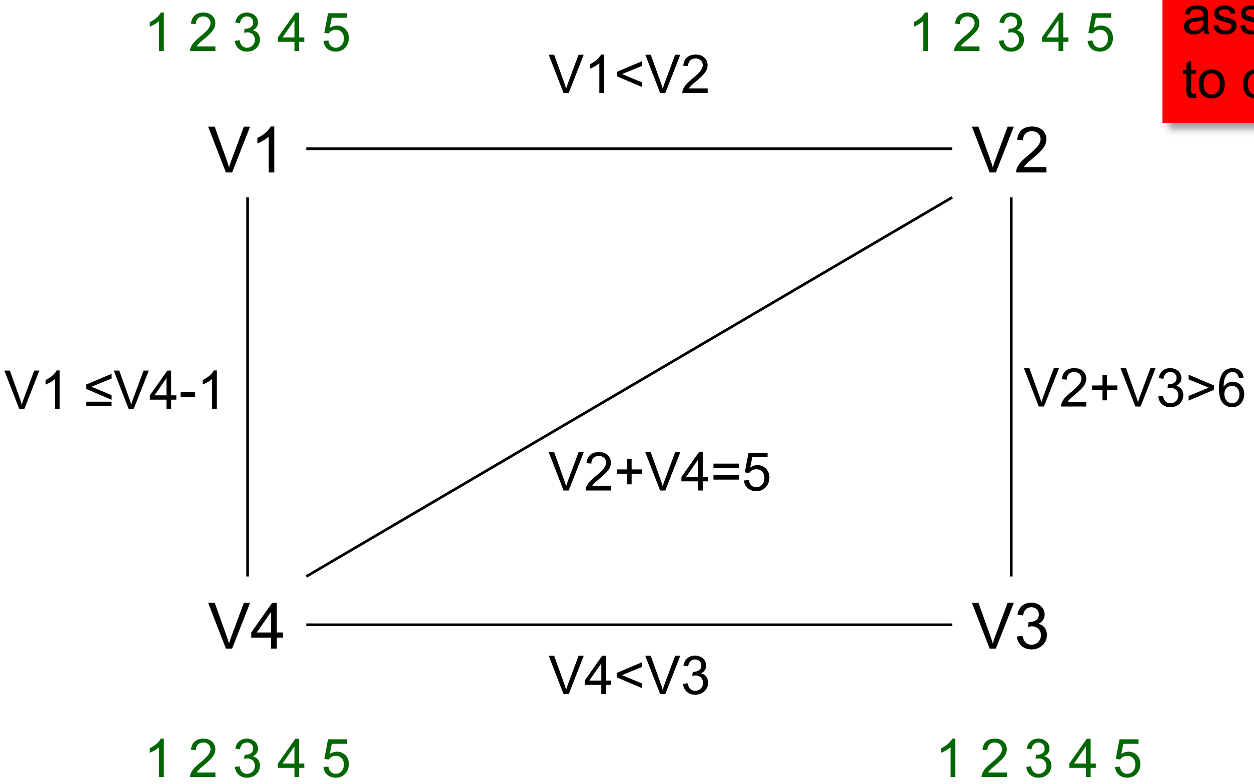
LECTURE 2

```java
import org.chocosolver.solver.Solver;
import org.chocosolver.solver.constraints.IntConstraintFactory;
import org.chocosolver.solver.trace.Chatterbox;
import org.chocosolver.solver.variables.IntVar;
import org.chocosolver.solver.variables.VariableFactory;

public class ARR3example {
  public static void main(String[] args) {
    //create a solver object
    Solver solver = new Solver();

    //create an array of 4 IntVars
    IntVar[] allVars = VariableFactory.enumeratedArray("allVars", 4, 1, 5, solver);

    //post the constraints
    solver.post(IntConstraintFactory.arithm(allVars[3], "-", allVars[0], ">=",  1));
    solver.post(IntConstraintFactory.arithm(allVars[0], "<", allVars[1]));
    solver.post(IntConstraintFactory.arithm(allVars[1], "+", allVars[2], ">",  6));
    solver.post(IntConstraintFactory.arithm(allVars[1], "+", allVars[3], "=",  5));
    solver.post(IntConstraintFactory.arithm(allVars[3], "<", allVars[2]));

    Chatterbox.showSolutions(solver);
    solver.findSolution();
    Chatterbox.printStatistics(solver);
  }
}
```

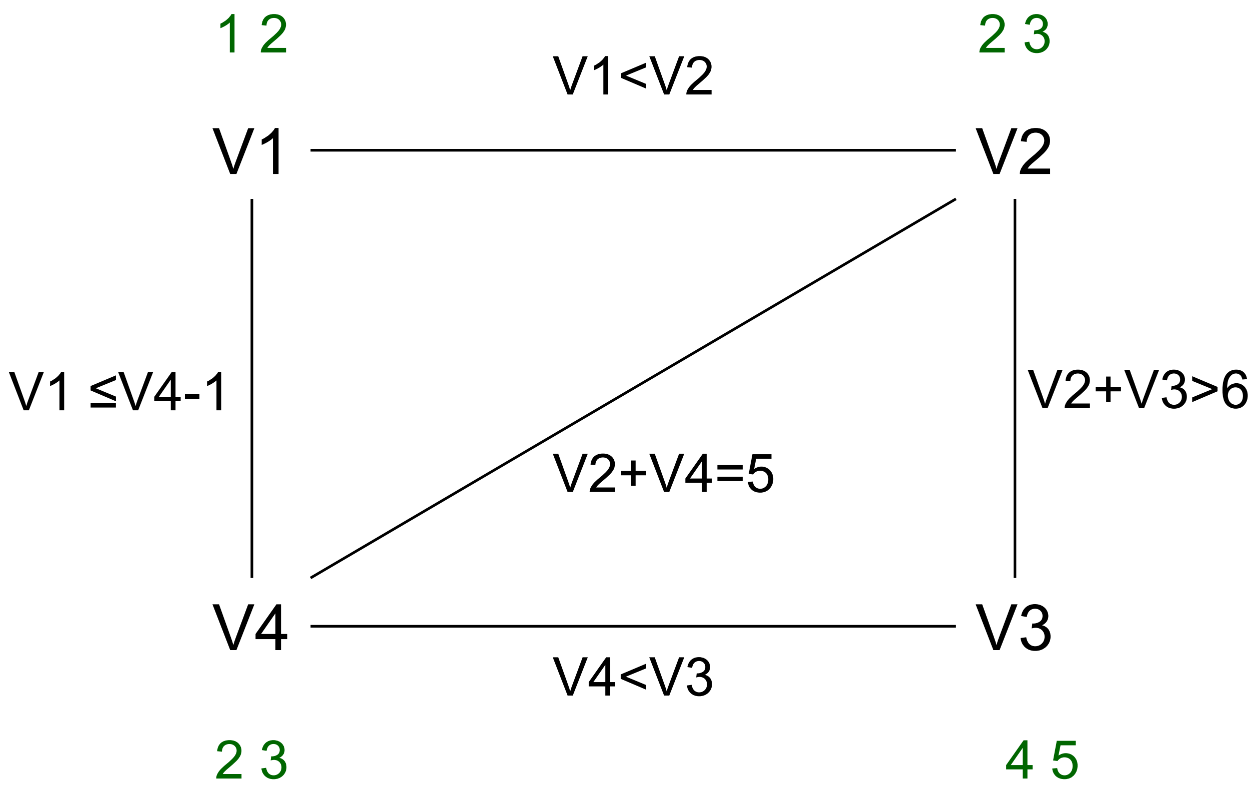
Note: represent V1 ≤ V4-1
as V4 – V1 ≥ 1

Solutions: {(1,2,5,3), (1,3,4,2), (1,3,5,2)}

If the domain sizes are smaller, there are fewer assignments to check ...



Before we start searching for a solution, can we simplify the problem, by removing any impossible values?


1 2
2 3
V1<V2
V1
V2
V1 ≤V4-1
V2+V3>6
V2+V4=5
V4
V3
V4<V3
2 3
4 5

Assumptions, definitions and notation

- Assume all constraints are *binary* – that is, each constraint acts on exactly two variables
- A constraint C_{XY} is $C_{XY} \subseteq D_X \times D_Y$
- Define the *scope* of a constraint C_{XY} to be $\{X,Y\}$
- The *inverse* of a constraint C_{XY} is just C_{YX}, such that $\forall v \in D_X \ \forall w \in D_Y$, $(v,w) \in C_{XY}$ if and only if $(w,v) \in C_{YX}$
- We will write
  - iff instead of "if and only if"
  - s.t. instead of "such that"
  - w.r.t. instead of "with respect to"
  - X<-v to mean the choice of value v for variable X

Arc consistency

Let C_{XY} be a constraint between two variables X and Y.

- A pair $(v,w) \in D_X \times D_Y$ is a *support* for C_{XY} iff $(v,w) \in C_{XY}$
- A value $v \in D_X$ is *arc consistent w.r.t. C_{XY}* iff there is a value $w \in D_Y$ s.t. (v,w) is a support for C_{XY}
- A value $v \in D_X$ is *arc consistent* iff it is arc consistent w.r.t. every constraint which contains X in its scope.
- C_{XY} is *arc consistent* iff for every value $v \in D_X$, v is arc consistent w.r.t. C_{XY}.
- A CSP = (V,D,C) is *arc consistent* iff for every constraint $C_{ij} \in C$, C_{ij} and C_{ji} are arc consistent

Why is arc consistency useful?

- A value that is not arc consistent cannot appear in a solution.
- So, before searching, why not remove every value that is not arc consistent? i.e. make every constraint arc consistent
- When we remove one value, we might make other values in other domains inconsistent, so we need to keep iterating

Can we formalise this as an algorithm?

- is the algorithm correct ? (removes all arc-inconsistent values, and never removes one that is arc-consistent)
- is the algorithm efficient ? (we don't want to waste time making a problem arc consistent if we could have solved it faster by some other method)

revise(C_{XY})

```
revise(C_XY)
Input: C_XY a constraint between two variables X and Y
Output: true if any value was removed; false otherwise
-------------------------------------------------------
changed := false                   //nothing deleted yet
for each v in D_X
   support := false                //no support yet for v
   for each w in D_Y while support == false
      if check(C_XY,v,w) == true  //is (v,w) a support in C_XY?
         support := true           //v∈X is supported
   if support == false             //now checked all w in D_Y
      D_X := D_X - {v}            //delete v from D_X
      changed := true              //Note that change happened
return changed
```

Note: revise C_{XY} only revises D_X. Need to revise the inverse C_{YX} to revise D_Y

AC1

```
AC1(V,D,C)
Input: A CSP; V variables, D non-empty domains, C constraints
Output: true if the CSP is made arc consistent without
        emptying any domain; false if a domain empties
-------------------------------------------------------
for each C_XY in C
   C := C + {C_YX}    //add all inverse constraints
changed := false
while changed == false
   Q := C
   while Q =/= {}
      C_XY := dequeue(Q)          //get the first constraint
      if revise(C_XY) == true     //revise it; if D_X changed ...
         if D_X == {}
            return false           //stop if emptied domain
         else changed := true      //note that a domain changed
return true                       //no more constraints to revise,
                                  //so AC, and no empty domains
```

AC1: analysis

- during the processing of all constraints, if a single value is removed from a single domain, we go back round again and revise every constraint.

- so we revise constraints even if the domains of their variables did not change (and so revising can't do anything for us)

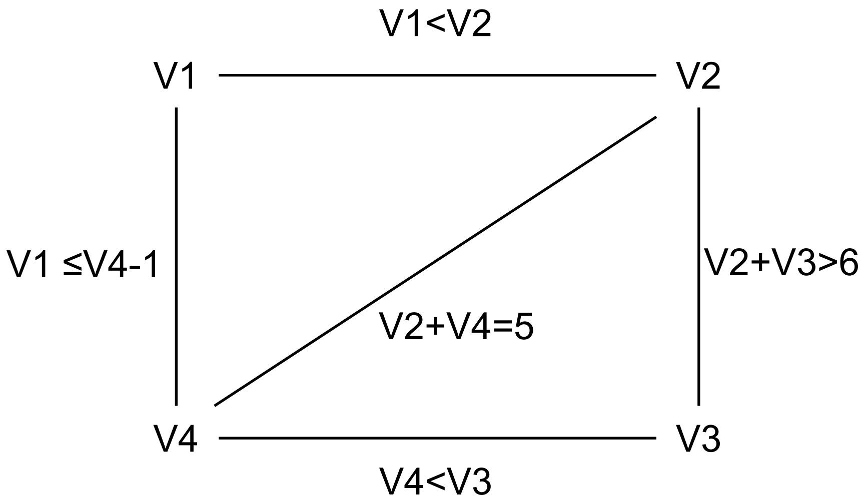
AC3

```
AC3(V,D,C)
Input: A CSP; V variables, D non-empty domains, C constraints
Output: true if the CSP is made arc consistent without
        emptying any domain; false if a domain empties
-------------------------------------------------------
for each C_XY in C
   C := C + {C_YX}   //add all inverse constraints
Q := C
while Q =/= {}
   C_XY := dequeue(Q)            //get the first constraint
   if revise(C_XY) == true       //revise it; if D_X changed ...
      if D_X == {}
         return false  //stop if emptied domain
      for each C_ZX in C, Z≠Y   //all constraints pointing to X
         Q := Q + {C_ZX}         //Note: added into Q
return true                     //no more constraints to revise,
                                //so AC, and no empty domains
```

only revise a constraint if one of its domains changed

Note: Q is a set and a queue – only enqueue an element if not already in Q

D1: 1 2 3 4 5
D2: 1 2 3 4 5
D3: 1 2 3 4 5
D4: 1 2 3 4 5



Queue: {C12, C21, C14, C41, C23, C32, C24, C42, C34, C43,

AC3: complexity O(ed^3)

Let d be the size of the biggest domain, and let e be the number of constraints in the problem.

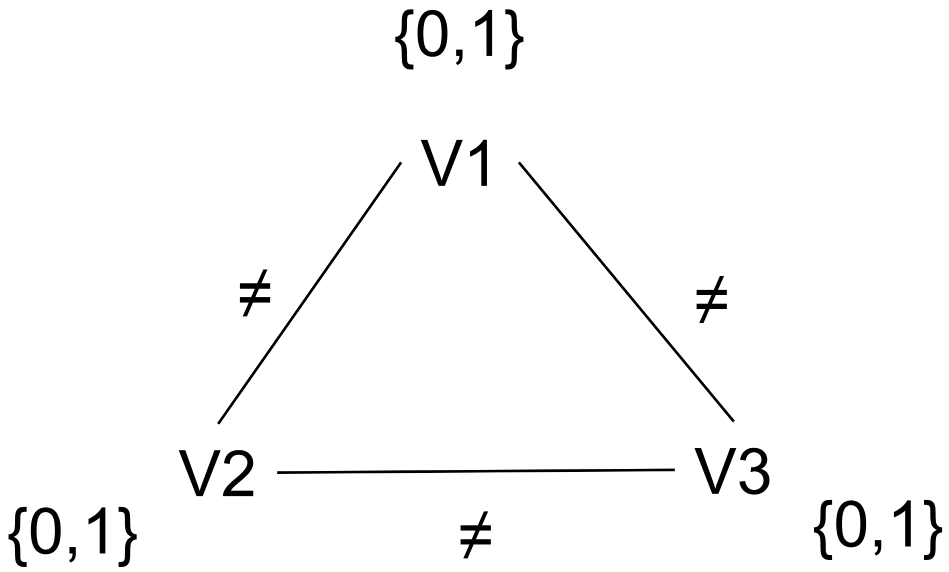
Each time a constraint is added to the queue, the 2nd variable has had values removed from its domain. At worst this happens d times, and there are $2e$ constraints that could be added to the queue (the originals plus inverses). So $2ed$ calls to revise(C_{XY}).

Each call to revise(C_{XY}) does at worst d^2 checks.

So at worst $2ed^3$ checks for AC3.

So AC3 has complexity O(ed^3).

Arc consistency alone is not enough



Beyond AC3

```
revise(CXY)
...
for each v in DX
   support := false
   for each w in DY while support == false
      if check(CXY,v,w) == true
...
```

We might call revise() for a single constraint many times. Each time we call it, for each value in D_X, we check for support by beginning at the start of D_Y ...

... even if the previous support in D_Y is still there?

... even if we know the first *k* checks in D_Y failed last time?

AC2001 (outline)

- Enforce a fixed order on the values in each domain
- The first time I establish support for X<-v from w in D_Y, store that value for C_{XY} with v in D_X
- The next time I am asked to check for support for X<-v in D_Y,
  - look to see if the recorded support (w) is still in D_Y
  - if it is, stop and report true
  - if it is has been deleted, start searching from the first value in D_Y after w's position

Analysis: if I have *n* variables, may require O(*nde)* storage
Worst case and average case runtime is O(ed^2)

Notes

- Mackworth(1977) first formalised arc consistency, and described AC3
- Mackworth & Freuder (1985) established AC3's complexity
- Bessiere & Régin (2001) and Zhang and Yap (2001) both described AC2001
- van Hentenryck, Deville & Teng (1992) described AC5
- Bessiere (2006) gives a wide ranging survey (and fomalisation of consistency in constraint programming

C. Bessiere and J. C. Régin, "Refining the basic constraint propagation algorithm", *Proc. IJCAI'01*, pp309—315, 2001.

C. Bessiere "Constraint Propagation", Chapter 3 of *Handbook of Constraint Programming* (eds. Rossi, van Beek and Walsh), Elsevier, 2006.

A. K. Mackworth "Consistency in networks of relations", *Artificial Intelligence*, 8:99—118, 1977

A. K. Mackworth and E. C. Freuder "The complexity of some polynomial network consistency algorithms for constraint satisfaction problems", *Artificial Intelligence,* 25:65—74, 1985.

P. van Hentenryck, Y. Deville and C. M. Teng. "A generic arc-consistency algorithm and its specializations", *Artificial Intelligence*, 57:291—321, 1992.

Y. Zhang and R. H. C. Yap, "Making AC-3 an optimal algorithm", *Proc. IJCAI'01*, pp316—321, 2001.

Next lecture ...

Search