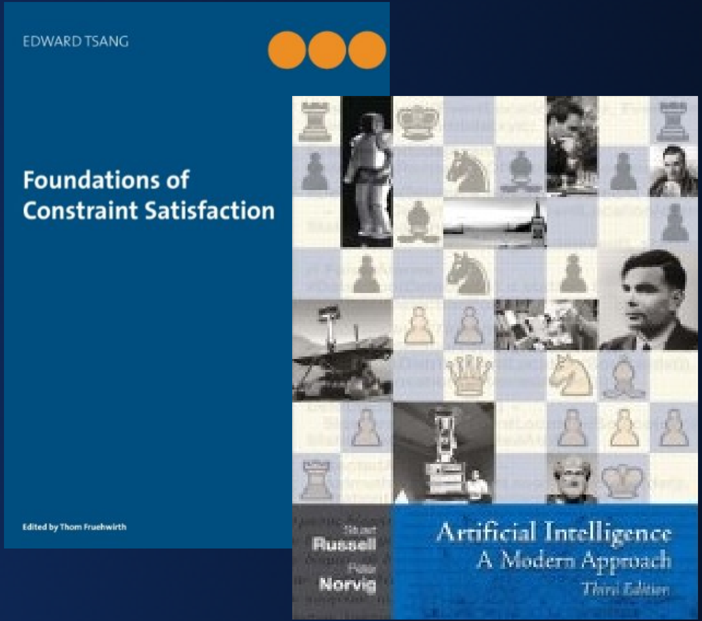
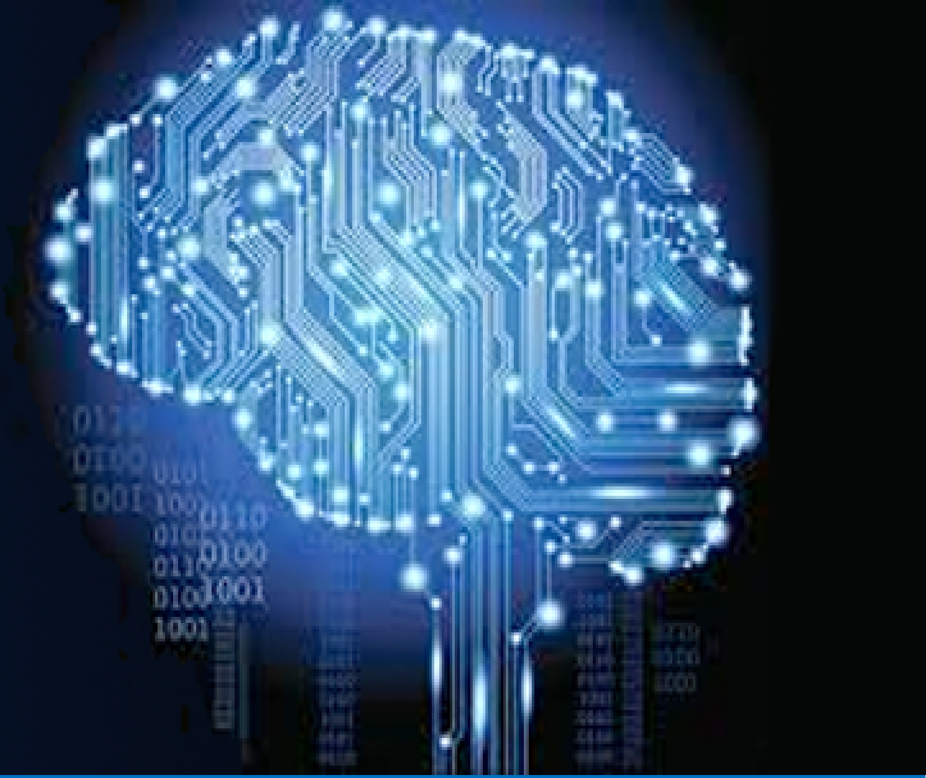




AI Fundamentals: Constraints Satisfaction Problems

Maria Simi

Constraint graphs

Constraint graphs

A binary CSP, is a CSP with unary and binary constraints only.

A binary CSP may be represented as an undirected graph (V, E):

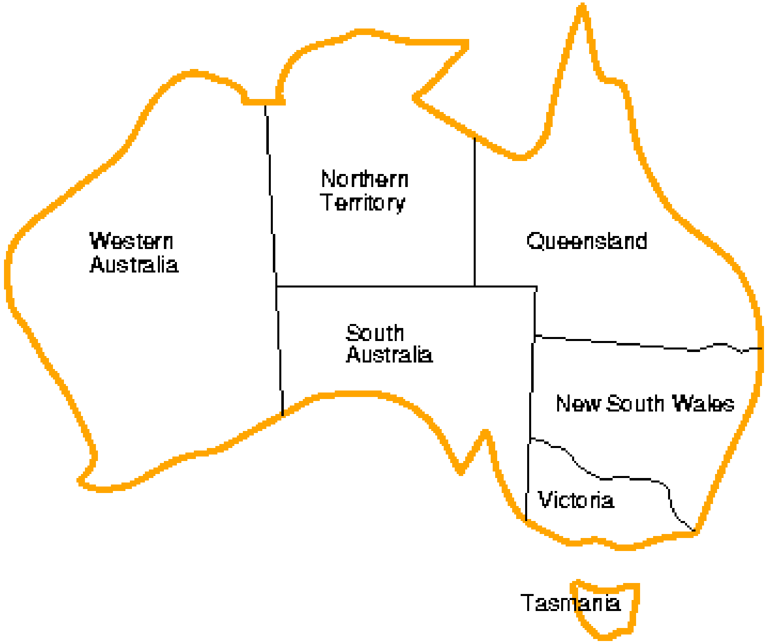
- Nodes correspond to variables (V)
- Edges correspond to binary constraints among variables $(E = V \times V)$

Note: **arcs** have a direction; **edges** are **undirected** arcs; an edge can be seen as a pair of arcs.

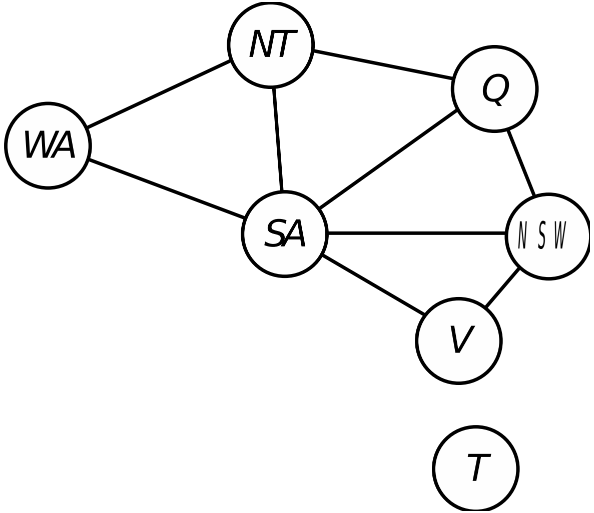
Node x is **adjacent** to node y if and only if (x, y) is in E

A graph is connected if there is a path among any two nodes

Map coloring: constraint graph



Binary constraint graph



Transformation into binary constraints

All problems can be transformed into binary constraint problems (not always worthwhile).

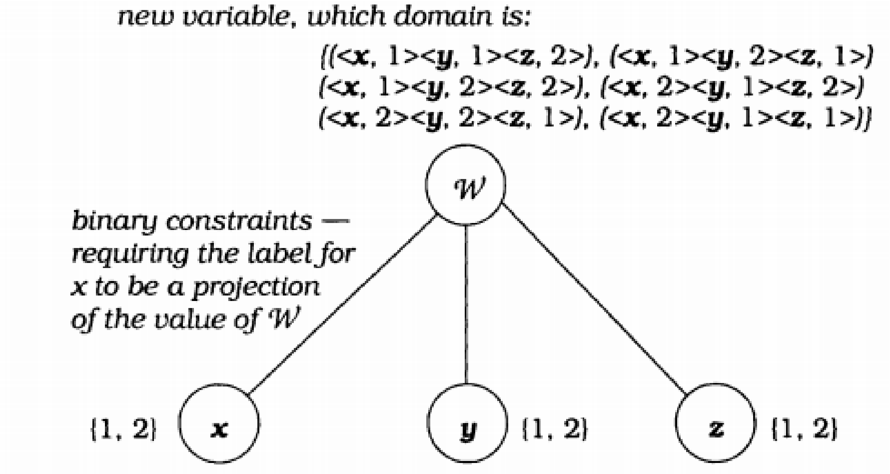
Example.

$X = \{x, y, z\}$

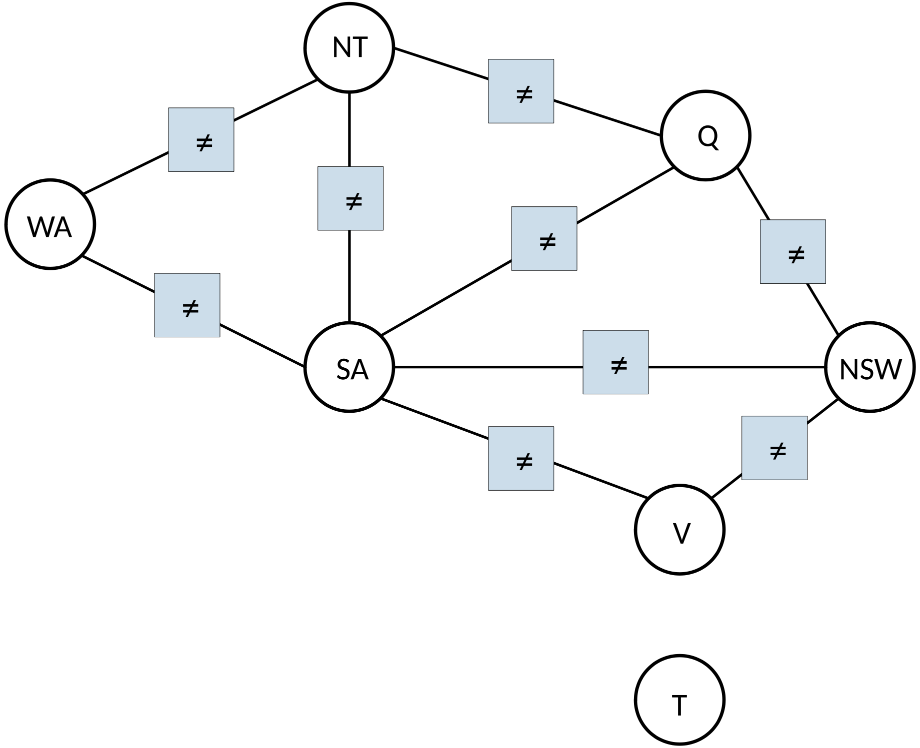
$D_x = D_y = D_z = \{1, 2\}$

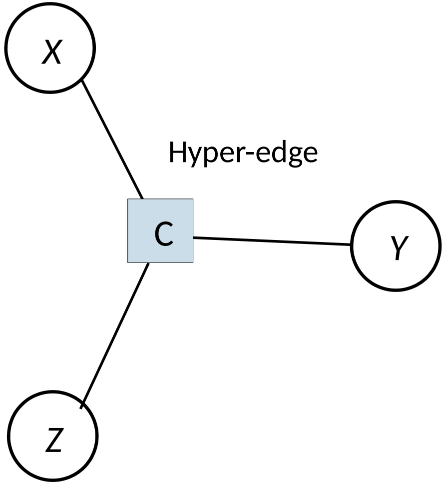
$$
\begin{aligned}
C = \{ & (\langle x,1\rangle, \langle y, 1\rangle, \langle z,2\rangle) \\
& (\langle x,1\rangle, \langle y,2\rangle, \langle z,2\rangle) \\
& (\langle x,1\rangle, \langle y,2\rangle, \langle z,1\rangle) \\
& (\langle x,2\rangle, \langle y,1\rangle, \langle z,2\rangle) \\
& (\langle x,2\rangle, \langle y,1\rangle, \langle z,1\rangle) \\
& (\langle x,2\rangle, \langle y,2\rangle, \langle z,1\rangle)\}
\end{aligned}
$$

Ternary constraint: not all three variables have the same values



Making constraints explicit, *hypergraphs*





A ternary constraint, e.g. $X + Y = Z$

Constraints hypergraphs

In general, every CSP is associated with a constraint hypergraph.

Hypergraphs are a generalization of graphs. In a hypergraph, each hyper-edge may connect more than two nodes.

The constraint hypergraph of a CSP $(X,\ D,\ C)$ is a hypergraph in which each node represents a variable in X, and each hyper-edge represents a higher order constraint in C.

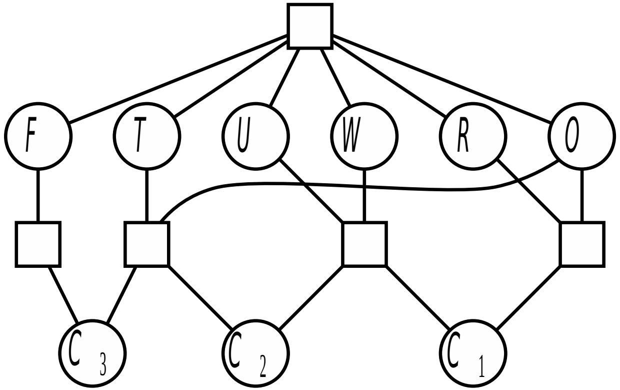
Example: Crypto-arithmetic

Each letter stands for a distinct digit;
the aim is to find a substitution of digits
for letters such that the resulting sum is arithmetically correct

$$\begin{array}{cccc} & T & W & O \\ + & T & W & O \\ \hline F & O & U & R \end{array}$$

Hypergraph: cryptoarithmetic example

$$
\begin{array}{ccccc}
 & C_3 & C_2 & C_1 & \\
 & & T & W & O \\
+ & & T & W & O \\
\hline
 & F & O & U & R
\end{array}
$$



Square nodes are hyper-edges representing *n*-ary constraints

The constraint hypergraph for the crypto-arithmetic problem, showing the **Alldiff** constraint (square box at the top) as well as the column addition constraints (four square boxes in the middle). The variables C_1 , C_2 , and C_3 represent the carry digits for the three columns. Constraints:

$O + O = R + 10 * C_1$

$W + W + C_1 = U + 10 * C_2$

$T + T + C_2 = O + 10 * C_3$

$F = C_3$

$Dom(C_1) = Dom(C_2) = Dom(C_3) = \{0, 1\}$

Dual graph transformation

An alternative way to convert an *n*-ary CSP to a **binary** one is the **dual graph transformation**:

1. Create a new graph in which there is one variable for each constraint in the original graph.
2. If two constraints share variables they are connected by an arc, corresponding to the constraint that the shared variables receive the same value.

Original CSP

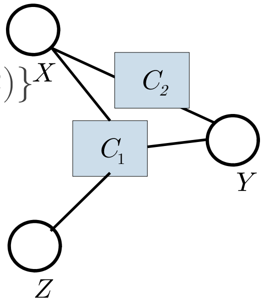
$Dom(x) = Dom(y) = Dom(z) = \{1, 2, 3\}$

$C_1 = \{\langle x, y, z \rangle, x + y = z\} =$

$= \{(1, 2, 3), (2, 1, 3), (1, 1, 2)\}$

$C_2 = \{\langle x, y \rangle, x < y\} =$

$= \{(1, 2), (1, 3), (2, 3)\}$

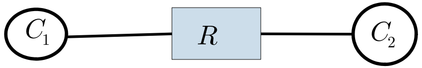


Dual CSP

$Dom(C_1) = \{(1, 2, 3), (2, 1, 3), (1, 1, 2)\}$

$Dom(C_2) = \{(1, 2), (1, 3), (2, 3)\}$

$R_{x,y} =$ *constraint that x and y receive the same values* $= \{(1, 2)\}$



Constraints Propagation

LESSON 2

CONSTRAINT PROPAGATION – LOCAL CONSISTENCY PROPERTIES.

Constraint propagation and related concepts

Contraint propagation

- Constraints are used to reduce the number of legal values for a variable, which in turn can reduce the legal values for another variable, and so on ...

Problem reduction techniques

- Techniques for transforming a CSP into equivalent problems which are hopefully easier to solve or recognizable as insoluble.

Enforcing local consistency

- The process of enforcing **local consistency** properties in a constraint graph causes inconsistent values to be eliminated
- Different types of local consistency have been studied

Problem reduction

Reducing a problem means removing from the constraints (legal assignments) those assignments which appear in no solution tuples.

Two CSP problems are **equivalent** if they have identical sets of variables and solutions.

A CSP problem P_1 is **reduced** to a problem P_2 when

1. P_1 is equivalent to P_2
2. Domains of variables in P_2 are subsets of those in P_1
3. The constraints in P_2 are at least as restrictive than in P_1

These conditions guarantee that a solution to P_2 is also a solution to P_1

Only **redundant** values and assignments are removed (no solution is lost).

The problem is easier to solve.

Problem reduction strategies

Problem reduction involves two possible tasks:

1. removing redundant values from the domains of the variables
2. tightening the constraints so that fewer compound labels satisfy them

Example: if $x < y$ is a constraint and $D_x = \{3, 4, 5\}$ and $D_y = \{1, 2, 4\}$ domains can be safely reduced to {3} and {4}.

Constraints are seen as sets, then this means removing redundant compound labels from the constraints. If the domain of any variable or any constraint is reduced to an empty set, then one can conclude that the problem is insoluble.

Problem reduction is also called *consistency checking/maintenance* since it relies on establishing local consistency properties.

Local consistency properties

- Node consistency
- Arc consistency
- Directional arc consistency
- Generalized arc consistency
- Path consistency
- K-consistency
- Forward Checking

All these operations do not change the set of the solutions, do not necessarily solve a problem but, used in conjunction with search, make the search more efficient by pruning the search tree.

Node consistency / domain consistency

A node is **consistent** if all the values in its domain satisfy unary constraints on the associated variable. In formula, given a unary constraint, we enforce the property

$$C_i = \langle (x_i), R_i \rangle \; D_i \subseteq R_i$$

A constraint network is **node-consistent** if all its nodes are consistent

Unary constraints can be easily satisfied by reducing the domains of variables as follows:

$$D_i \leftarrow D_i \cap R_i$$

The algorithm, called NC-1, is O($d.n$)

Example: in the map coloring problem of Australia

- Suppose South Australia dislikes green: (*SA ≠ green*) is a unary constraint.
- SA starts with domain {*red* , *green*, *blue*}, and we can make it *node-consistent* by eliminating *green*, leaving *SA* with the reduced domain {*red* , *blue*}

Arc consistency (for binary constraints)

A variable in a CSP is **arc-consistent** if every value in its domain satisfies the binary constraints on this variable.

x_i is **arc-consistent** with respect to another variable x_j if for every value in its domain D_i there is some value in the domain D_j that satisfies the binary constraint on the arc (x_i, x_j).

Example: $X = \{x, y\}$ $\quad D_X = D_Y = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

Constraint: $\langle(x, y), \{(0, 0), (1, 1), (4, 2), (9, 3)\}\rangle$ i.e. $x = y^2$ $\qquad x \rightarrow y$

To make x arc-consistent with respect to y, we reduce the domain of x to $\{0, 1, 4, 9\}$.

If we also make y consistent with respect to x, then y's domain becomes $\{0, 1, 2, 3\}$ and the whole **edge** is consistent.

A relational algebra view

Consider variable x_i with associated domain D_i. We further assume a constraint between x_i and x_j, expressed by relation $R_{i,j}$.

Arc $x_i \rightarrow x_j$ is arc consistent iff $D_i \subseteq \pi_i \left(R_{i,j} \bowtie D_j\right)$

Where $\bowtie$ and π are the join and projection operator of relational algebra. The operation is a *left semijoin* ($\ltimes$)

Arc $x_i \rightarrow x_j$ can be made *arc consistent* by computing:

$$D_i \leftarrow D_i \cap \pi_i \left(R_{i,j} \bowtie D_j\right)$$

Example:

$$\pi_x \left(R_{x,y} \bowtie D_x\right) = \pi_x \left(\{(0, 0), (1, 1), (2, 4), (3, 9))\} \bowtie \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}\right)$$
$$= \pi_x \left(\{(0, 0), (1, 1), (2, 4), (3, 9))\}\right) = \{0, 1, 2, 3\}$$

$$D_x \leftarrow D_x \cap \{0, 1, 2, 3\} = \{0, 1, 2, 3\}$$

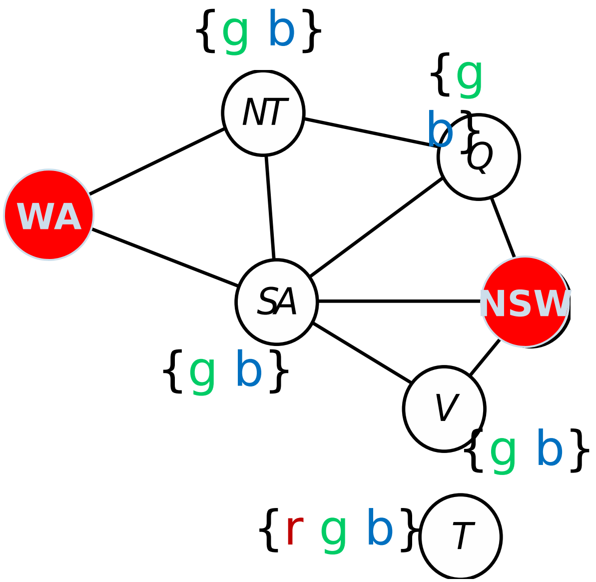
Arc consistent but no solutions

Arc consistency does not guarantee a solution.

In this case all the arcs are consistent but there is no solution

NT ≠ Q, Q ≠ SA, SA ≠ NT

Impossible to color three fully connected nodes with two colors



Algorithm for arc consistency (AC-3)

The most popular algorithm for arc consistency is called AC-3 [Mackworth, 1977]

AC-3(*csp*) maintains a queue of arcs to consider (actually it is a **set**); initially all the arcs in *csp*. An **edge** produces two arcs.

AC-3 pops off an arc (x_i, x_j) from the queue and makes x_i **arc-consistent** with respect to x_j

1. If this step leaves D_i unchanged, the algorithm just moves on to the next arc.
2. If D_i is made smaller, then we need to add to the queue all arcs (x_k, x_i) where x_k is a neighbor of x_i different from x_j
3. If D_i becomes empty, then we conclude that the whole CSP has no solution.

When there are no more arcs to consider, we are left with a CSP that is equivalent to the original CSP, but simpler.

AC-3: AIMA pseudo-code

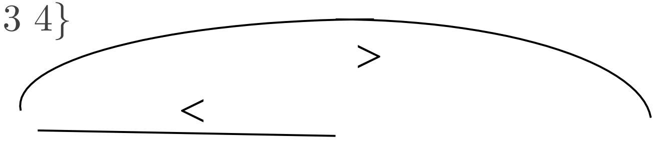
```
function AC-3(csp) returns false if an inconsistency is found and true otherwise
   inputs: csp, a binary CSP with components (X, D, C)
   local variables: queue, a queue of arcs, initially all the arcs in csp

   while queue is not empty do
      (X_i, X_j) ← REMOVE-FIRST(queue)
      if REVISE(csp, X_i, X_j) then
         if size of D_i = 0 then return false
         for each X_k in X_i.NEIGHBORS - {X_j} do
            add (X_k, X_i) to queue
   return true
```

```
function REVISE(csp, X_i, X_j) returns true iff we revise the domain of X_i
   revised ← false
   for each x in D_i do
      if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j then
         delete x from D_i
         revised ← true
   return revised
```

Arc consistency: an example

- Variables $A\ \{1, 2, 3, 4\}$ $B\ \{1, 2, 3, 4\}$ $C\ \{1, 2\ ,3\ 4\}$
- Constraints A < B ; A > C

A B C



QUEUE ARC ARC DOMAIN

$\{(A, B), (B, A), (A , C), (C , A)\}$

$\{(B, A), (A , C), (C , A)\}$ (A, B) $A = \{1, 2, 3, 4\}$

$\{(A, C), (C , A)\}$ (B, A) $B = \{1, 2, 3, 4\}$

$\{(C, A)\}$ (A, C) $A = \{1, 2, 3\}$

$\{(B, A), (C , A)\}$ $add\ (B, A)\ for\ checking$

$\{(C, A)\}$ (B, A) $B = \{2, 3, 4\}$

$\{\ \}$ (C, A) $C = \{1, 2, 3, 4\}$

At the end: $A = \{2, 3\}$ $B = \{3, 4\}$ $C = \{1, 2\}$

Complexity of AC-3

Assume a CSP with n variables, each with domain size at most d, and with c binary constraints (arcs).

Each arc (x_k, x_i) can be inserted in the queue only d times because x_i has at most d values to delete.

Checking consistency of an arc can be done in O(d^2) time, so we get O($c\, d^3$) total worst-case time

Complexity: O(cd^3) ... polynomial time

The algorithm AC-4 is an improved version of AC-3, based on the notion of **support**, that doesn't need to consider all the incoming arcs. Some more information must be kept. O(cd^2).

Directional Arc Consistency

Directional Arc Consistency (DAC) is defined wrt a **total ordering of the variables**.

A CSP is **directional arc consistent** (DAC) under an ordering of the variables if and only if for every label $\langle x, a \rangle$ which satisfies the constraints on x, there exists a compatible label $\langle y, b \rangle$ for every variable y, **which is after** x according to the ordering.

In the algorithm for establishing DAC (DAC-1), each arc is examined exactly once, by proceedings from the last in the ordering, so the complexity is O(cd^2).

We will see later an use of this property.

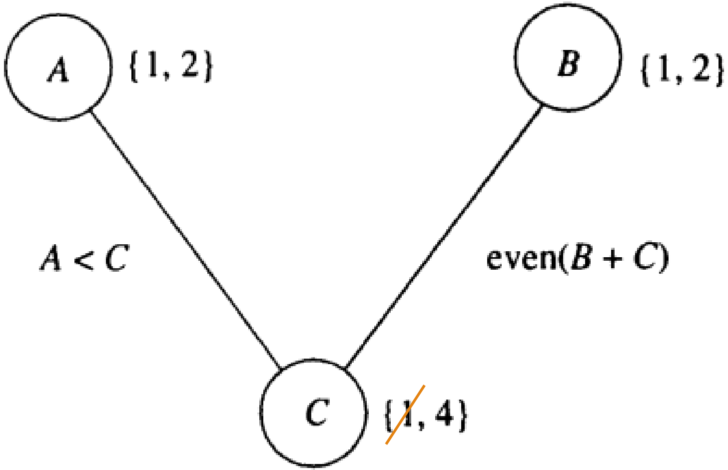
Warning: AC cannot always be achieved by running DAC-1 in both directions.

DAC in both directions weaker than AC

After achieving DAC with orderings (A, B, C) and (C, B, A), the only effect is to delete 1 from the C domain.

However the resulting graph is not arc consistent.

In fact Arc BC is not consistent: the value 1 should be deleted from the domain of B to make it consistent.



Generalized Arc Consistency (GAC)

An extension of the notion of arc consistency to handle *n*-ary rather than just binary constraints (also called *hyper-arc* consistency).

A variable x_i is **generalized arc consistent** with respect to a *n*-ary constraint if for every value v in the domain of x_i there exists a tuple of values that is a member of the constraint and has its x_i component equal to v.

For example, if all variables have the domain {0, 1, 2, 3}, then to make the variable X consistent with the ternary constraint $X < Y < Z$, we would have to eliminate 2 and 3 from the domain of X because the constraint cannot be satisfied when $X = 2 \text{ or } X = 3$.

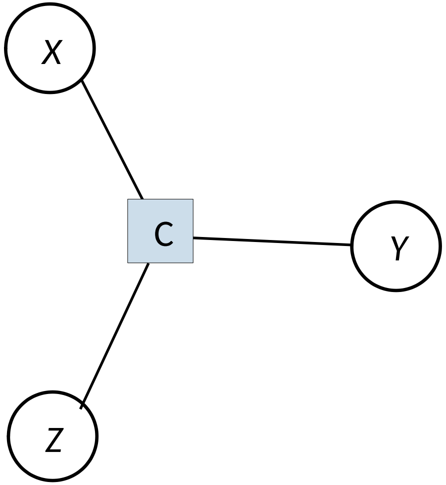
GAC algorithm

The GAC algorithm is a generalization of AC-3. It uses hypergraphs.

The arcs considered are:

$$\langle X,\ X+Y=Z\rangle$$
$$\langle Y,\ X+Y=Z\rangle$$
$$\langle Z,\ X+Y=Z\rangle$$

In general, if constraint c has scope $\{X,\ Y_1,\ \ldots\ Y_k\}$, arc $\langle X,\ c\rangle$ is arc consistent when for each value x in D_x there are values $y_1, \ldots y_k$ in $D_{Y1}\ \ldots\ D_{yk}$ such that $(x,\ y_1, \ldots\ y_k)$ satisfies c.



C is the ternary constraint $X+Y=Z$

Path consistency [Montanari]

Arc consistency tightens down the domains (unary constraints) using the arcs (binary constraints).

Path consistency is a stronger notion: it tightens the binary constraints by using implicit constraints that are inferred by looking at triples of variables.

A path of length 2 between variables $\{x_i, x_j\}$ is **path-consistent** with respect to a third variable x_m if, for every consistent assignment $\{x_i = a, x_j = b\}$, there is an assignment to x_m that satisfies the constraints on $\{x_i, x_m\}$ and $\{x_m, x_j\}$.

In relational algebra:

$$R_{i,j} \subseteq \pi_{i,j}\,(R_{i,m} \bowtie \mathrm{D}_m \bowtie R_{m,j})$$

Path consistency algorithm and properties

To achieve path consistency:

$$R_{i,j} \leftarrow R_{i,j} \cap \pi_{i,j}(R_{i,m} \bowtie D_m \bowtie R_{m,j})$$

The algorithm is called PC-2.

If all path of length 2 are made consistent, then all path of any length are consistent [Montanari 1974], so longer path need not be considered.

This is called **path consistency** because one can think of it as looking at a path from x_i to x_j with x_m in the middle.

Path consistency: example

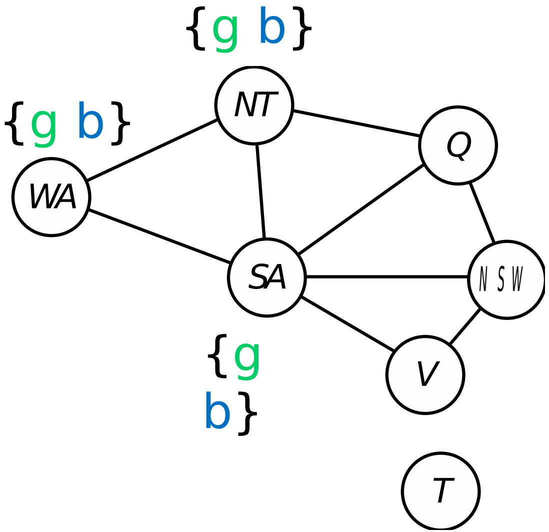
Coloring the Australia map with two colors is impossible, but arc-consistency is not able to discover it.

If we try to make the set {WA, SA} *path consistent* with respect to NT.

The consistent assignments for WA and SA are only two:

1. {WA = green, SA = *blue*}
2. {WA = *blue*, SA = *green*}

Neither of them is compatible with NT=*green* nor NT=*blue*, so the domains of WA and SA become empty and we can conclude that there are no solutions.



k-consistency

Stronger forms of propagation can be defined with the notion of **k-consistency**, a generalization of the other properties.

A CSP is *k-consistent* if, for any set of $k - 1$ variables and for any consistent assignment to those variables, a consistent value can always be assigned to any k^{th} variable.

1-*consistency* says that, given the empty set, we can make any set of one variable consistent: this is what we called node consistency.

2-*consistency* is the same as arc consistency. For binary constraint networks.

3-*consistency* is the same as path consistency.

Strong k-consistency

A CSP is **strongly** k-*consistent* if it is k-*consistent* and is also (k − 1)-*consistent*, (k − 2)-*consistent*, . . . all the way down to 1-*consistent*.

Now suppose we have a CSP with n nodes and make it strongly n-consistent. We can then solve the problem as follows: First, we choose a consistent value for x_1. We are then guaranteed to be able to choose a value for x_2 because the graph is 2-consistent, for x_3 because it is 3-consistent, and so on.

For each variable x_i, we need only search through the *d* values in the domain to find a value consistent with $x_1, \ldots, x_{i-1}$. We are guaranteed to find a solution.

BUT: Any algorithm for establishing k-consistency must take time exponential in k in the worst case. Worse, k-consistency also requires space that is exponential in k.

Domain splitting / case analysis

Split a problem into a number of disjoint cases and solve each case separately.

Example 1: Boolean variable X with domain $\{\mathrm{t}, \mathrm{f}\}$. Solve with $X = \mathrm{f}$ and with $X = \mathrm{t}$.

Combine solutions of simpler problems or stop as soon as a solution is found.

Example 2: $Dom(A) = \{1, 2, 3, 4\}$

1. A case for each value: $A = 1$, $A = 2$, $A = 3$, $A = 4$ (like searching)
2. Two disjoint subsets: $A \in \{1, 2\}$ and $A \in \{3, 4\}$

This strategy can be combined with arc consistency.

Variable elimination

The Variable Elimination (VE) strategy simplifies the network **by removing variables** (not values).

You can eliminate x, having taken into account constraints of x with other variables and obtain a simpler network.

Best understood with relational algebra.

- Consider variable X and all the constraints involving X
- Compute the join of the relations expressing constraints on x and Y (the neighboring variables) then project into Y.

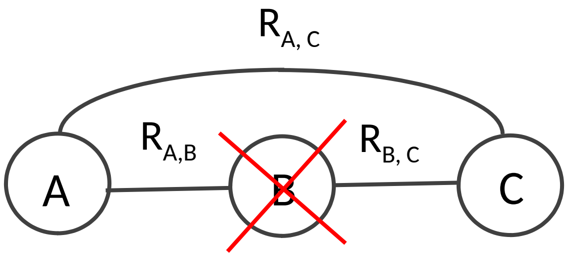
Continue to eliminate variables until only one variable is left.

The algorithm is further described in [AI-FCA, Ch. 4.6]

Example of Variable Elimination

Example: $D_A = D_B = D_C = \{1, 2, 3, 4\}$

Constraints: $A < B$ and $B < C$.



$R_{A,B}$

A	B
1	2
1	3
1	4
2	3
2	4
3	4

$\bowtie$

$R_{B,C}$

B	C
1	2
1	3
1	4
2	3
2	4
3	4

=

A	B	C
1	2	3
1	2	4
1	3	4
2	3	4

$\xrightarrow{\pi_{A,C}}$

A	C
1	3
1	4
2	4

Specialized global constraints [AIMA]

A **global constraint** is one involving an arbitrary number of variables (but not necessarily all variables). Special algorithms can be much more efficient.

Consider two cases:

1. *Alldiff* constraint: all the variables involved must have distinct values (Crypto-arithmetic, Sudoku).
2. Resource constraint: when you have a limited quantity of resources to allocate.

Solutions for *AllDiffs*

Simple form of inconsistency detection for *Alldiff*:

if m variables are involved in the constraint, and if they have n possible distinct values altogether, and $m > n$, then the constraint cannot be satisfied.

Remove any variable in the constraint that has a singleton domain. Delete that variable's value from the domains of the remaining variables. Repeat as long as there are singleton variables. If at any point an empty domain is produced or there are more variables than domain values left, then an inconsistency has been detected.

Example: in the map coloring problem, the assignment {WA=red , NSW =red}
and AC-3 do not detect the inconsistency on the variables NT, Q, SA. The *Alldiff* constraint is instead effective.

Resource constraints

Example: $Atmost(10, P_1, P_2, P_3, P_4)$, meaning that 10 is the maximum of personnel units to be assigned to 4 tasks. This constraint may be checked by summing the minimum requirement for personnel for each task.

Domains can be represented by upper and lower bounds and managed by **bounds propagation**.

Example: air scheduling with two flights F_1 and F_2. You need to carry 420 people.

$D_1 = [0, 165]$ and $D_2 = [0, 385]$ with additional constraint $F_1 + F_2 = 420$

By propagating bounds constraints, we reduce the domains to

$D_1 = [35, 165]$ and $D_2 = [255, 385]$

Forward checking (FC)

A very weak, local and quick form of consistency checking which is triggered during the search process.

When you assign a value v to a variable x in the process of searching for a consistent assignment, check the neighbor's variables and exclude values that are not compatible with v from their domains.

Conclusions

- We have looked at problem reduction techniques which work by enforcing local consistency properties of different strength and complexity.
- These are properties that make the problem simpler: the more effort you put, the simpler the problem becomes.
- These techniques will be used in connection with search algorithms which is the topic of the next lecture.

References

Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach* (3rd edition). Pearson Education 2010 [Cap 6 – CSP]

Handbook of Constraint Programming, Edited by F. Rossi, P. van Beek and T. Walsh. Elsevier 2006.

Edward Tsang, *Foundations of Constraints Satisfaction* [Cap 3]