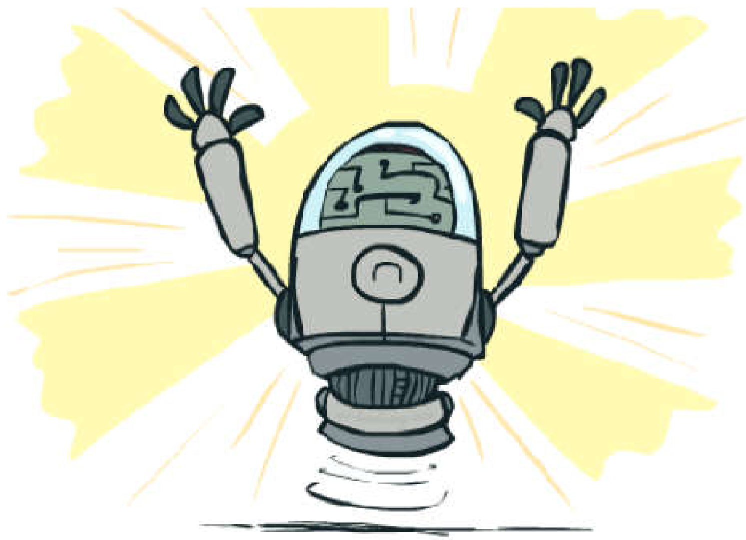
Improving Backtracking

- General-purpose ideas give huge gains in speed

- Ordering:
  - Which variable should be assigned next?
  - In what order should its values be tried?
  - [In what order should constraints be checked?]

- Filtering: Can we detect inevitable failure early?
- Structure: Can we exploit the problem structure?

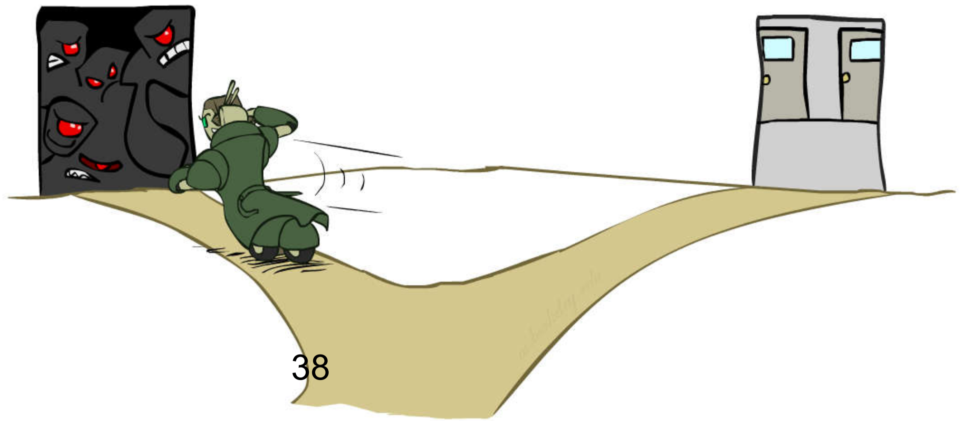


Ordering

Ordering: Minimum Remaining Values

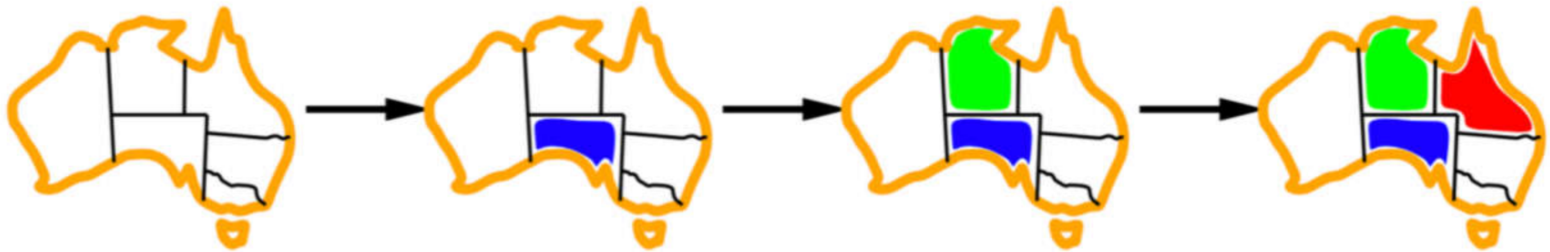
- Variable Ordering: Minimum remaining values (MRV):
  - Choose the variable with the fewest legal left values in its domain

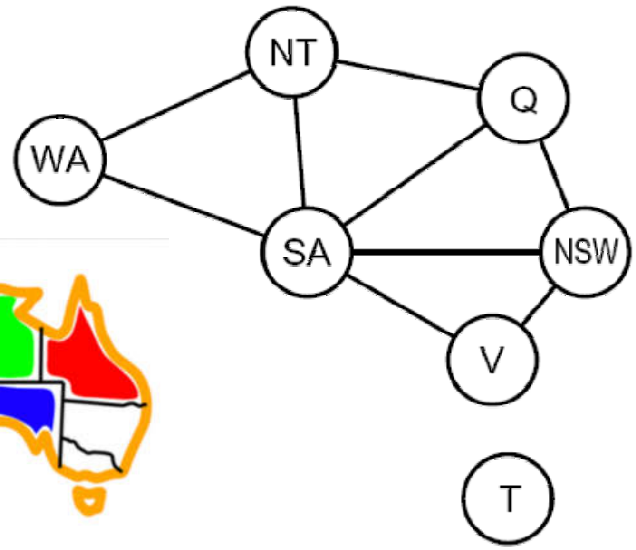
- Why min rather than max?
  - Find bad assignment quickly(fast converge)
- Also called "most constrained variable"
- "Fail-fast" ordering



Ordering: Degree heuristic

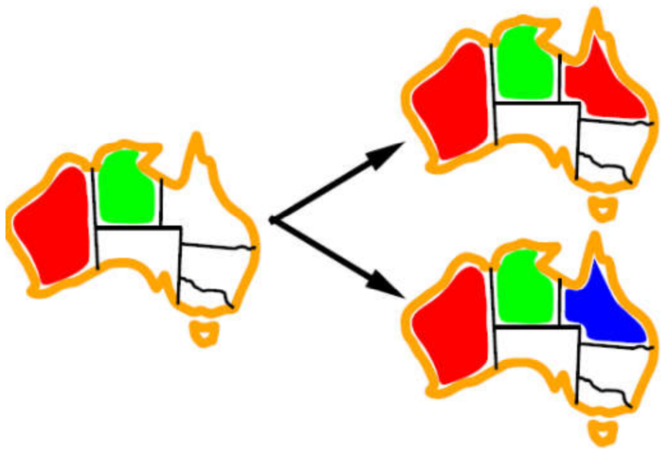
- Variable Ordering: when MRV are same
- Choose the variable with the most constraints on remaining variables





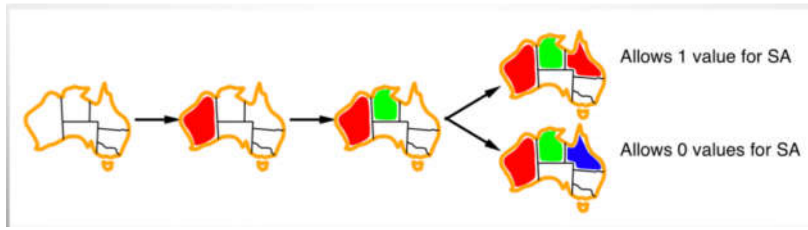
Ordering: Least Constraining Value

- Value Ordering: Least Constraining Value
  - Given a choice of variable, choose the *least constraining value*
  - I.e., the one that rules out the fewest values in the remaining variables
  - Note that it may take some computation to determine this! (E.g., rerunning filtering)



- Why least rather than most?

- Combining these ordering ideas makes 1000 queens feasible

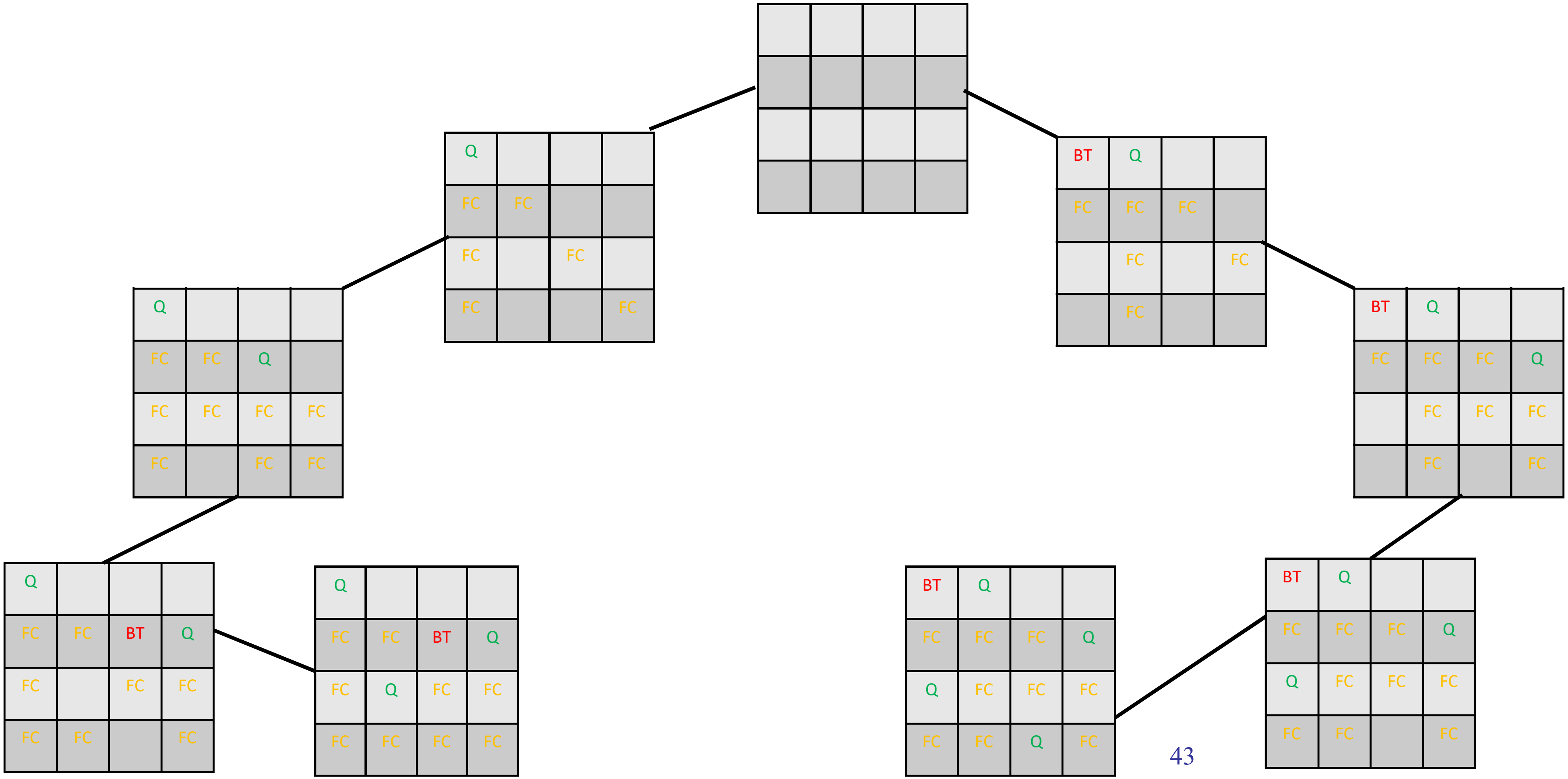


Filtering

Filtering: Forward Checking

- Idea1:
  - Filtering: Keep track of domains for unassigned variables and cross off bad options
  - Forward checking: Cross off values that violate a constraint when added to the existing assignment
  - Stop when there is no legal option for domains of a unassigned variable(save time just one step)

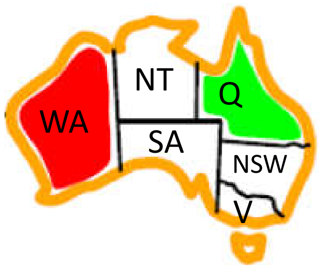
Forward Checking

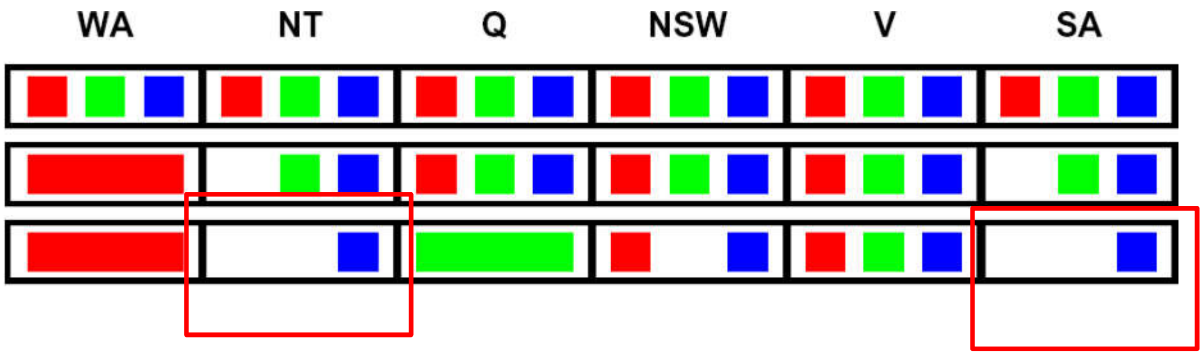


[40-417, SUT]

Filtering: Constraint Propagation

- **Forward checking** propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:
  - NT and SA cannot both be blue!
  - Why didn't we detect this yet?

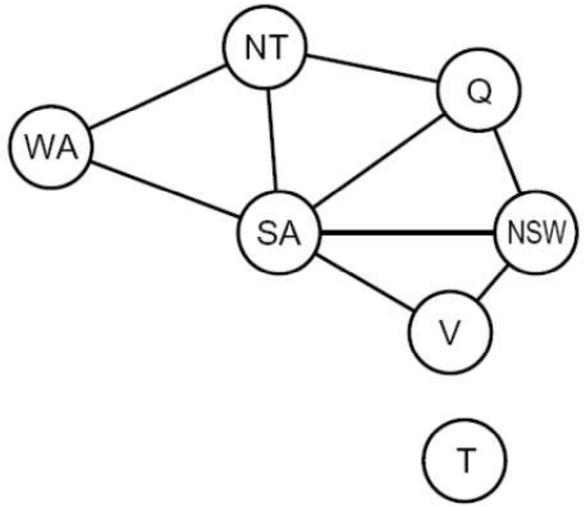




- Idea2: check constraint in next unassigned variable(save more time)
- *Constraint propagation:* reason from constraint to constraint

Consistency of A Arc

- The idea of forward checking can be generalized into
  the principle of arc consistency
- Each undirected edge of the constraint graph as two directed edges pointing in opposite directions.
- Each of these directed edges is called an arc.

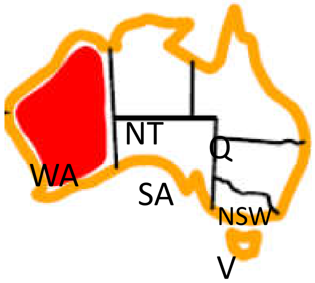


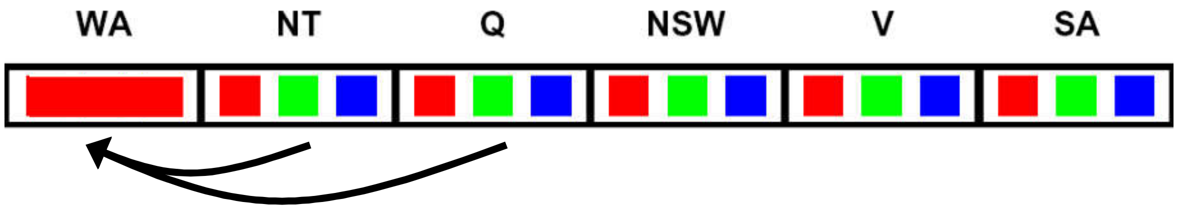
- Arc -> constraint check direction

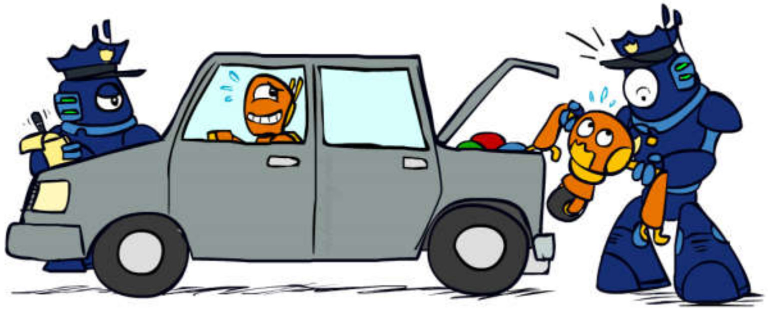
- adding all arcs between unassigned variables
  sharing a constraint to a queue Q

Consistency of A Single Arc

- An arc X $\rightarrow$ Y is consistent iff for *every* x there is *some* y which could be assigned without violating a constraint





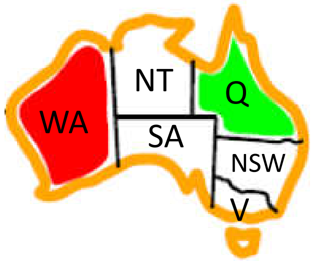


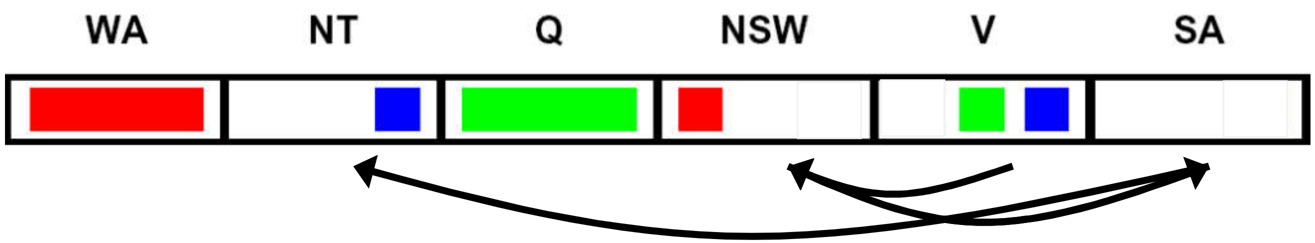
Delete from the tail!

- Forward checking: Enforcing consistency of arcs pointing to each new assignment

Arc Consistency of an Entire CSP

- A simple form of propagation makes sure all arcs are simultaneously consistent:





- Arc consistency detects failure earlier than forward checking
- Important: If X loses a value, neighbors of X need to be rechecked!
- Can be run as a preprocessor or after each assignment
- What's the downside of enforcing arc consistency?

Remember: Delete from the tail!

Enforcing Arc Consistency in a CSP

function AC-3(csp) **returns** the CSP, possibly with reduced domains
  inputs: csp, a binary CSP with variables $\{X_1,\ X_2,\ \ldots,\ X_n\}$
  local variables: $queue$, a queue of arcs, initially all the arcs in csp

  while $queue$ is not empty **do**
    $(X_i,\ X_j) \leftarrow$ REMOVE-FIRST($queue$)
    if REMOVE-INCONSISTENT-VALUES($X_i,\ X_j$) **then**
      for each X_k **in** NEIGHBORS[X_i] **do**
        add $(X_k,\ X_i)$ to $queue$

---

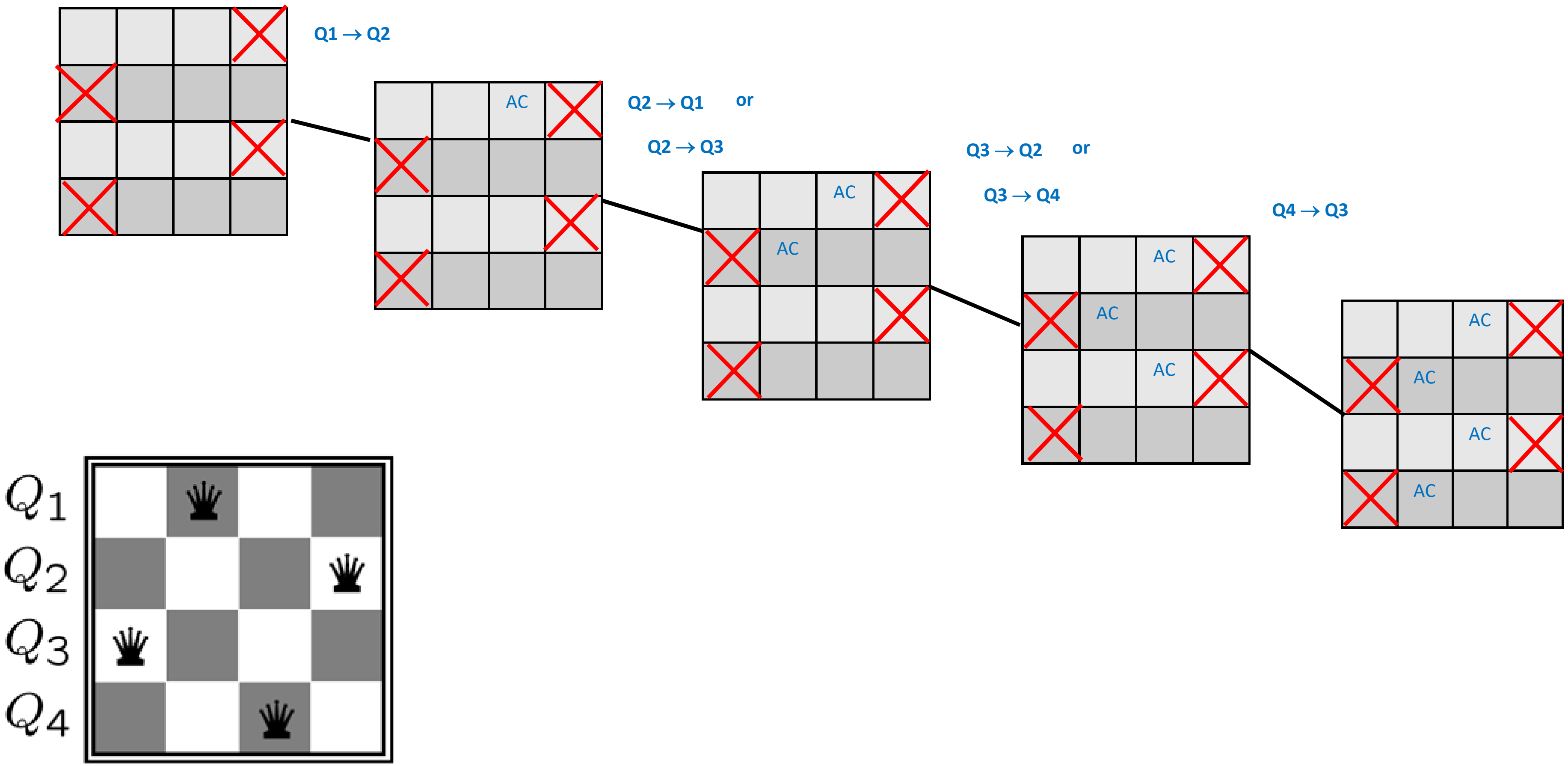
function REMOVE-INCONSISTENT-VALUES($X_i,\ X_j$) **returns** true iff succeeds
  $removed \leftarrow false$
  for each x **in** DOMAIN[X_i] **do**
    if no value y in DOMAIN[X_j] allows (x,y) to satisfy the constraint $X_i \leftrightarrow X_j$
      then delete x from DOMAIN[X_i]; $removed \leftarrow true$
  return $removed$

- Runtime: $O(n^2d^3)$, can be reduced to $O(n^2d^2)$
- … but detecting all possible future problems is NP-hard – why?

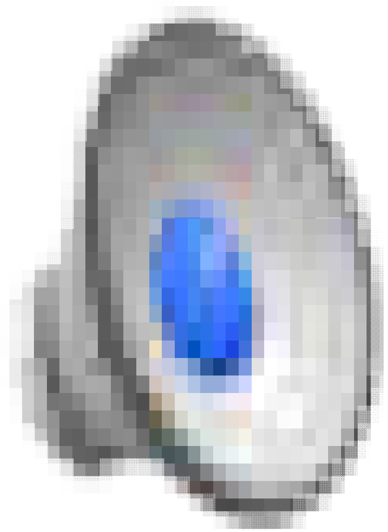
AC-3: time complexity

- Time complexity (n variables, c binary constraints, d domain size): $O(cd^3)$
  - Each arc (X_k, X_i) is inserted in the queue at most d times.
    - At most all values in domain X_i can be deleted.
  - Checking consistency of an arc: $O(d^2)$

Arc Consistency



Video of Demo Arc Consistency – CSP Applet – n Queens



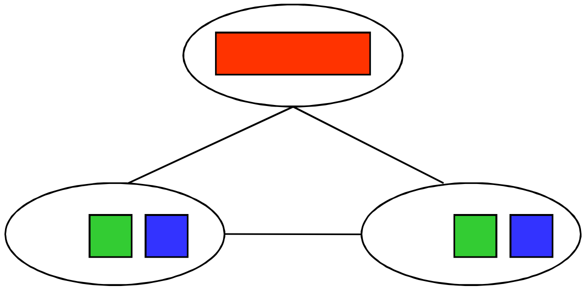
Ordering: Most-constraining-constraint

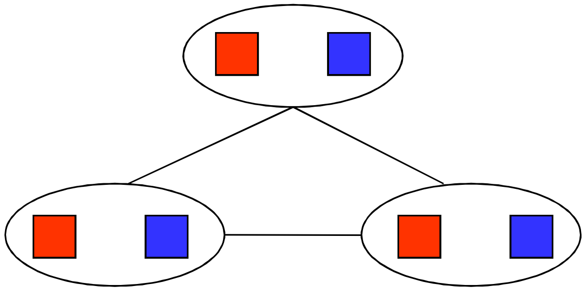
- Constraint Ordering: Most Constraining constraint
- Prefer testing constraints that are more difficult to
- Called Fail First Principle = “Fail-fast” ordering

Limitations of Arc Consistency

- After enforcing arc consistency:
  - Can have one solution left
  - Can have multiple solutions left
  - Can have no solutions left (and not know it)

- Arc consistency still runs inside a backtracking search!



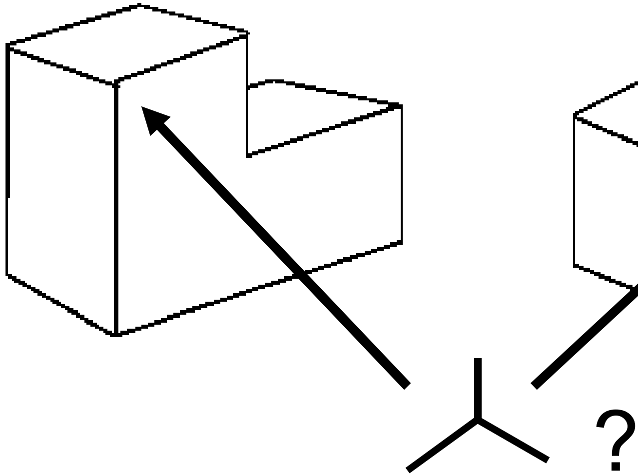


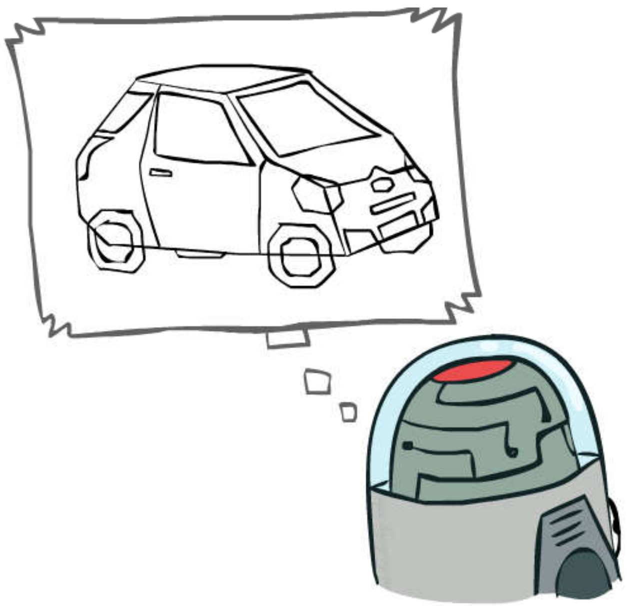
What went wrong here?

[Demo: coloring -- forward checking]
[Demo: coloring -- arc consistency]

Example: The Waltz Algorithm

- The Waltz algorithm is for interpreting line drawings of solid polyhedra as 3D objects
- An early example of an AI computation posed as a CSP

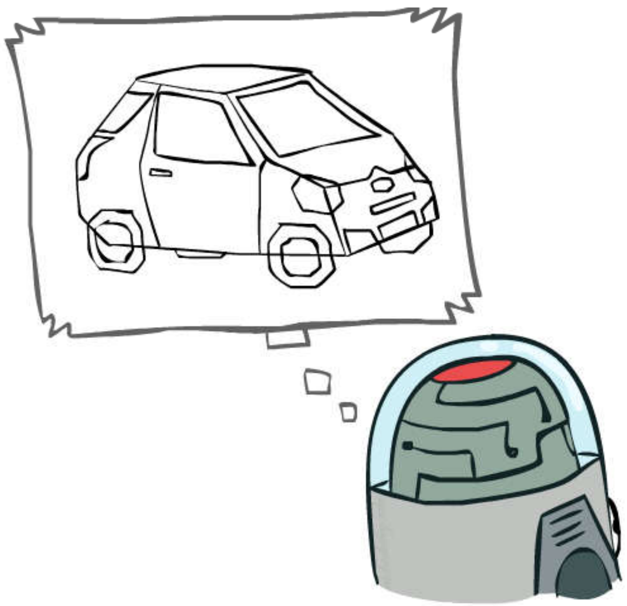


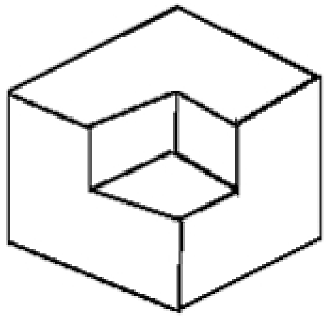


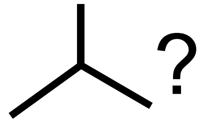
- Approach:
  - Each intersection is a variable
  - Adjacent intersections impose constraints on each other
  - Solutions are physically realizable 3D interpretations

Example: The Waltz Algorithm

- The Waltz algorithm is for interpreting line drawings of solid polyhedra as 3D objects
- An early example of an AI computation posed as a CSP

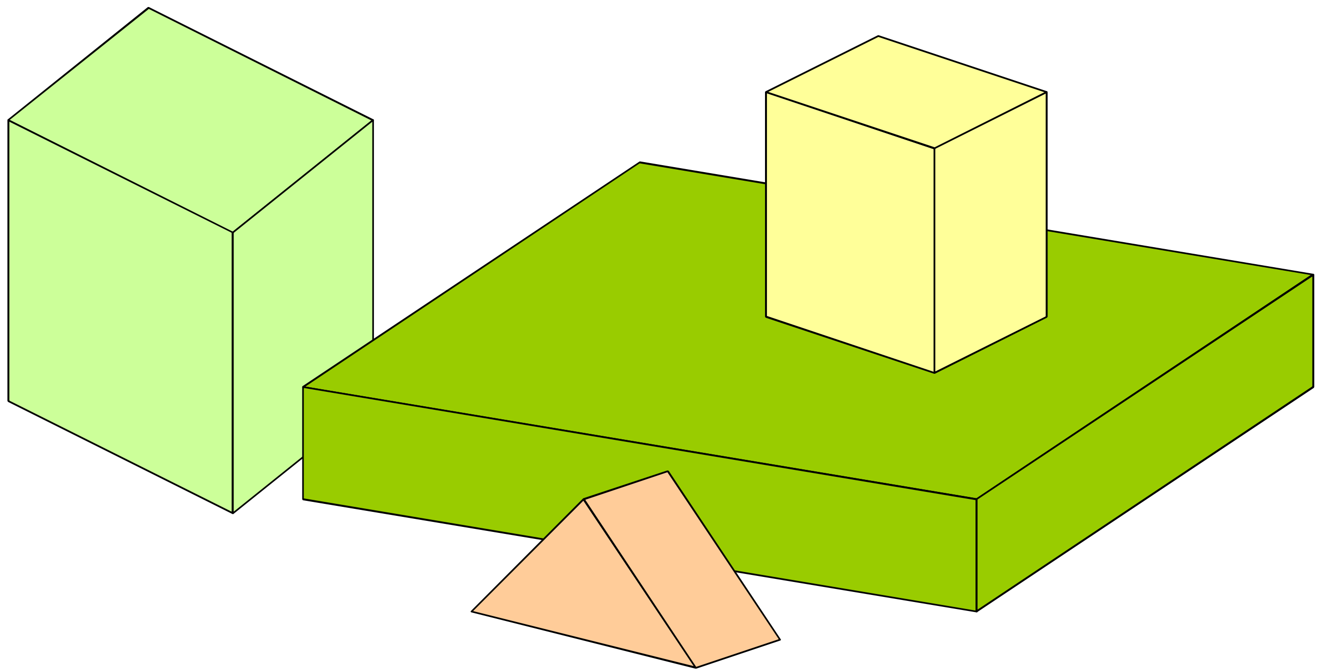




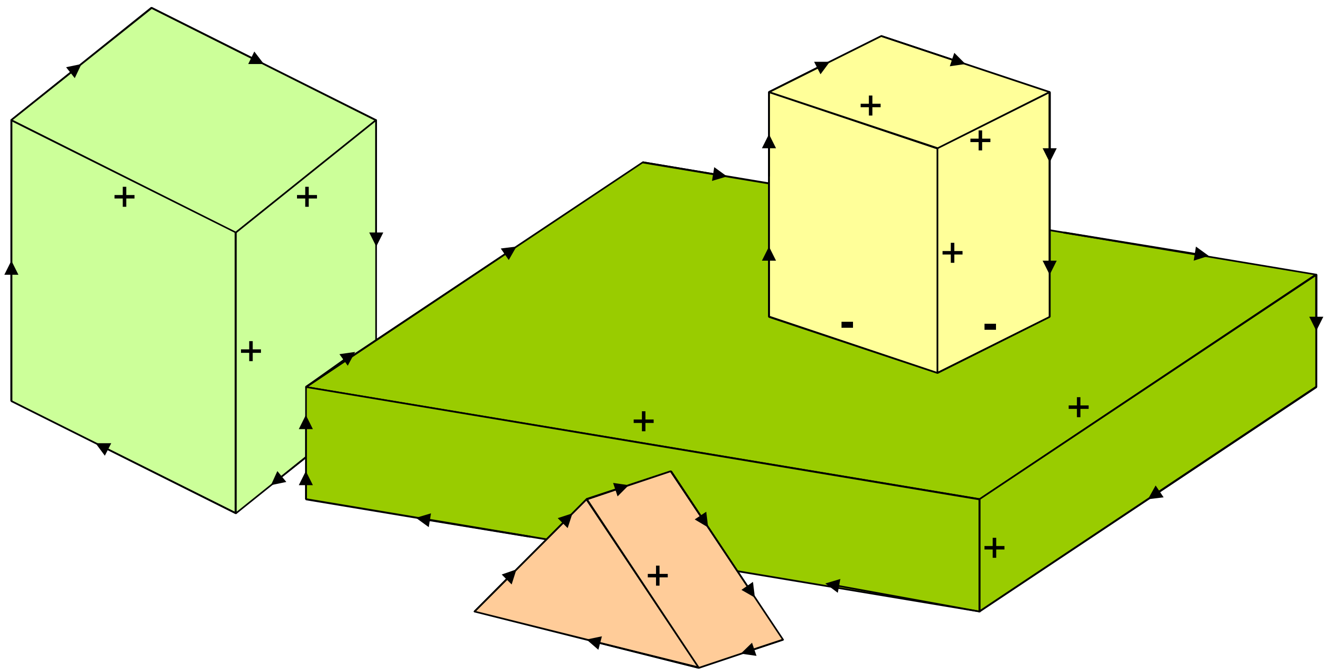


- Approach:
  - Each intersection is a variable
  - Adjacent intersections impose constraints on each other
  - Solutions are physically realizable 3D interpretations

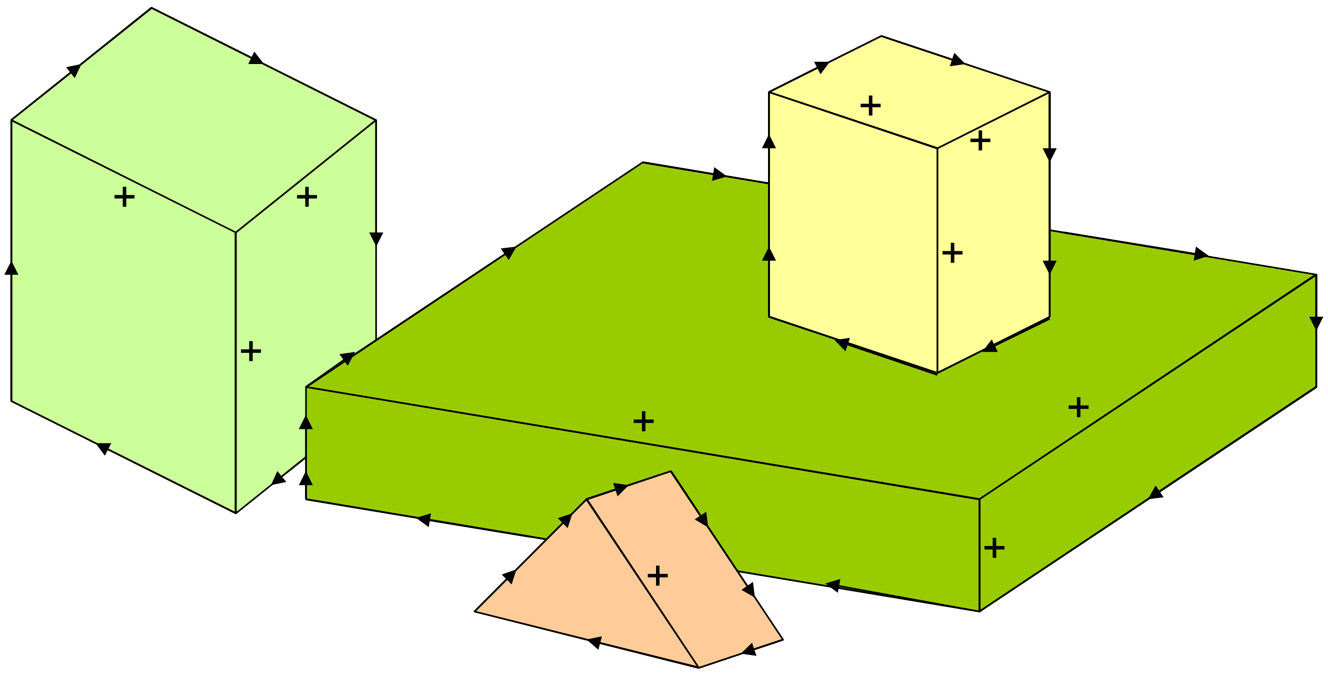
Edge Labeling in Computer Vision



Edge Labeling in Computer Vision



Edge Labeling in Computer Vision

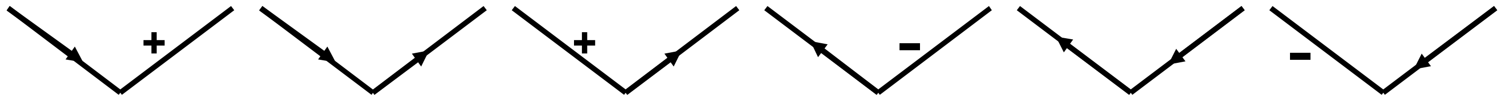


Labels of Edges

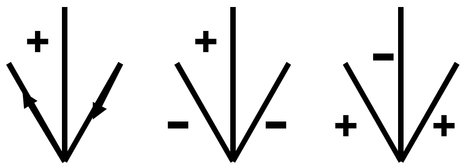
- Convex edge:
  - two surfaces intersecting at an angle greater than 180°
- Concave edge
  - two surfaces intersecting at an angle less than 180°
- + convex edge, both surfaces visible
- − concave edge, both surfaces visible
- ← convex edge, only one surface is visible (i.e., on the right side of ←)

Junction Label Sets

L-junctions

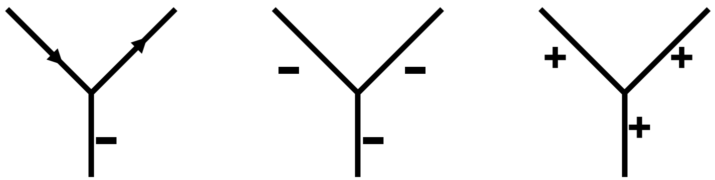


Arrow-junctions

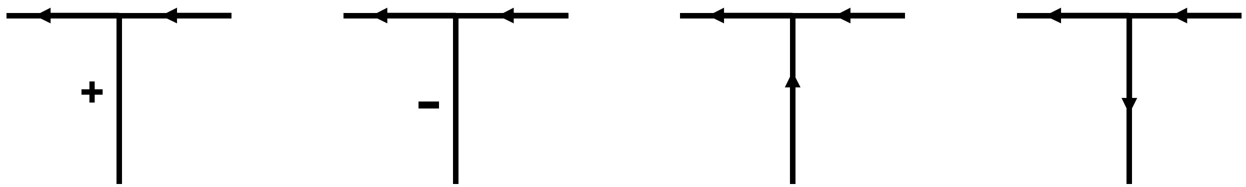


From the total of 208 ($4^2 + 4^3 + 4^3 + 4^3$) possible cases, only 16 is valid

Y-junctions



T-junctions

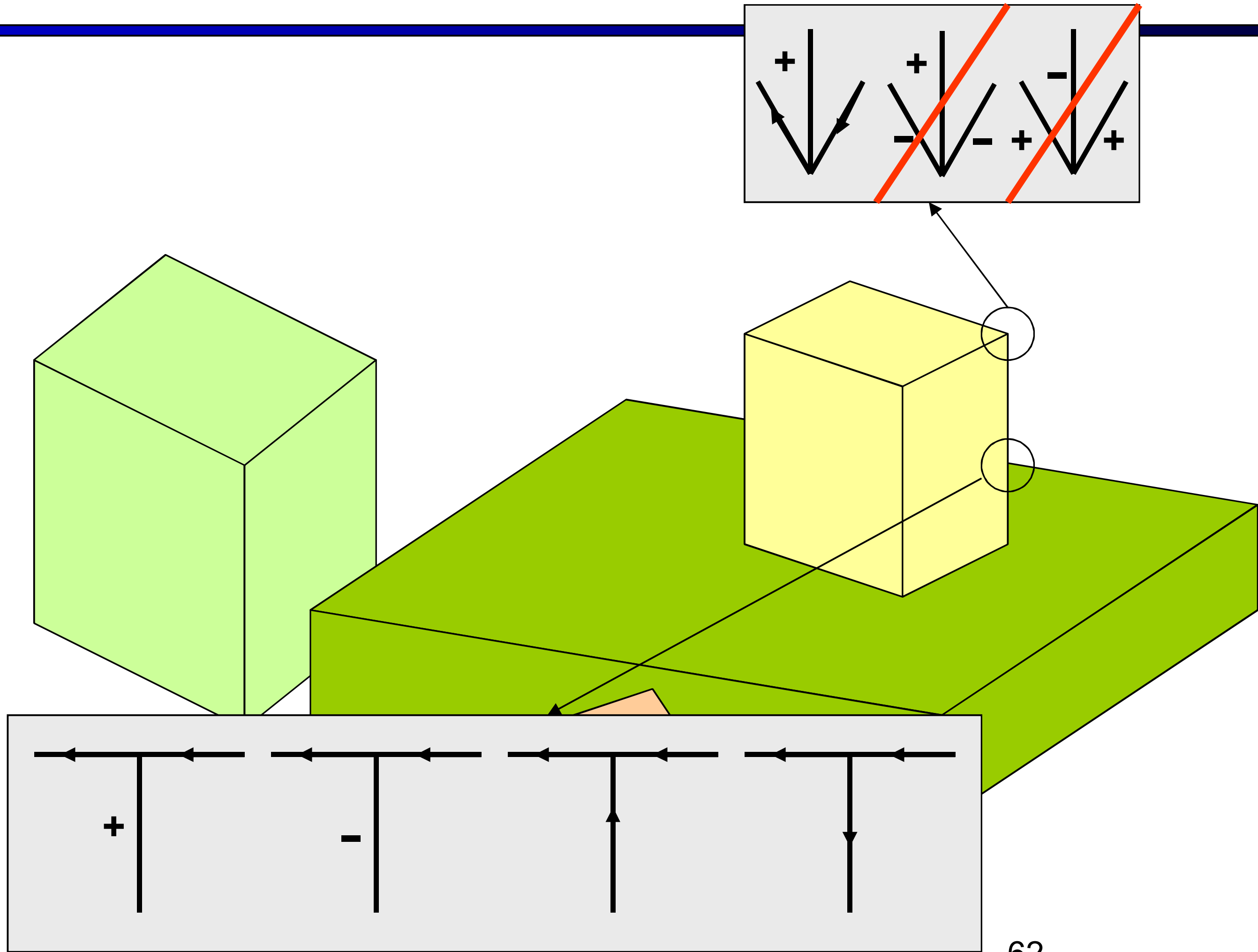


(Waltz, 1975; Mackworth, 1977)

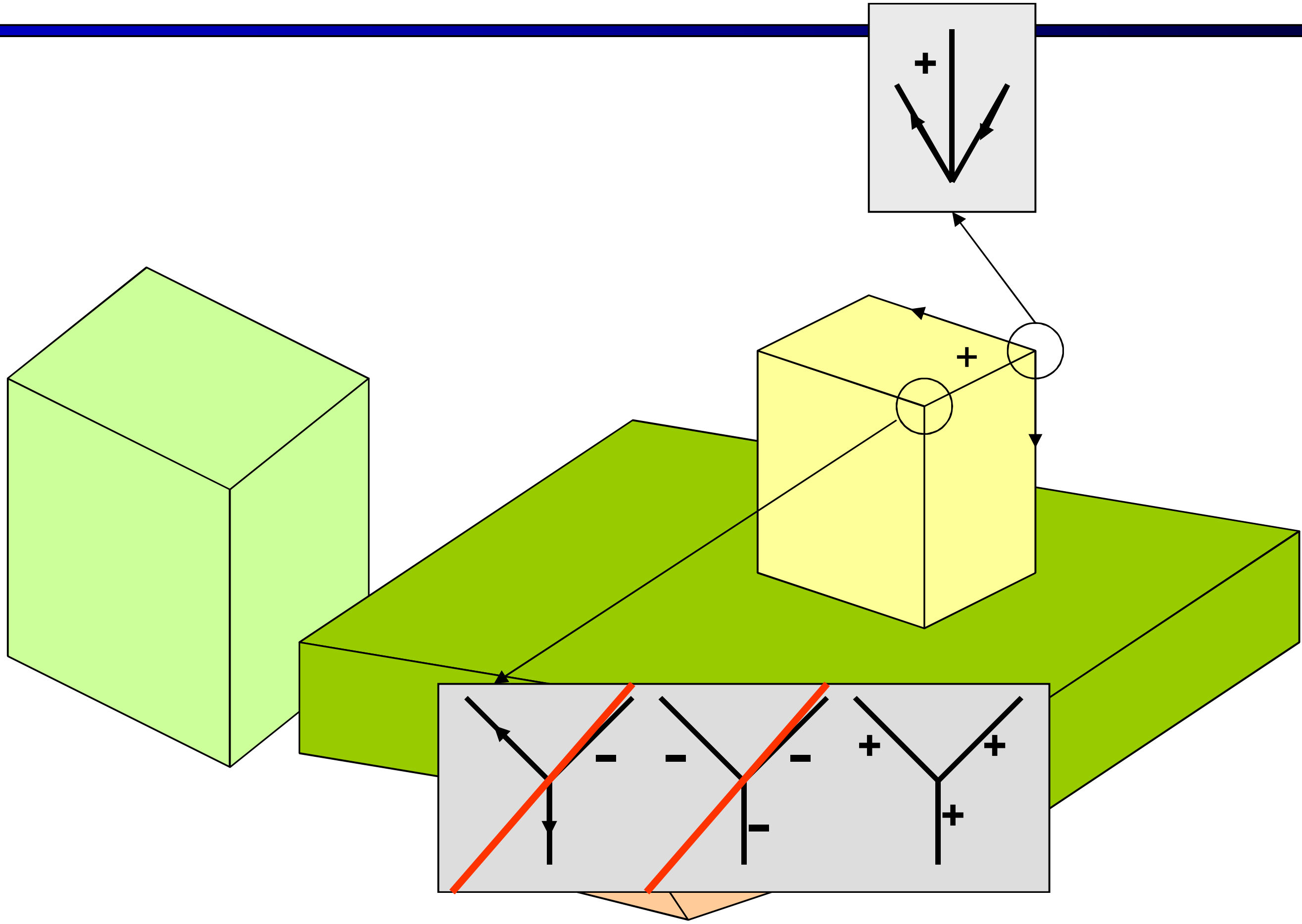
Edge Labeling as a CSP

- A variable is associated with each junction
- The domain of a variable is the label set of the corresponding junction
- Each constraint imposes that the values given to two adjacent junctions give the same label to the joining edge

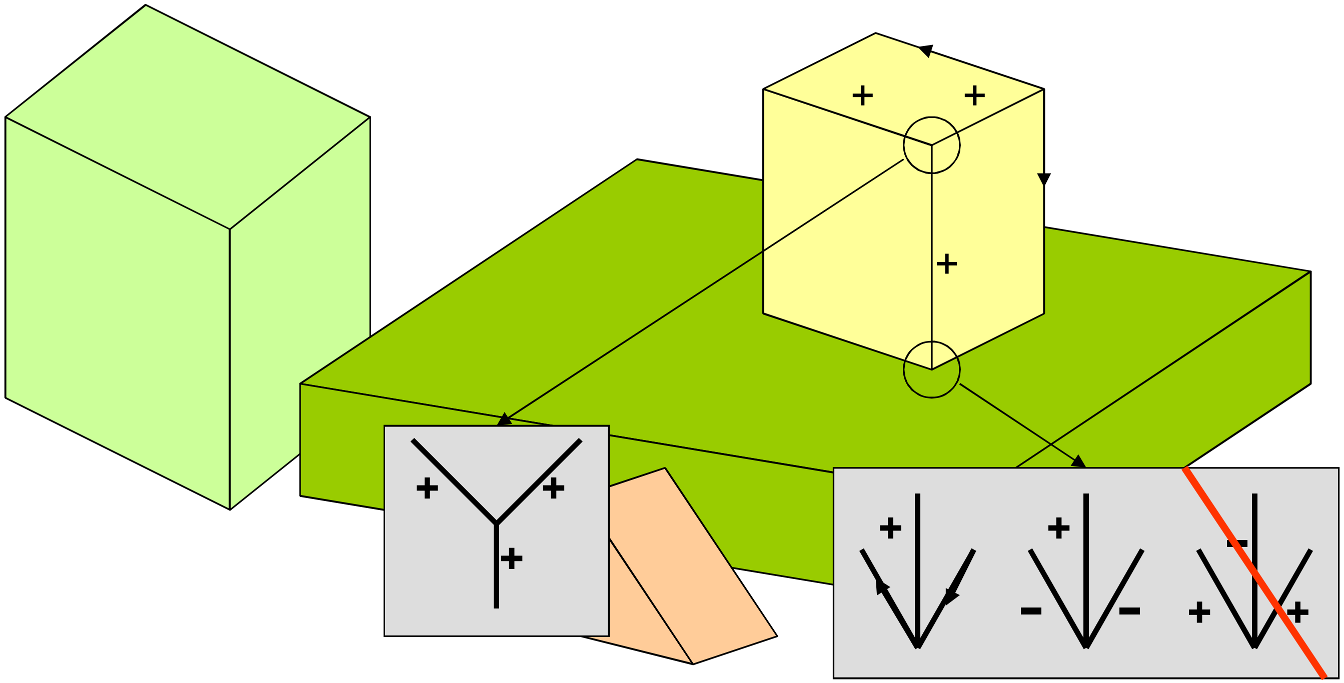
Edge labeling example



Edge labeling example



Edge labeling example



Edge labeling example