Local Search and Constraint Satisfaction

Big Picture - AI

- Problem Solving/Search
- Perception
- Learning
- Language Understanding
- Knowledge Representation
- Reasoning (using Knowledge)
- Robotics
- Etc…

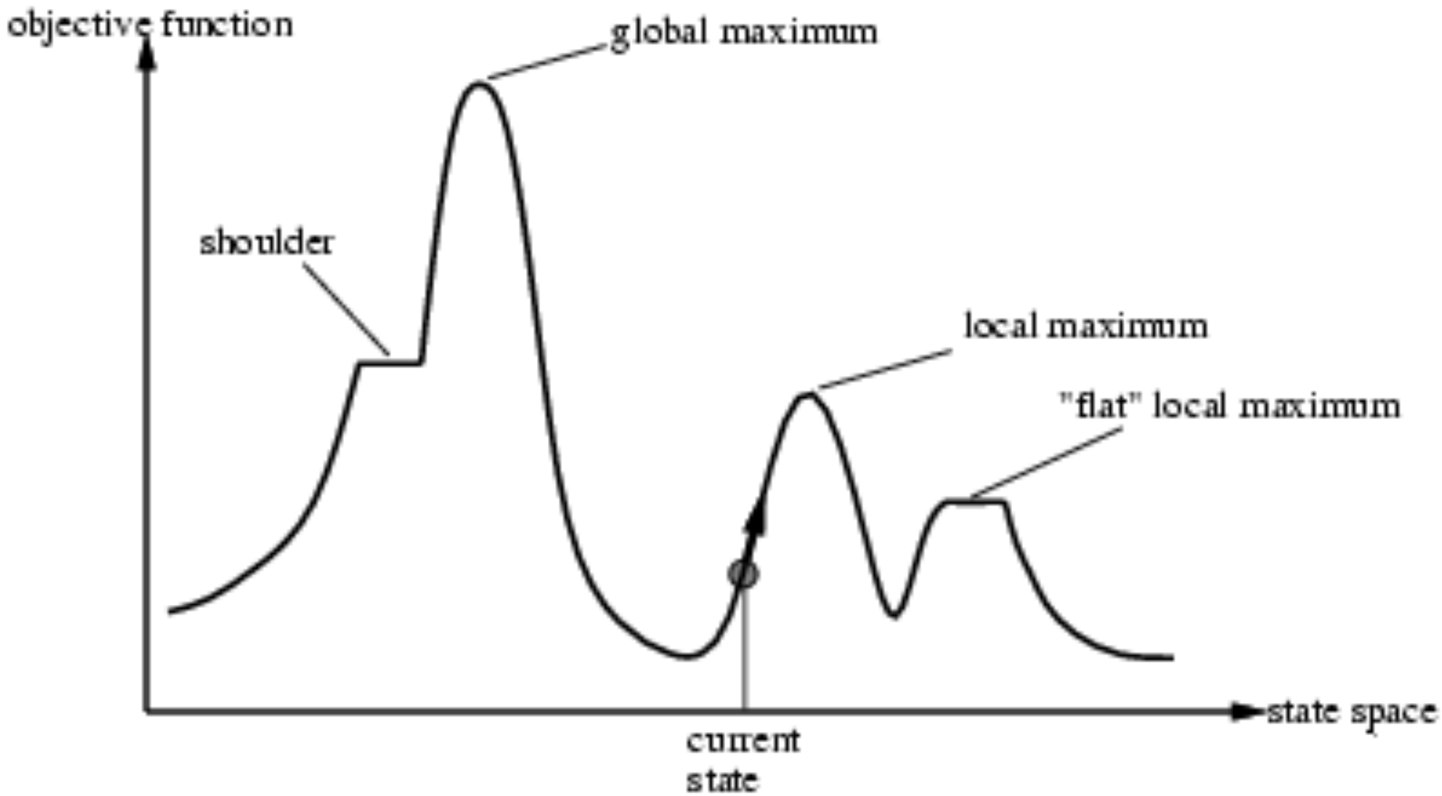
Local Search algorithms

- In many optimization problems, the path to the goal is irrelevant; the goal state itself is the solution

- In such cases, we can use local search algorithms
- keep a single "current" state, try to improve it

•Hill-climbing
•Simulated annealing
•Local Beam Search
•Stochastic Beam Search
•Genetic Algorithms

Problems with hill-climbing?



Hill-climbing Variants

- Stochastic Hill Climbing
- First-choice hill climbing
- Random-restart hill climbing

Simulated annealing search

- Idea: escape local maxima by allowing some "bad" moves but gradually decrease their frequency

```
function SIMULATED-ANNEALING( problem, schedule) returns a solution state
   inputs: problem, a problem
           schedule, a mapping from time to "temperature"
   local variables: current, a node
                    next, a node
                    T, a "temperature" controlling prob. of downward steps

   current ← MAKE-NODE(INITIAL-STATE[problem])
   for t ← 1 to ∞ do
       T ← schedule[t]
       if T = 0 then return current
       next ← a randomly selected successor of current
       ΔE ← VALUE[next] – VALUE[current]
       if ΔE > 0 then current ← next
       else current ← next only with probability e^(Δ E/T)
```

Local beam search

- Keep track of *k* states rather than just one
- Start with *k* randomly generated states
- At each iteration, all the successors of all *k* states are generated
- If any one is a goal state, stop; else select the *k* best successors from the complete list and repeat.

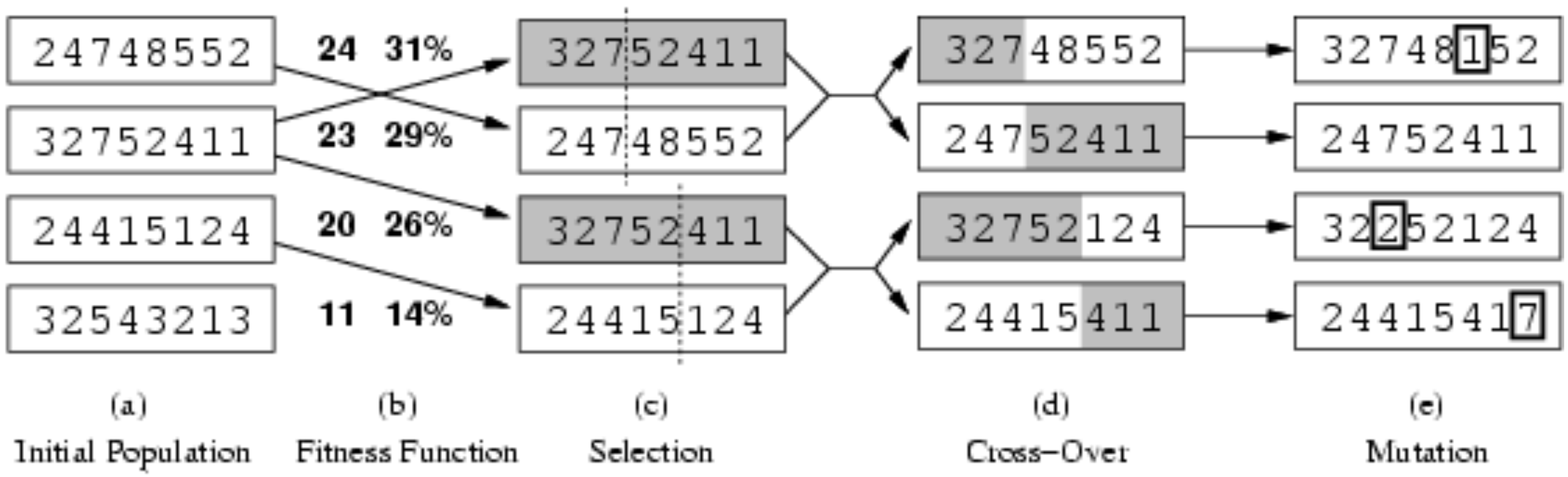
Stochastic Beam Search

- Instead of choosing the *k* best from pool, choose *k* at “random”
- Like natural selection
  - Successors = offspring
  - State = organism
  - Value = fitness

Genetic algorithms

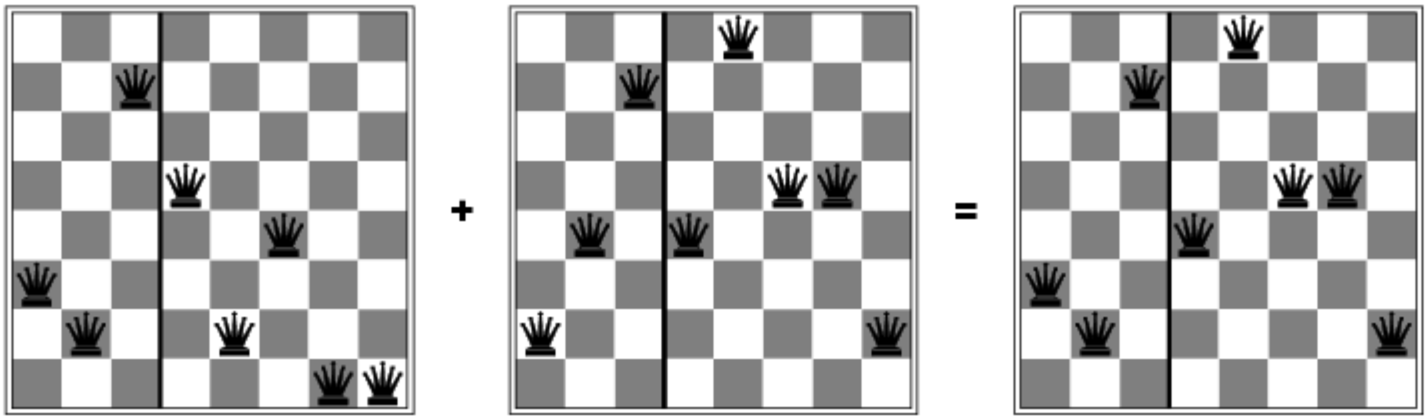
- A successor state is generated by combining two parent states
- Start with *k* randomly generated states (population)
- A state is represented as a string over a finite alphabet (often a string of 0s and 1s)
- Evaluation function (fitness function). Higher values for better states.
- Produce (breed) the next generation of states by selection, crossover, and mutation

Genetic algorithms



- Fitness function: number of non-attacking pairs of queens
- 24/(24+23+20+11) = 31%
- 23/(24+23+20+11) = 29% etc

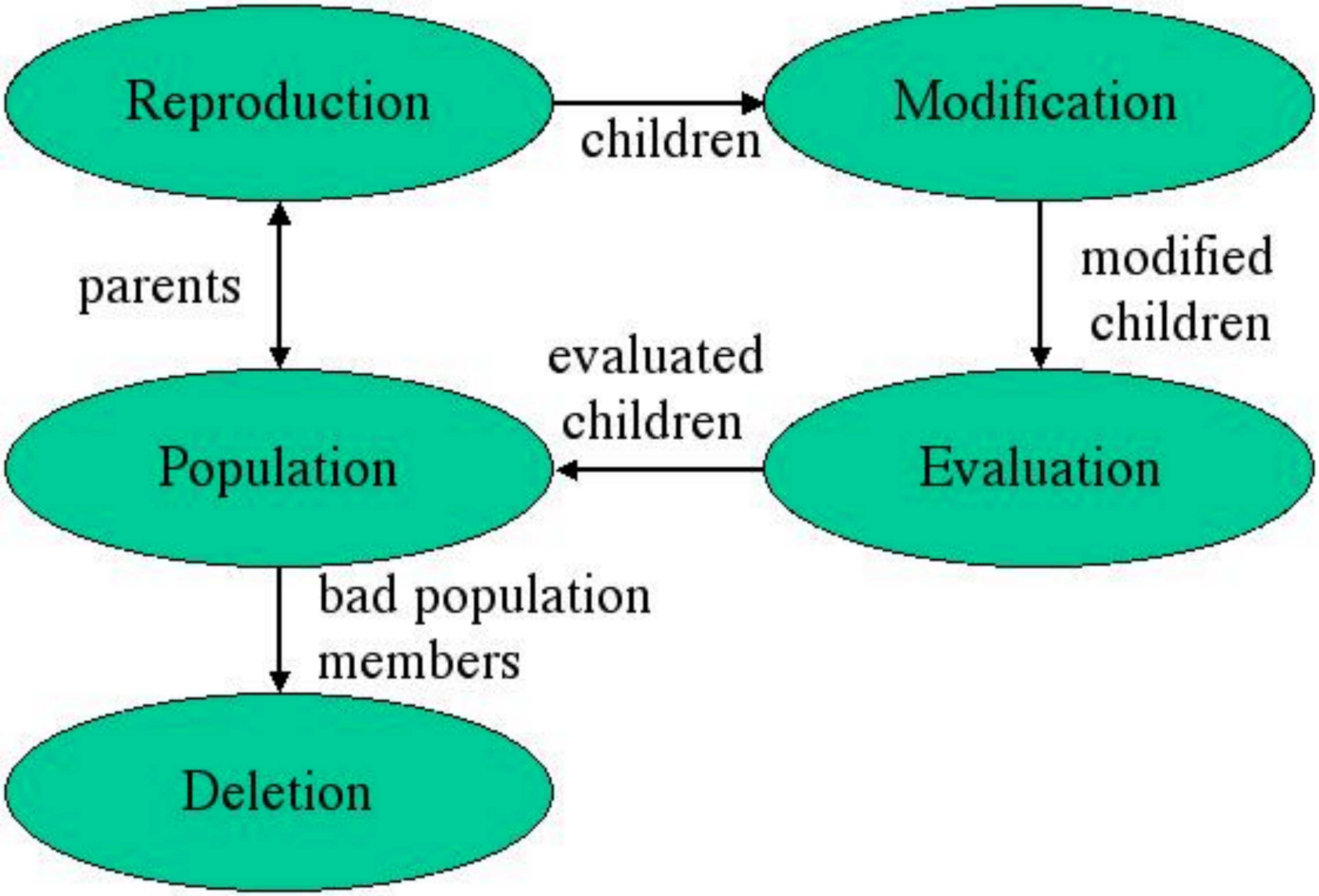
Genetic algorithms



Genetic Algorithms Continued…

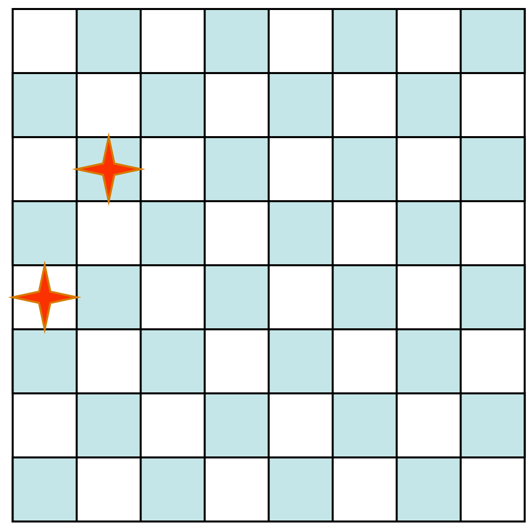
1. Choose initial population
2. Evaluate fitness of each in population
3. Repeat the following until we hit a terminating condition:
   1. Select best-ranking to reproduce
   2. Breed using crossover and mutation
   3. Evaluate the fitnesses of the offspring
   4. Replace worst ranked part of population with offspring

Anatomy of a Genetic Algorithm



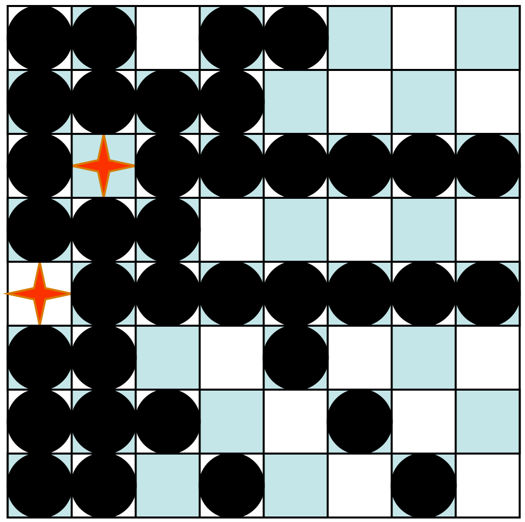
Edge Labeling in Computer Vision - a CSP

Intro Example: 8-Queens



Generate-and-test, with no redundancies → "only" 8^8 combinations

Intro Example: 8-Queens



What is Needed?

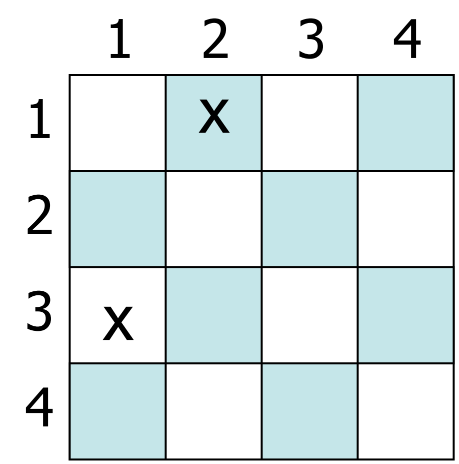
- Not just a successor function and goal test
- But also a means to propagate the constraints imposed by one queen on the others and an early failure test
- → Explicit representation of constraints and constraint manipulation algorithms

Constraint Satisfaction Problem

- Set of variables $\{X_1, X_2, \ldots, X_n\}$
- Each variable X_i has a domain D_i of possible values
  - Usually D_i is discrete and finite
- Set of constraints $\{C_1, C_2, \ldots, C_p\}$
  - Each constraint C_k involves a subset of variables and specifies the allowable combinations of values of these variables
- Assign a value to every variable such that all constraints are satisfied

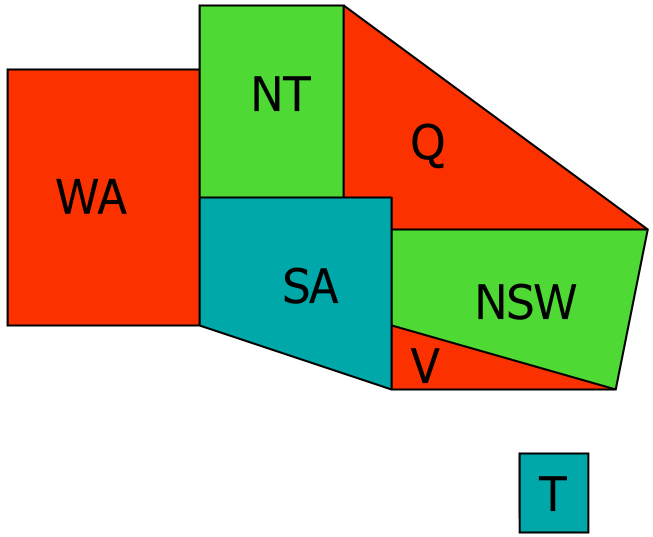
Example: 8-Queens Problem

- 8 variables X_i, i = 1 to 8
- Domain for each variable {1,2,…,8}
- Constraints are of the forms:
  - $X_i = k \Rightarrow X_j \neq k$ for all j = 1 to 8, $j \neq i$
  - $X_i = k_i, X_j = k_j \Rightarrow |i-j| \neq |k_i - k_j|$
    - for all j = 1 to 8, $j \neq i$



$X_1=3$, $X_2=1$

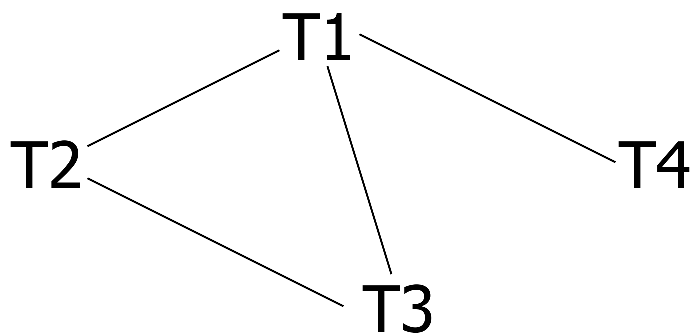
Example: Map Coloring



- 7 variables {WA,NT,SA,Q,NSW,V,T}
- Each variable has the same domain {red, green, blue}
- No two adjacent variables have the same value:

WA≠NT, WA≠SA, NT≠SA, NT≠Q, SA≠Q, SA≠NSW, SA≠V,Q≠NSW, NSW≠V

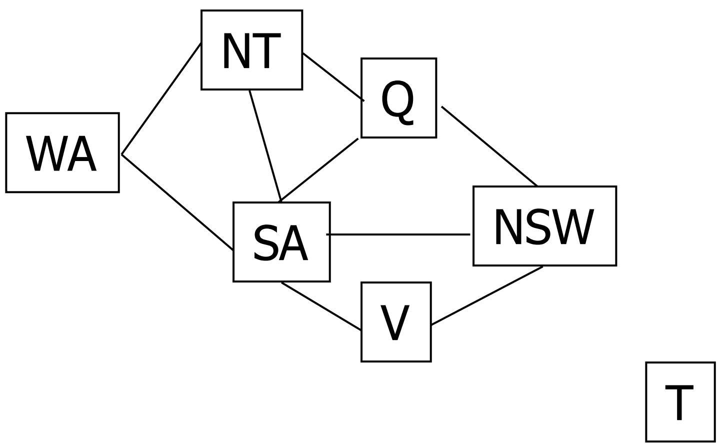
Example: Task Scheduling

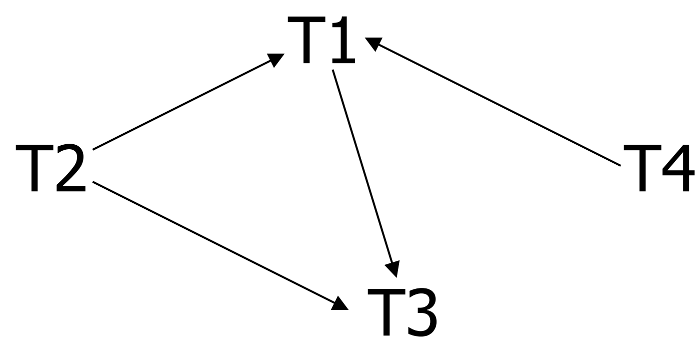


T1 must be done during T3
T2 must be achieved before T1 starts
T2 must overlap with T3
T4 must start after T1 is complete

Constraint Graph

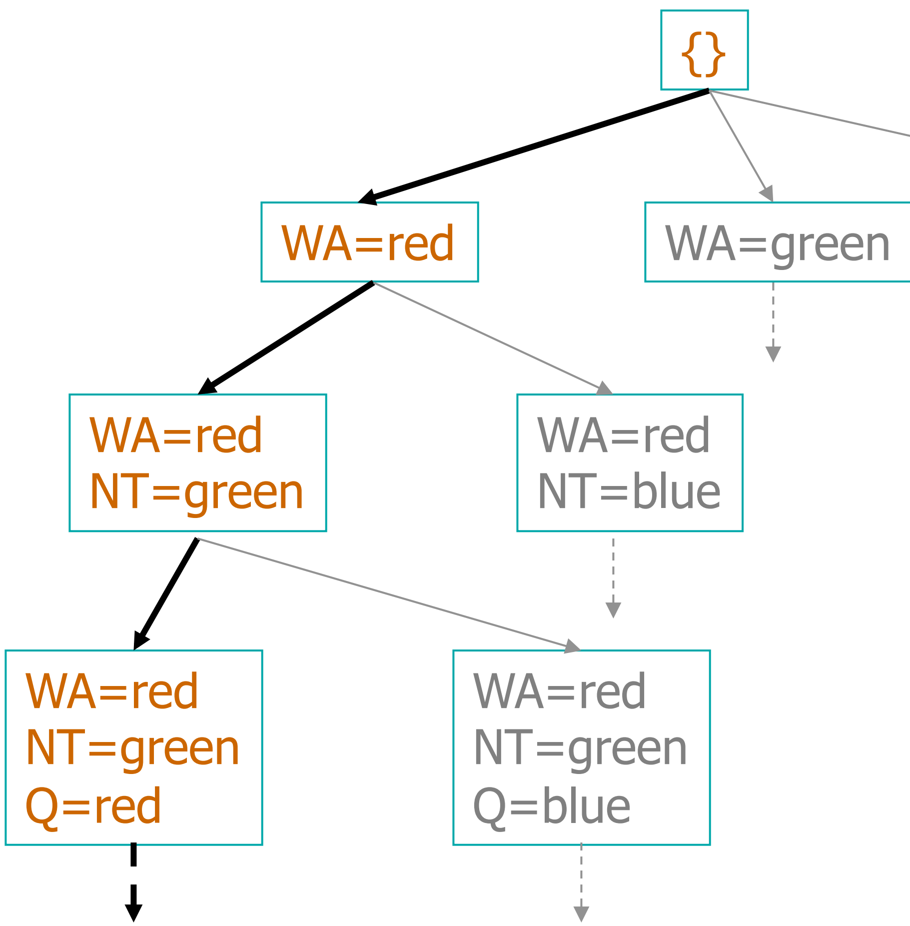
Binary constraints

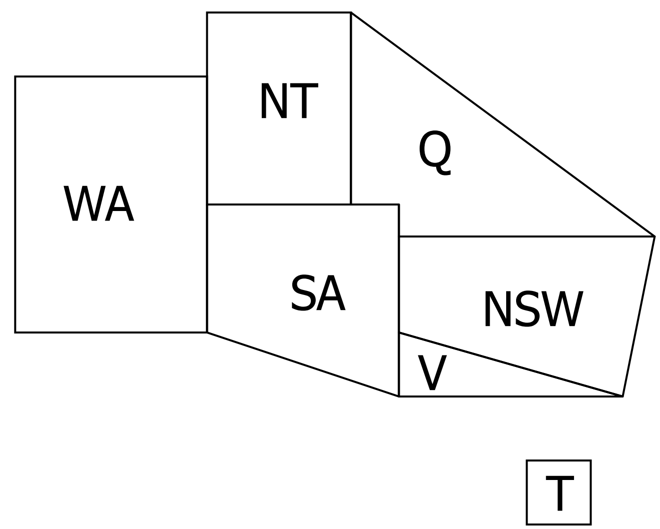




Two variables are adjacent or neighbors if they are connected by an edge or an arc

Map Coloring





Backtracking Algorithm

CSP-BACKTRACKING(PartialAssignment a)

- If a is complete then return a
- X ← select an unassigned variable
- D ← select an ordering for the domain of X
- For each value v in D do
  - If v is consistent with a then
    - Add (X= v) to a
    - result ← CSP-BACKTRACKING(a)
    - If result ≠ *failure* then return result
- Return *failure*

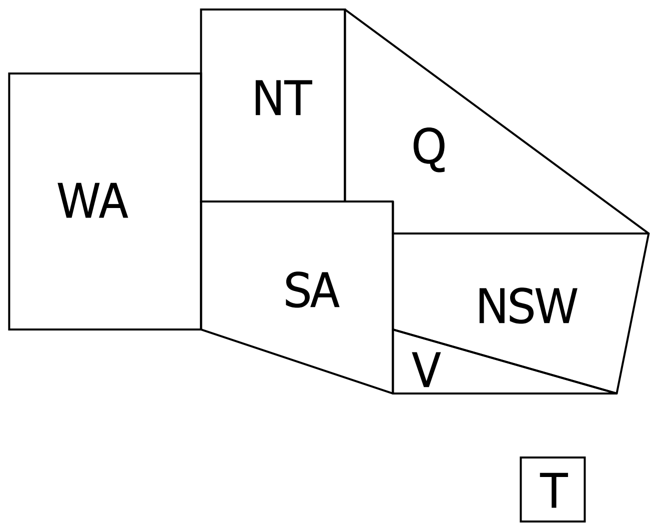
CSP-BACKTRACKING({})

Questions

- Which variable X should be assigned a value next?
- In which order should its domain D be sorted?
- What are the implications of a partial assignment for yet unassigned variables?

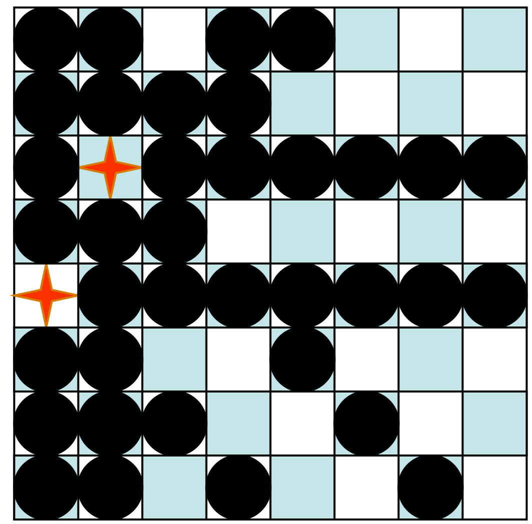
Choice of Variable

- Map coloring

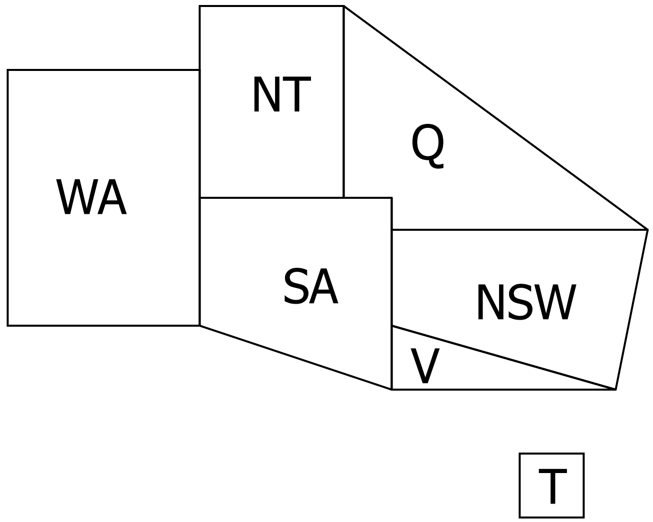


Choice of Variable

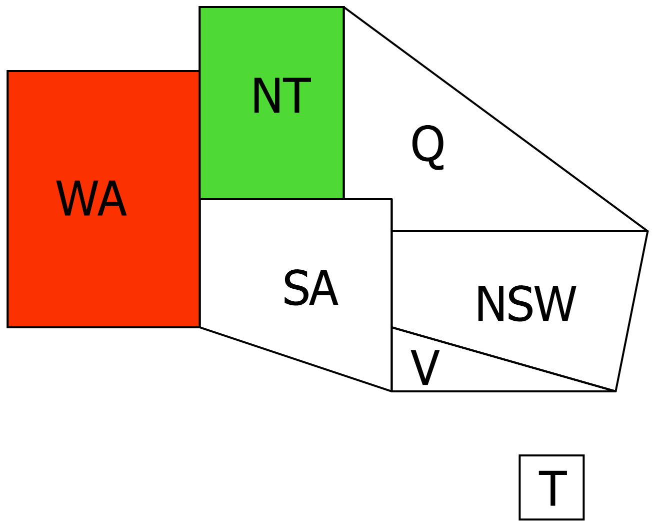
- 8-queen



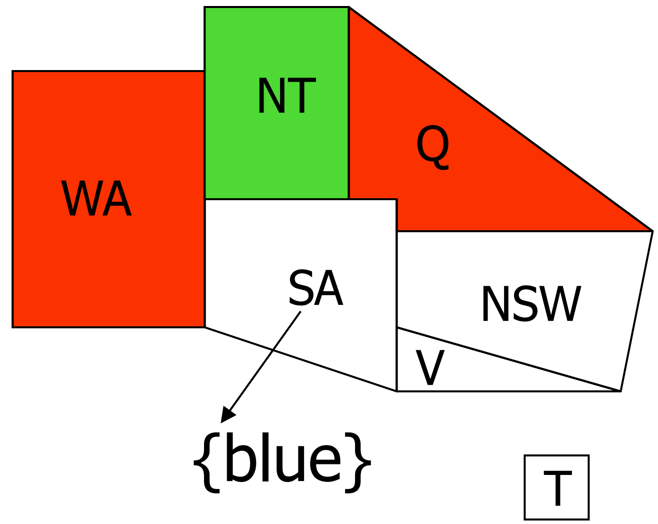
Choice of Variable



Choice of Variable



Choice of Value



Least-constraining-value heuristic:

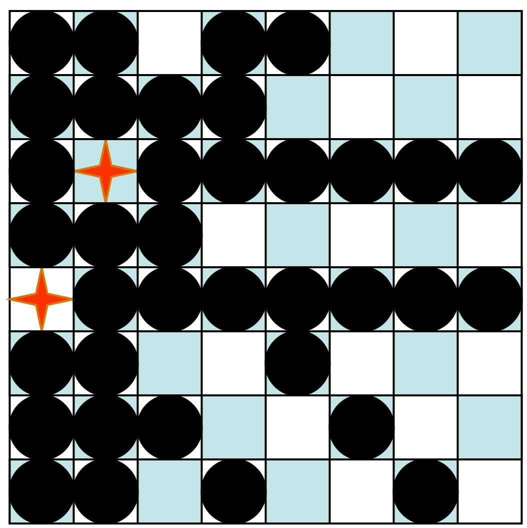
Prefer the value that leaves the largest subset of legal values for other unassigned variables

Eliminating wasted search

- Our goal is to avoid searching branches that will ultimately dead-end
- How can we use the information available at the beginning of the assignment to help with this process?

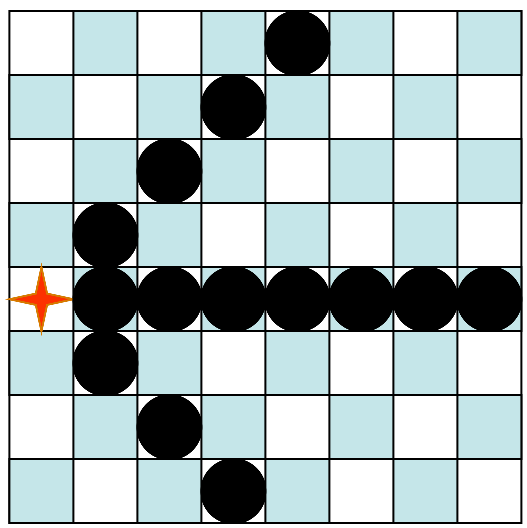
Constraint Propagation …

… is the process of determining how the possible values of one variable affect the possible values of other variables

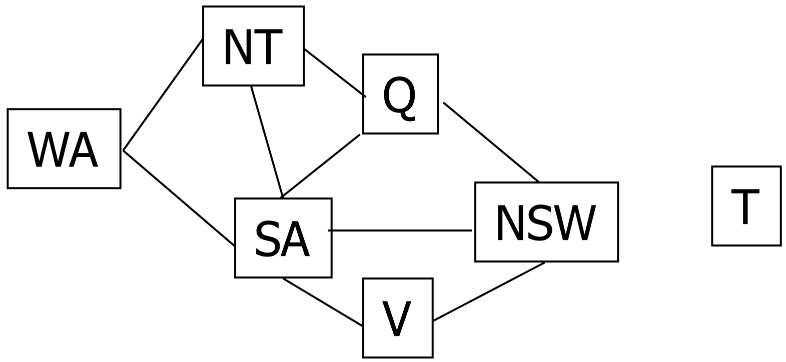


Forward Checking

After a variable X is assigned a value v, look at each unassigned variable Y that is connected to X by a constraint and deletes from Y's domain any value that is inconsistent with v

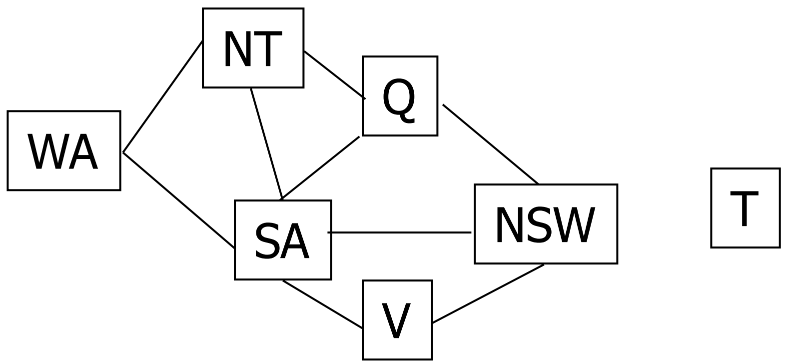


Map Coloring



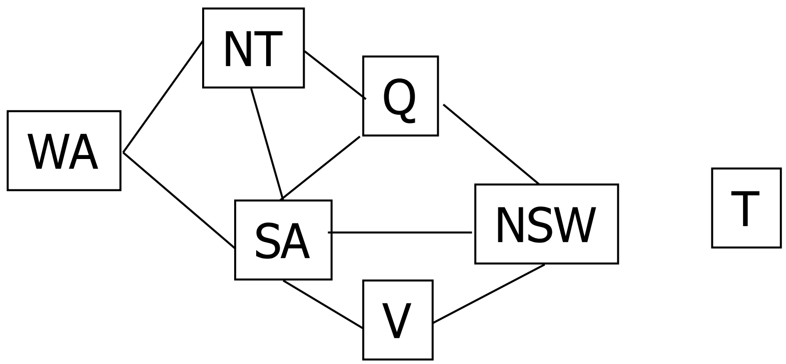
WA	NT	Q	NSW	V	SA	T
RGB	RGB	RGB	RGB	RGB	RGB	RGB

Map Coloring



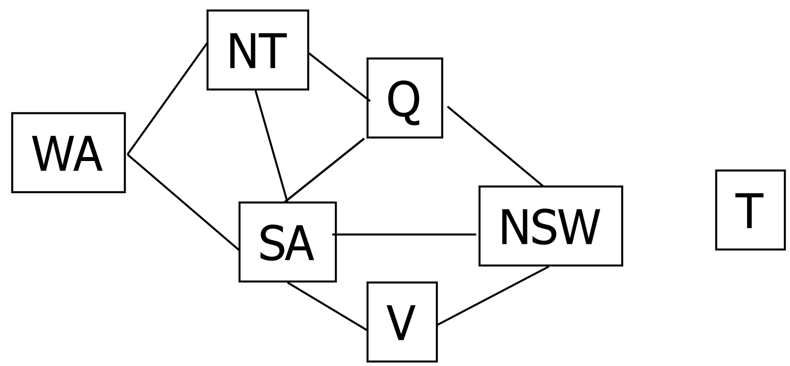
WA	NT	Q	NSW	V	SA	T
RGB	RGB	RGB	RGB	RGB	RGB	RGB
R	GB	RGB	RGB	RGB	GB	RGB

Map Coloring



WA	NT	Q	NSW	V	SA	T
RGB	RGB	RGB	RGB	RGB	RGB	RGB
R	GB	RGB	RGB	RGB	GB	RGB
R	B	G	RB	RGB	B	RGB

Map Coloring



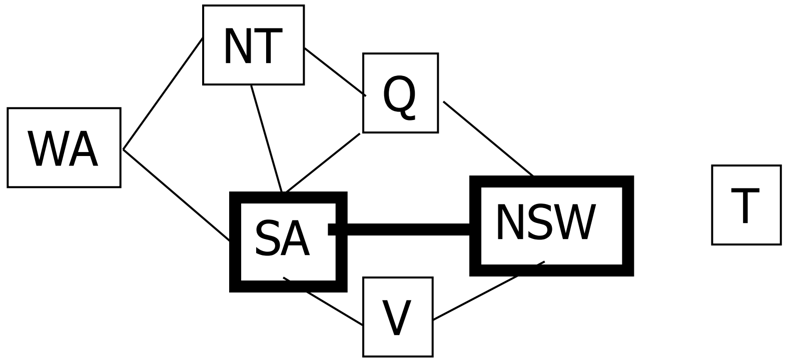
WA	NT	Q	NSW	V	SA	T
RGB	RGB	RGB	RGB	RGB	RGB	RGB
R	GB	RGB	RGB	RGB	GB	RGB
R	B	G	RB	RGB	B	RGB
R	B	G	R	B		RGB

Removal of Arc Inconsistencies

REMOVE-ARC-INCONSISTENCIES(J,K)

- removed ← *false*
- X ← label set of J
- Y ← label set of K
- For every label y in Y do
  - If there exists no label x in X such that the constraint (x,y) is satisfied then
    - Remove y from Y
    - If Y is empty then contradiction ← *true*
    - removed ← *true*
- Label set of K ← Y
- Return removed

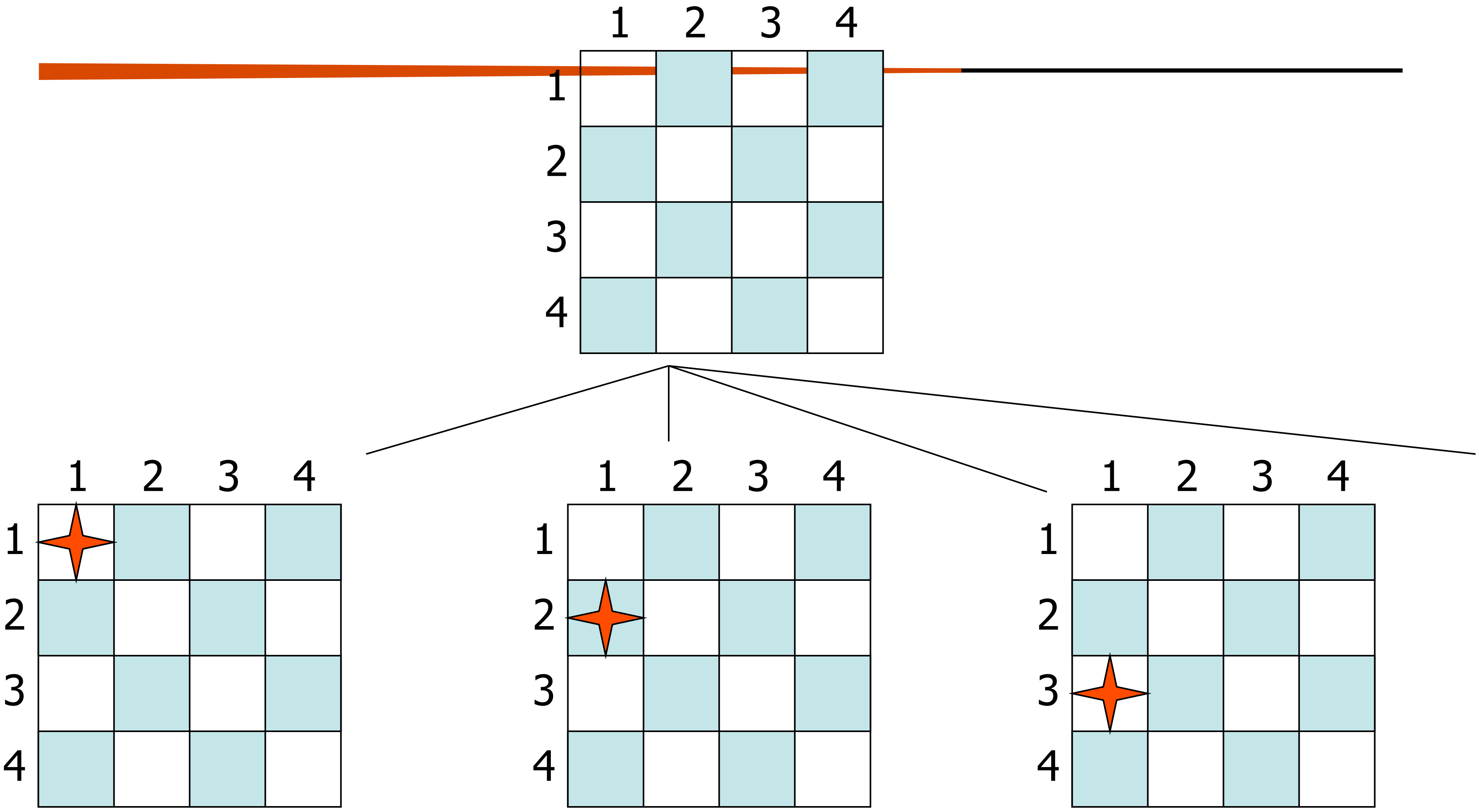
Map Coloring



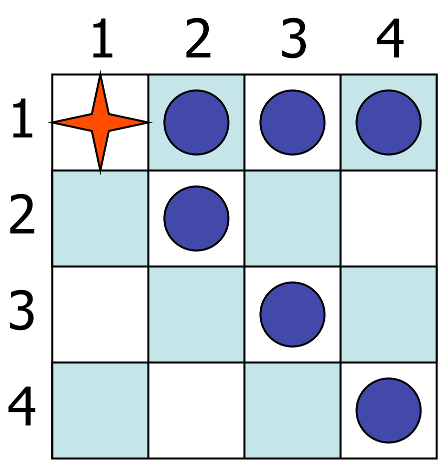
WA	NT	Q	NSW	V	SA	T
RGB	RGB	RGB	RGB	RGB	RGB	RGB
R	GB	RGB	RGB	RGB	GB	RGB
R	B	G	RB	RGB	B	RGB

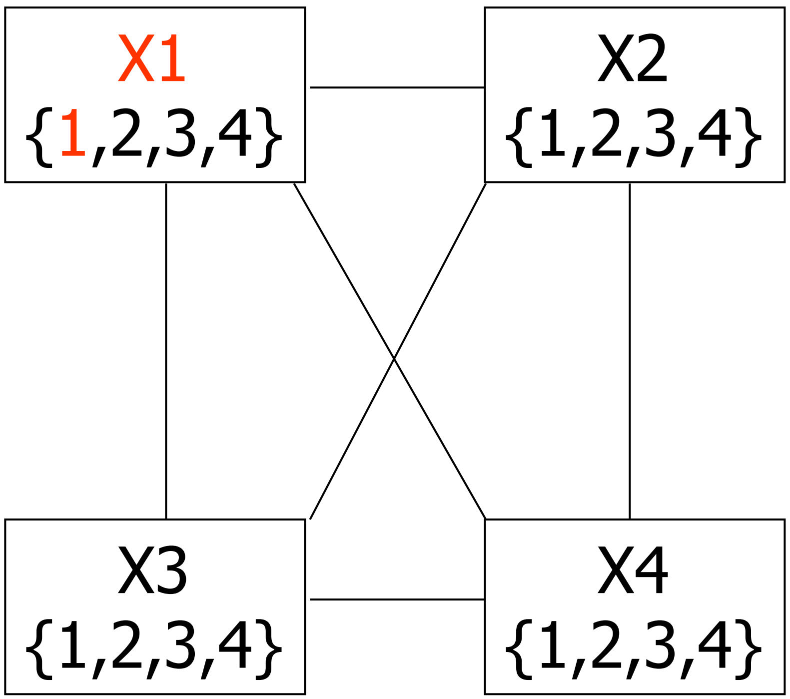
Solving a CSP

- Search:
  - can find good solutions, but must examine non-solutions along the way
- Constraint Propagation:
  - can rule out non-solutions, but this is not the same as finding solutions:
- Interweave constraint propagation and search
  - Perform constraint propagation at each search step.

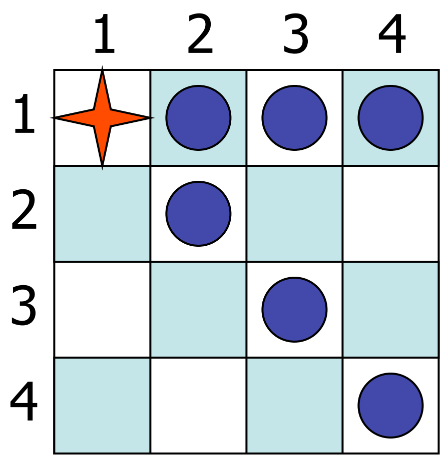

1 2 3 4
1
2
3
4
1 2 3 4
1
2
3
4
1 2 3 4
1
2
3
4
1 2 3 4
1
2
3
4

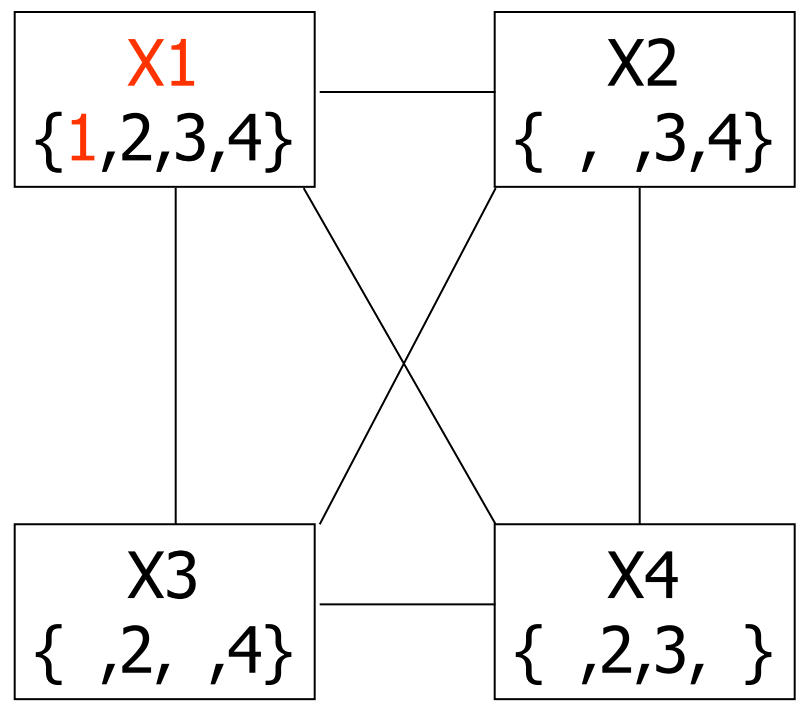
4-Queens Problem



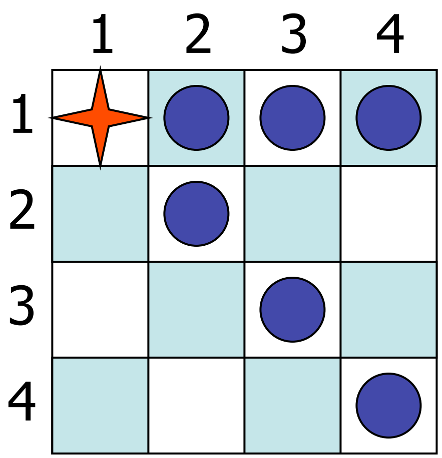


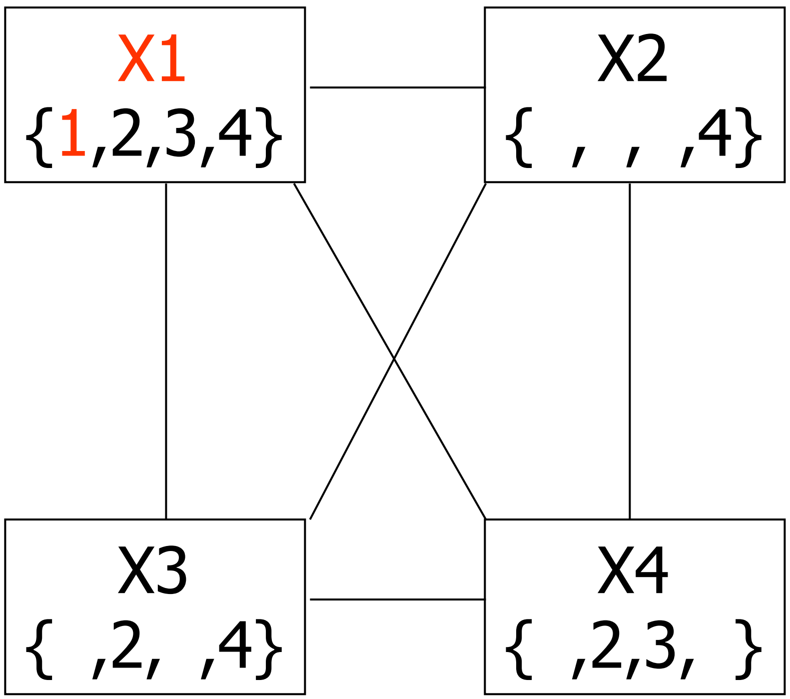
4-Queens Problem



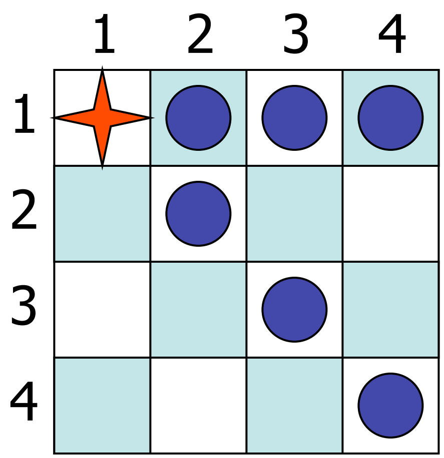


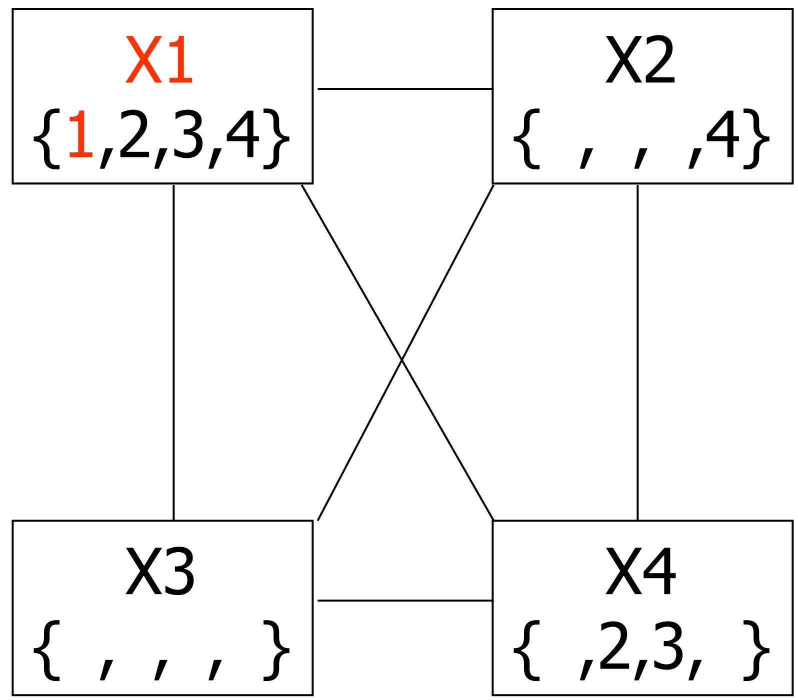
4-Queens Problem

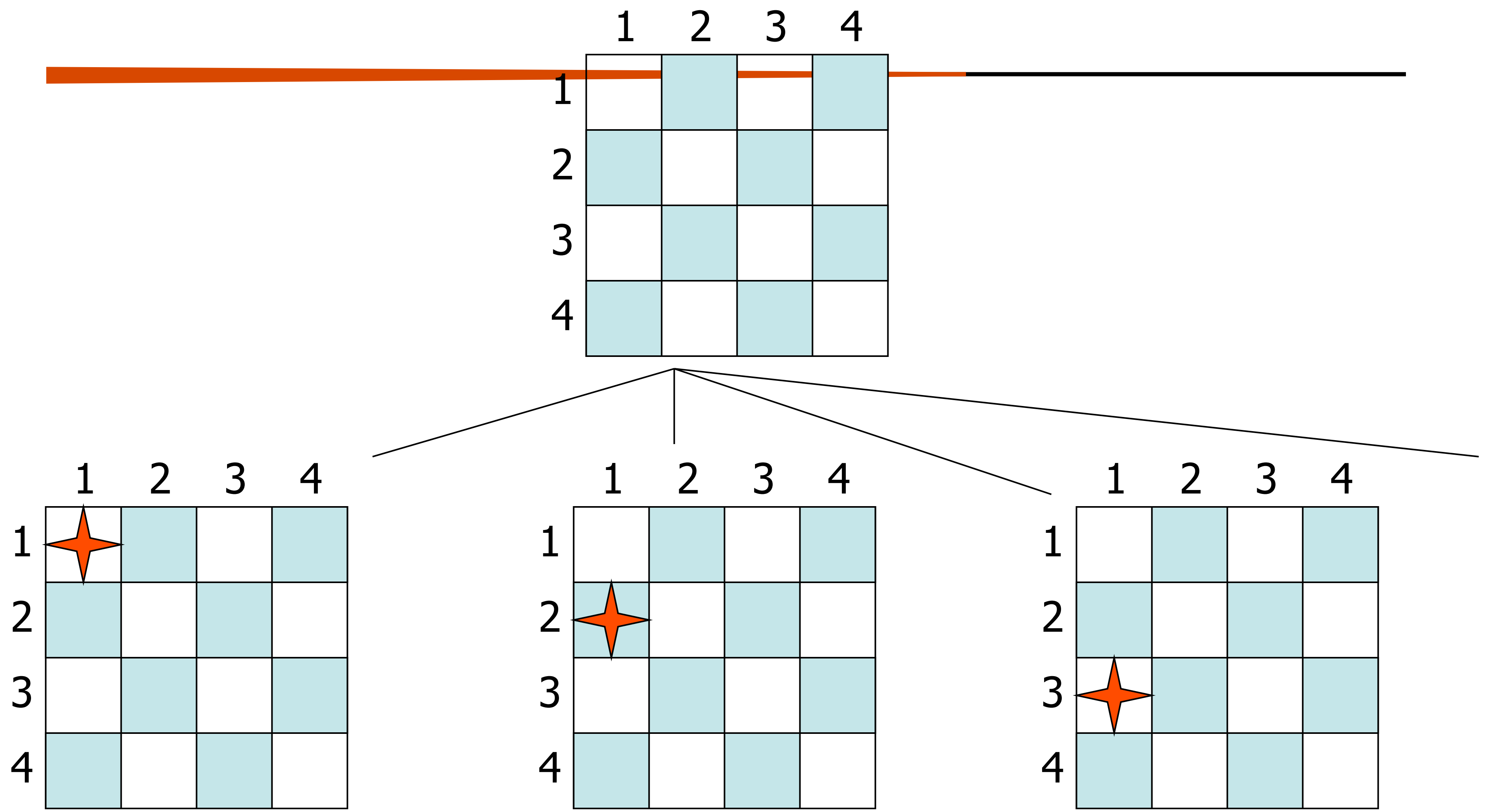




4-Queens Problem

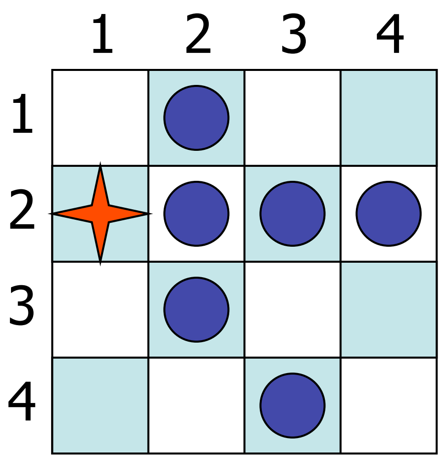


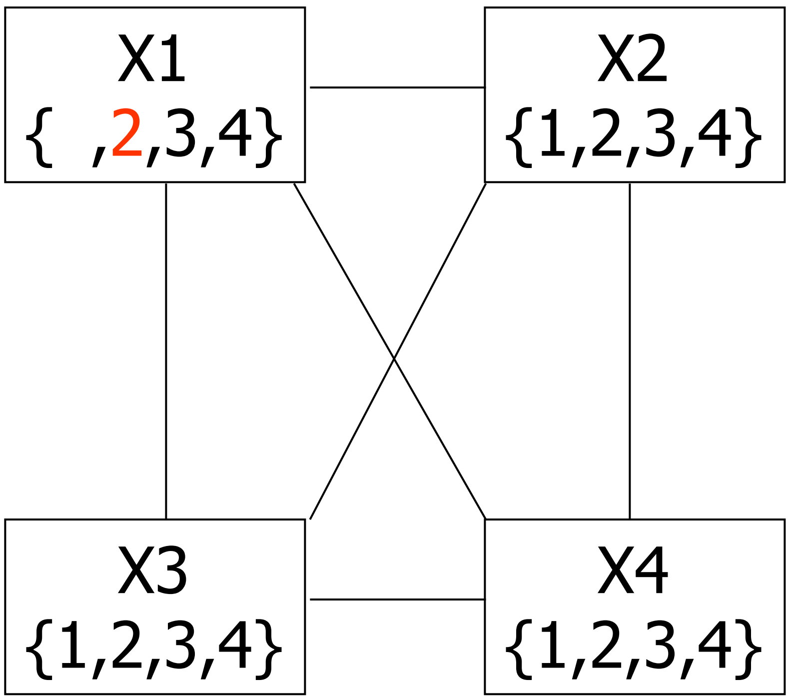




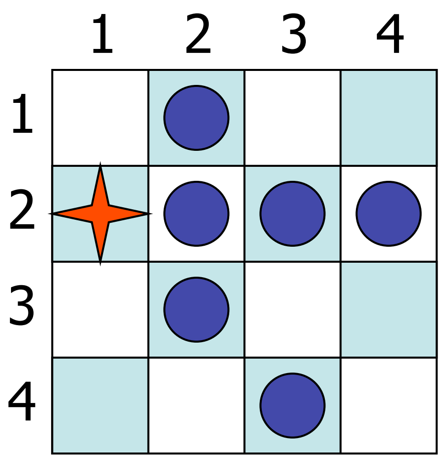
1 2 3 4
1
2
3
4
1 2 3 4
1
2
3
4
1 2 3 4
1
2
3
4
1 2 3 4
1
2
3
4

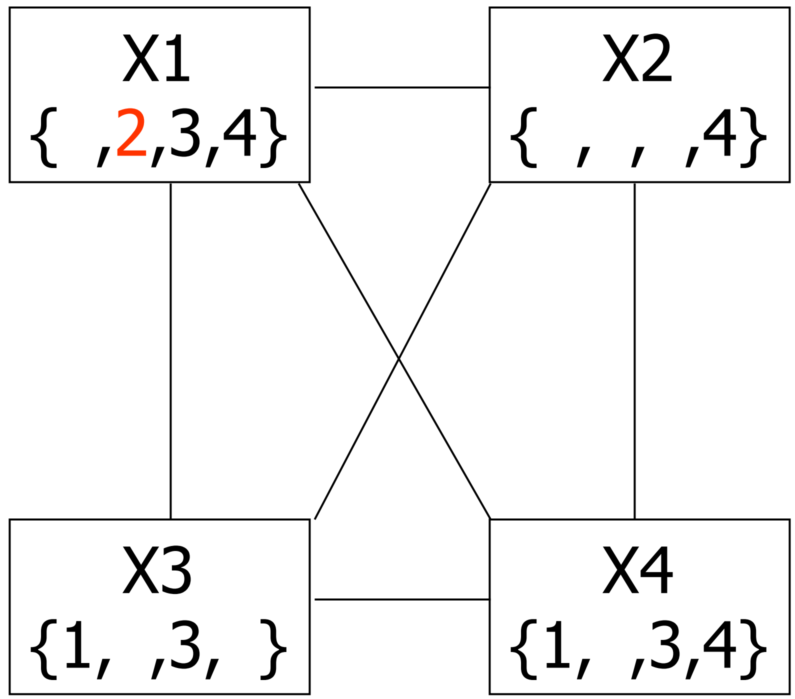
4-Queens Problem



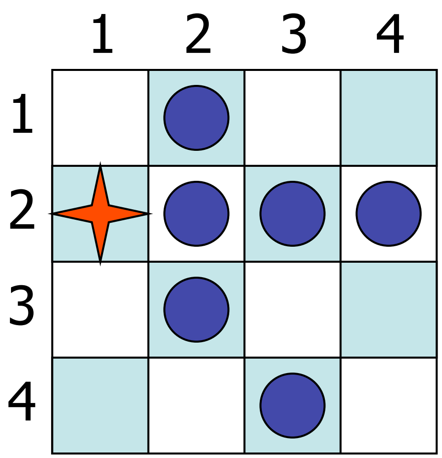


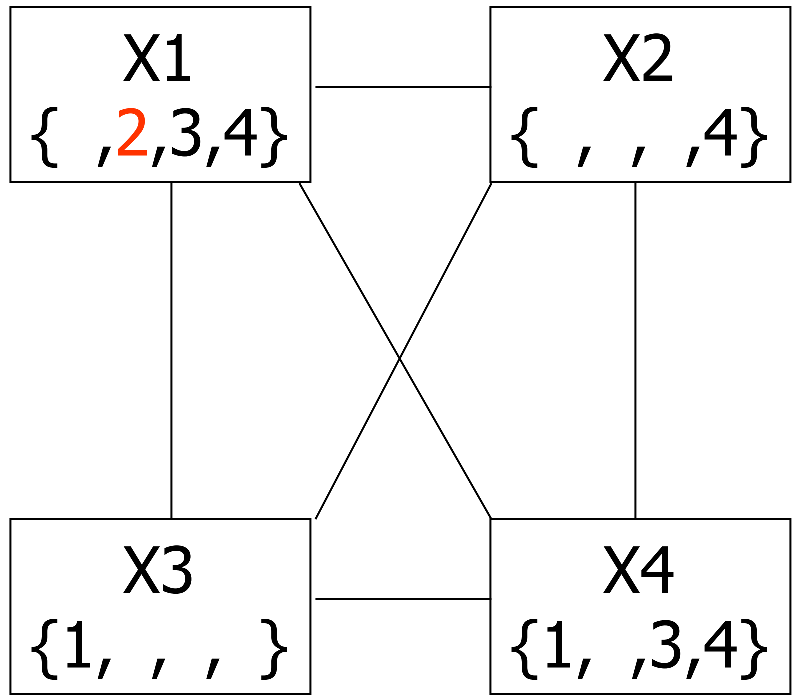
4-Queens Problem



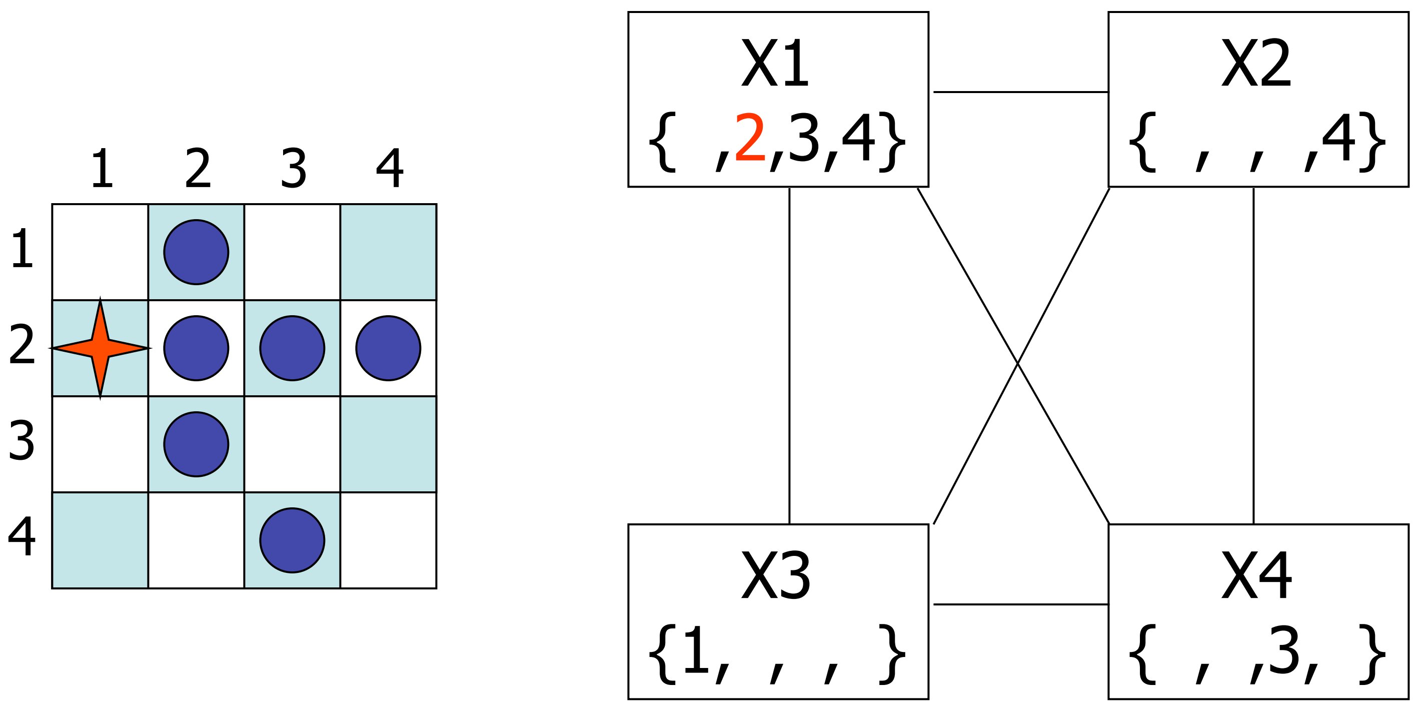


4-Queens Problem

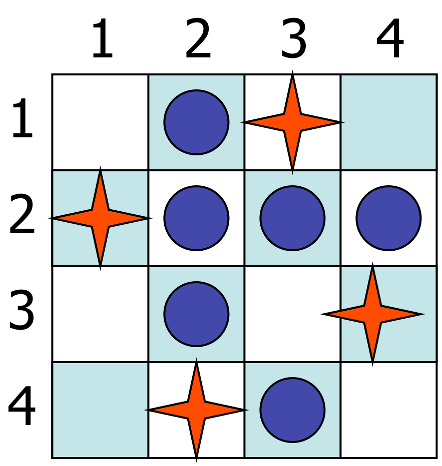


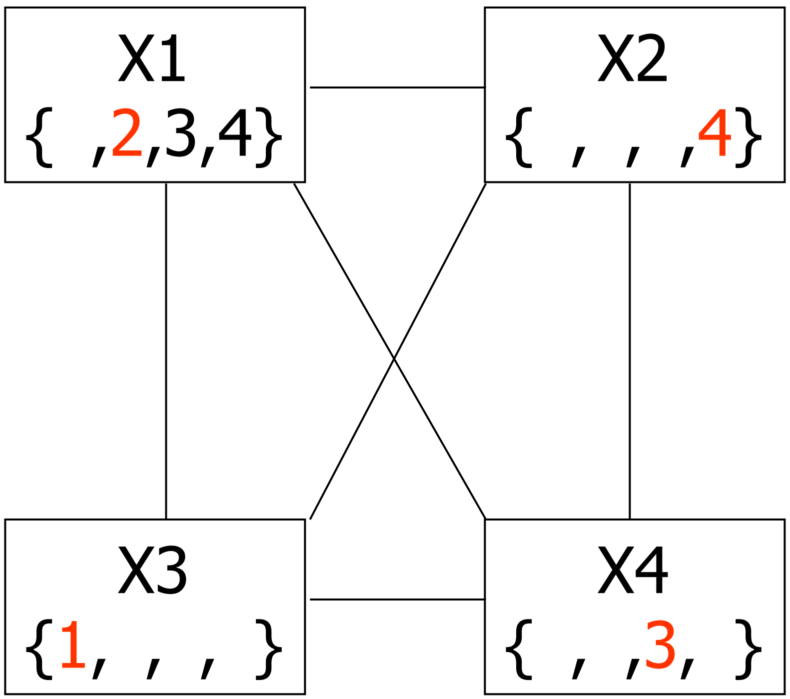


4-Queens Problem



4-Queens Problem





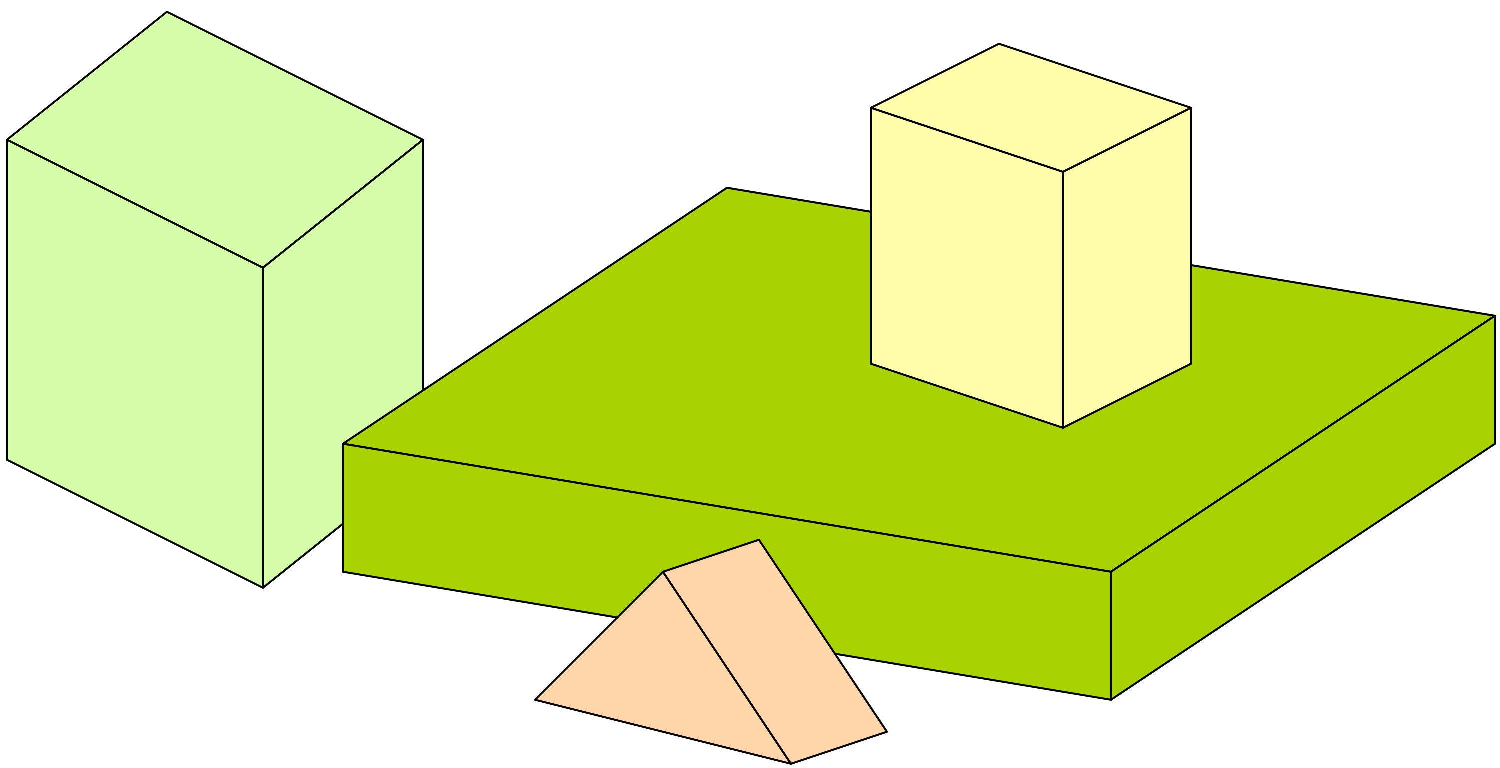
Summary

- Constraint Satisfaction Problems (CSP)
- CSP as a search problem
  - Backtracking algorithm
  - General heuristics
- Forward checking
- Removing Arch Inconsistencies
- Interweaving CP and backtracking

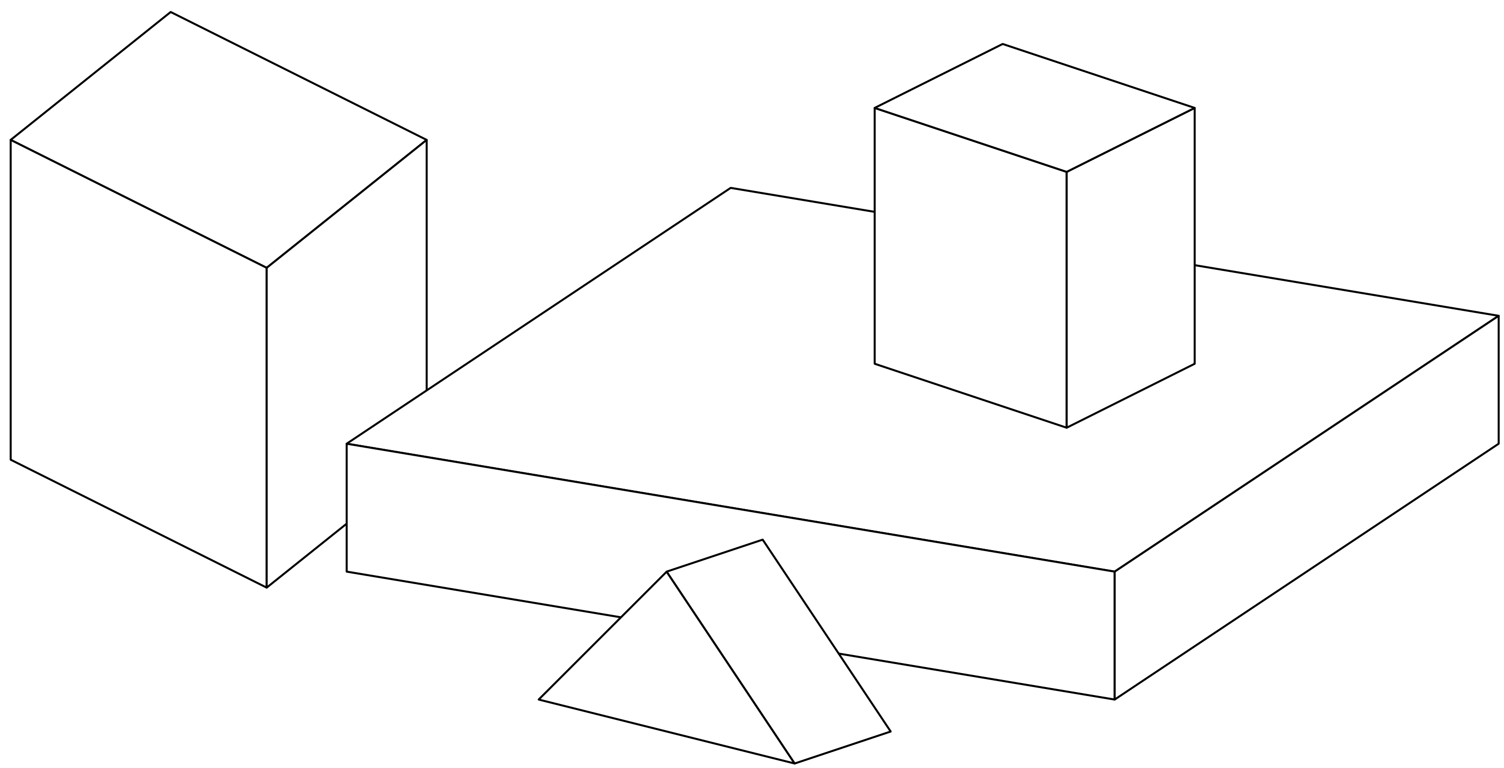
Edge Labeling in Computer Vision

Russell and Norvig: Chapter 24

Edge Labeling



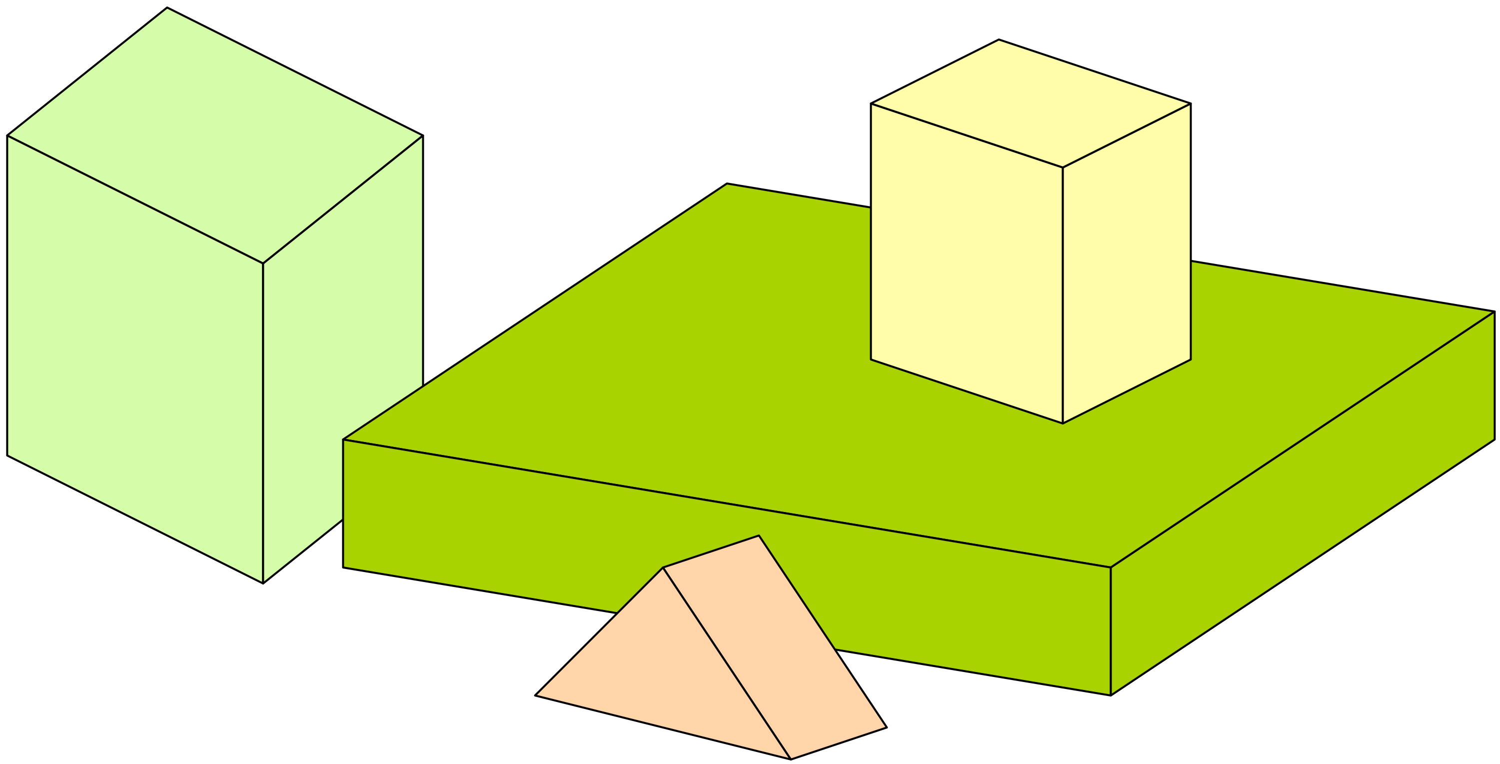
Edge Labeling



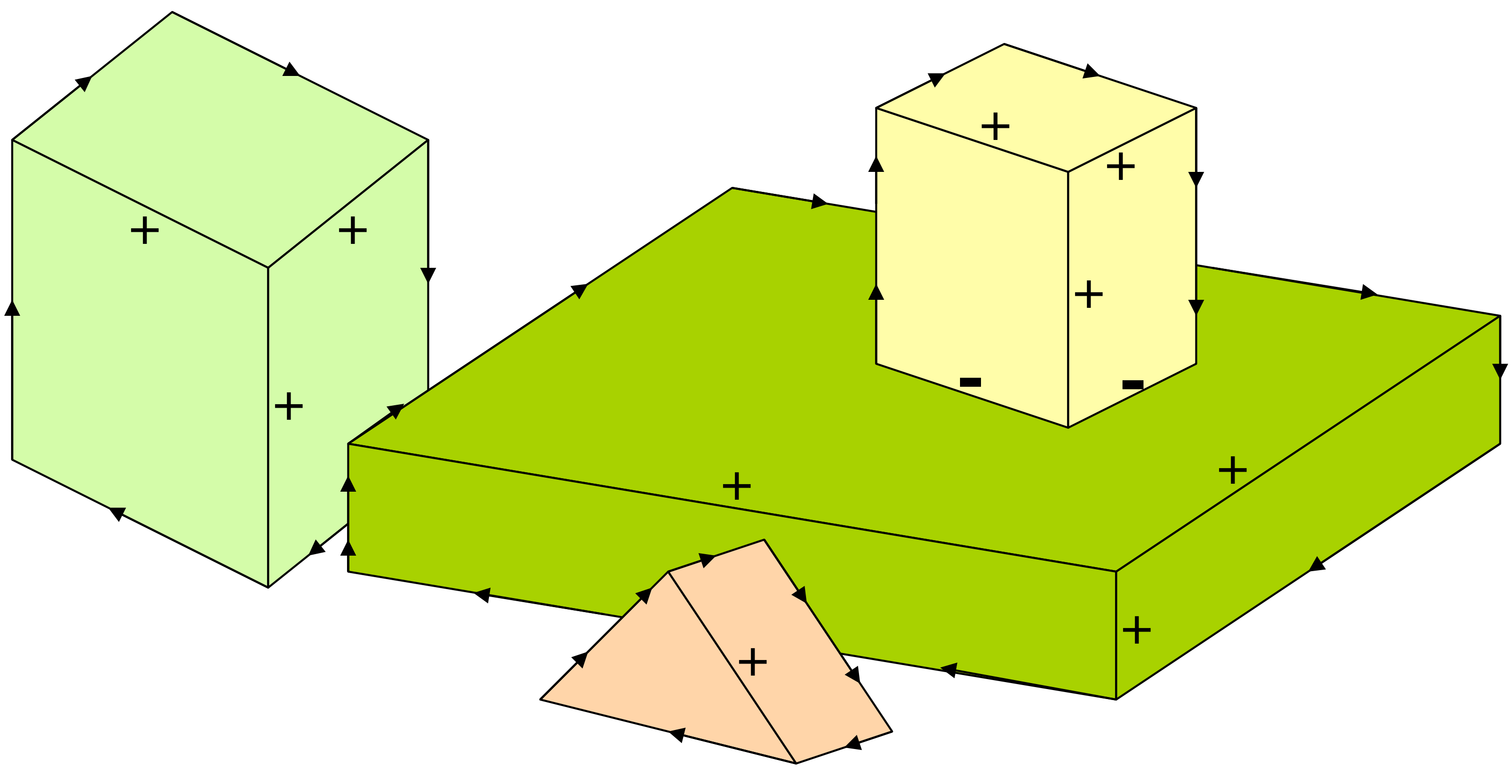
Labels of Edges

- 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 and it is on the right side of ←

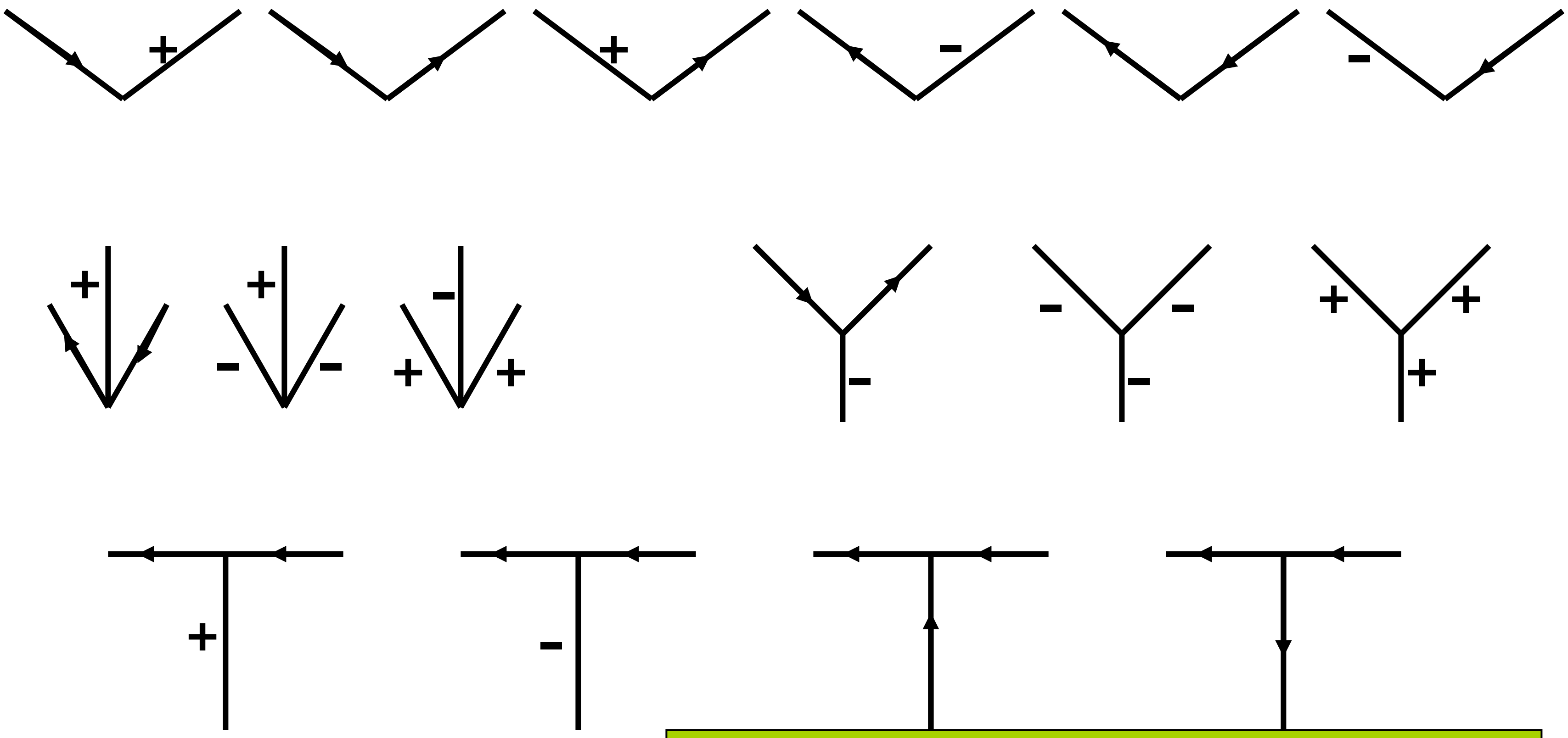
Edge Labeling



Edge Labeling



Junction Label Sets

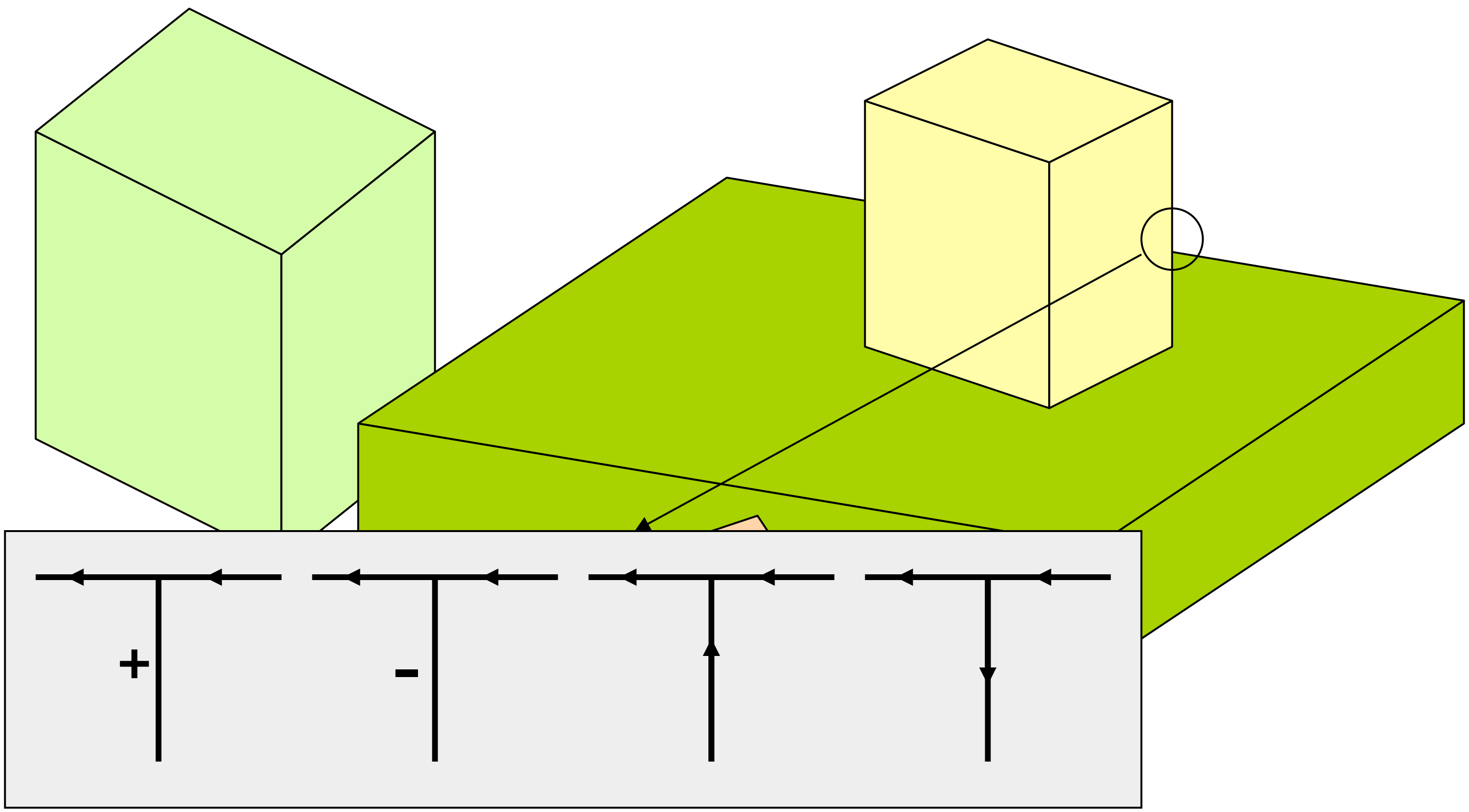


(Waltz, 1975; Mackworth, 1977)

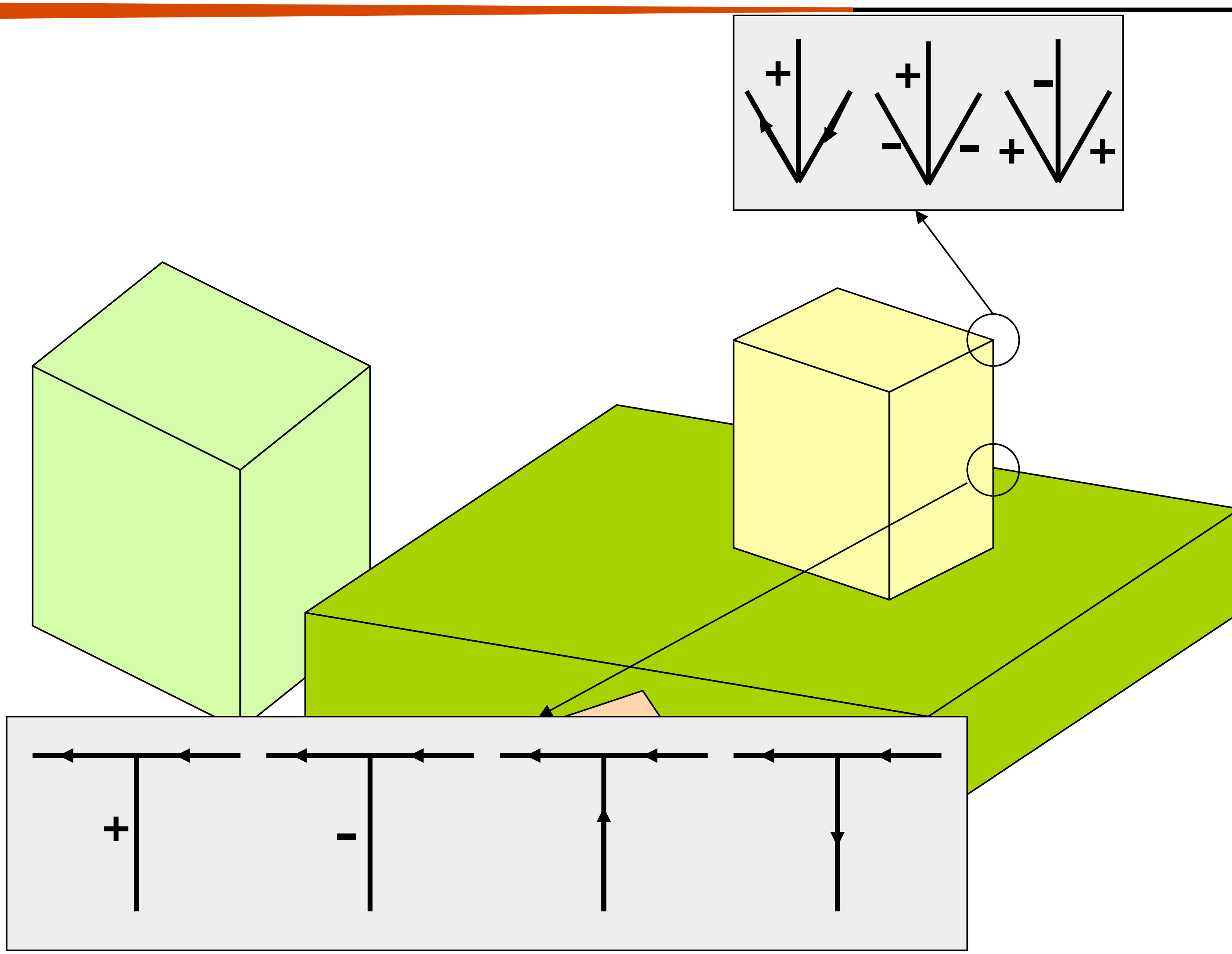
Edge Labeling as a CSP

- 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

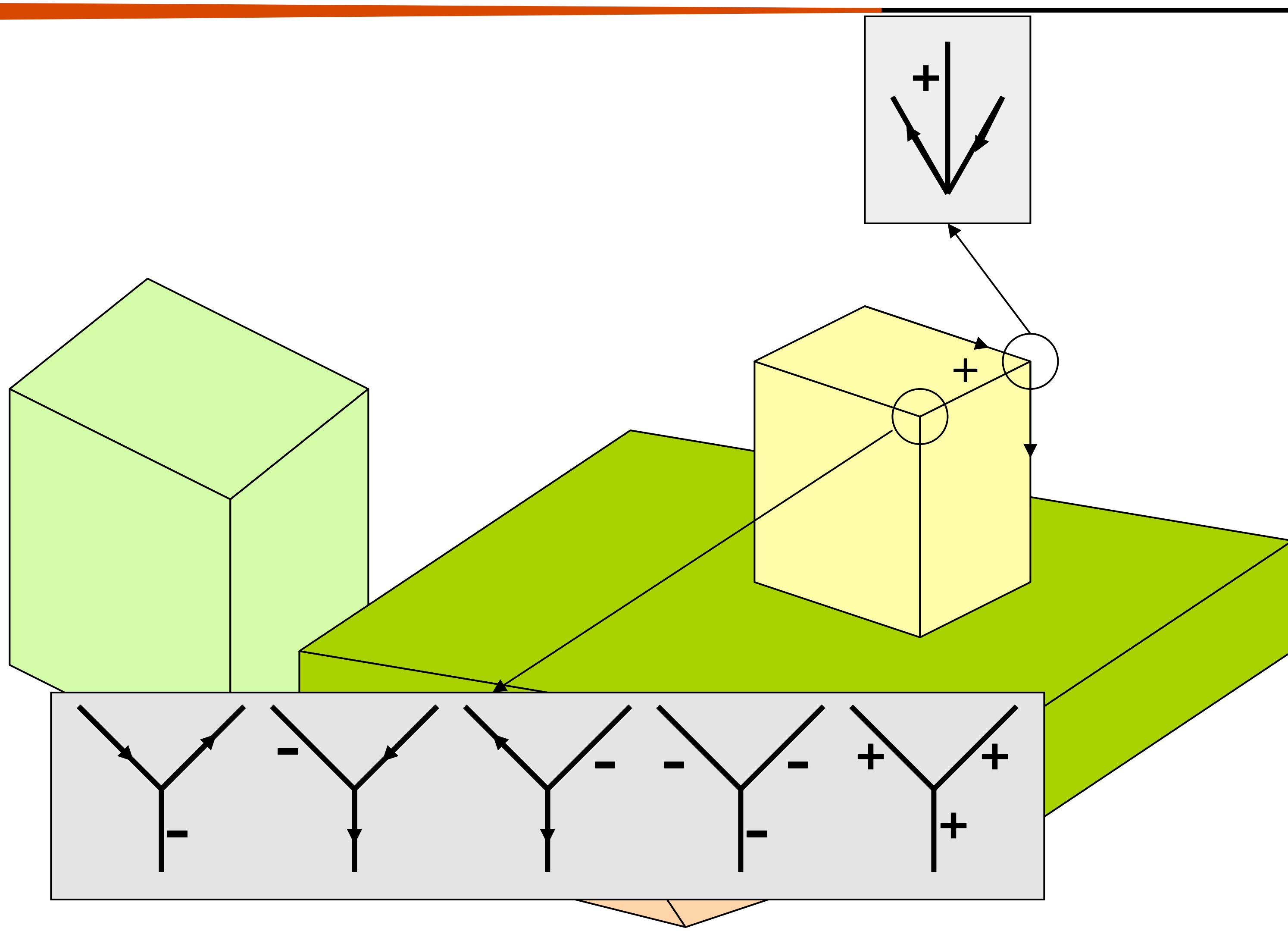
Edge Labeling



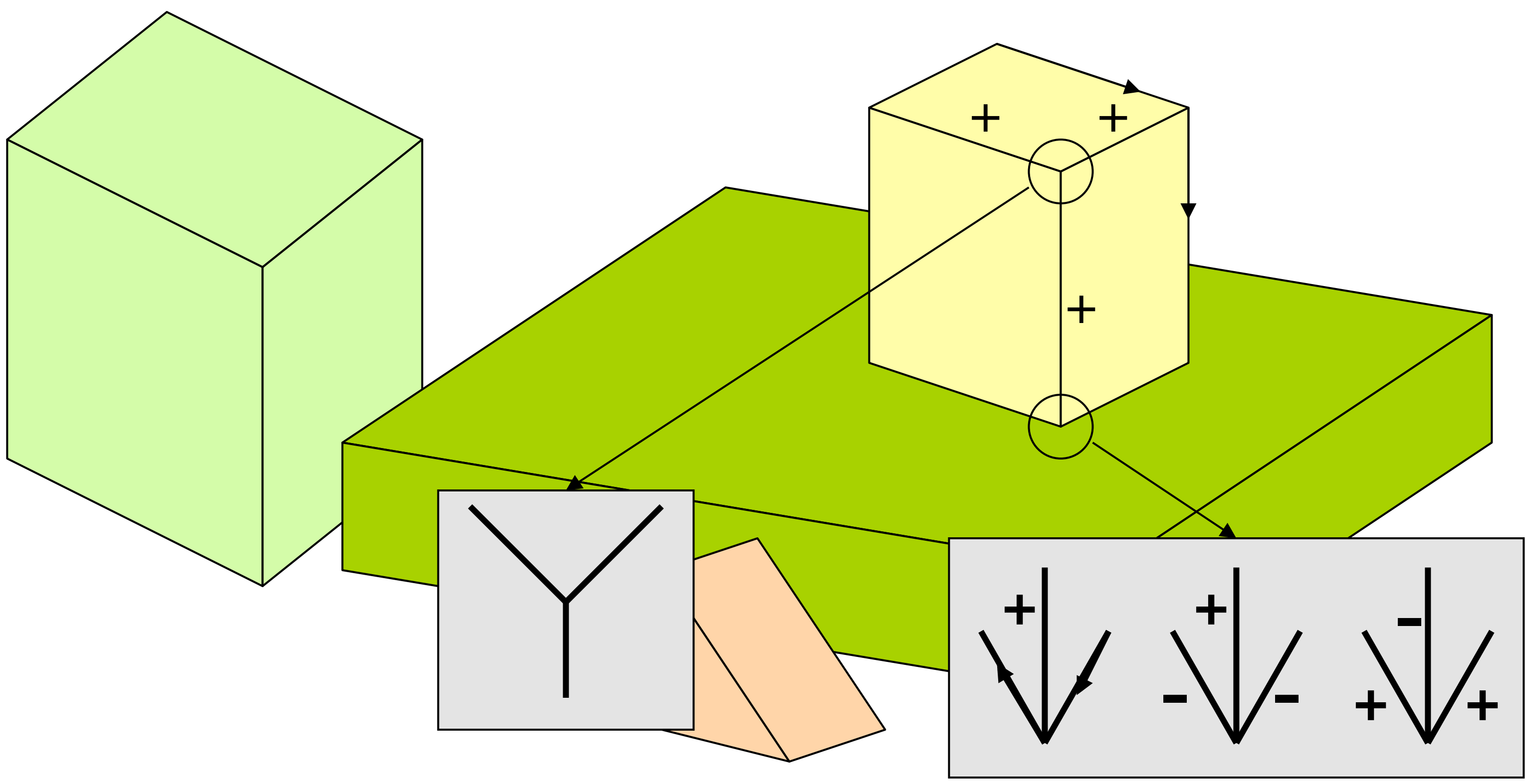
Edge Labeling



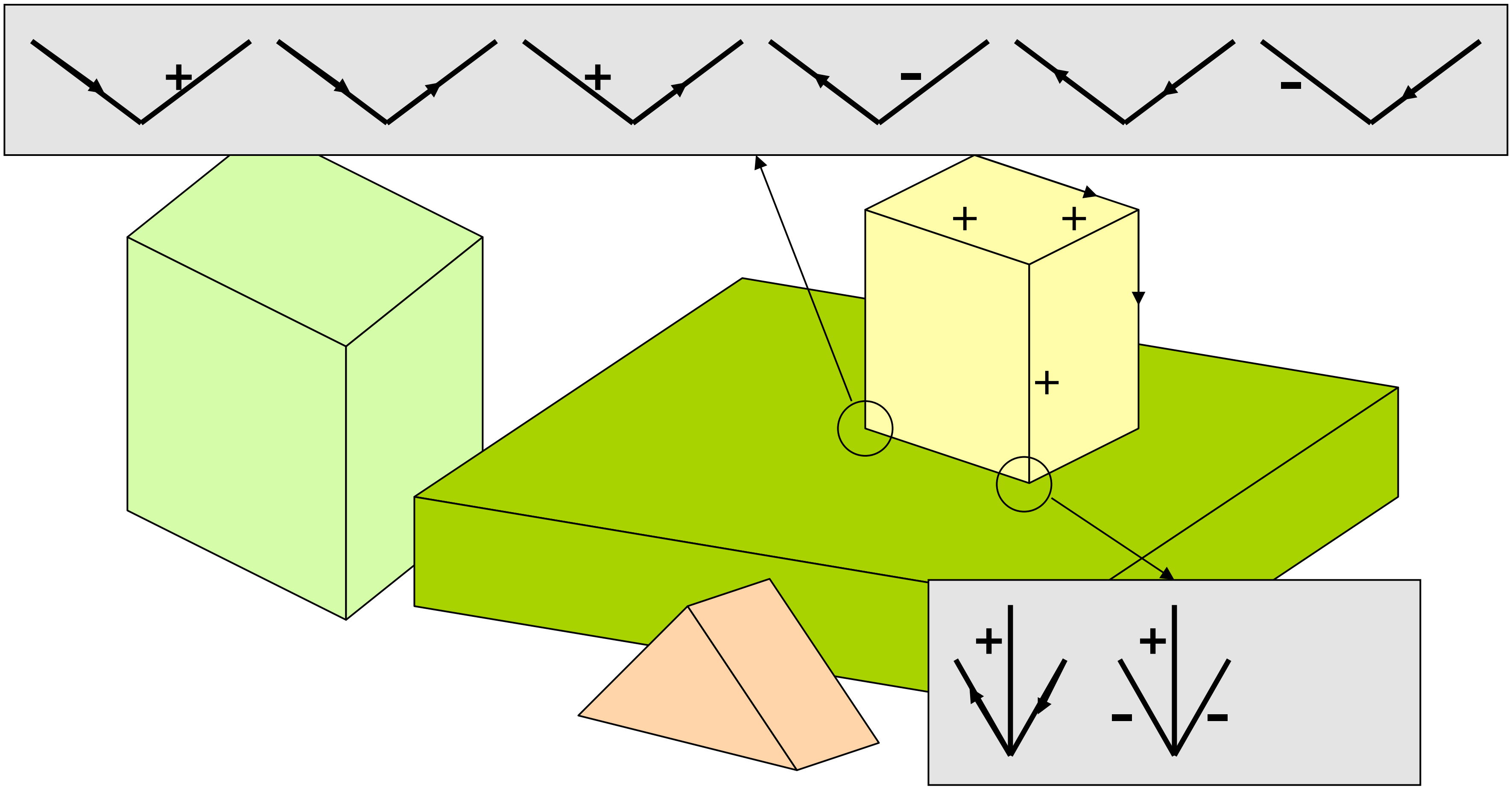
Edge Labeling



Edge Labeling



Edge Labeling



Removal of Arc Inconsistencies

REMOVE-ARC-INCONSISTENCIES(J,K)

- removed ← *false*
- X ← label set of J
- Y ← label set of K
- For every label y in Y do
  - If there exists no label x in X such that the constraint (x,y) is satisfied then
    - Remove y from Y
    - If Y is empty then contradiction ← *true*
    - removed ← *true*
- Label set of K ← Y
- Return removed

CP Algorithm for Edge Labeling

- Associate with every junction its label set
- Q ← stack of all junctions
- while Q is not empty do
  - J ← UNSTACK(Q)
  - For every junction K adjacent to J do
    - If REMOVE-ARC-INCONSISTENCIES(J,K) then
      - If K's domain is non-empty then STACK(K,Q)
      - Else return false

(Waltz, 1975; Mackworth, 1977)