

- ⊙ The 4-houses puzzle:
- ⊙ 4 families A, B, C and D live next to each other in houses numbered 1, 2, 3 and 4.
- ➔ D lives in a house with lower number than B,
 - ➔ B lives next to A in a house with higher number,
 - ➔ There is at least one house between B and C,
 - ➔ D does not live in the house with number 2,
 - ➔ C does not live in the house with number 4.
- ⊙ Which family lives in which house ?

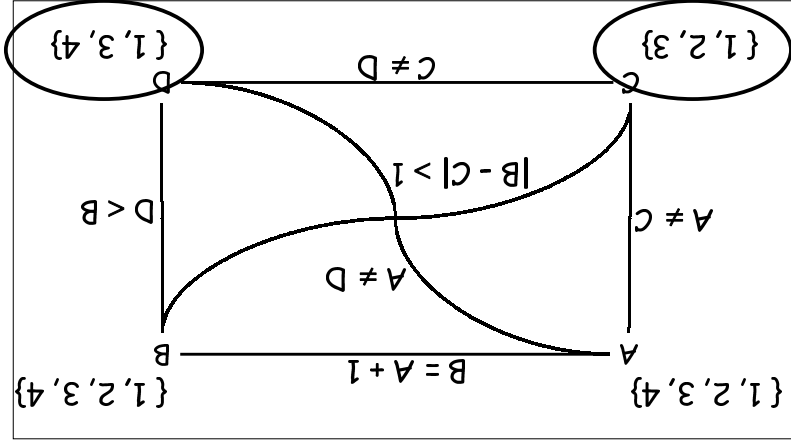
Running example

Node consistency
 Forward check
 Lookahead check
 AC1
 AC3
 Path-consistency

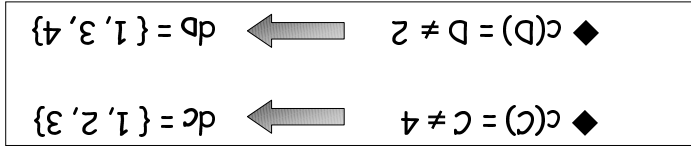
Relaxation

Relaxation
and
Hybrid constraint processing

Different relaxation techniques
 Some popular hybrid techniques

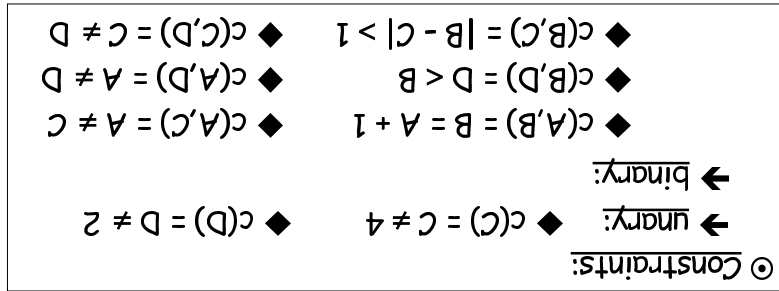


The constraint network:



- ⊙ Or: 1-consistency
- ➔ (only 1 variable is involved)
- ⊙ Unary constraints are eliminated through domain-reductions:

Node-consistency:



- ⊙ The variables: A, B, C and D
- ⊙ The domains: $d_A = d_B = d_C = d_D = \{1, 2, 3, 4\}$

Representation:

Weak relaxation

Forward Check
Lookahead Check

8

◎ Assume we fix the value for 1 variable z_i : $z_i = a$
 ◎ Forward Check(z_i) =
 → activate each constraint $c(z_i, z_j)$ or $c(z_j, z_i)$ once to remove the inconsistent values for $z_i = a$
 ◎ Our example: assume $A = 2$:

Forward check:

- ◎ Requires that 1 variable already obtained a value
- suggests use in combination with backtracking
- ◎ Does not produce a consistent state
- not all relaxation is done

9

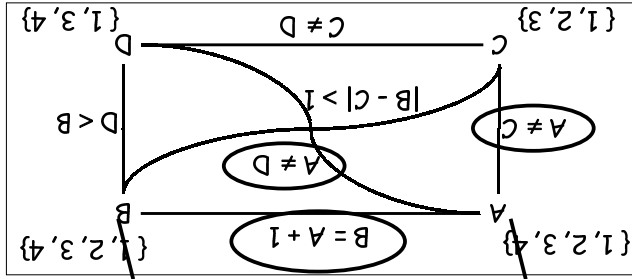
◎ Requires that 1 variable already obtained a value
 → suggests use in combination with backtracking
 ◎ Does not produce a consistent state
 → not all relaxation is done

A stronger (weak) relaxation method

Look ahead check

⊙ Look Ahead Check =
 → activate each constraint $c(z_i, z_j)$ exactly once
 to remove the inconsistent values from the
 domains D_i and D_j .

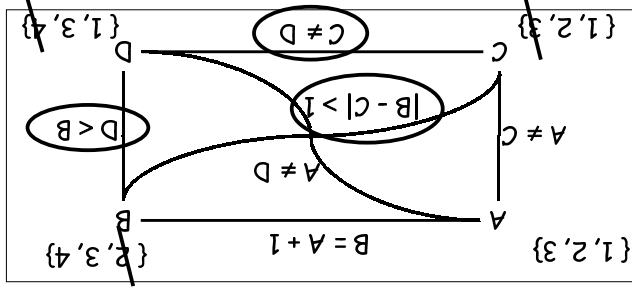
⊙ Our example:



11

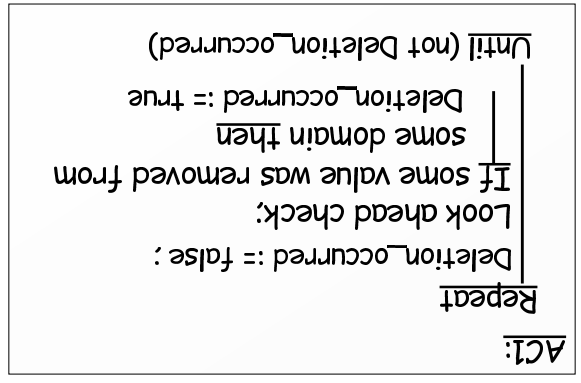
Example continued:

⊙ The 3 other constraints:



12

⊙ Forces Look ahead to reach a consistent state
↳ by reactivating Look ahead until consistency

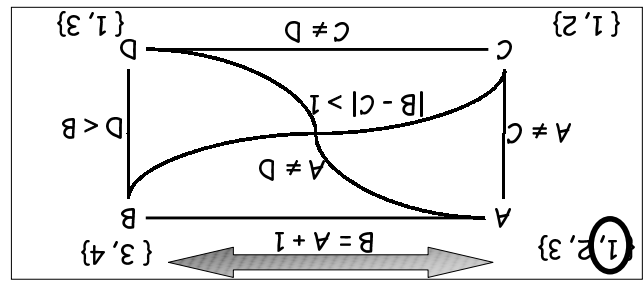


AC 1 (Mackworth)

Techniques that reduce domains to a state that is consistent for each constraint (or arc).
Also called: 2-consistency techniques

Arc consistency techniques

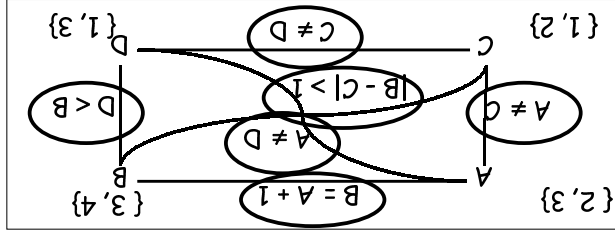
⊙ Still does not produce a consistent state
↳ not all relaxation is done
⊙ Result may depend on the order in which constraints are dealt with.
↳ Removing some values first may allow to find others inconsistent



Look ahead: final results:

- ⊙ Result: A (2 or 3), B (3 or 4), C (1 or 2), D (1 or 3)
- ⊙ Consistent, but NOT REALLY A SOLUTION !!

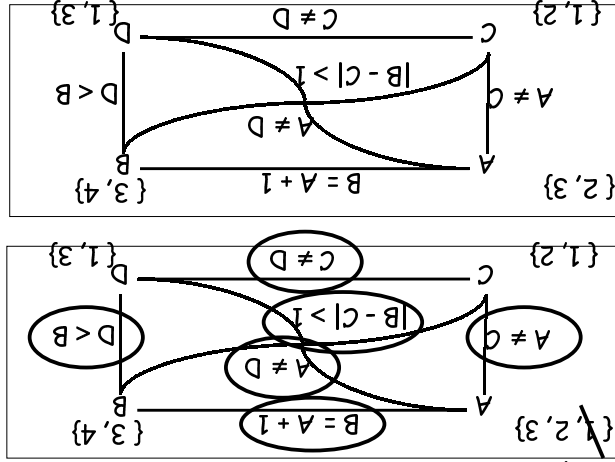
⊙ Deletion_occurred := false



⊙ Third pass:

The example (3):

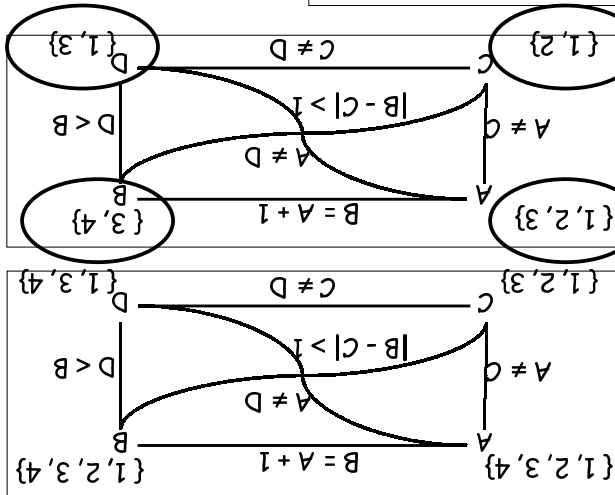
⊙ Deletion_occurred := true



⊙ Second pass:

The example (2):

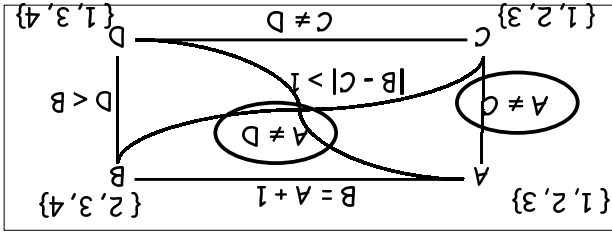
⊙ Deletion_occurred := true



⊙ First pass (= Look ahead check):

The example (1):

⊙ QUEUE = {c(B,C), c(B,D), c(C,D)}



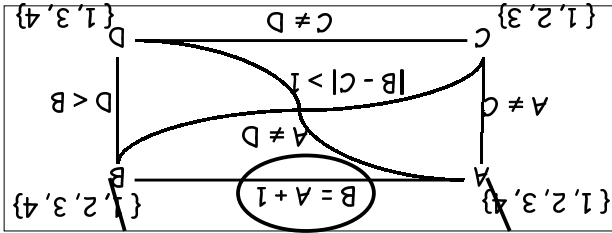
⊙ ~~QUEUE = {c(A,C), c(A,D), c(B,C), c(B,D), c(C,D)}~~

The example (2):

⊙ QUEUE = {c(A,C), c(A,D), c(B,C), c(B,D), c(C,D)}

All already in QUEUE!

⊙ To be added: c(A,C), c(A,D), c(B,C), c(B,D)



⊙ ~~QUEUE = {c(A,B), c(A,C), c(A,D), c(B,C), c(B,D), c(C,D)}~~

The example (1):

AC3:

QUEUE := {all constraints in the problem}

While not empty(QUEUE) DO

Remove c(x,y) from QUEUE;

Remove all inconsistent values from domains Dx and Dy with respect to c(x,y);

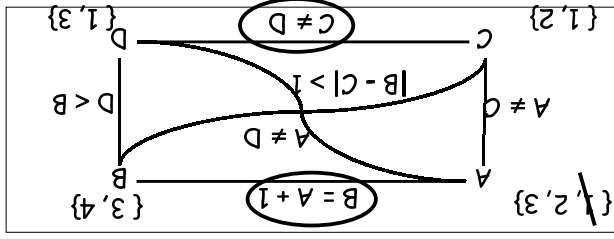
If some value was removed from Dx (or Dy) then add all other constraints involving x (or y) to QUEUE;

End-While

AC-3 (Mackworth)
More efficient arc-consistency:

⊙ QUEUE = {c(A,C), c(A,D)}

⊙ To be added: c(A,C), c(A,D)

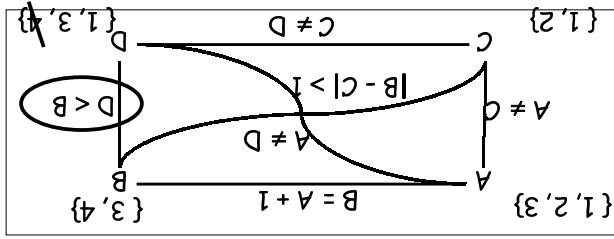


~~⊙ QUEUE = {c(A,B), c(A,C), c(A,D)}~~

The example (5):

⊙ QUEUE = {c(C,D), c(A,B), c(A,C), c(A,D)}

⊙ To be added: c(A,D), c(C,D)

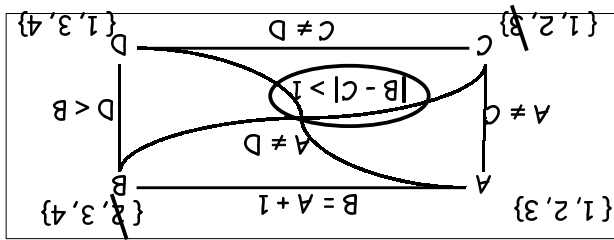


~~⊙ QUEUE = {c(B,D), c(C,D), c(A,B), c(A,C)}~~

The example (4):

⊙ QUEUE = {c(B,D), c(C,D), c(A,B), c(A,C)}

⊙ To be added: c(A,B), c(A,C), c(B,D), c(C,D)



~~⊙ QUEUE = {c(B,C), c(B,D), c(C,D)}~~

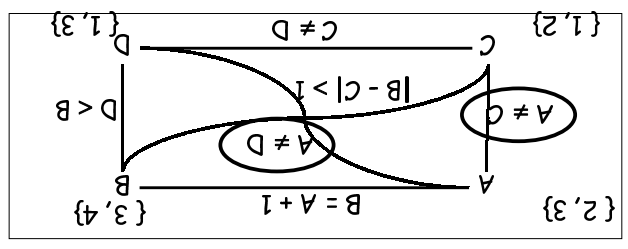
The example (3):

The example (6):

⊙ QUEUE = ~~{(A,C), (A,D)}~~:

⊙ QUEUE = empty

STOP!



Comparison:

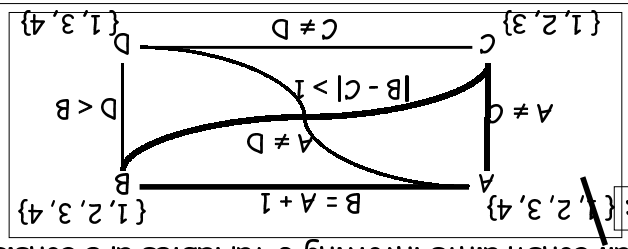
⊙ Same result: full arc-consistency:
 ↪ A = {2,3}, B = {3,4}, C = {1,2}, D = {1,3}

⊙ Efficiency:

- ↪ AC1:
- ◆ 3 times 6 checks = 18
- ↪ AC3:
- ◆ 9 constraint checks

- ⊙ 1-consistency (node-consistency): unary constraints (on 1 variable) are consistent
- ↪ 2-consistency (arc-consistency): binary constraints (on 2 variables) are consistent
- ⊙ 3-consistency: all constraints involving 3 variables are consistent

Example: {1, 2, 3, 4}



⊙ A value remains in the domain if there are consistent values in the domains of the 2 other variables (for all connecting constraints)

Backtracking combined
with Forward Check

Forward checking

Combine the power of
exhaustive (backtrack) search
with (relaxation) pruning

Hybrid constraint processing

- ⊙ Checking k -consistency for $k > 2$ is very hard to do efficiently !!
- ⊙ Example: 4-consistency for the 4-houses puzzle is equivalent to finding solutions to the original problem.

Practicality of k -consistency:

Forward checking:

Forward Checking:
 Execute Standard Backtracking
 BUT
 After each assignment of a
 value to a variable z_i DO
 Forward Check(z_i)

Forward checking at work



Lookahead checking

Backtracking combined
 with Look ahead check

◉ Usually: forward checking is best trade-off
 ◉ For VERY strongly constrained problems:
 ◉ Lookahead checking is needed to prune more

- ◉ Forward checking:
 - ➔ does less consistency checking
 - ➔ has more branching
 - ◆ closer to backtracking
- ◉ Lookahead checking:
 - ➔ spends more work on consistency
 - ➔ tries less alternative values

Which is best?



Lookahead checking at work

Lookahead Checking:
 Look Ahead Check :
 Execute Standard Backtracking
 BUT
 After each assignment of a
 value to some variable DO
 Look Ahead Check

Lookahead checking:

Alternative techniques

⊙ Linear programming

➔ numerical techniques for solving systems of

linear equations (and inequalities) + optimization

problems

◆ Ex.: simplex algorithm

⊙ Works VERY well for 'linear' constraints

$4 * X - 3 * Y > Z + 2$	YES!
$X^3 - 3 * Y > Z^2 + 2$	NO!

⊙ Works, but not VERY well, for discrete problems

⊙ In such cases: Constraint Processing is a better option

⊙ Also: for constraint problems on non-numerical data!

38

Applications:

⊙ All combinatorial search problems

⊙ Scheduling problems:

➔ Ex.: reschedule the trains when some railway

problem has occurred

⊙ Rostering problems:

➔ Ex.: compute work-shifts, given various

expertise constraints and personal preferences

⊙ Production planning:

➔ Ex.: schedule the optimal workflow traversal

⊙ Loading problems:

➔ Ex.: optimize truck-space given various types of

loads

37