Artificial Intelligence
ENCS 434

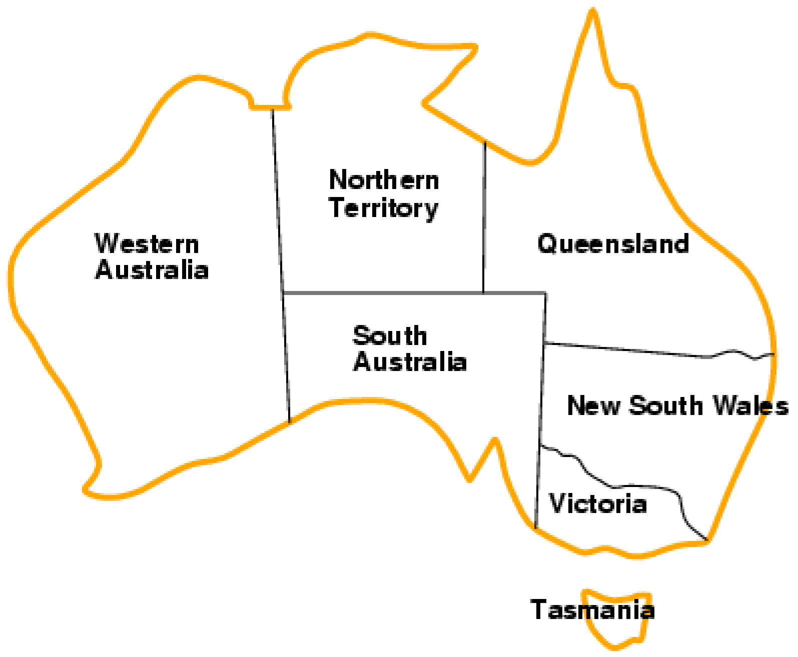
Constraint Satisfaction Problems

Aziz M. Qaroush - Birzeit University

Constraint Satisfaction

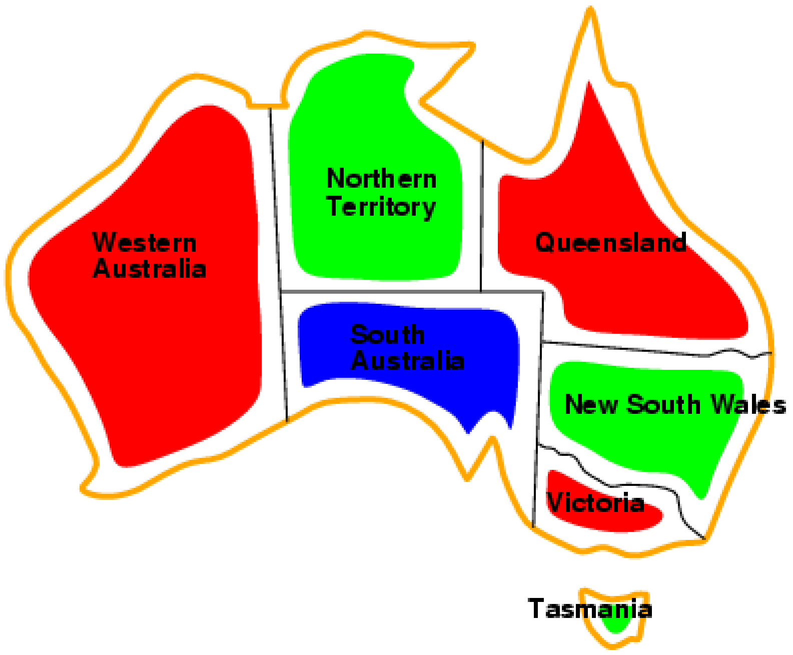
- satisfies additional structural properties of the problem
  - may depend on the representation of the problem
- the problem is defined through a set of variables and a set of domains
  - variables can have possible values specified by the problem
  - constraints describe allowable combinations of values for a subset of the variables
- *state* in a CSP
  - defined by an assignment of values to some or all variables
- *solution* to a CSP
  - must assign values to all variables
  - must satisfy all constraints
  - solutions may be ranked according to an objective function

Example: Map-Coloring



- Variables *WA, NT, Q, NSW, V, SA, T*
- Domains $D_i = \{\text{red, green, blue}\}$
- Constraints: adjacent regions must have different colors
  - e.g., WA ≠ NT

Example: Map-Coloring



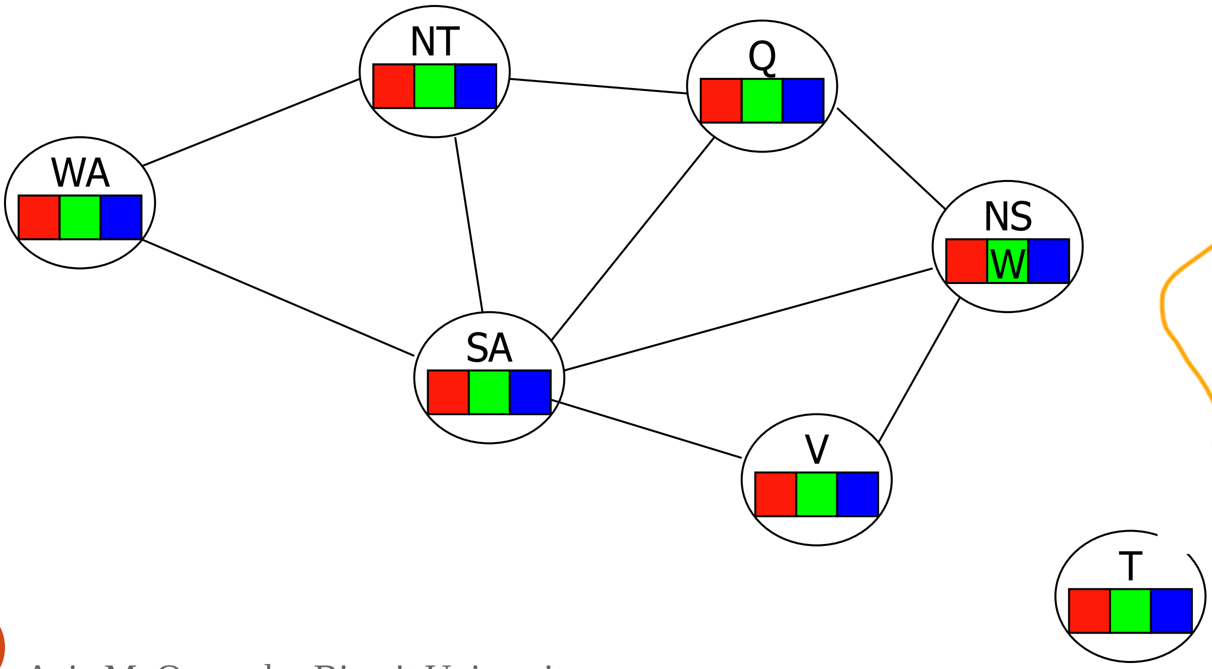
- Solutions are complete and consistent assignments, e.g.,

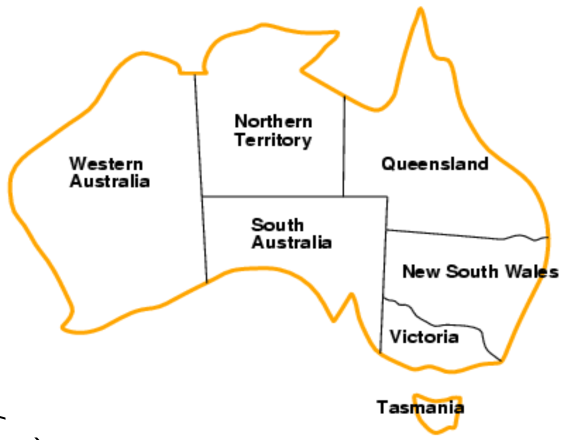
  WA = red, NT = green, Q = red, NSW = green, V = red, SA = **blue**, T = green

- A state may be incomplete e.g., just WA=red

Constraint graph

- It is helpful to visualize a CSP as a **constraint graph**
  - **Binary CSP:** each constraint relates two variables
  - **Constraint graph:** nodes are variables, arcs are constraints





Varieties of CSPs

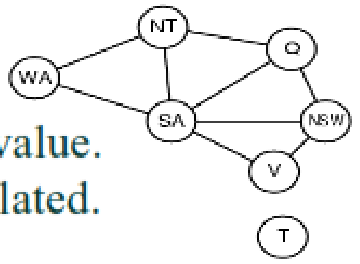
- Discrete variables
  - finite domains:
    - n variables, domain size d □ O(dn) complete assignments
    - e.g., Boolean CSPs, incl.~Boolean satisfiability (NP-complete)
  - infinite domains:
    - integers, strings, etc.
    - e.g., job scheduling, variables are start/end days for each job
    - need a constraint language, e.g., StartJob1 + 5 ≤ StartJob3
- Continuous variables
  - e.g., start/end times for Hubble Space Telescope observations
  - linear constraints solvable in polynomial time by linear programming

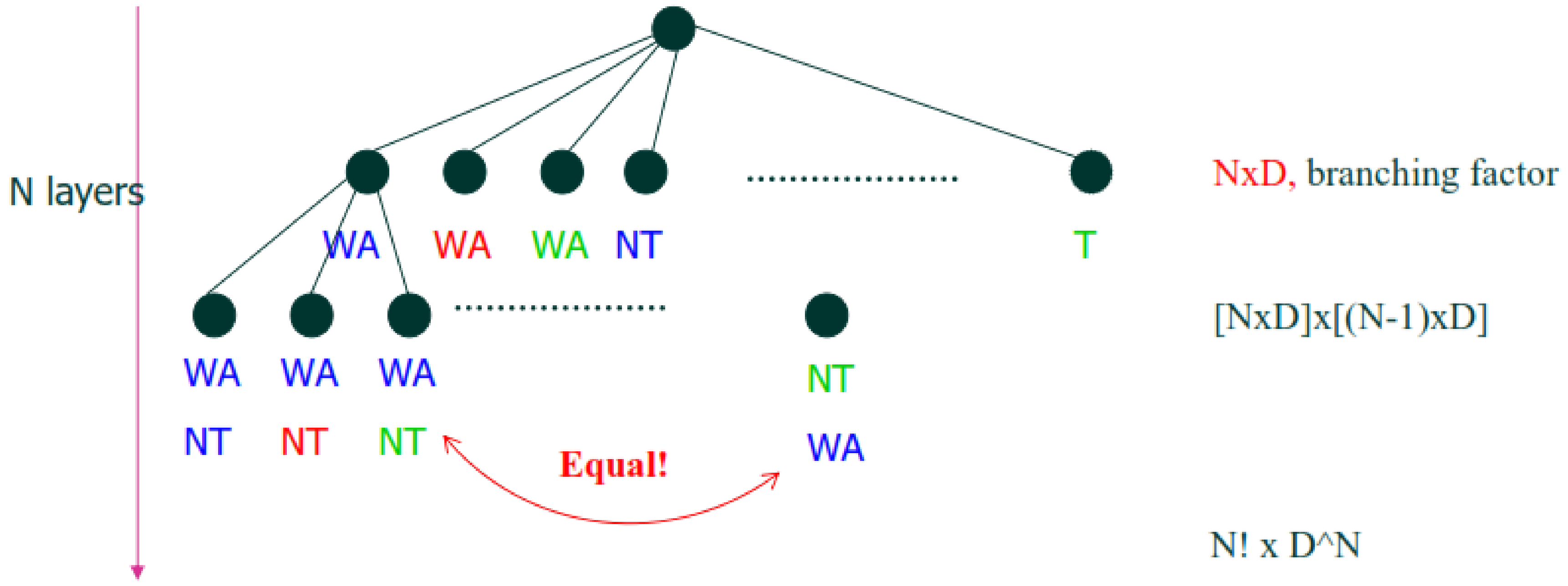
CSP as Incremental Search Problem

- initial state
  - all (or at least some) variables unassigned
- successor function
  - assign a value to an unassigned variable
  - must not conflict with previously assigned variables
- goal test
  - all variables have values assigned
  - no conflicts possible
    - not allowed in the successor function
- path cost
  - e.g. a constant for each step
  - may be problem-specific

Constraint graph Formulation

- Node: variable
- Arc: constraint
- Initial state: none of the variables has a value (color)
- Successor state: assign a value to one of the variables without a value.
- Goal: all variables have a value and none of the constraints is violated.





There are N! x D^N nodes in the tree but only D^N distinct states

CSPs and Search

- in principle, any search algorithm can be used to solve a CSP
  - awful branching factor
    - $n*d$ for n variables with d values at the top level, $(n-1)*d$ at the next level, etc.
  - not very efficient, since they neglect some CSP properties
    - commutativity: the order in which values are assigned to variables is irrelevant, since the outcome is the same

Backtracking Search for CSPs

- a variation of depth-first search that is often used for CSPs
    - values are chosen for one variable at a time
    - if no legal values are left, the algorithm backs up and changes a previous assignment
    - very easy to implement
        - initial state, successor function, goal test are standardized
    - not very efficient
        - can be improved by trying to select more suitable unassigned variables first

Improving backtracking efficiency

- General-purpose methods can give huge gains in speed:
    - Which variable should be assigned next?
    - In what order should its values be tried?
    - Can we detect inevitable failure early?

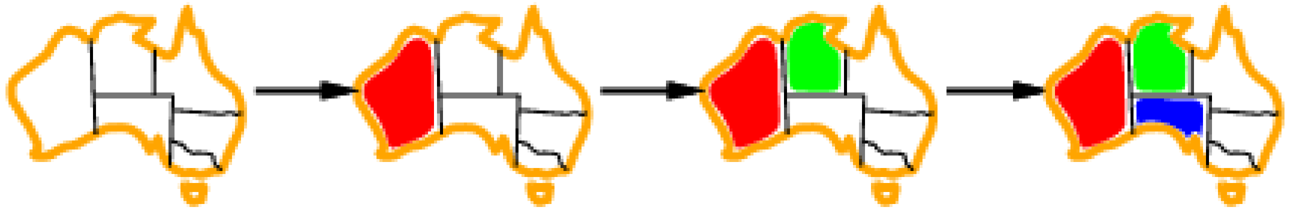
Heuristics for CSP

- most-constrained variable (minimum remaining values, “fail-first”)
  - variable with the fewest possible values is selected
  - tends to minimize the branching factor
- most-constraining variable
  - variable with the largest number of constraints on other unassigned variables
- least-constraining value
  - for a selected variable, choose the value that leaves more freedom for future choices

Most constrained variable

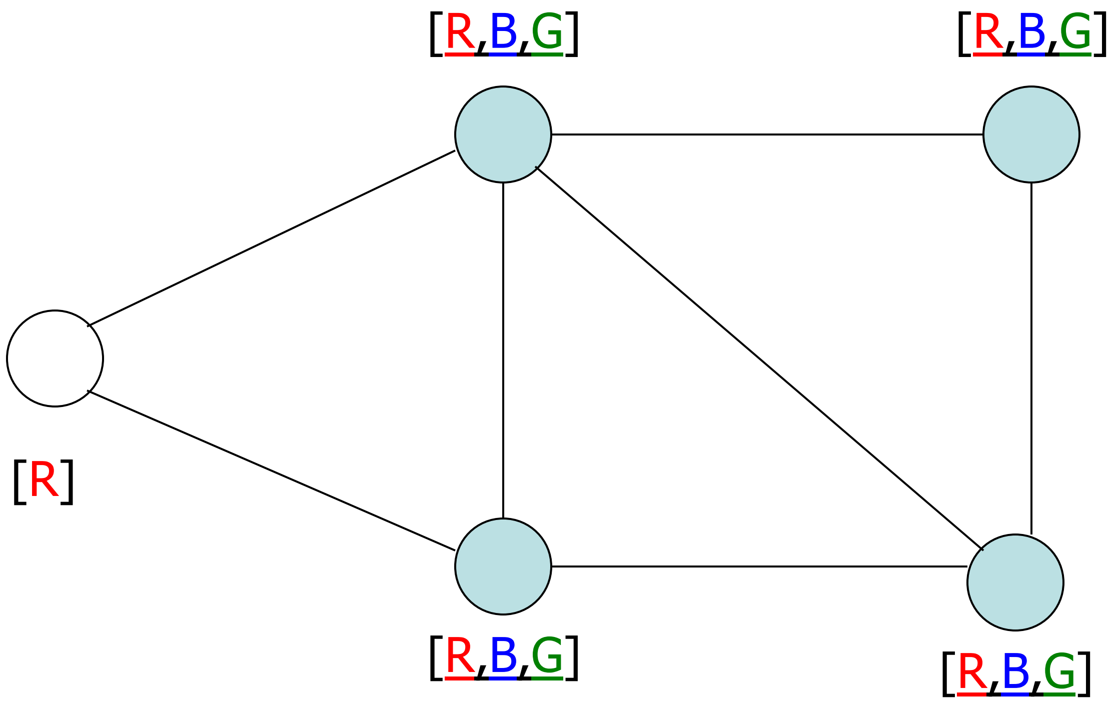
Minimum Remaining Values (MRV)

- Most constrained variable:
  choose the variable with the fewest legal values

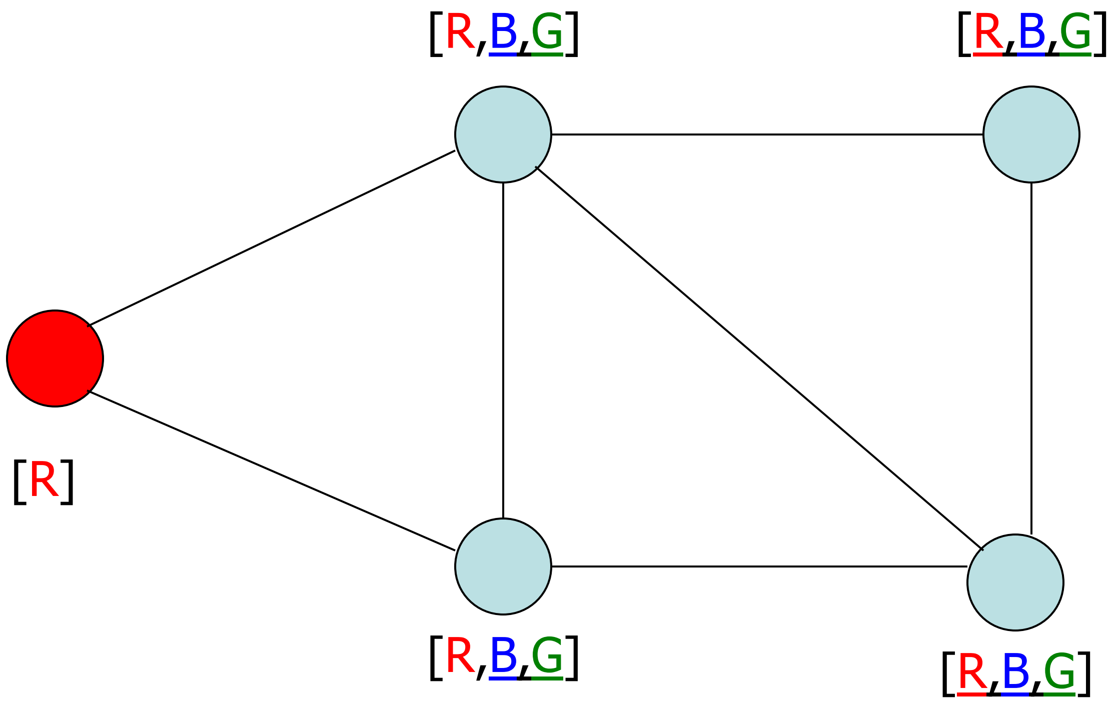


- Called minimum remaining values (MRV) heuristic
- “fail-first” heuristic: Picks a variable which will cause failure as soon as possible, allowing the tree to be pruned.

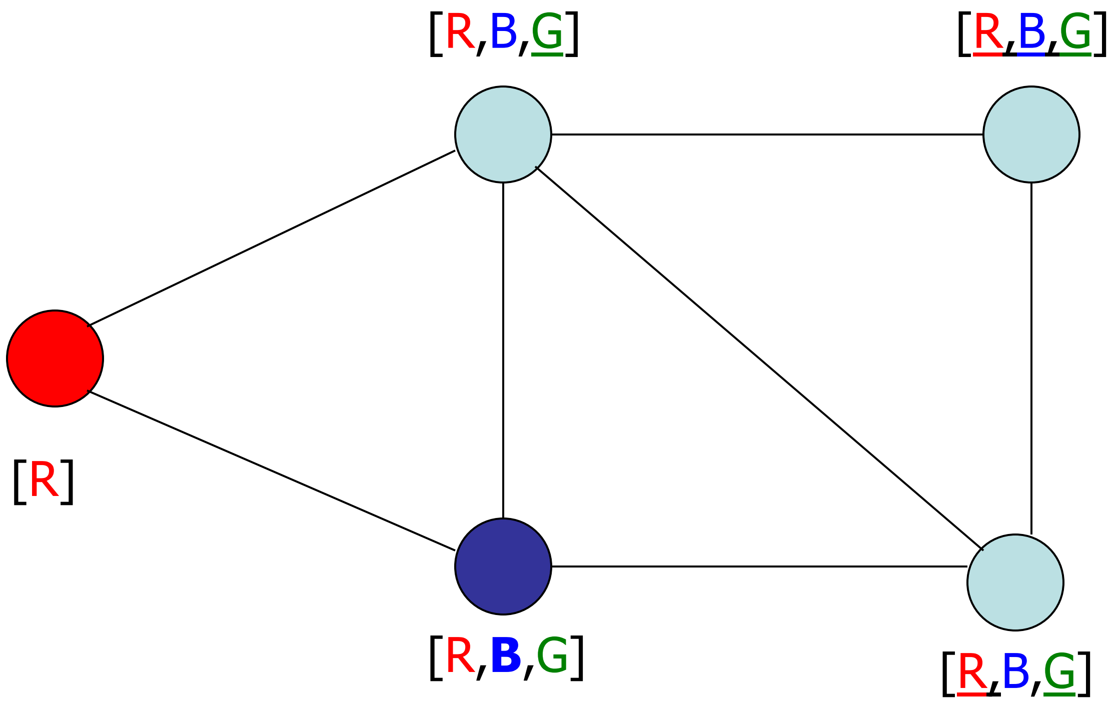
Backpropagation - MRV



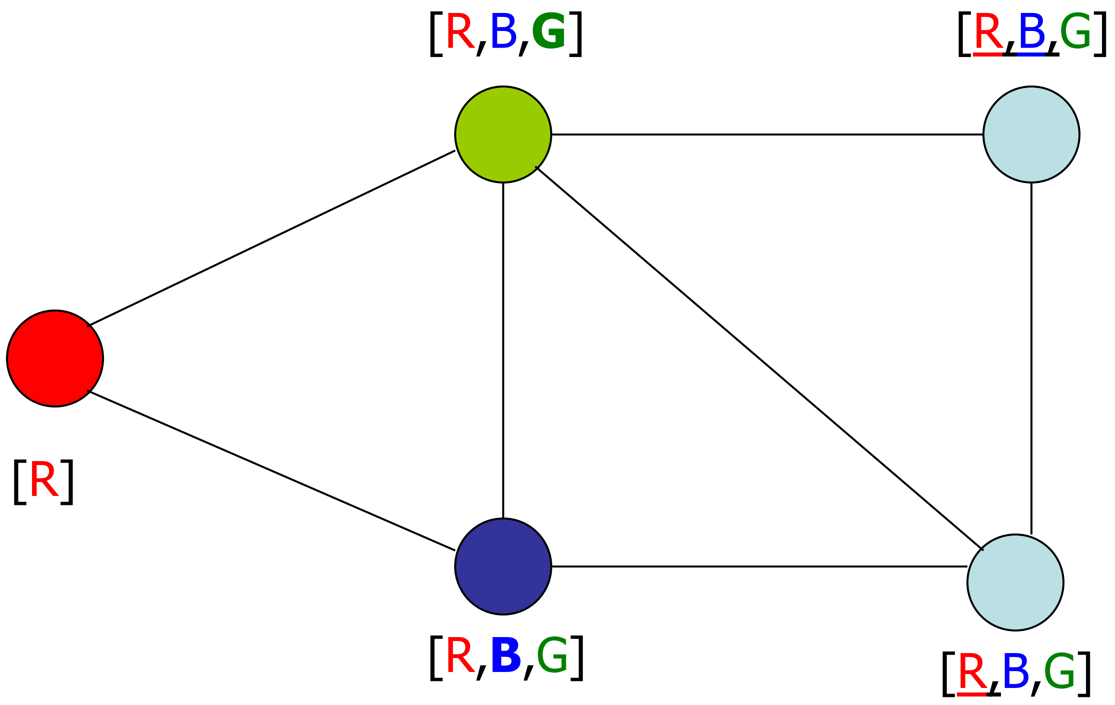
Backpropagation - MRV



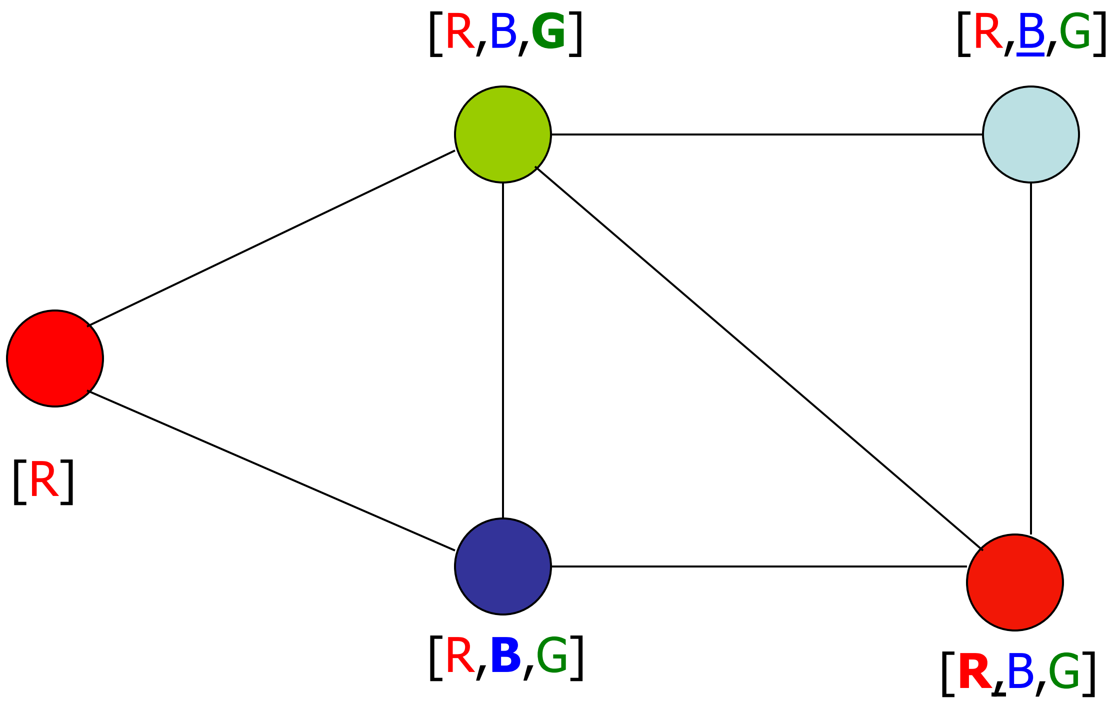
Backpropagation - MRV



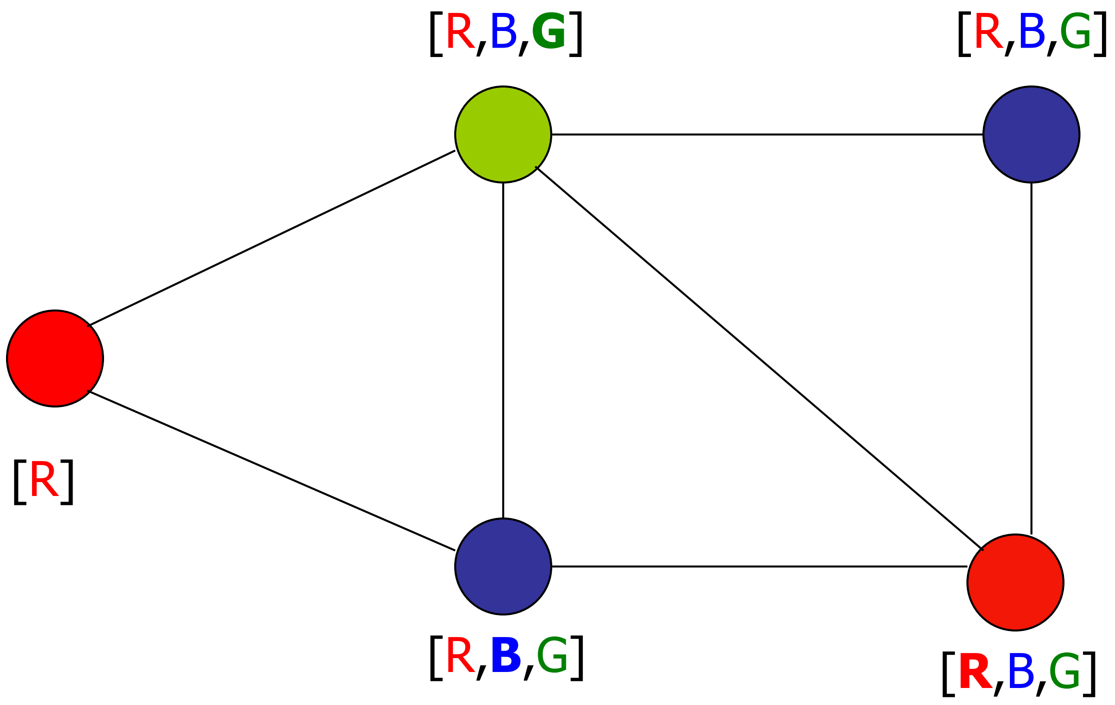
Backpropagation - MRV



Backpropagation - MRV



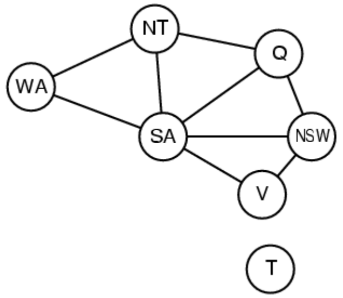
Backpropagation - MRV

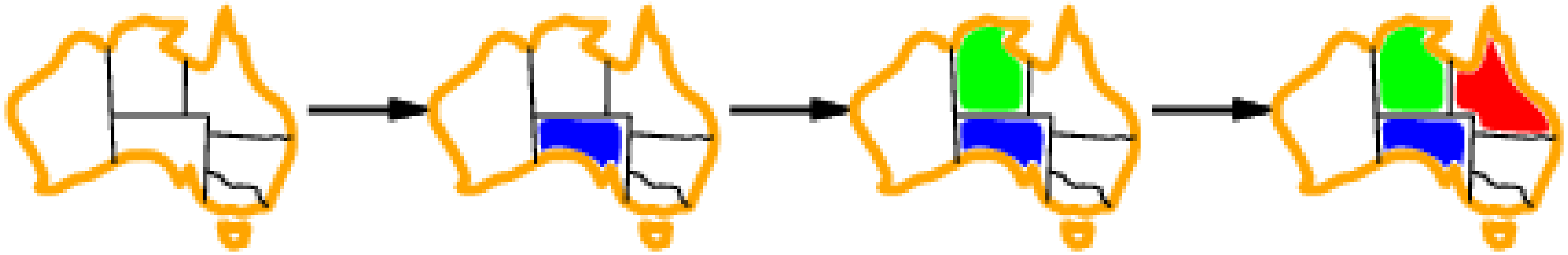


Solution !!!

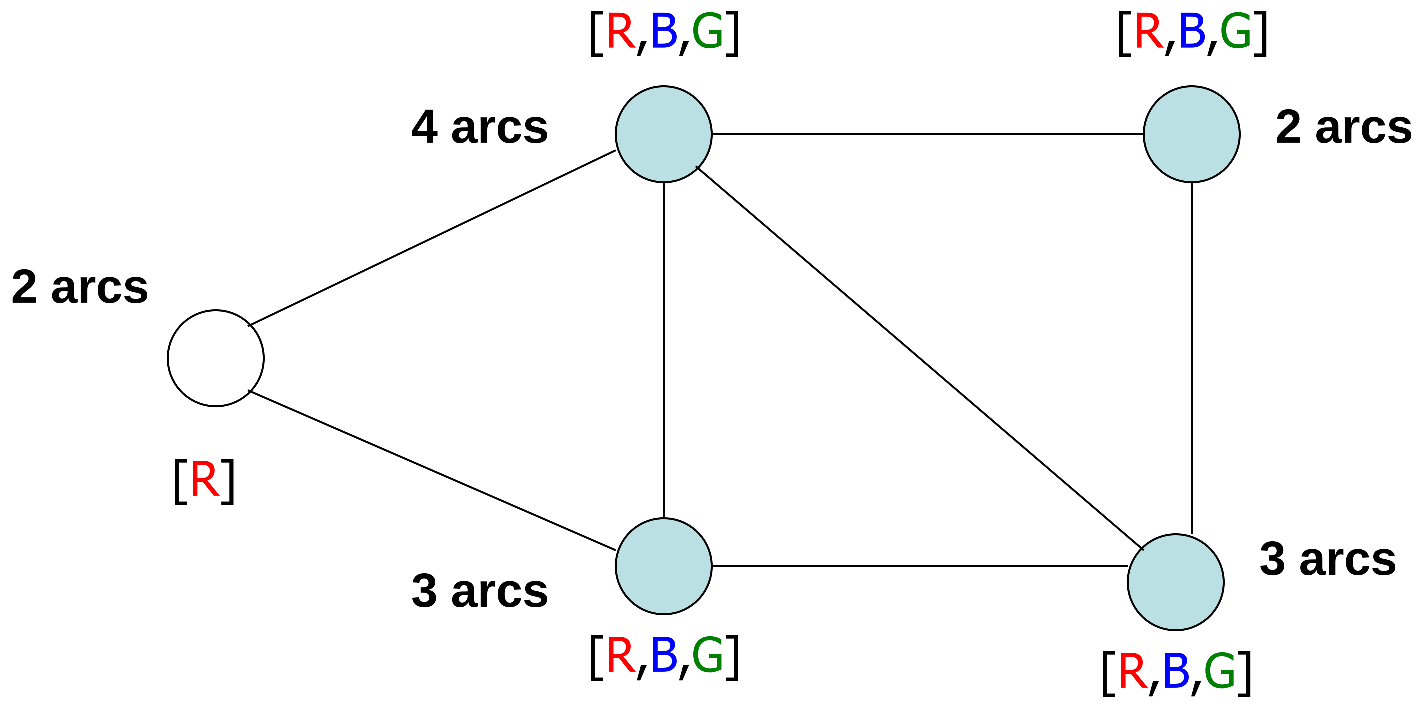
Most constraining variable - MCV

- Tie-breaker among most constrained variables
- Most constraining variable:
  - choose the variable **with the most constraints on remaining variables** (select variable that is involved in the largest number of constraints - edges in graph on other unassigned variables)

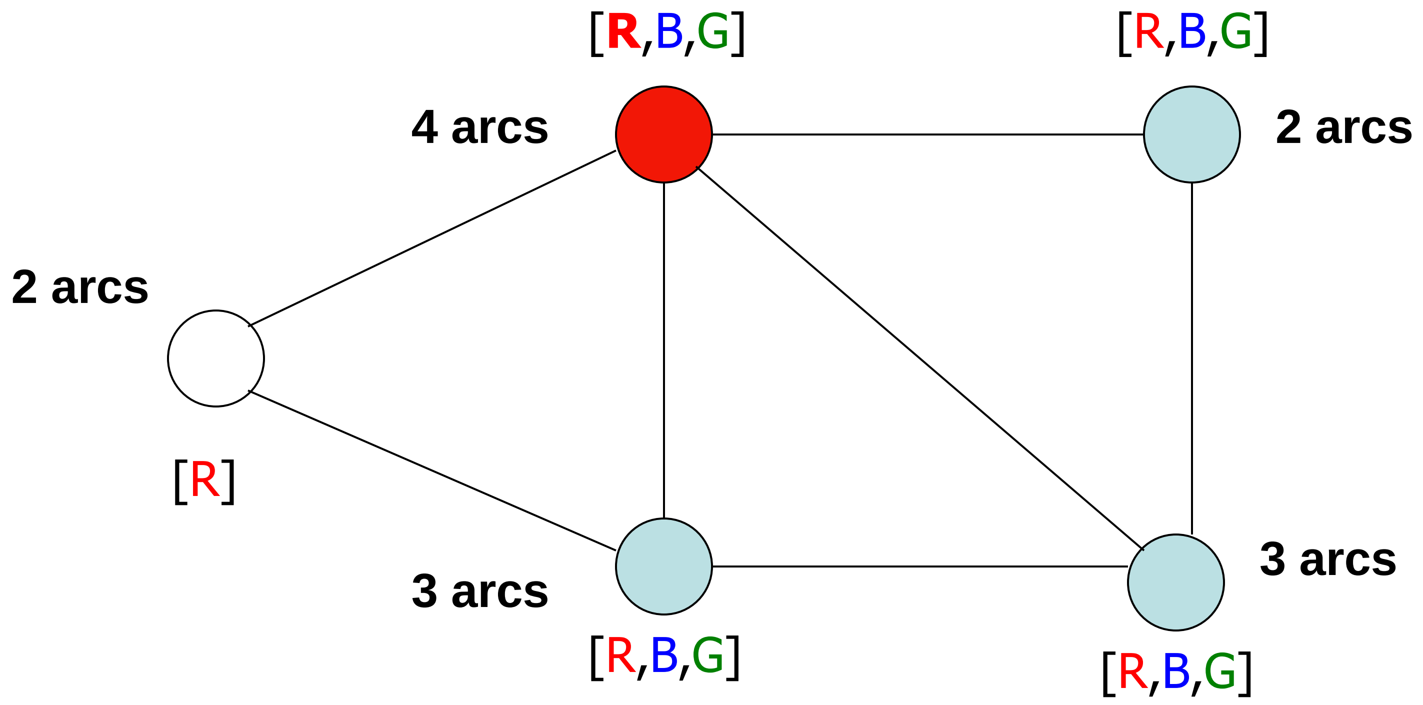




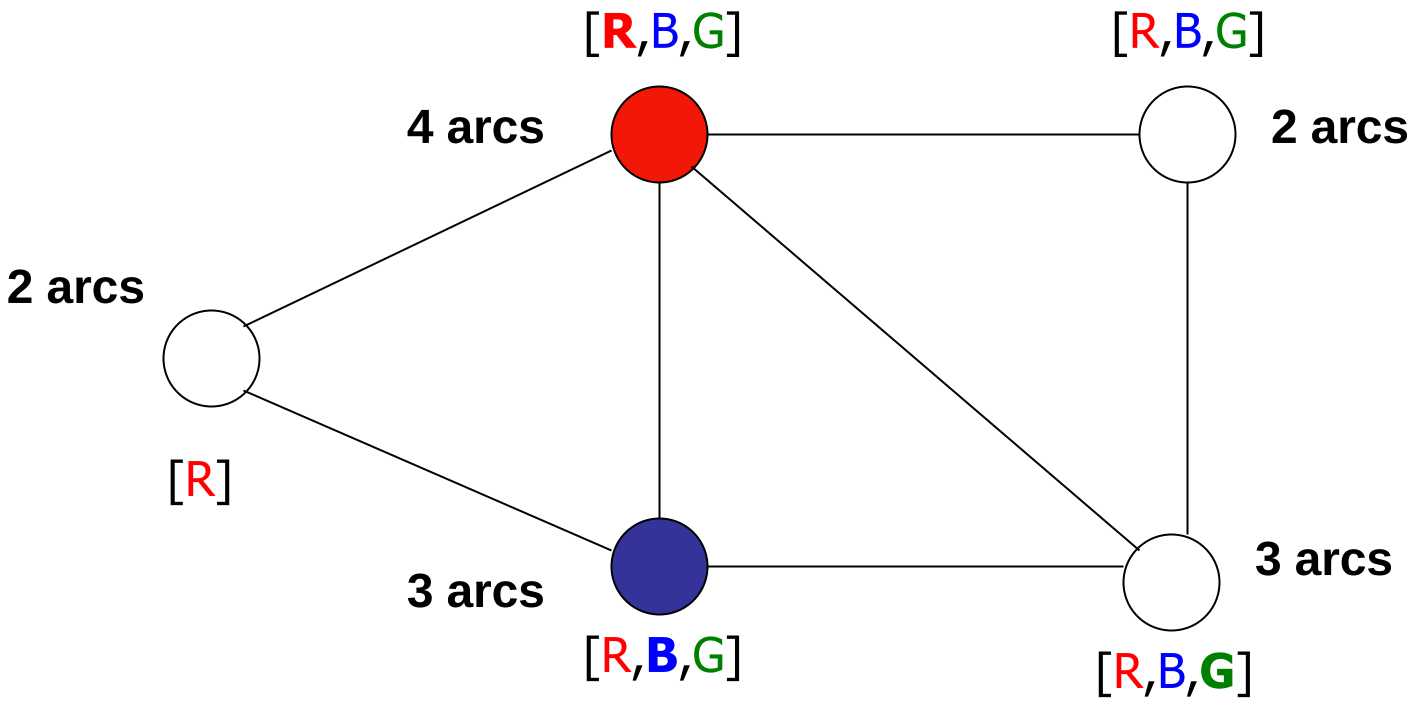
Backpropagation - MCV



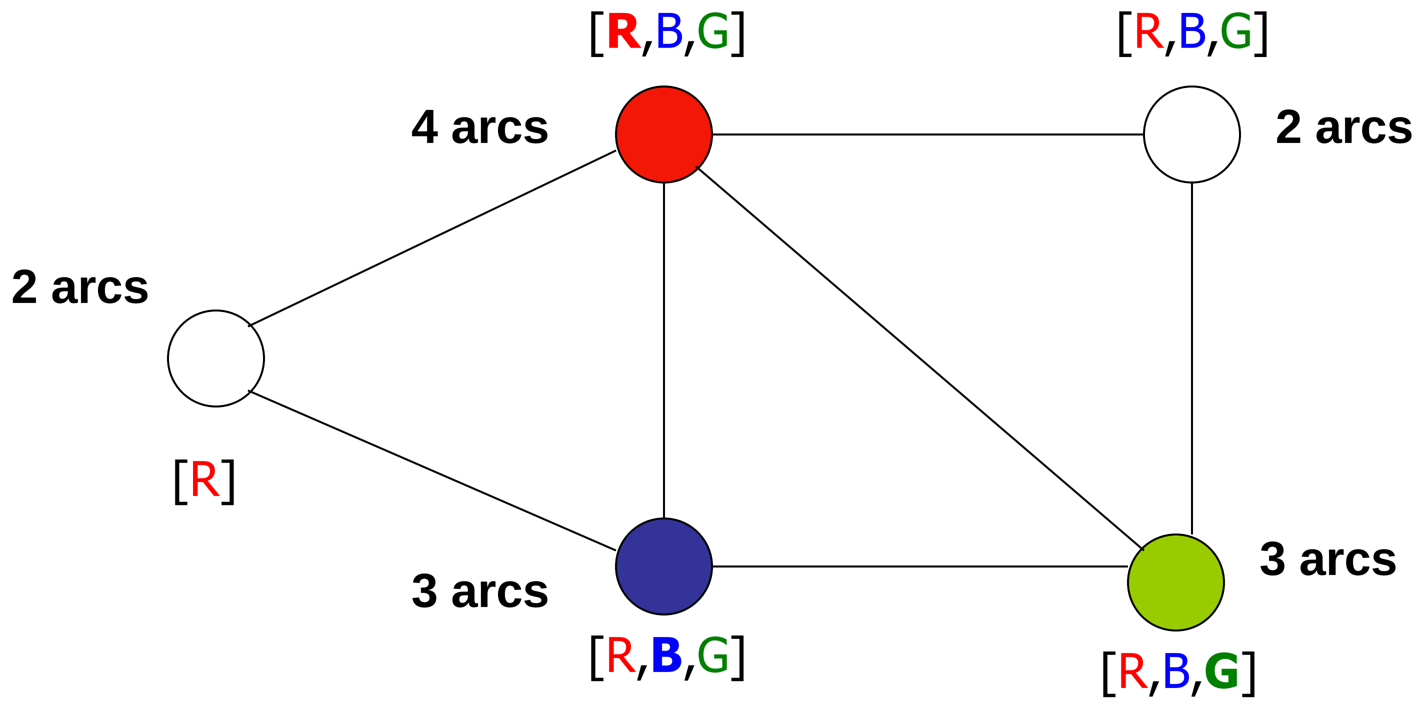
Backpropagation - MCV



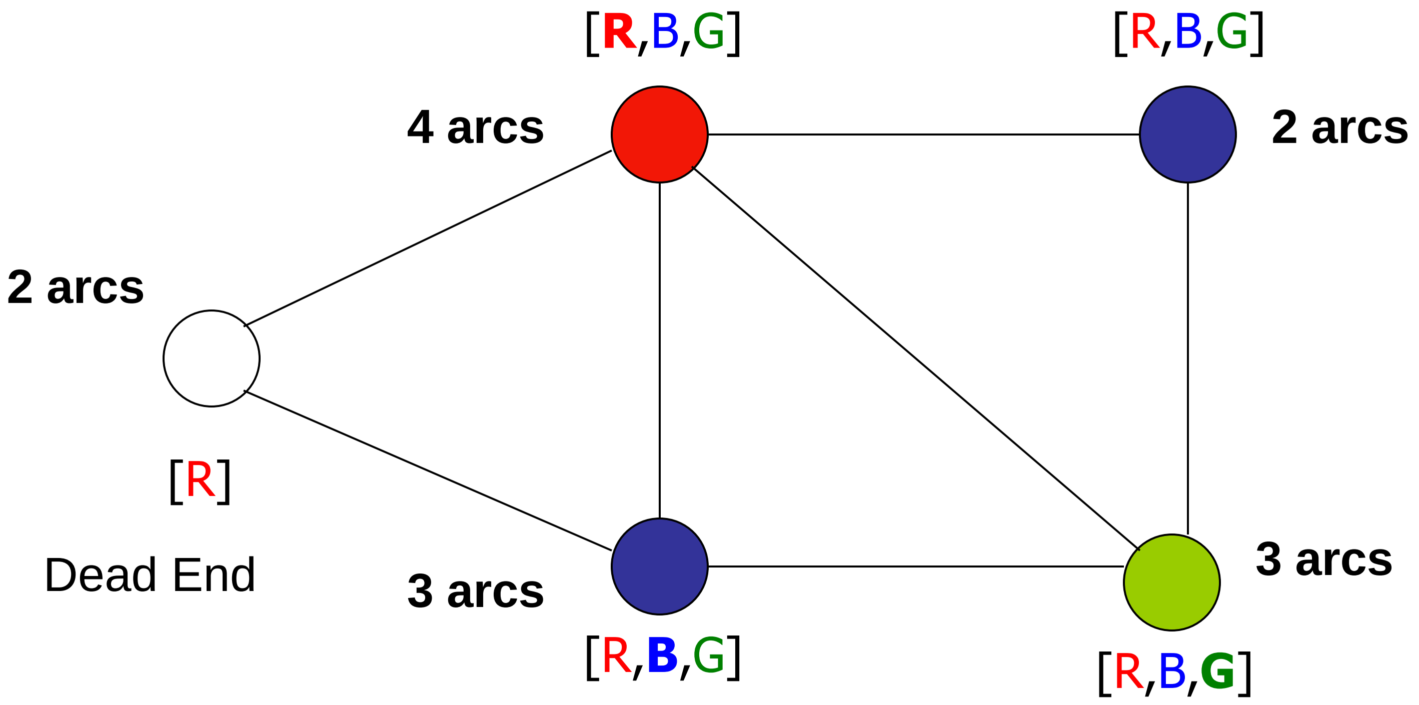
Backpropagation - MCV



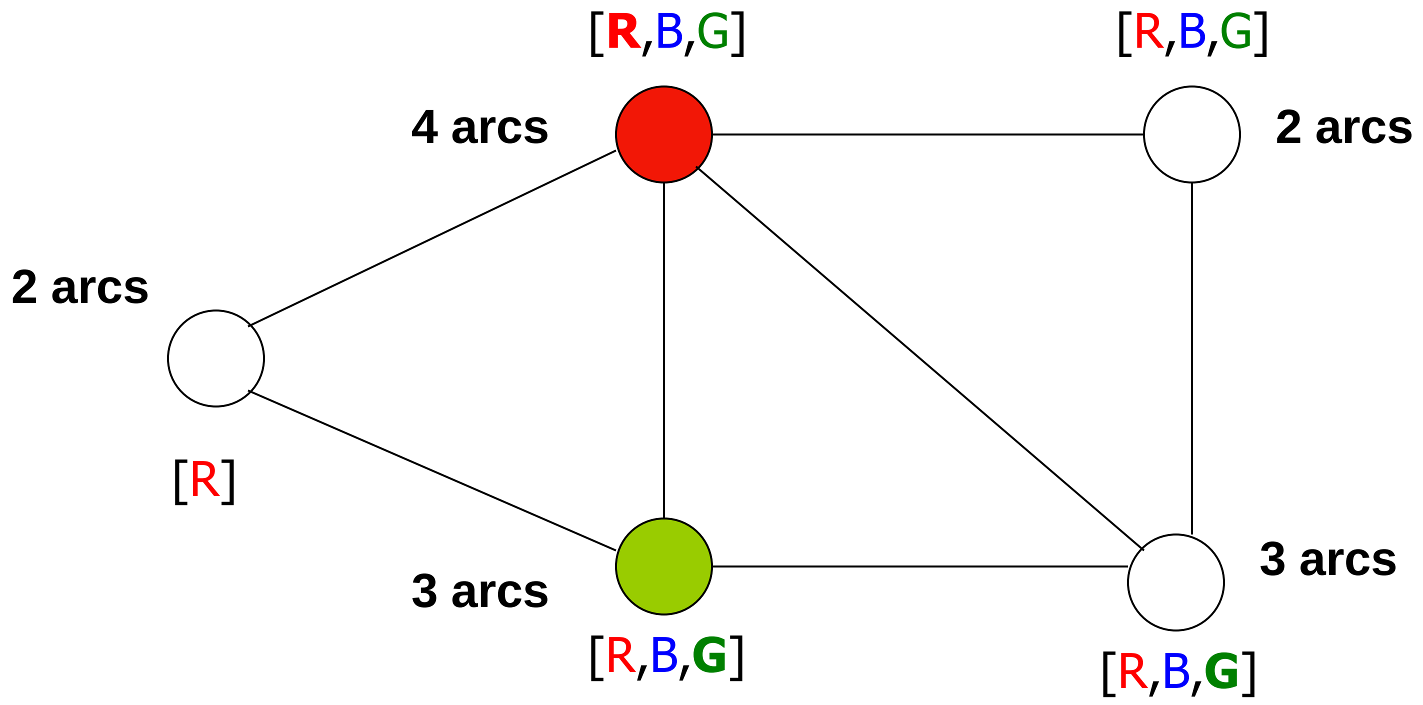
Backpropagation - MCV



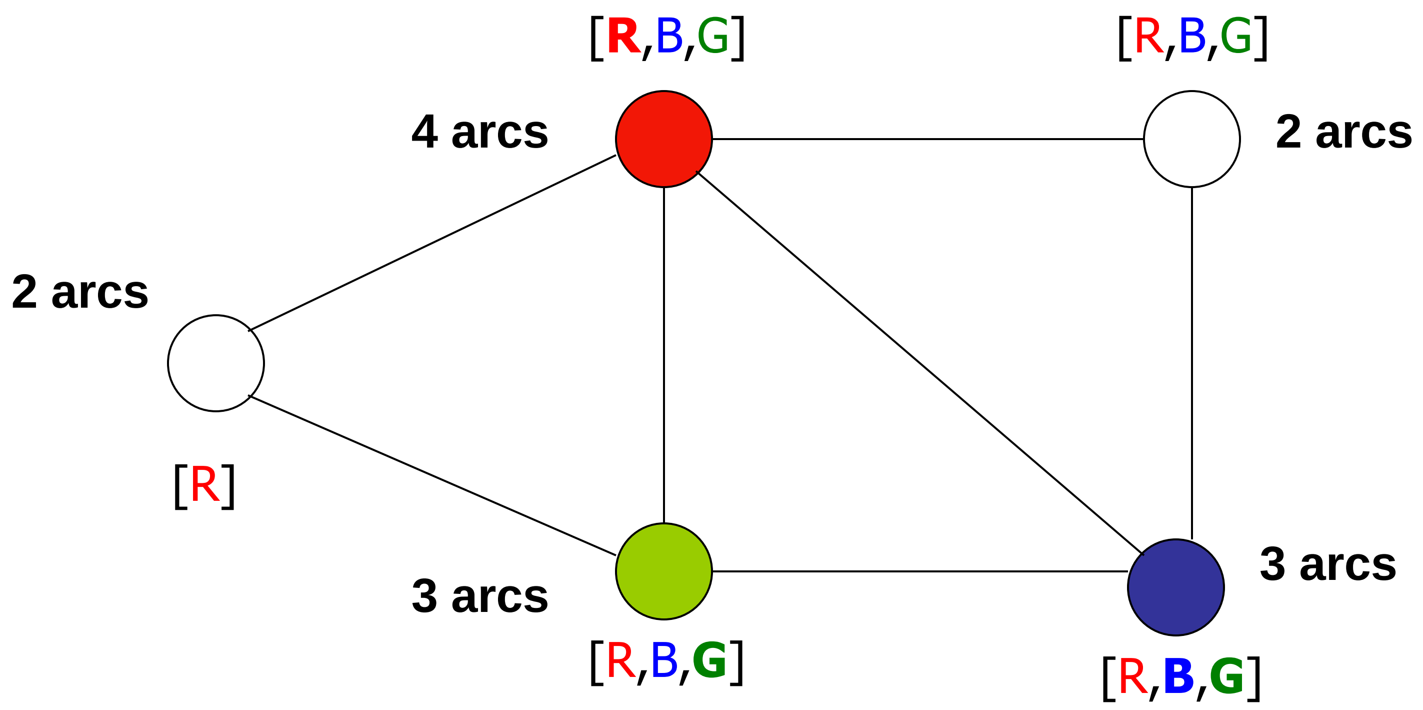
Backpropagation - MCV



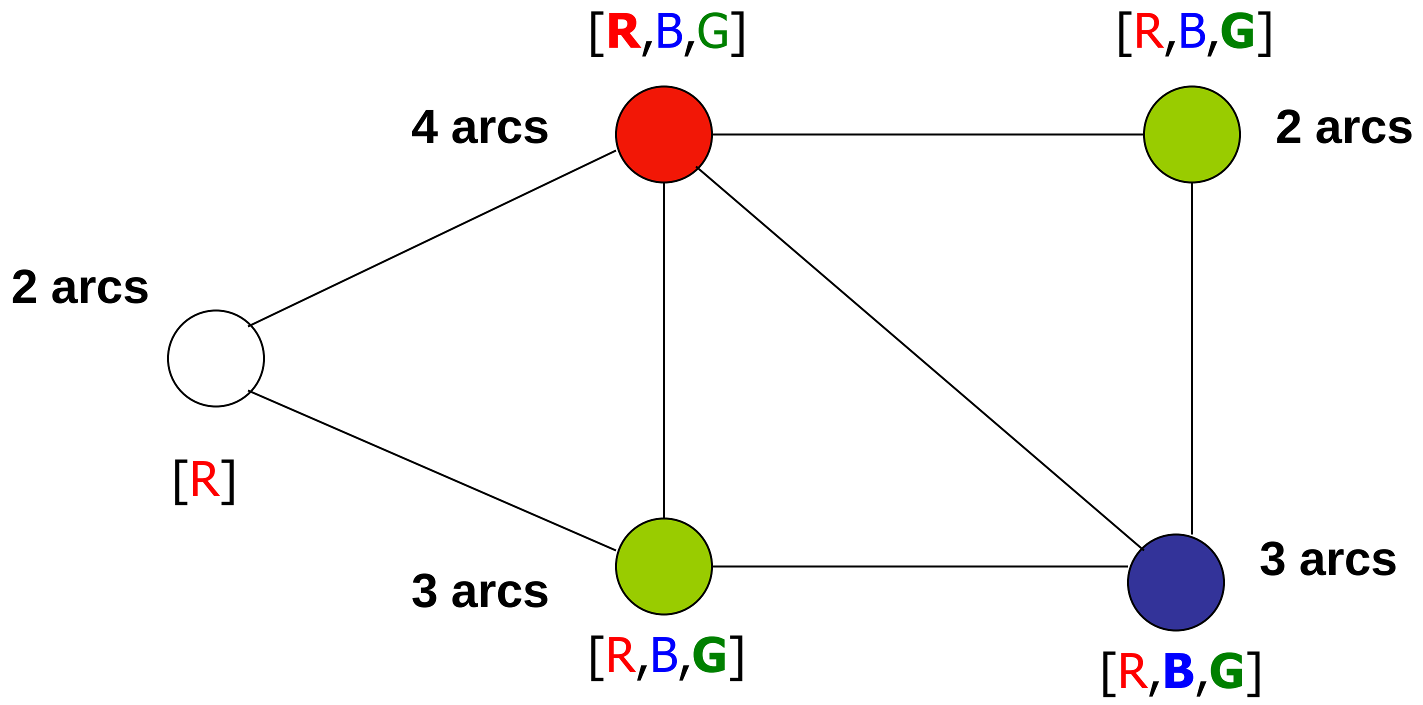
Backpropagation - MCV



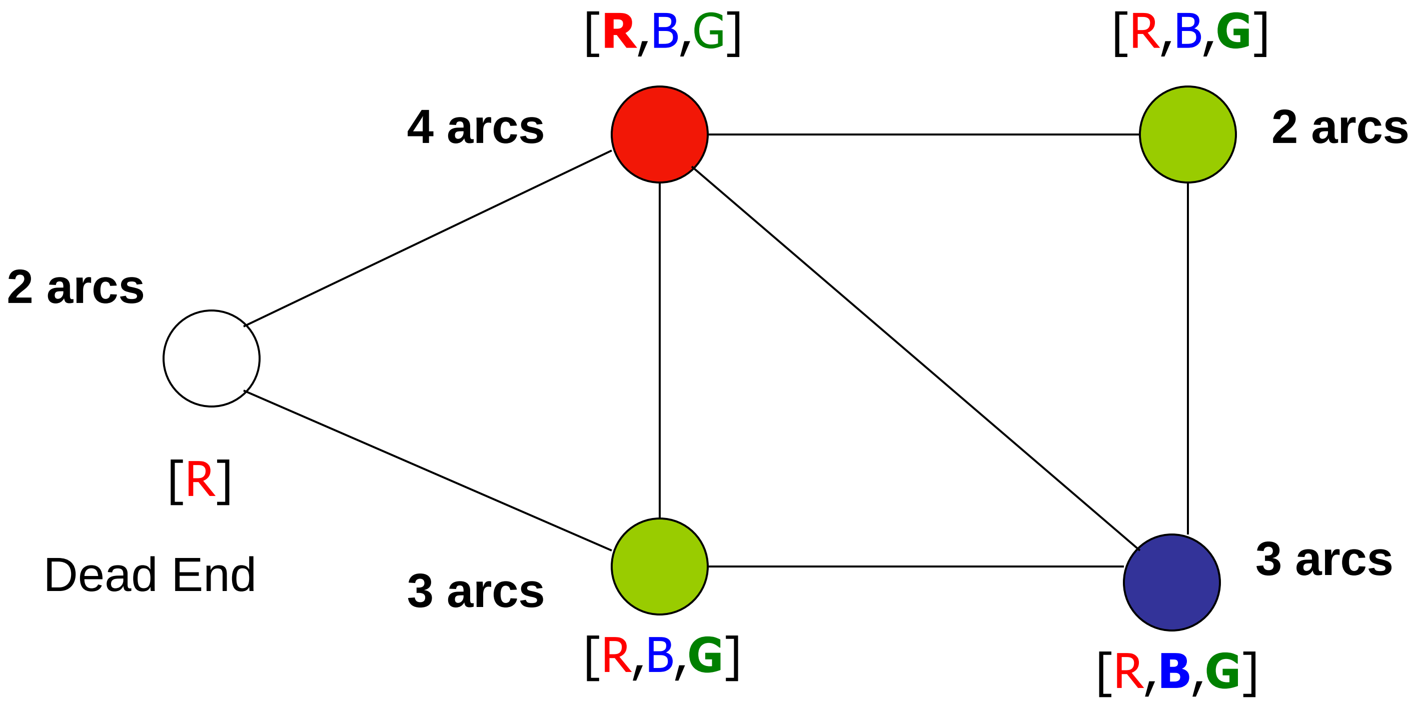
Backpropagation - MCV



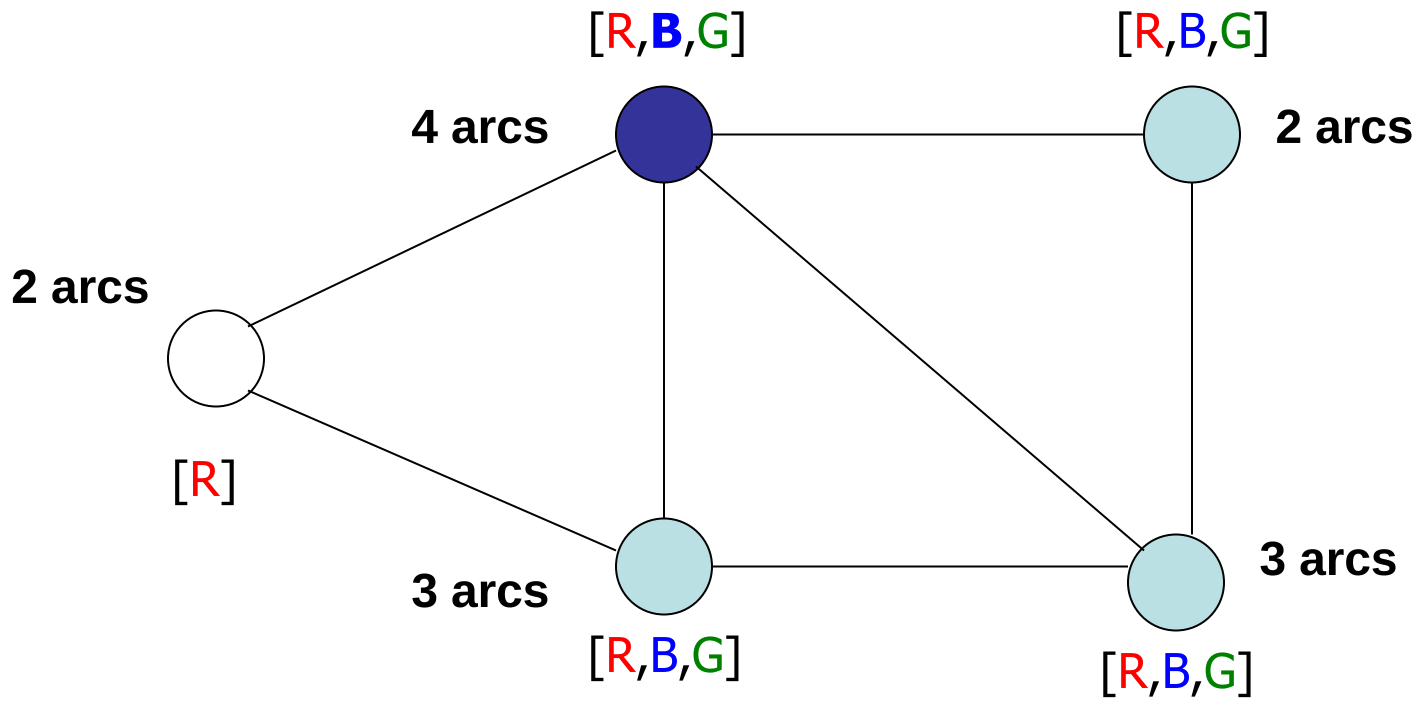
Backpropagation - MCV



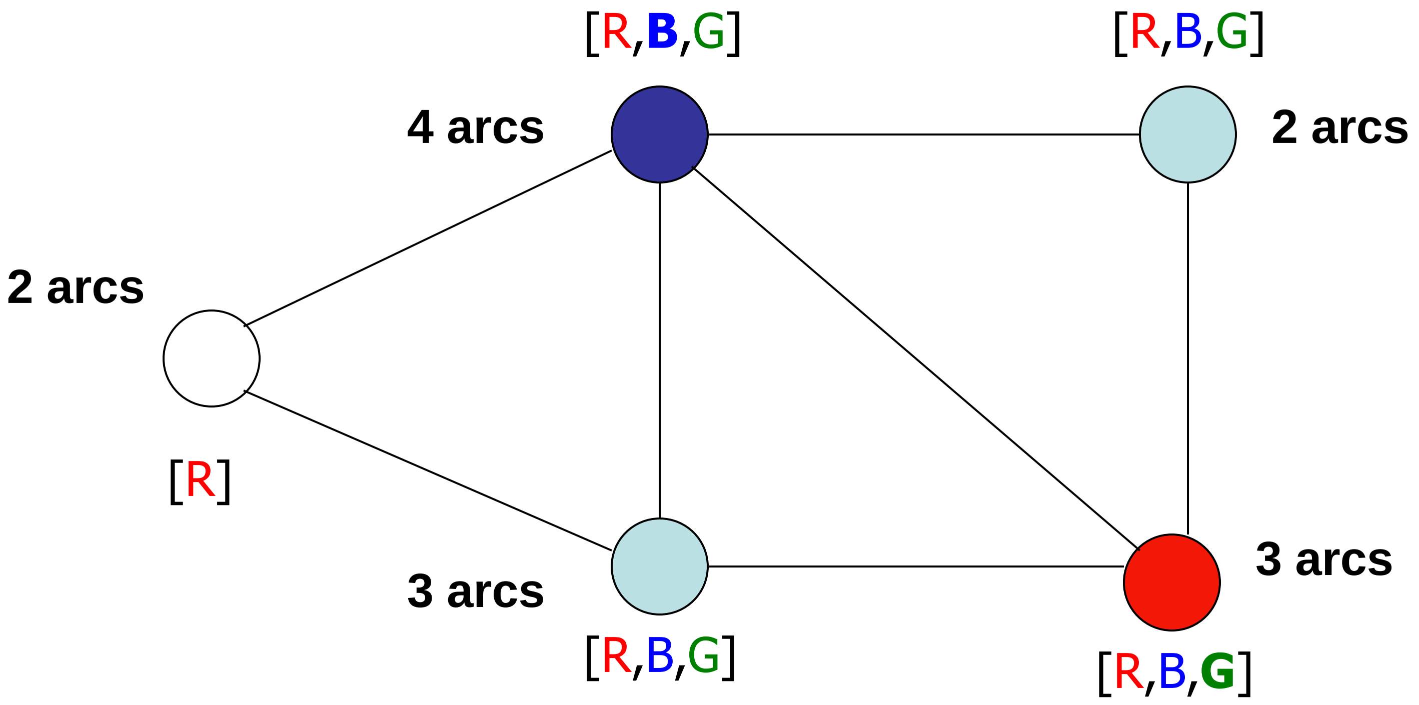
Backpropagation - MCV



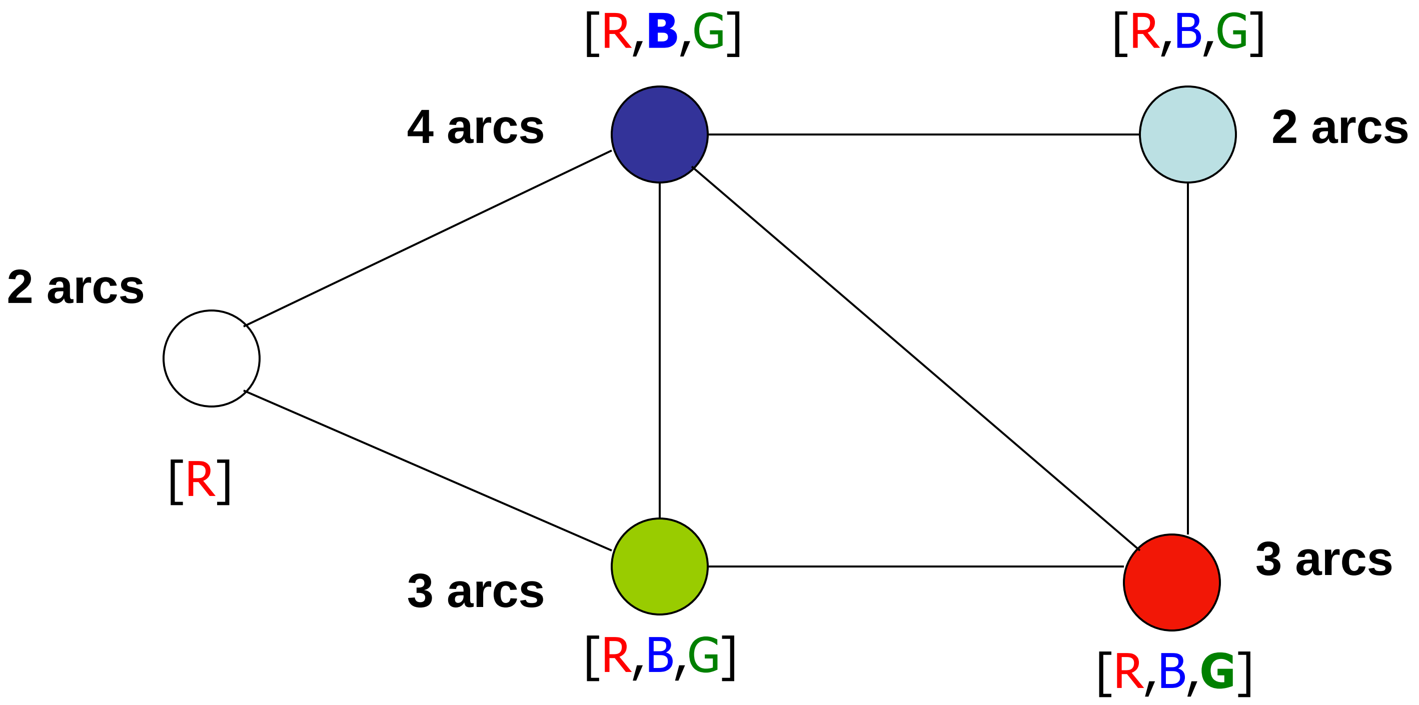
Backpropagation - MCV



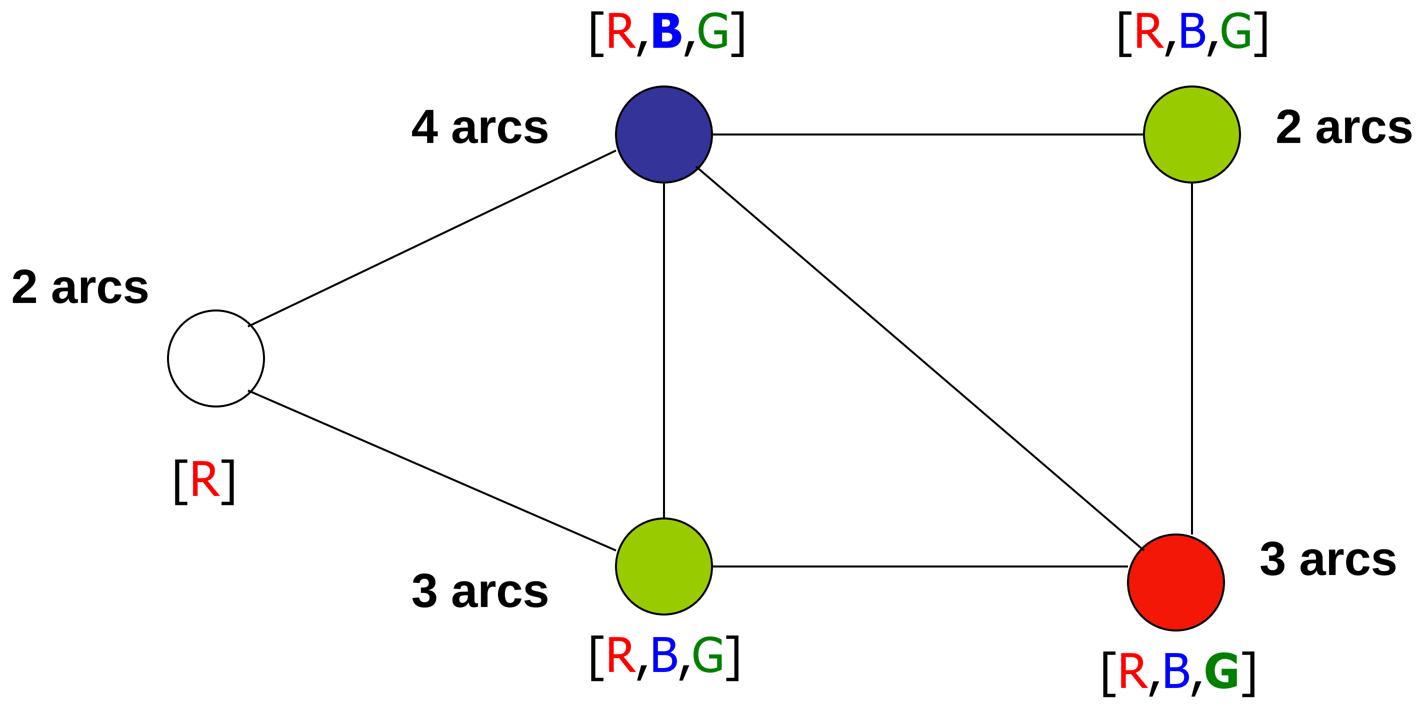
Backpropagation - MCV



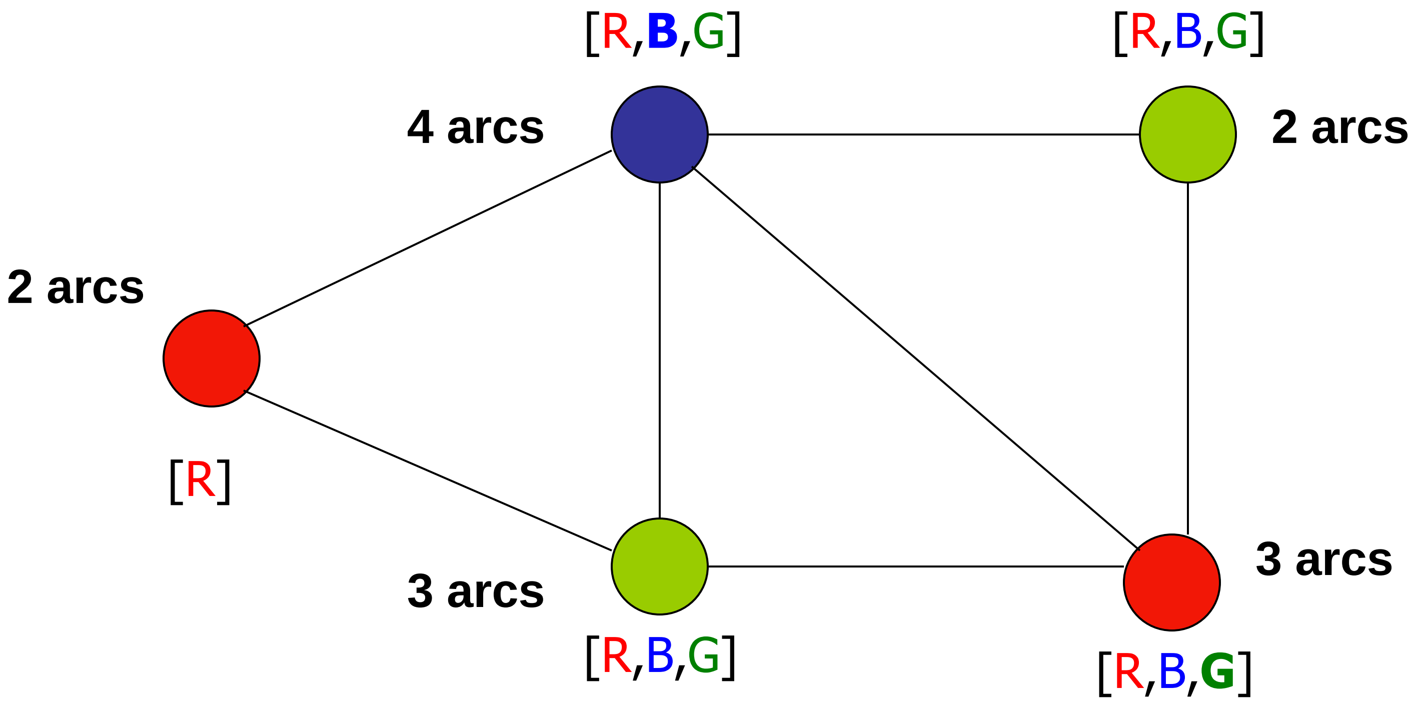
Backpropagation - MCV



Backpropagation - MCV



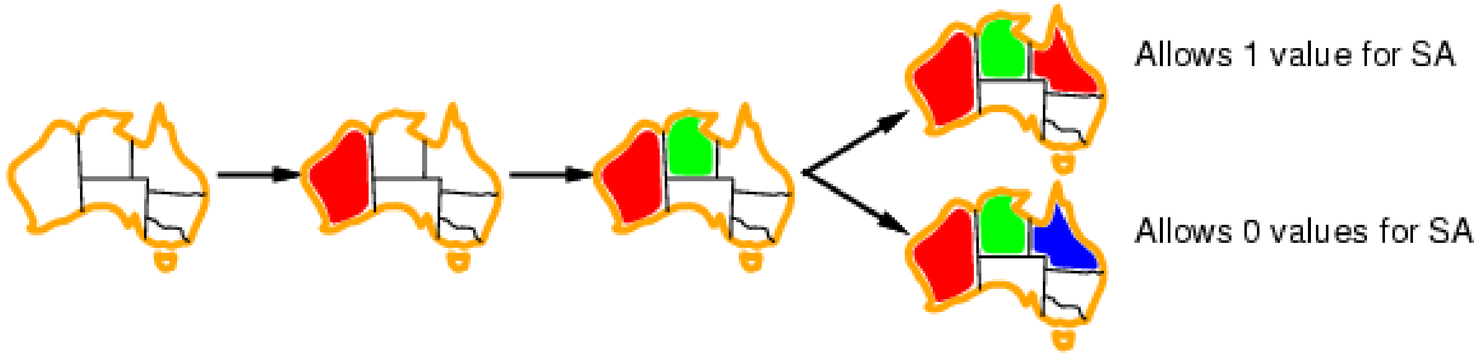
Backpropagation - MCV



Solution !!!

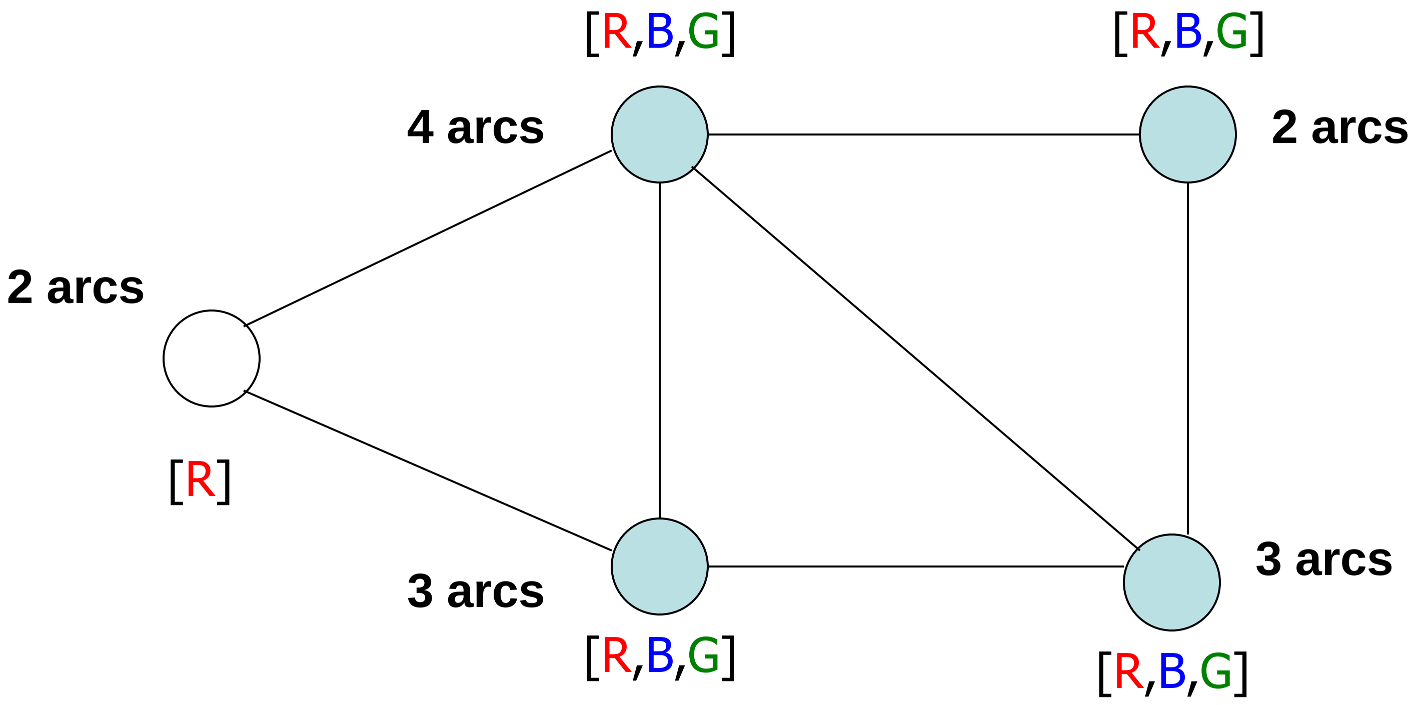
Least constraining value - LCV

- Given a variable, choose the least constraining value:
  - the one that rules out (eliminate) the fewest values in the remaining variables

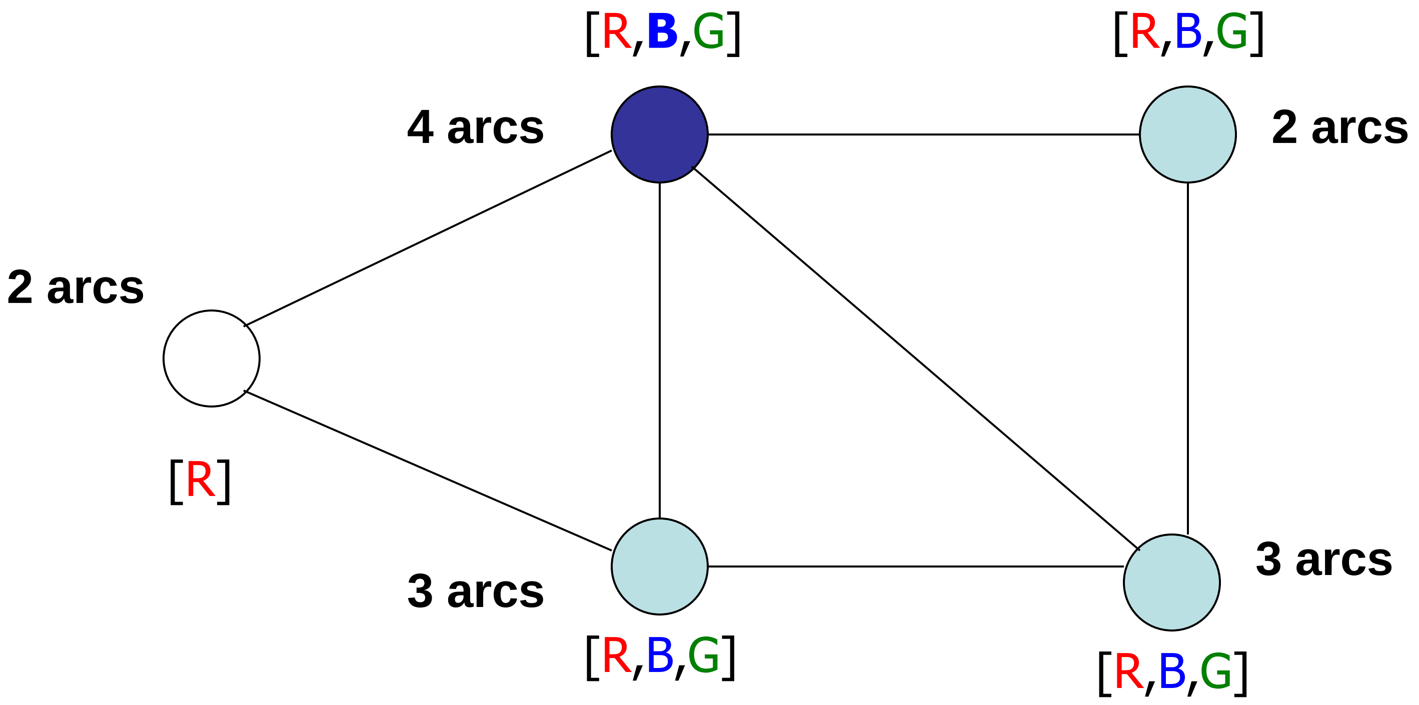


- Combining these heuristics makes 1000 queens feasible

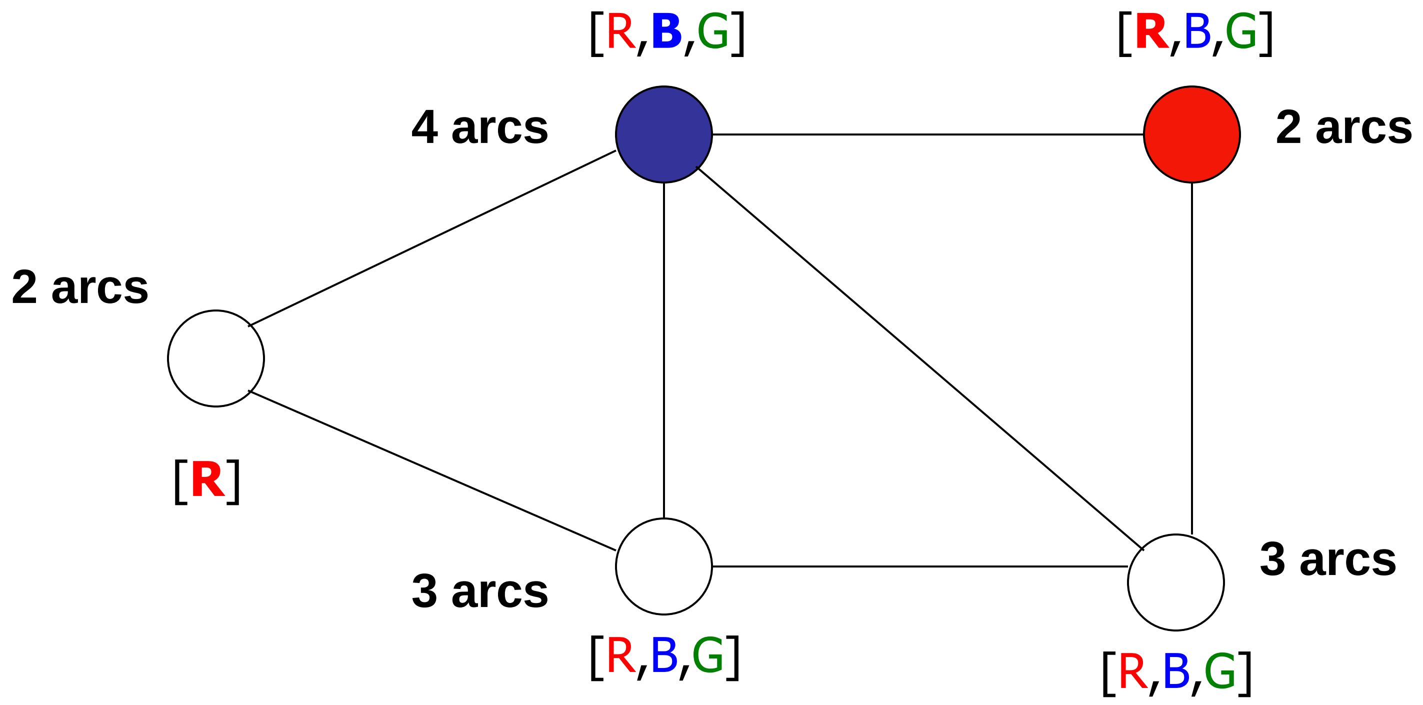
Backpropagation - LCV



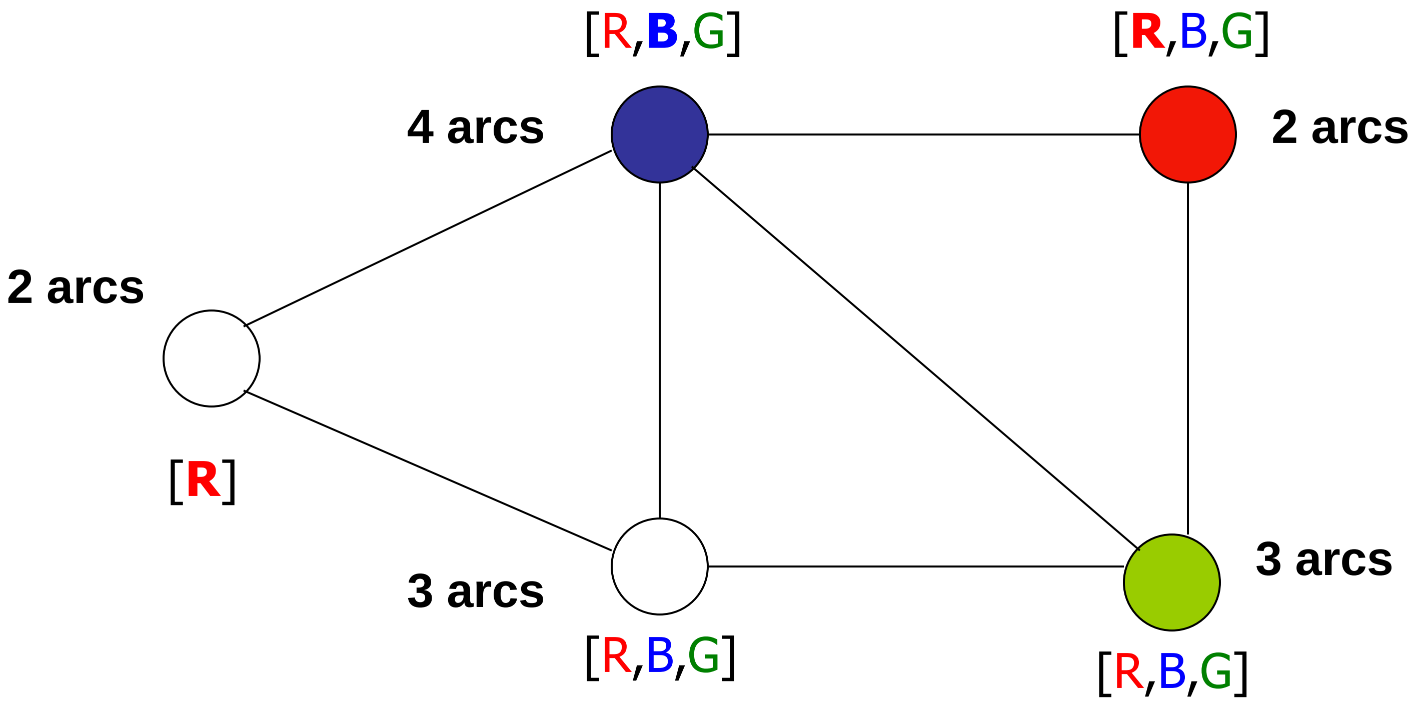
Backpropagation - LCV



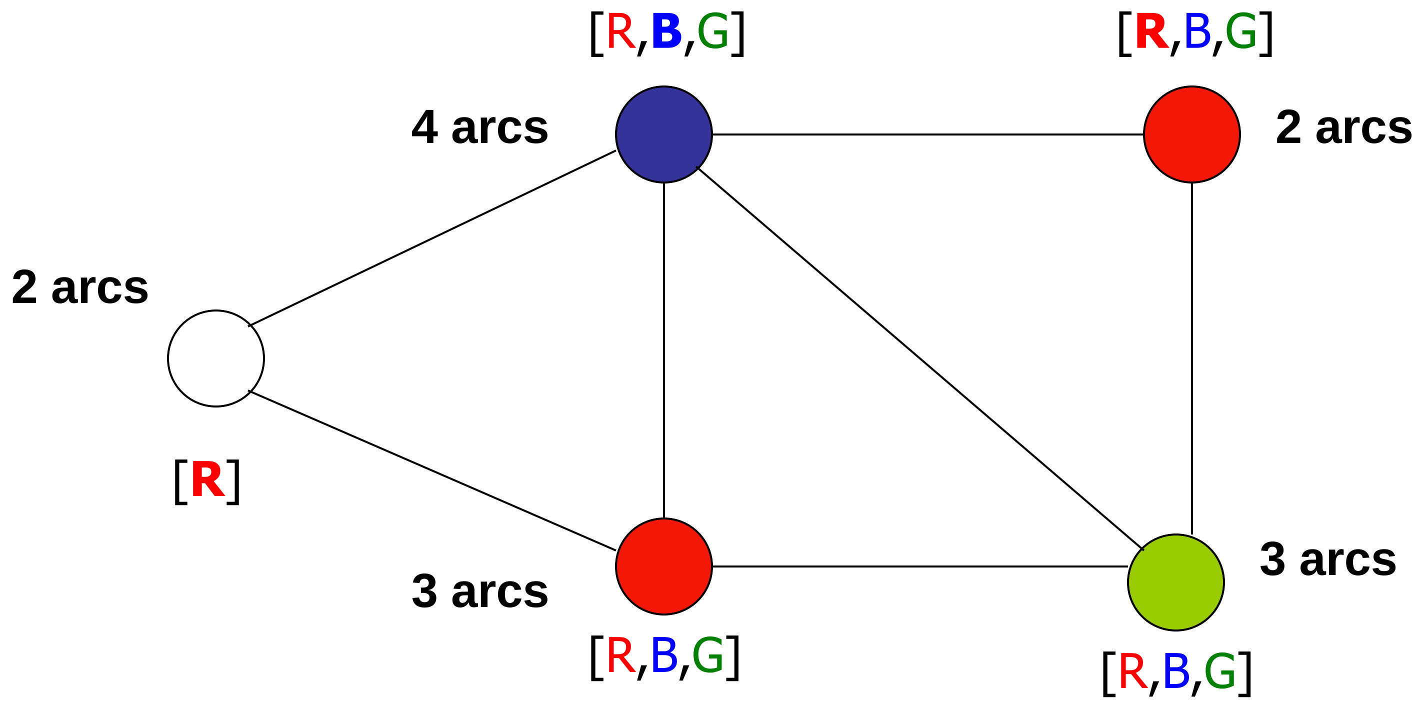
Backpropagation - LCV



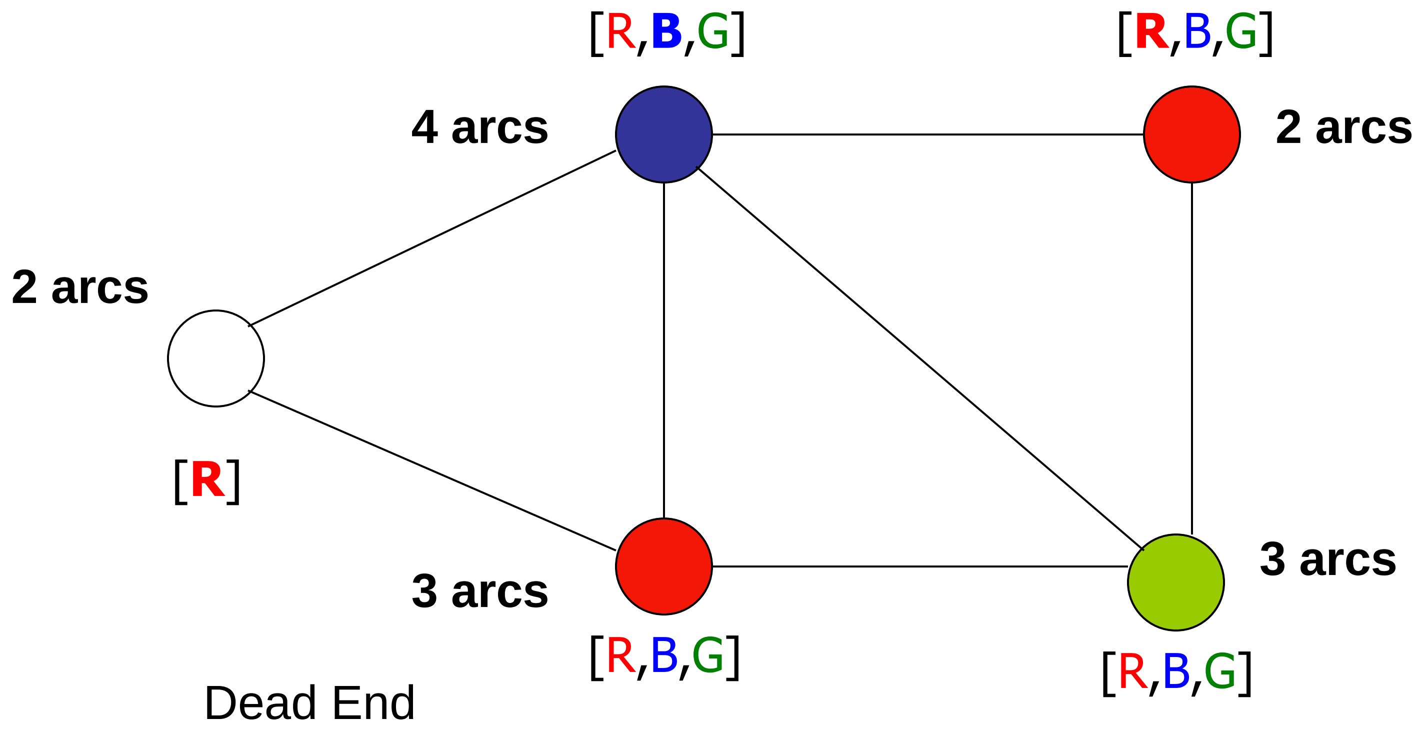
Backpropagation - LCV



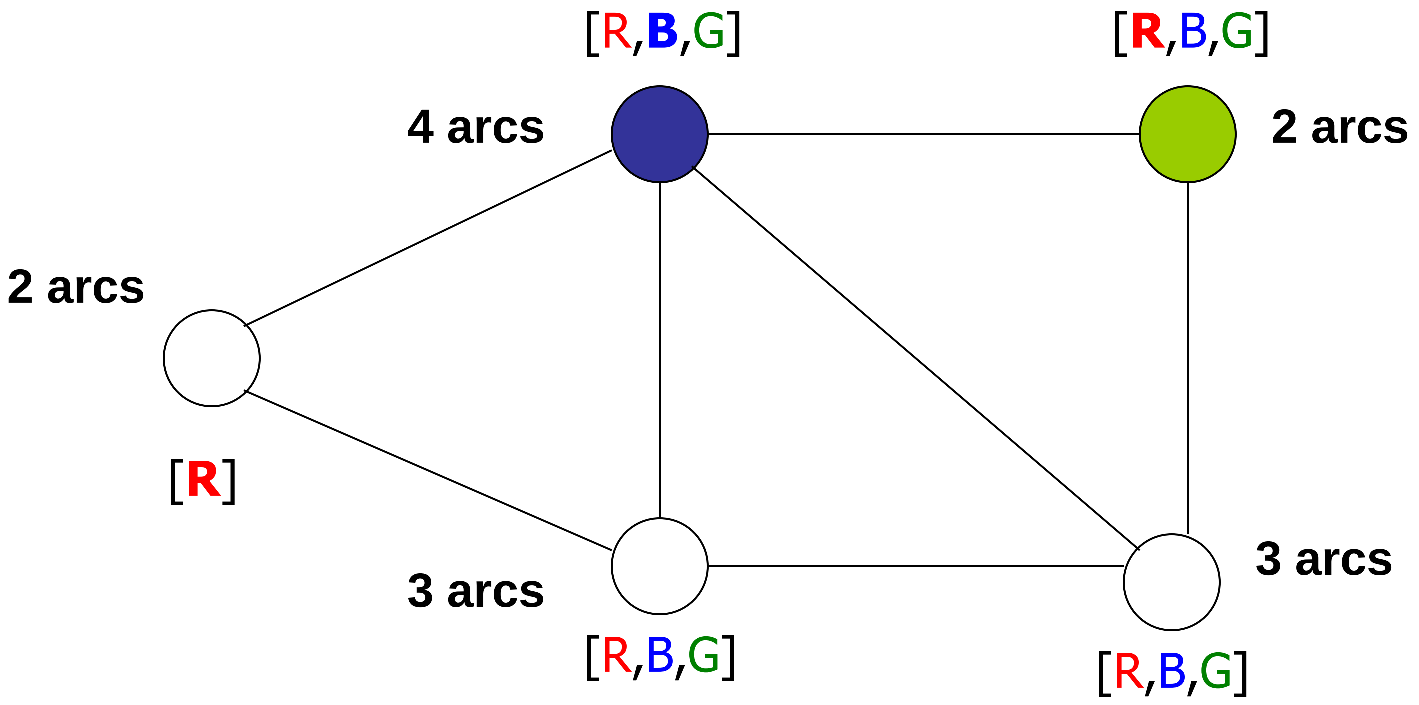
Backpropagation - LCV



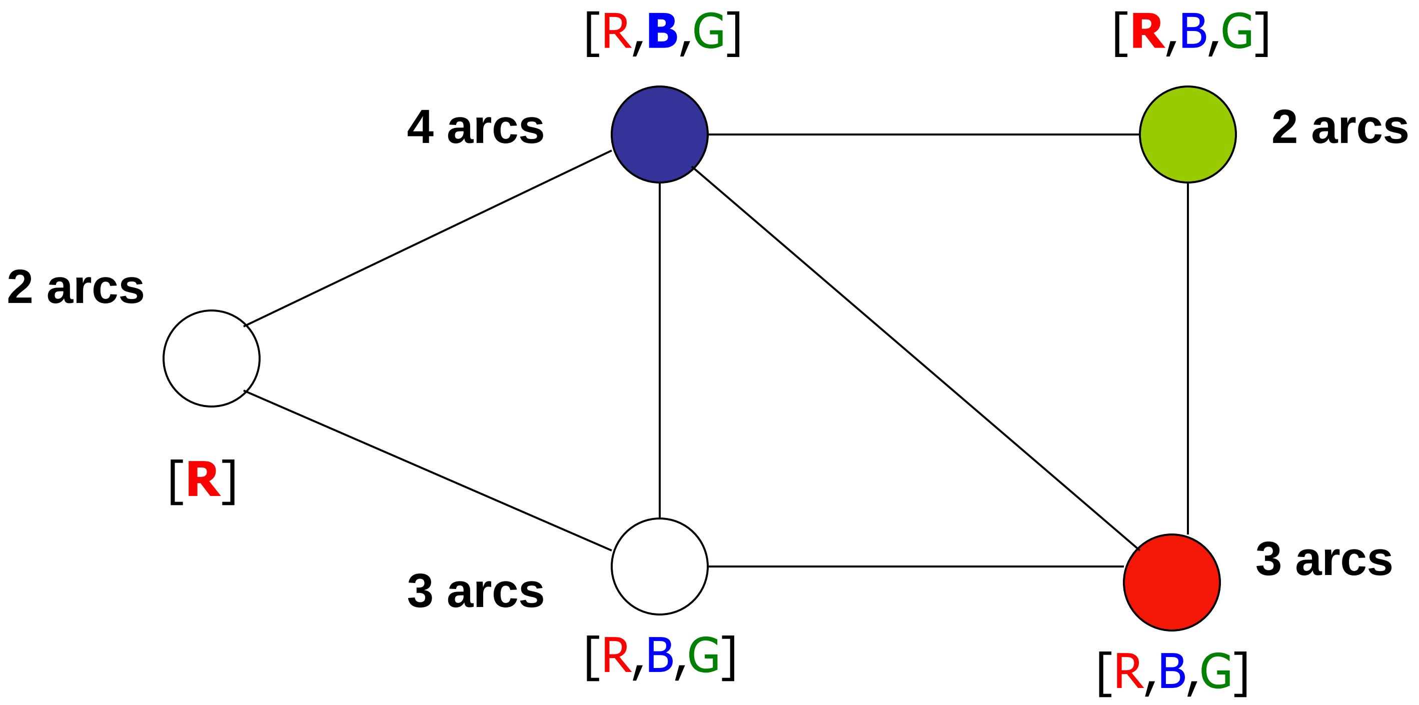
Backpropagation - LCV



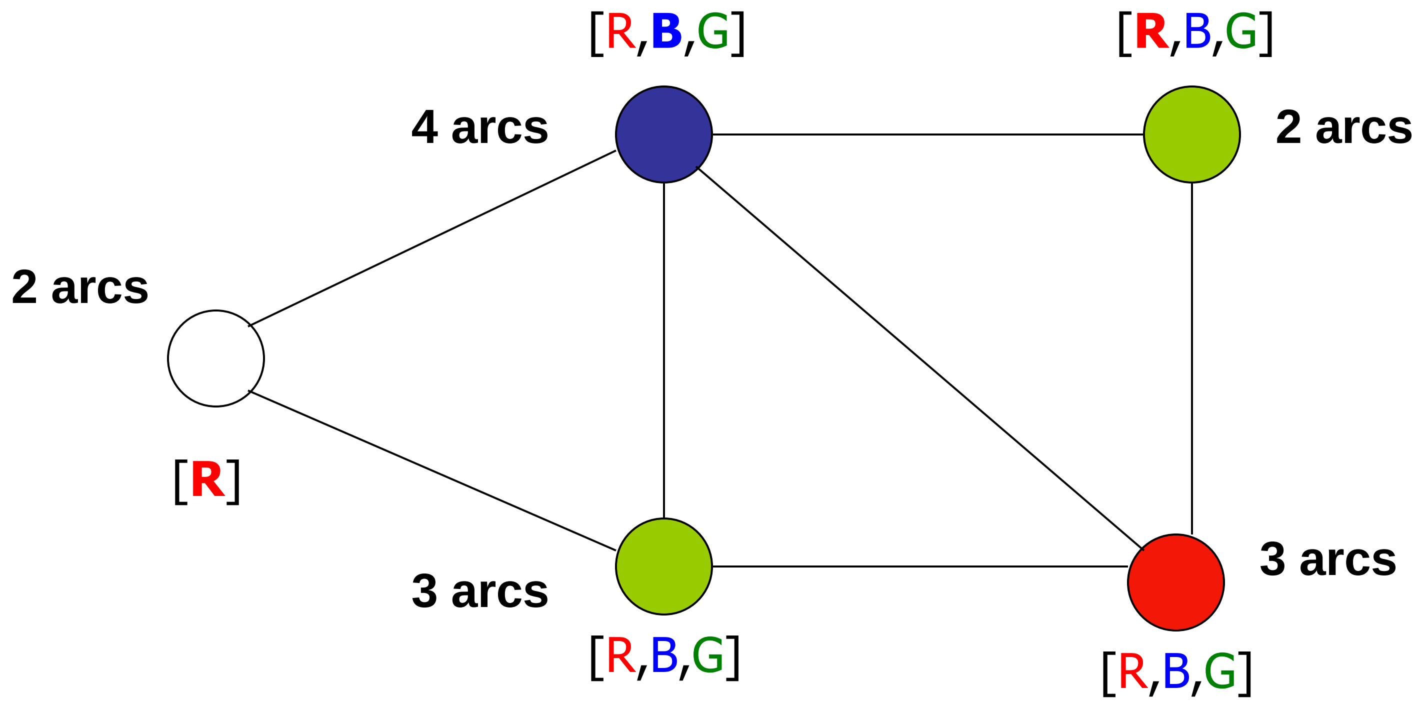
Backpropagation - LCV



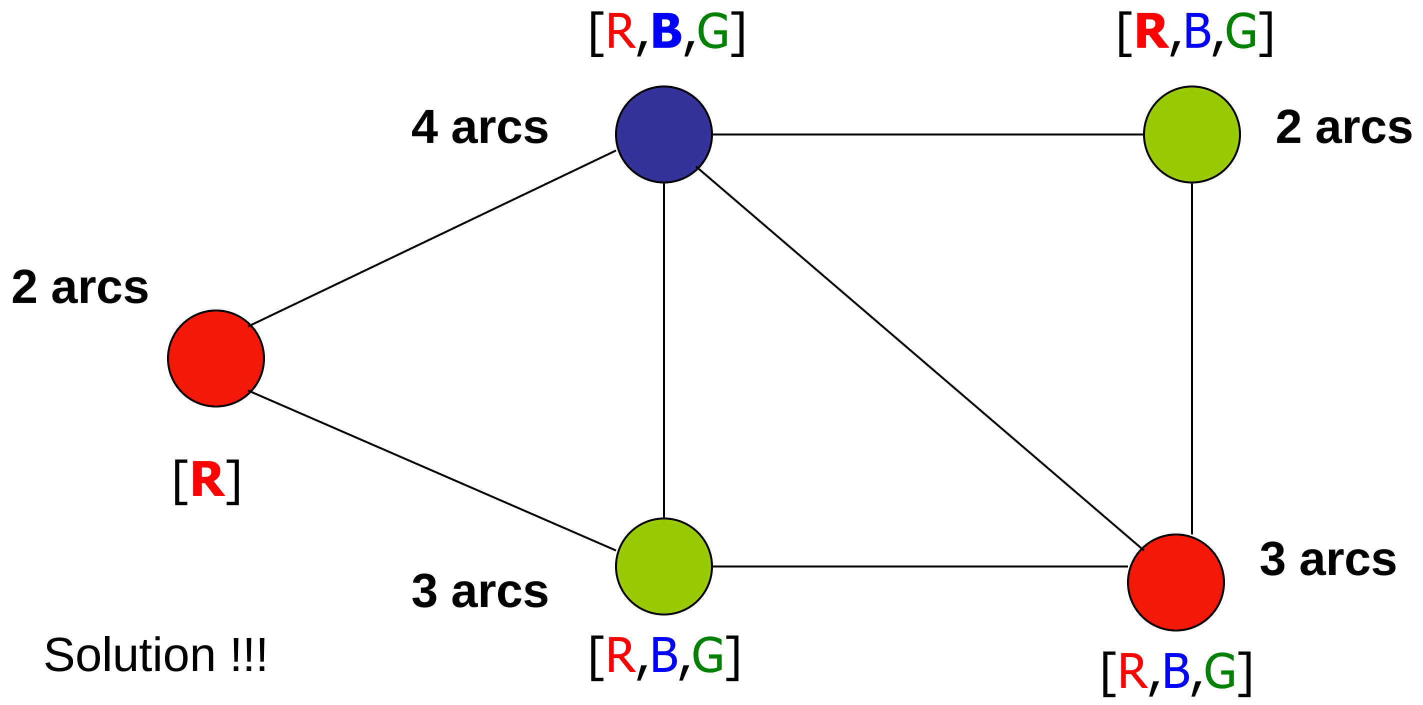
Backpropagation - LCV



Backpropagation - LCV



Backpropagation - LCV



Solution !!!

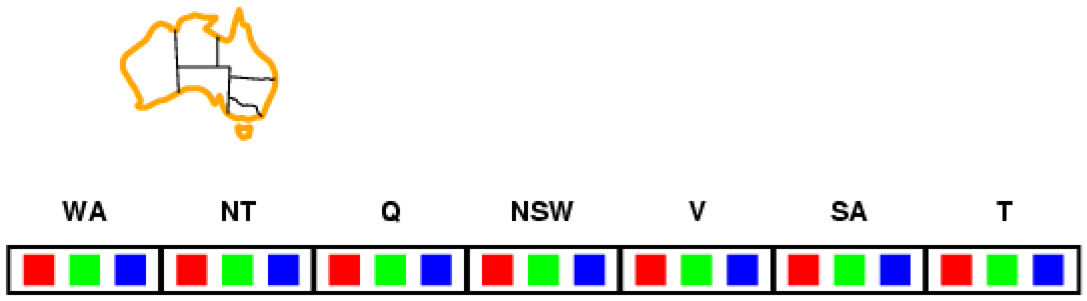
Analyzing Constraints

- forward checking
  - when a value *X* is assigned to a variable, inconsistent values are eliminated for all variables connected to *X*
    - identifies "dead" branches of the tree before they are visited

- constraint propagation
  - analyses interdependencies between variable assignments via *arc consistency*
    - an arc between *X* and *Y* is consistent if for every possible value *x* of *X*, there is some value *y* of *Y* that is consistent with *x*
    - more powerful than forward checking, but still reasonably efficient
    - but does not reveal every possible inconsistency

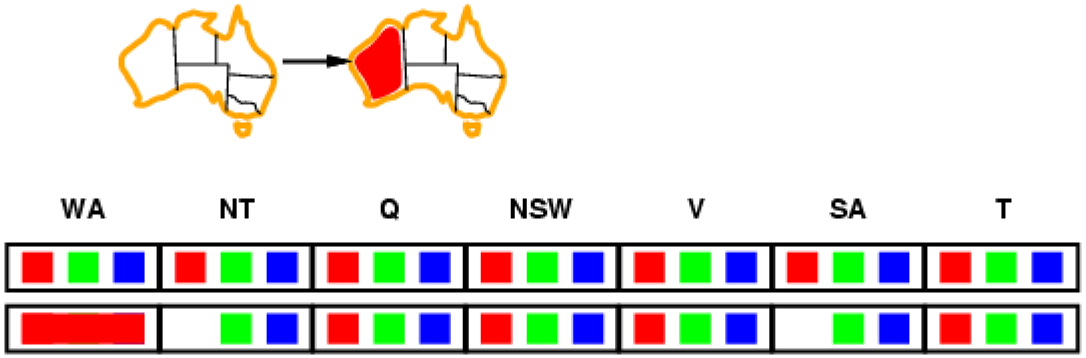
Forward checking

- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values



Forward checking

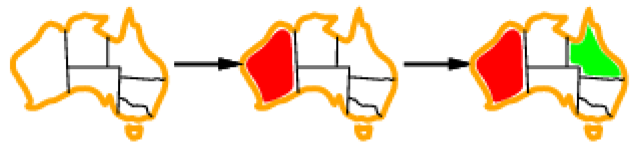
- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values



WA	NT	Q	NSW	V	SA	T
🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦
🟥	🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟩🟦	🟥🟩🟦

Forward checking

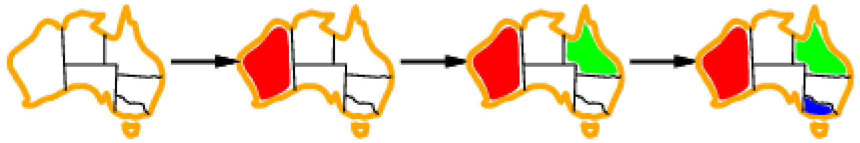
- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values



WA	NT	Q	NSW	V	SA	T
🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦
🟥	🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟩 🟦	🟥 🟩 🟦
🟥	🟦	🟩	🟥 🟦	🟥 🟩 🟦	🟦	🟥 🟩 🟦

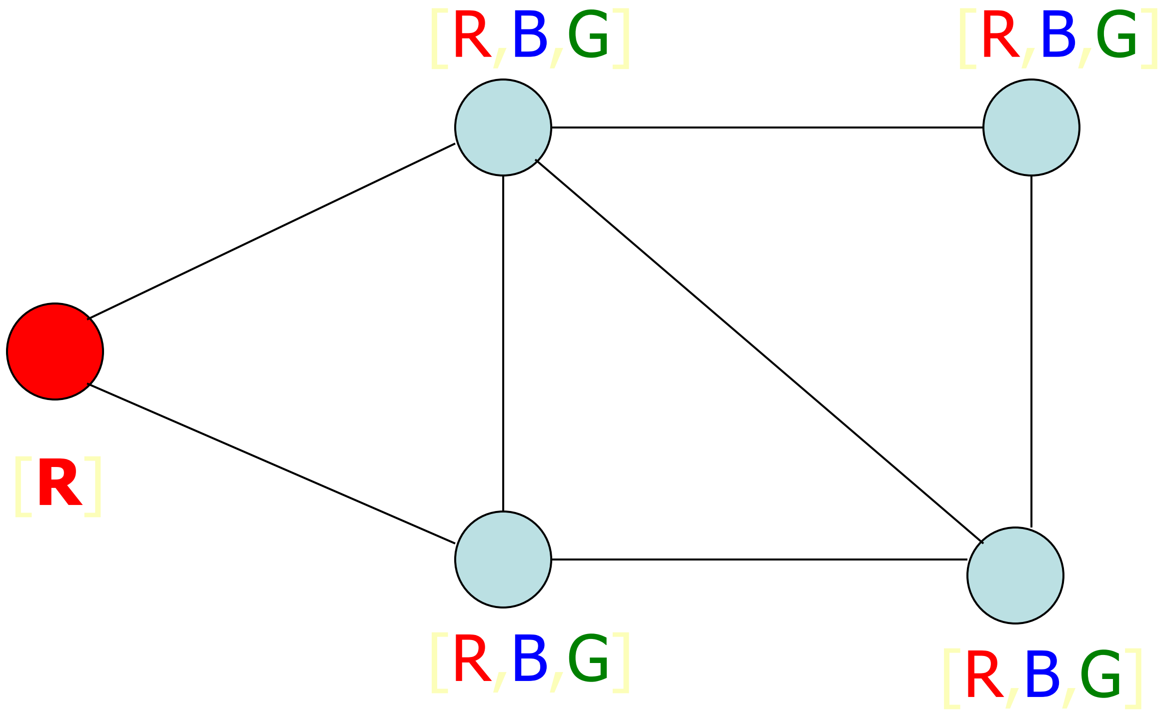
Forward checking

- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values

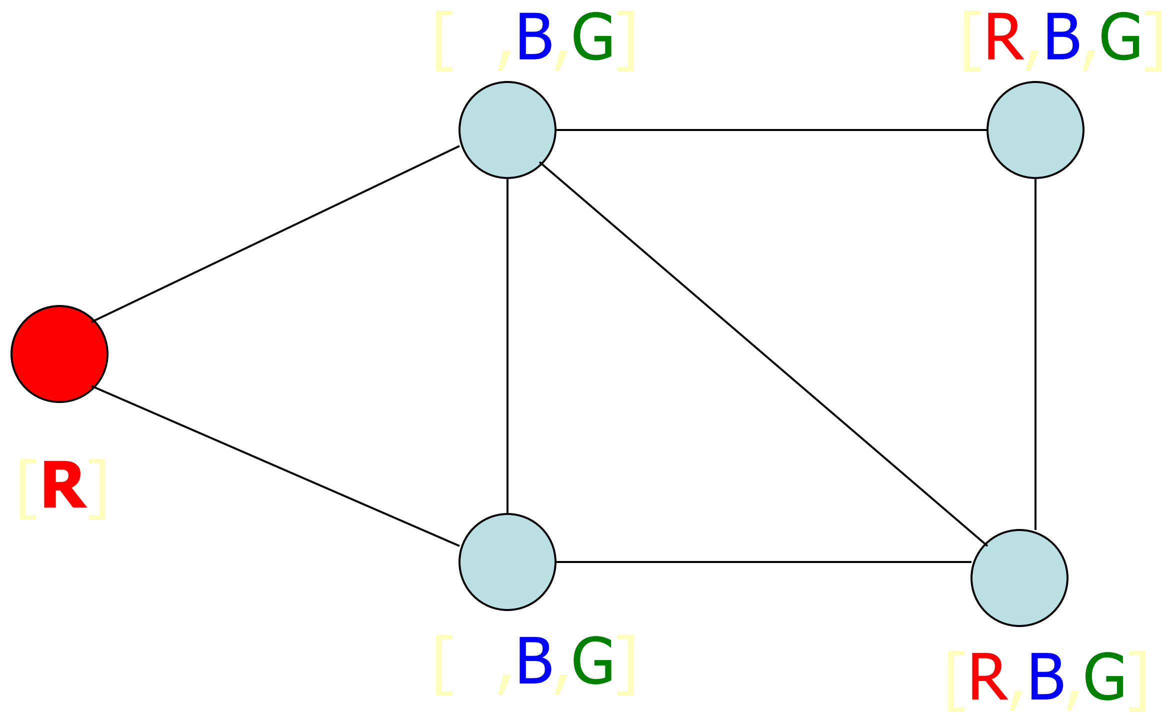


WA	NT	Q	NSW	V	SA	T
🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦
🟥	🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟩🟦	🟥🟩🟦
🟥	🟦	🟩	🟥🟦	🟥🟩🟦	🟦	🟥🟩🟦
🟥	🟦	🟩	🟥	🟦		🟥🟩🟦

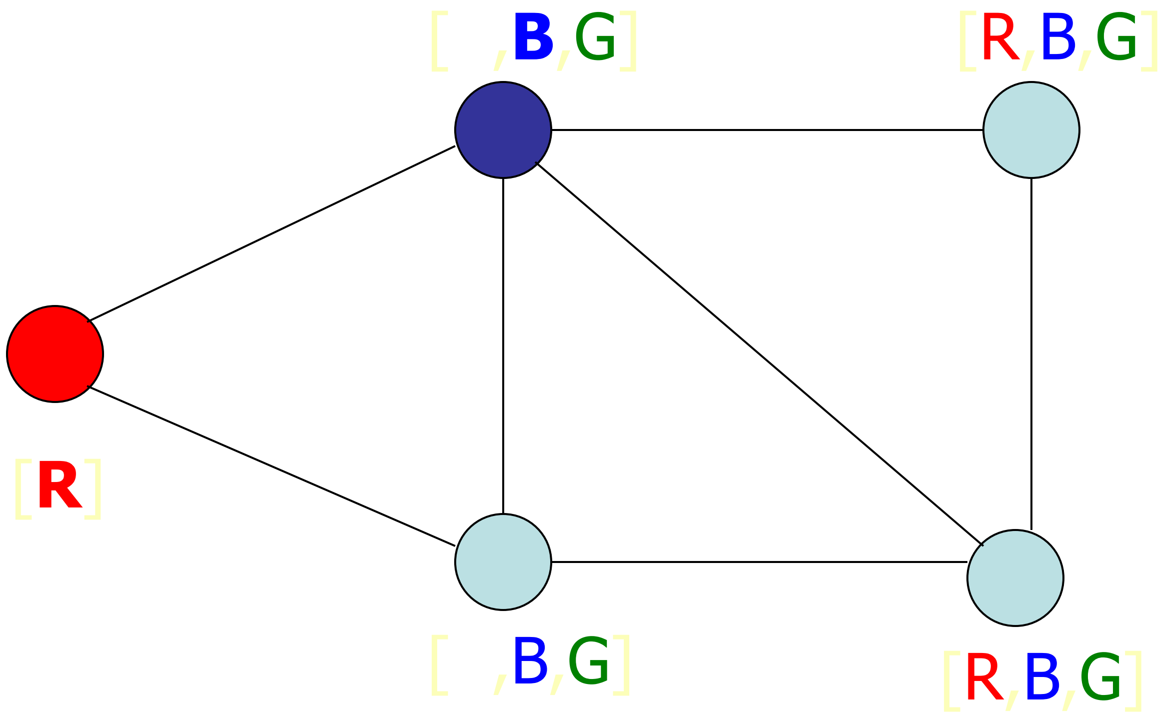
Forward Checking



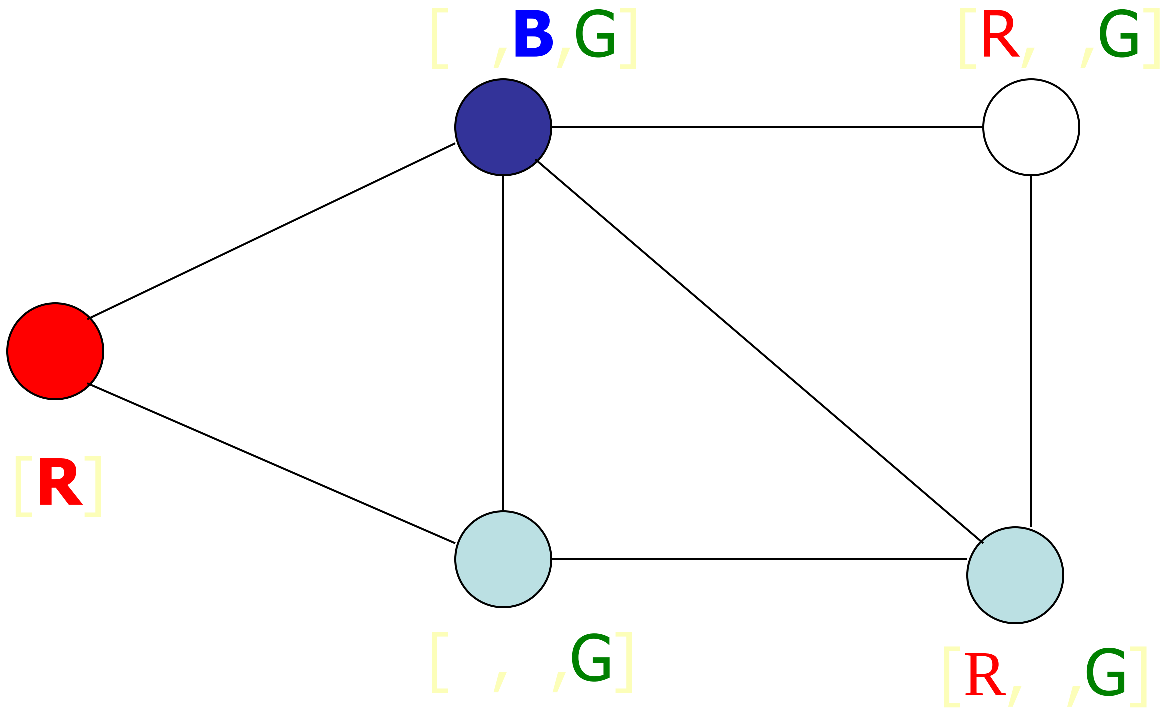
Forward Checking



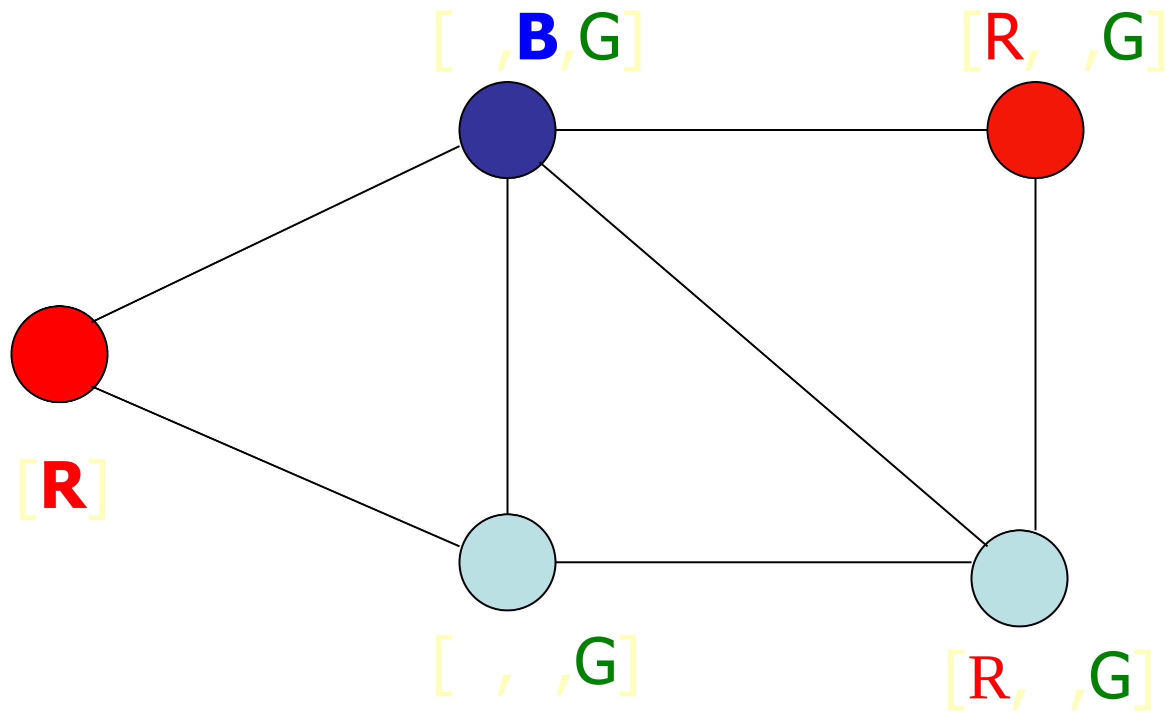
Forward Checking



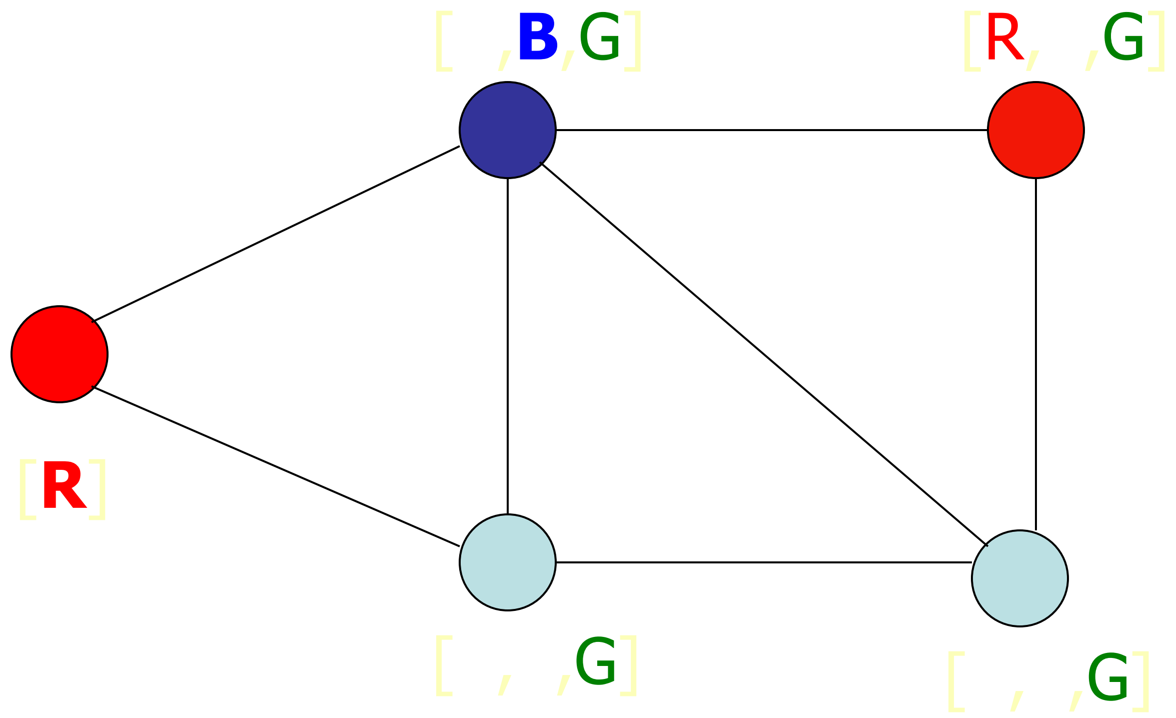
Forward Checking



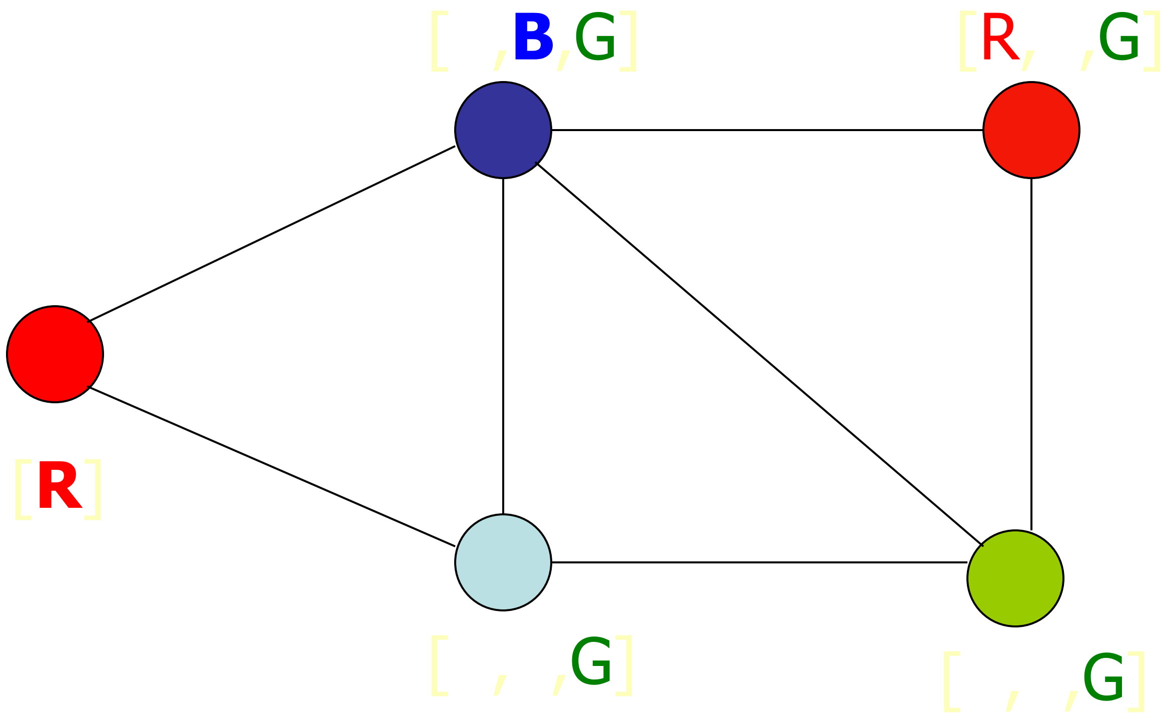
Forward Checking



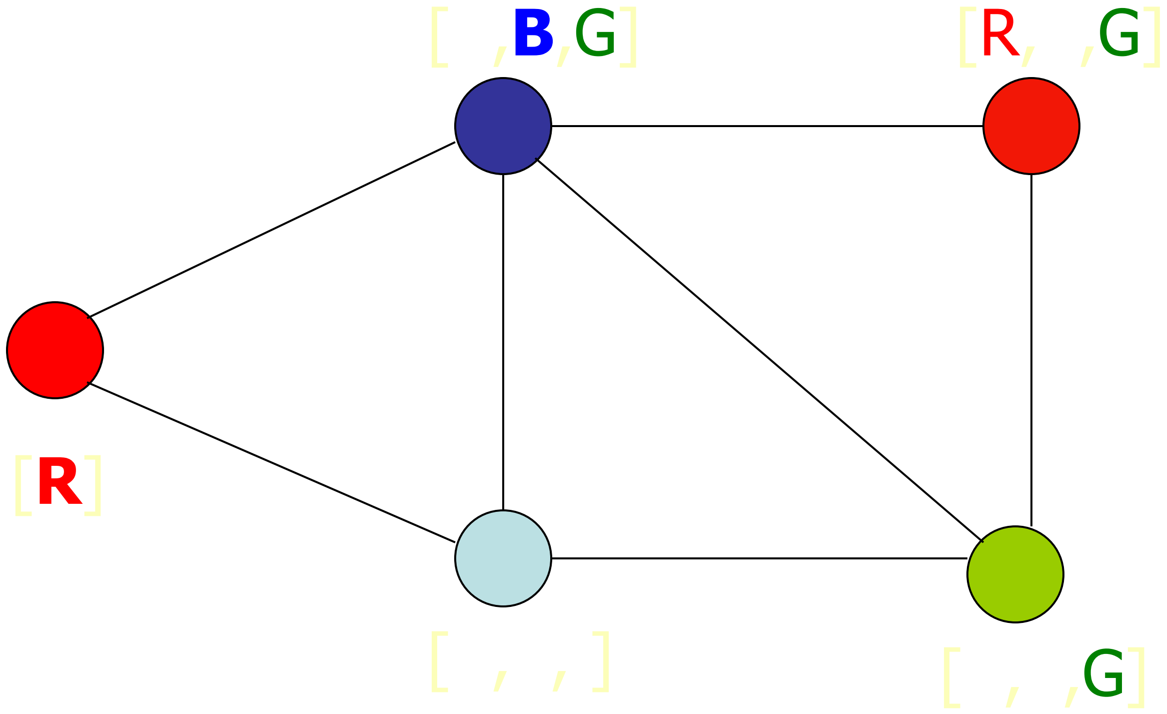
Forward Checking



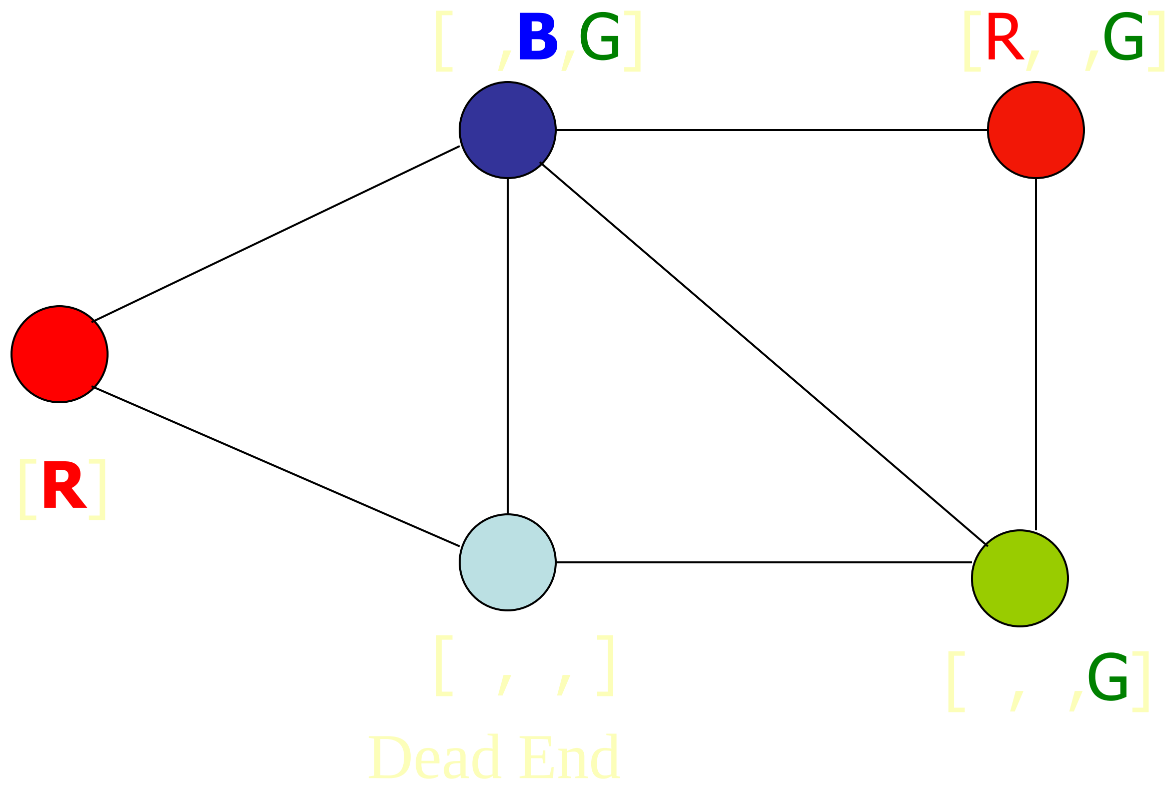
Forward Checking



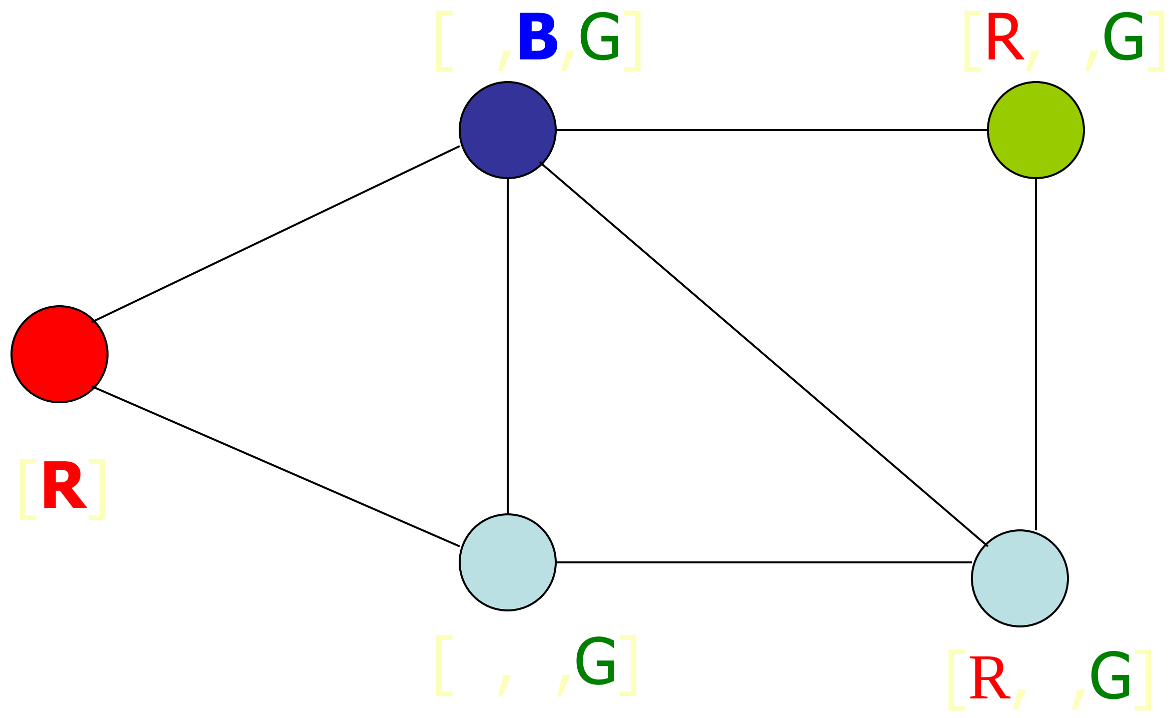
Forward Checking



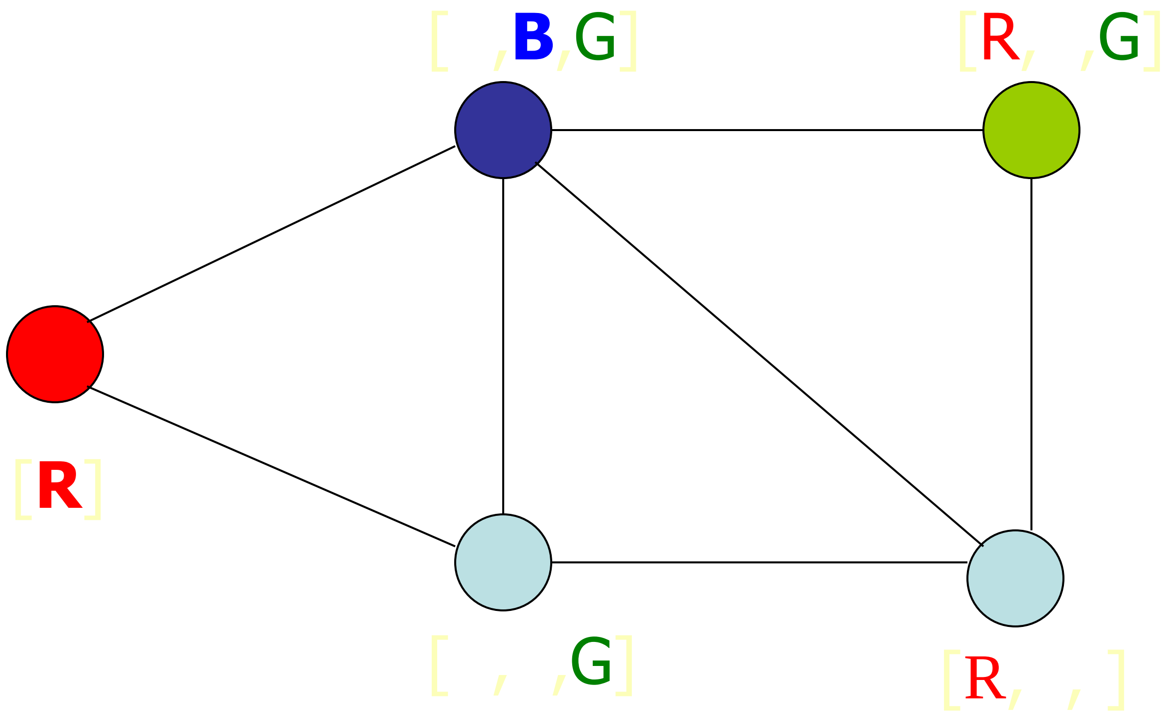
Forward Checking



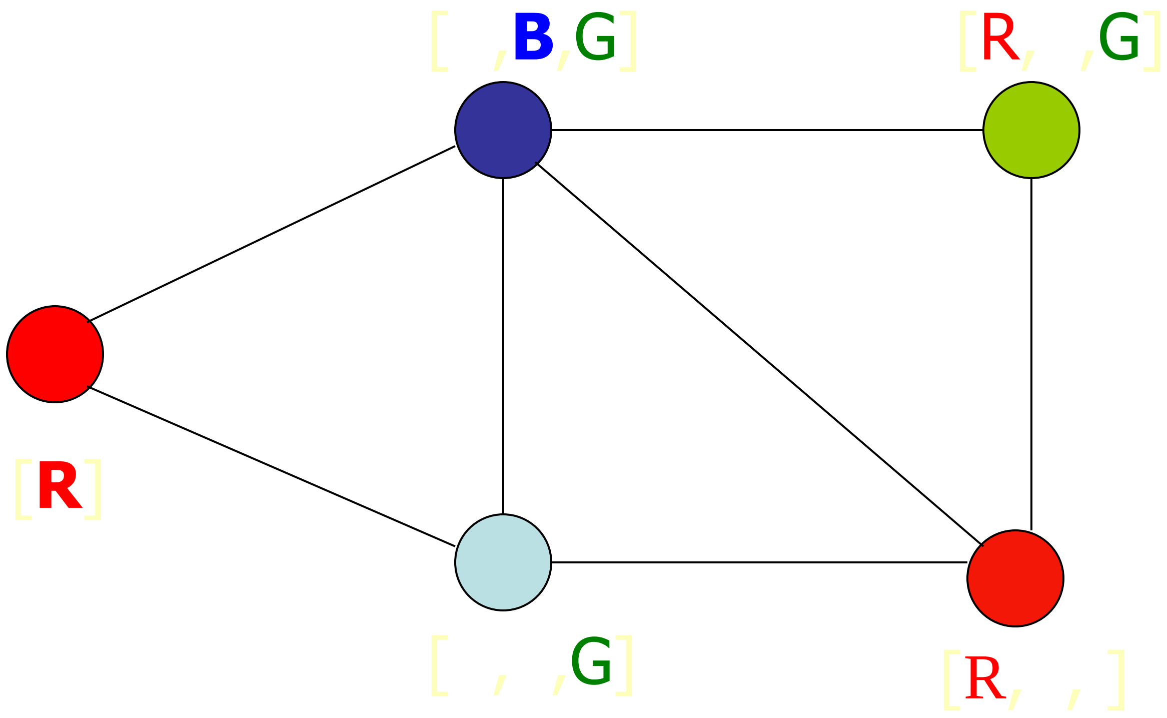
Forward Checking



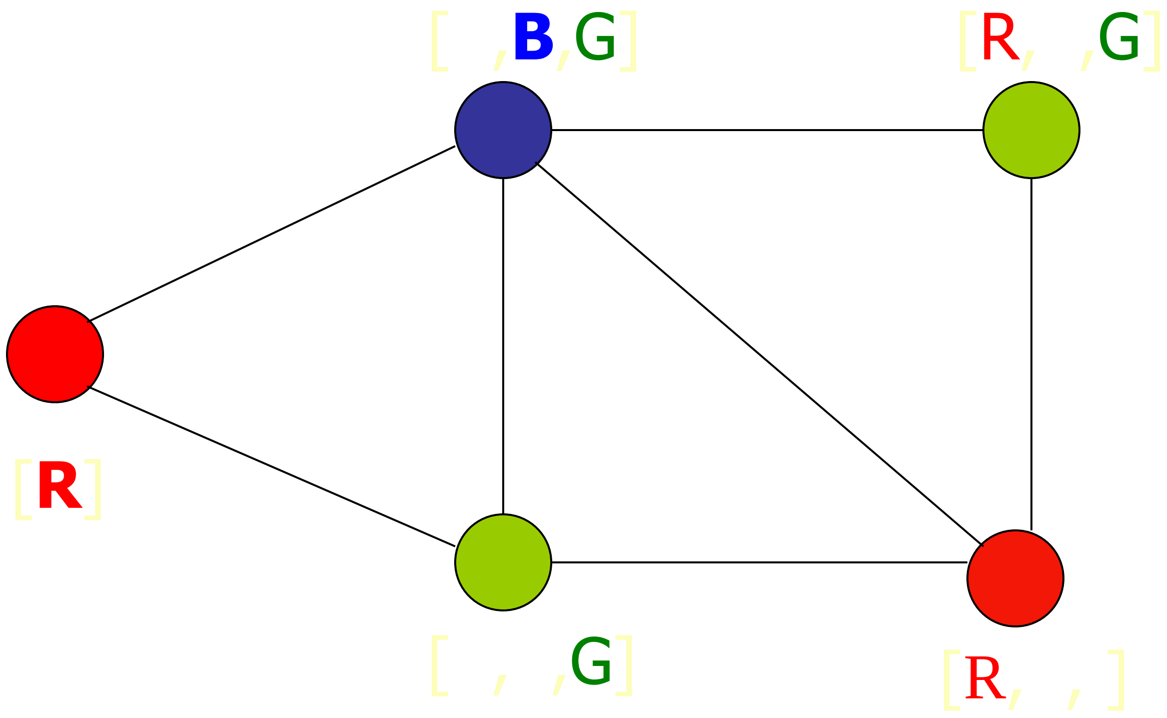
Forward Checking



Forward Checking

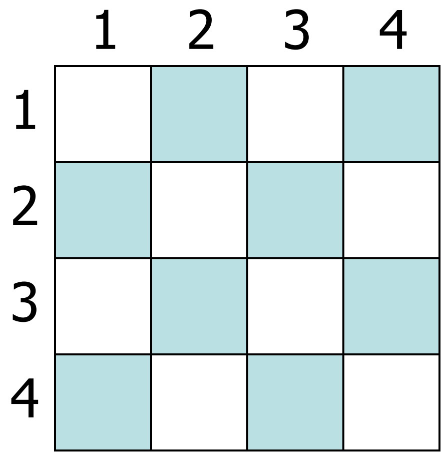


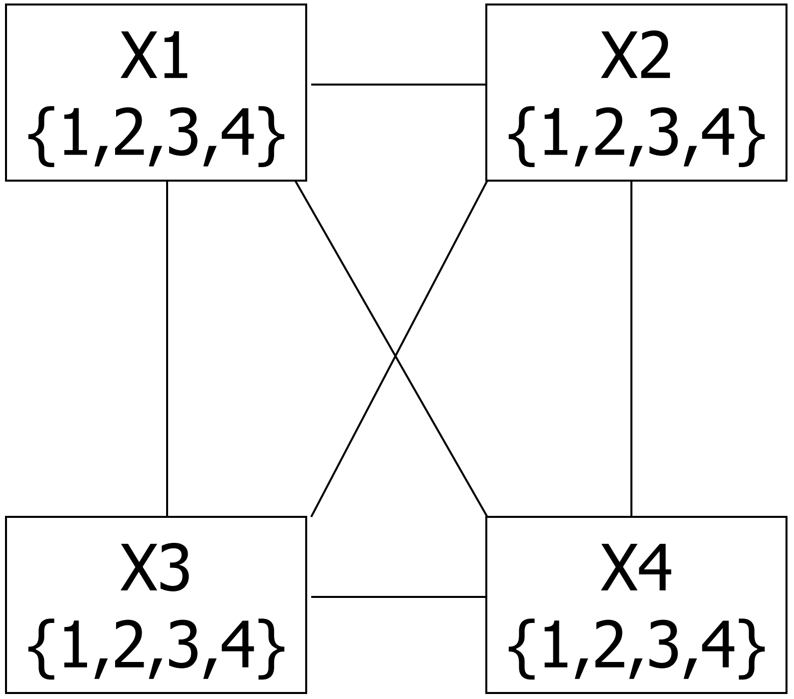
Forward Checking



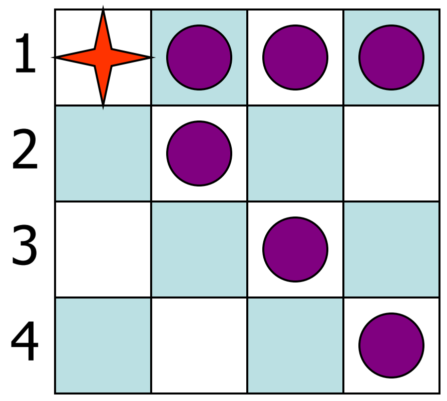
Solution !!!

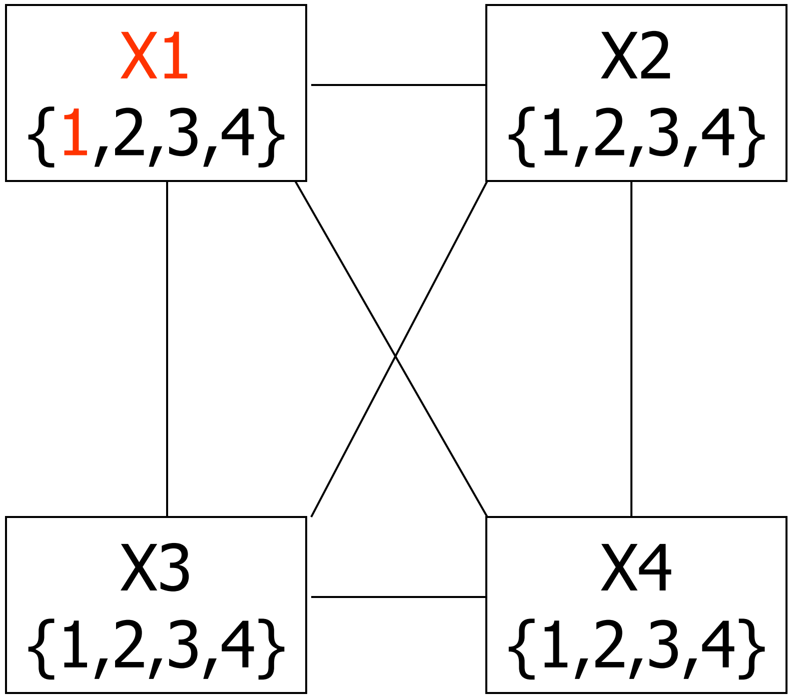
Example: 4-Queens Problem



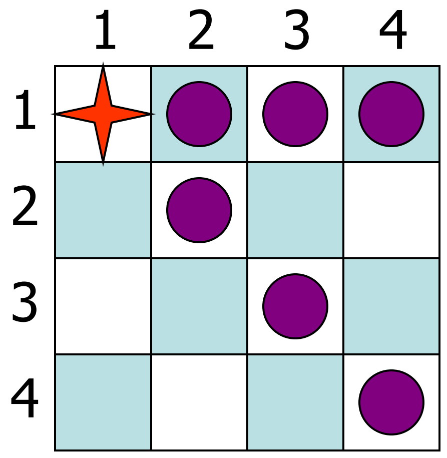


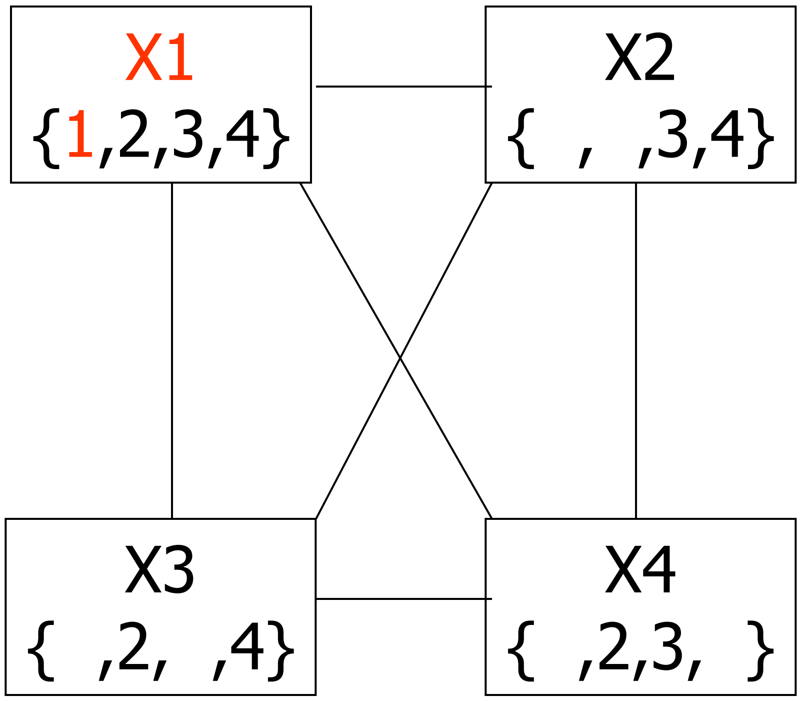
Example: 4-Queens Problem



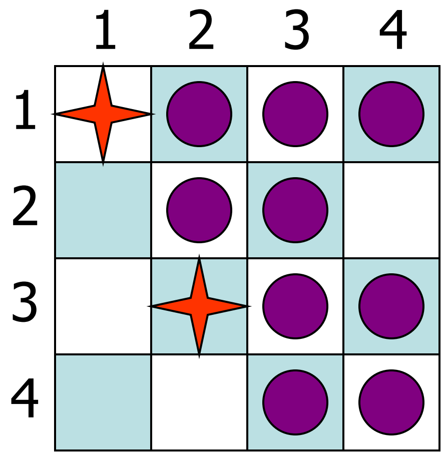


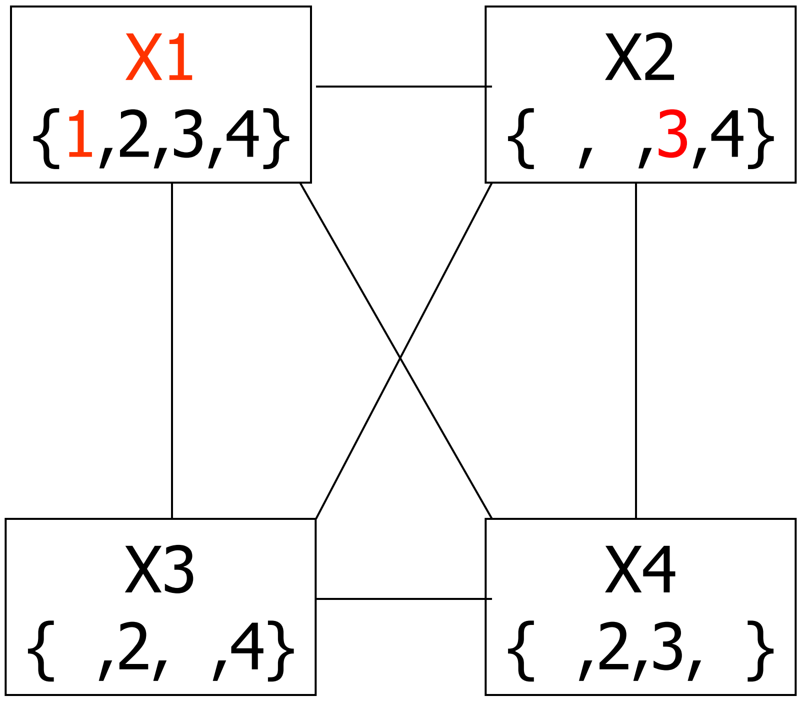
Example: 4-Queens Problem



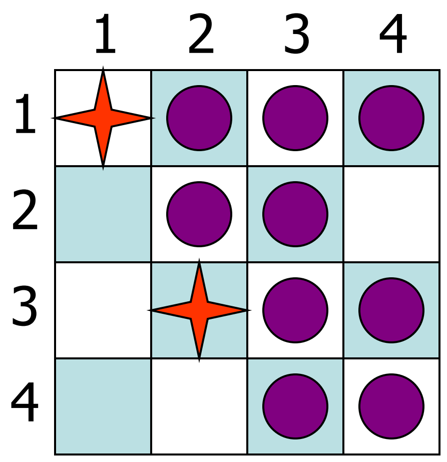


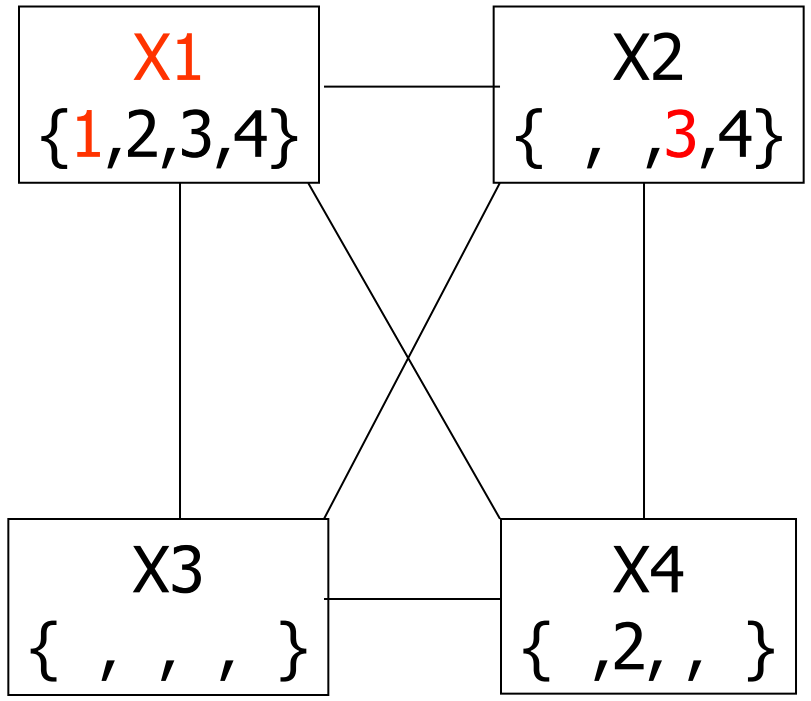
Example: 4-Queens Problem





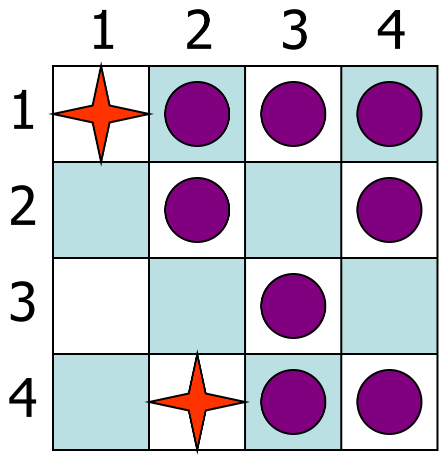
Example: 4-Queens Problem

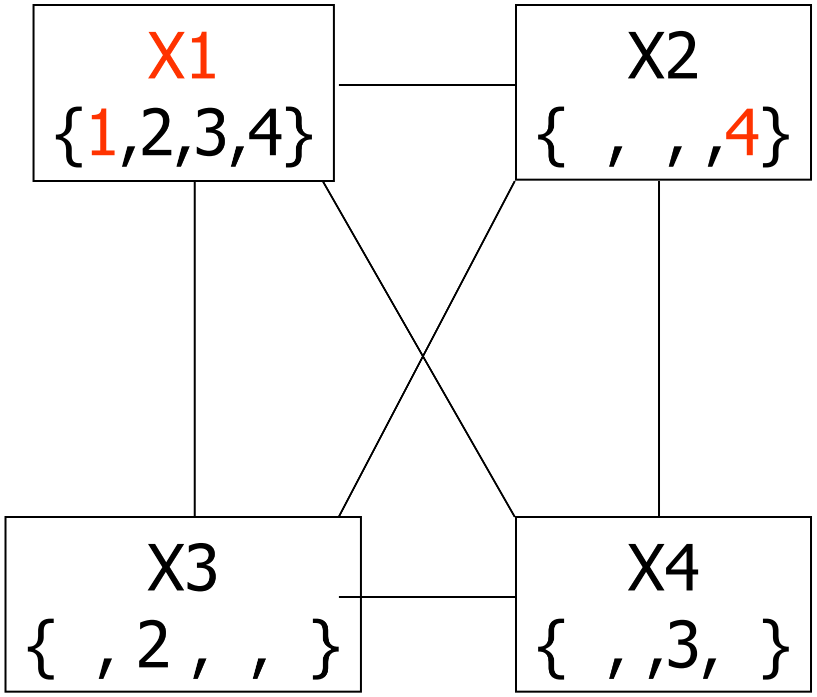




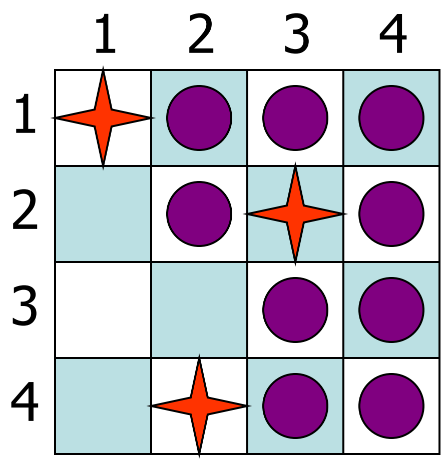
Dead End $\longrightarrow$ Backtrack

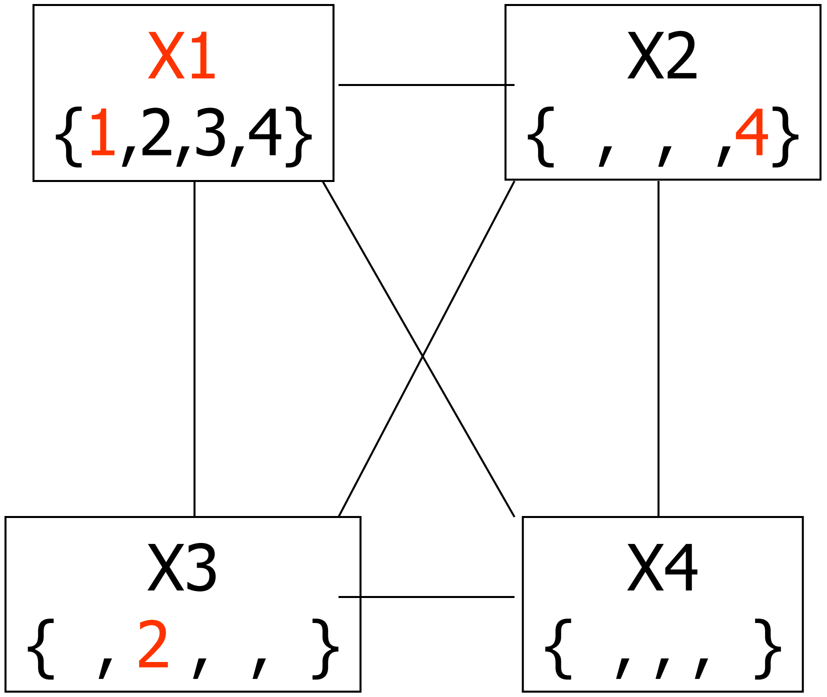
Example: 4-Queens Problem





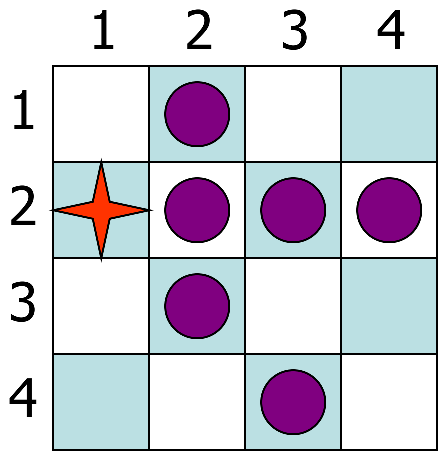
Example: 4-Queens Problem

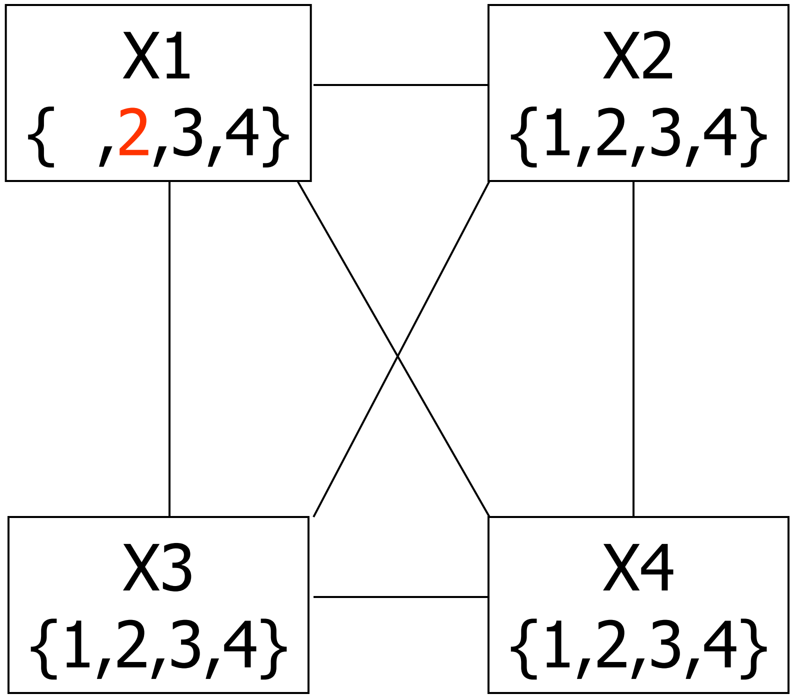




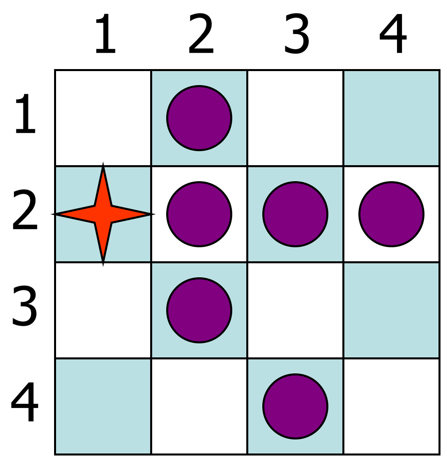
Dead End ⟶ Backtrack

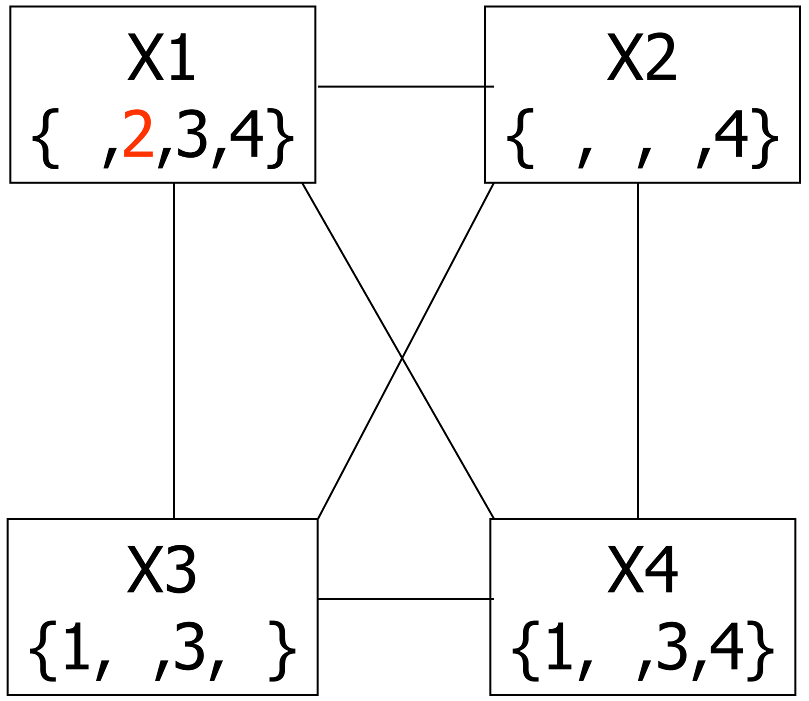
Example: 4-Queens Problem



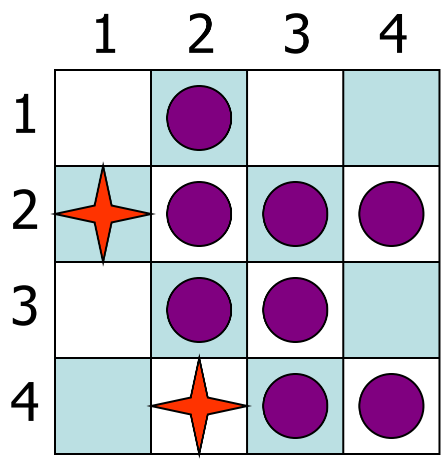


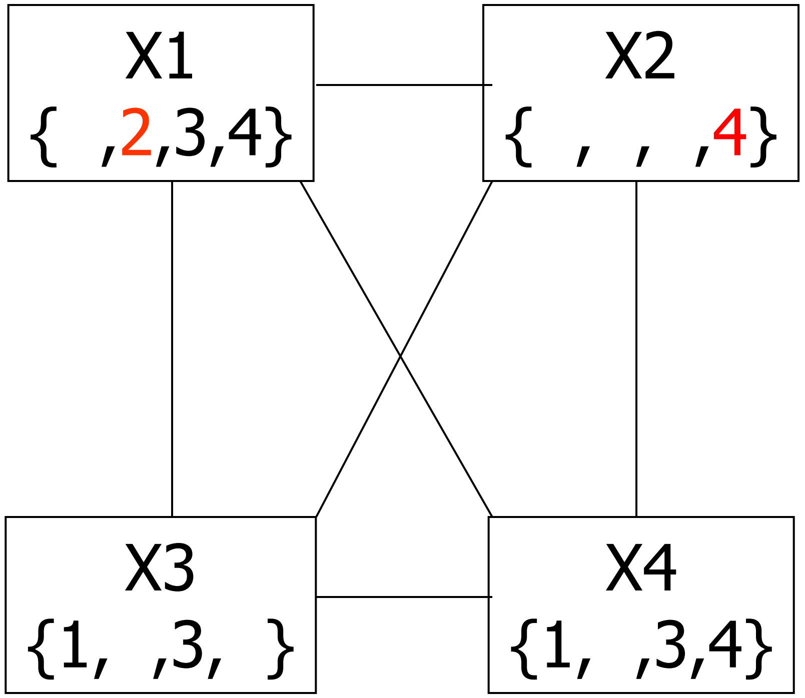
Example: 4-Queens Problem



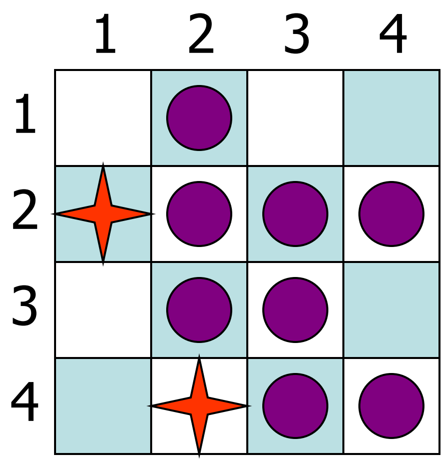


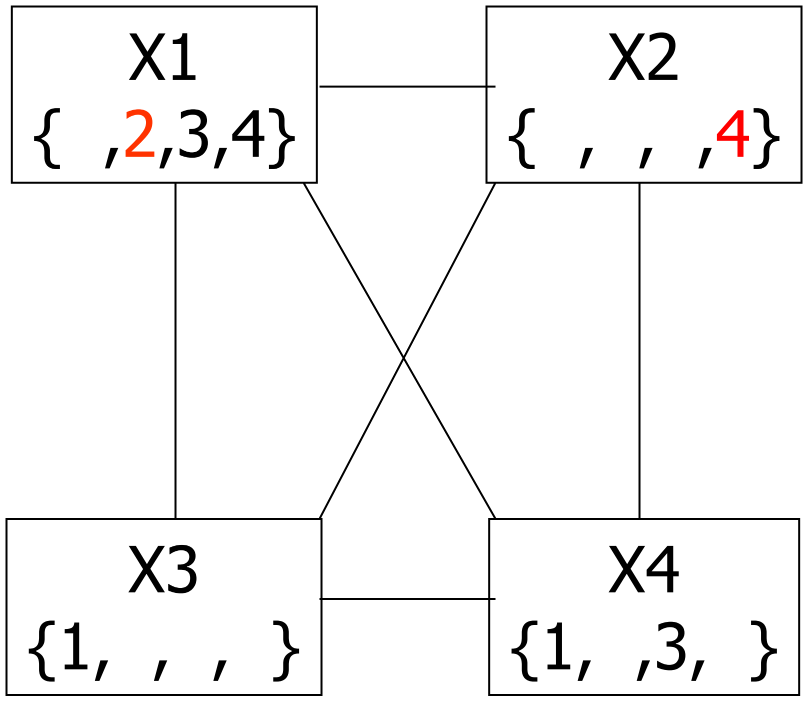
Example: 4-Queens Problem



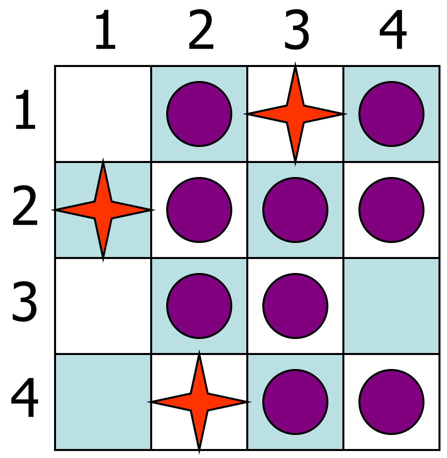


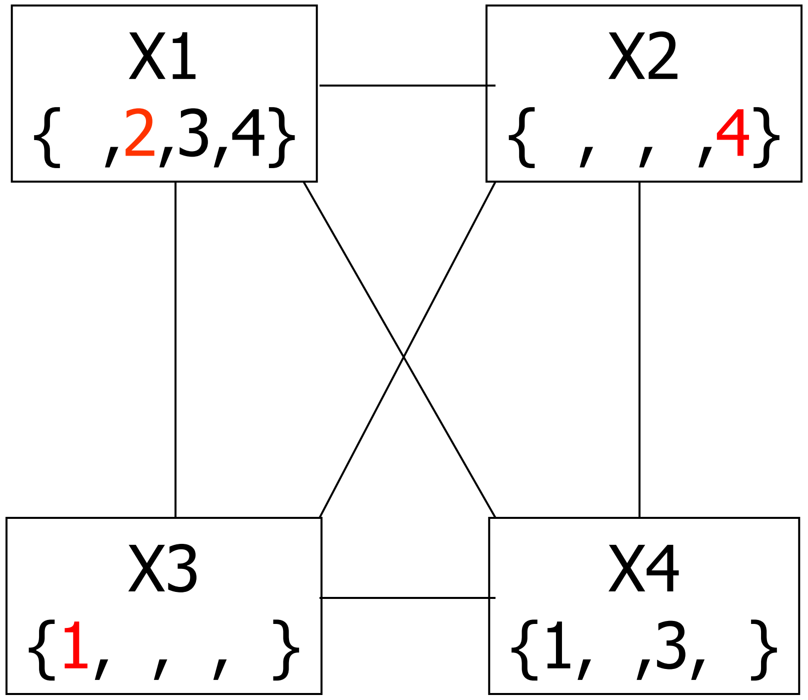
Example: 4-Queens Problem



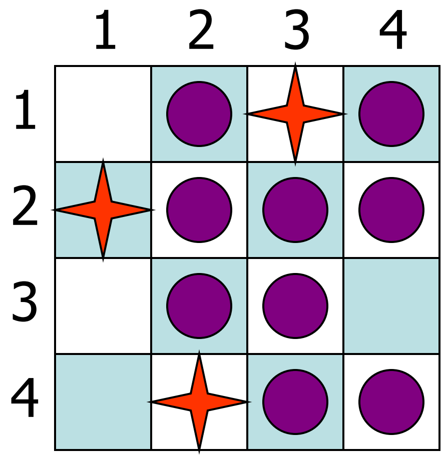


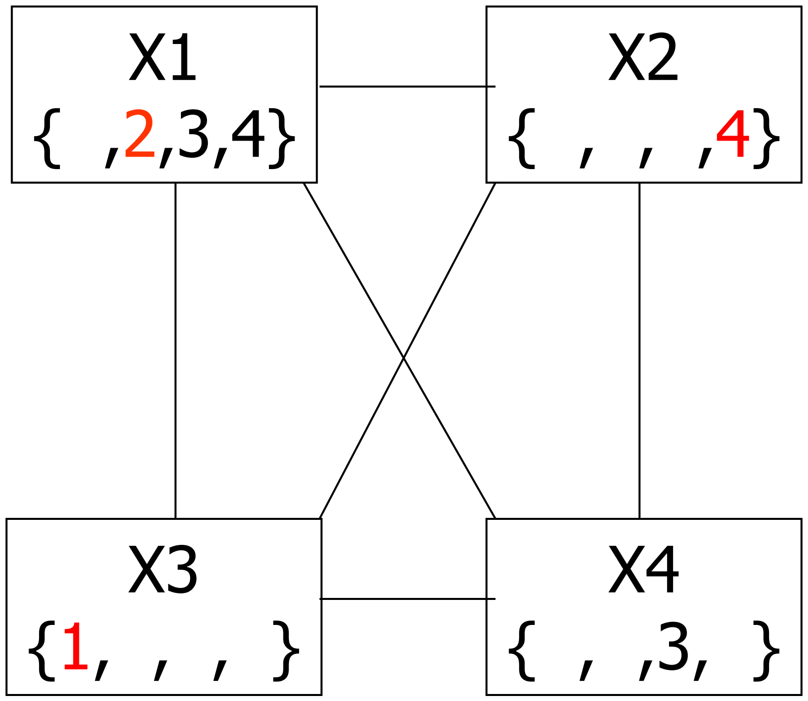
Example: 4-Queens Problem



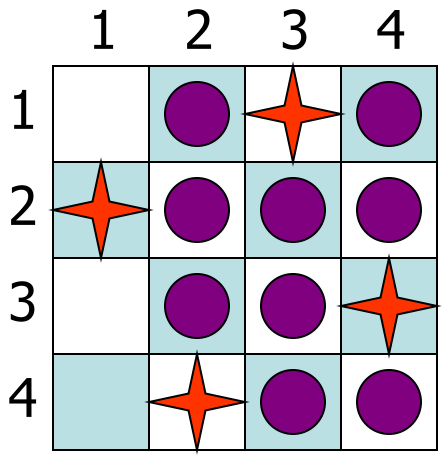


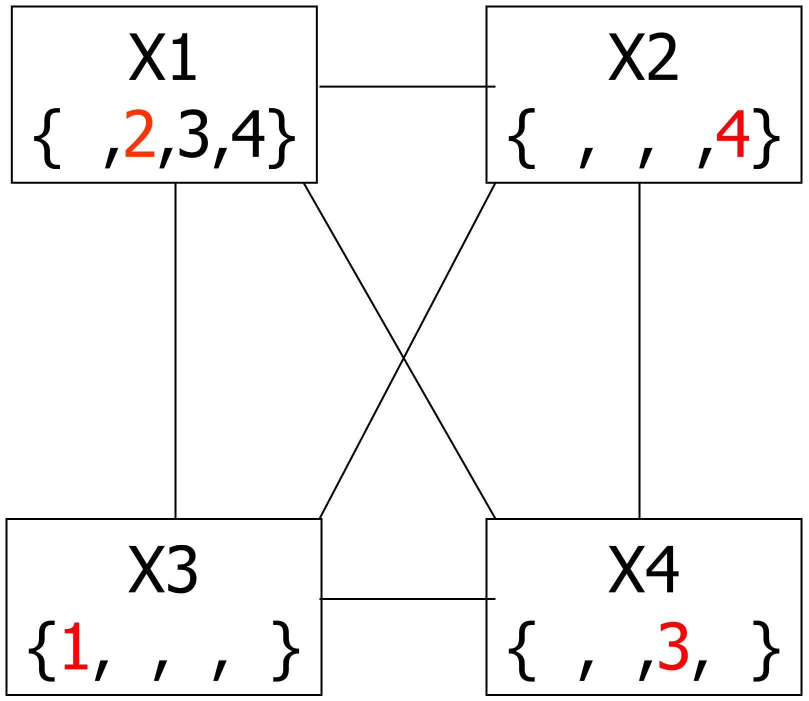
Example: 4-Queens Problem





Example: 4-Queens Problem

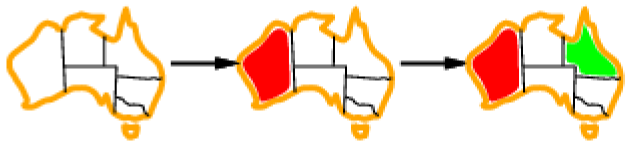




Solution !!!!

Constraint propagation

- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:



WA	NT	Q	NSW	V	SA	T
🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦
🟥	🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟩🟦	🟥🟩🟦
🟥	🟦	🟩	🟥🟦	🟥🟩🟦	🟦	🟥🟩🟦

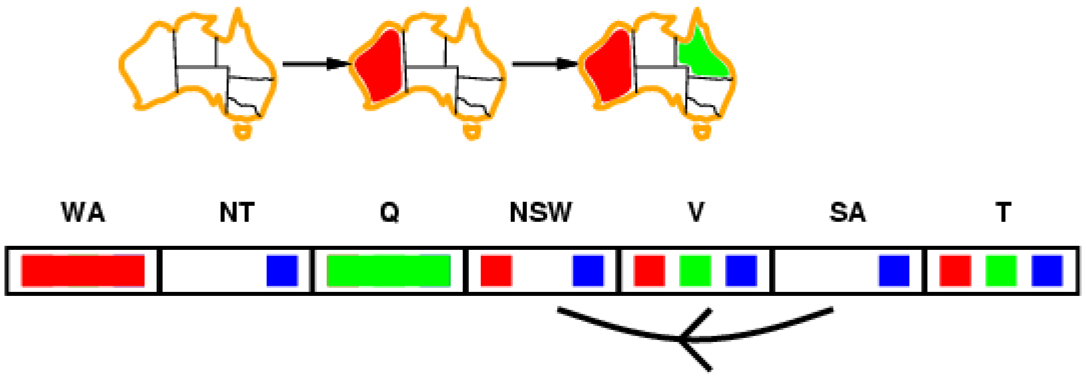
- NT and SA cannot both be blue!

- Constraint propagation repeatedly enforces constraints locally

Arc consistency

- Simplest form of propagation makes each arc consistent
- X →Y is consistent iff

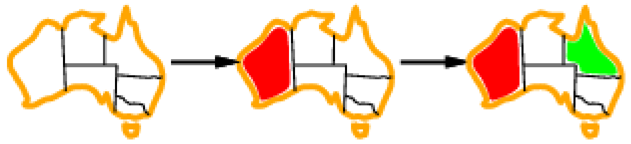
for every value x of X there is some allowed y

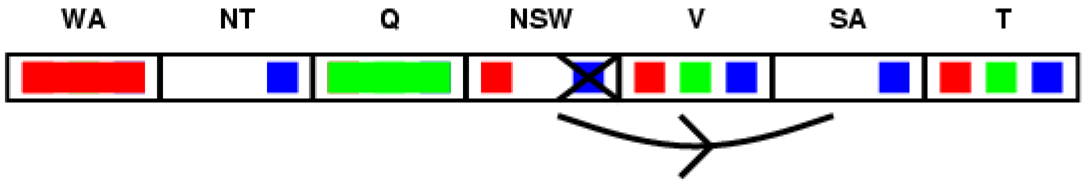


Arc consistency

- Simplest form of propagation makes each arc consistent
- X →Y is consistent iff

for every value x of X there is some allowed y

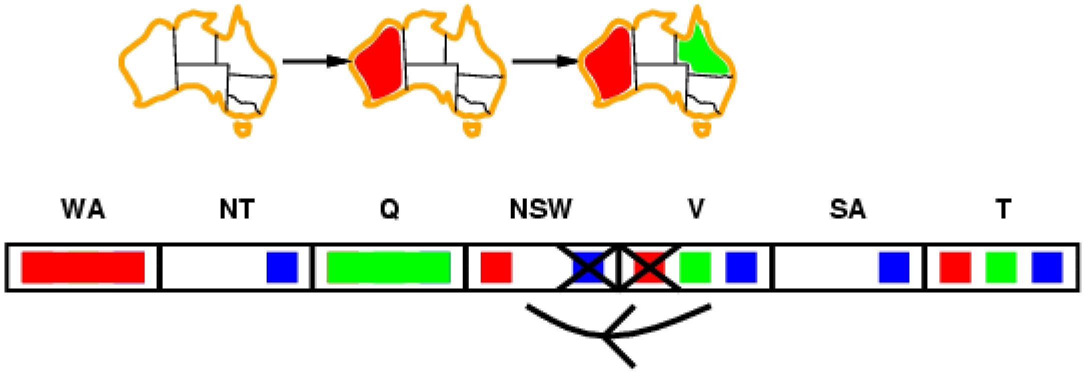




Arc consistency

- Simplest form of propagation makes each arc consistent
- X →Y is consistent iff

  for every value x of X there is some allowed y

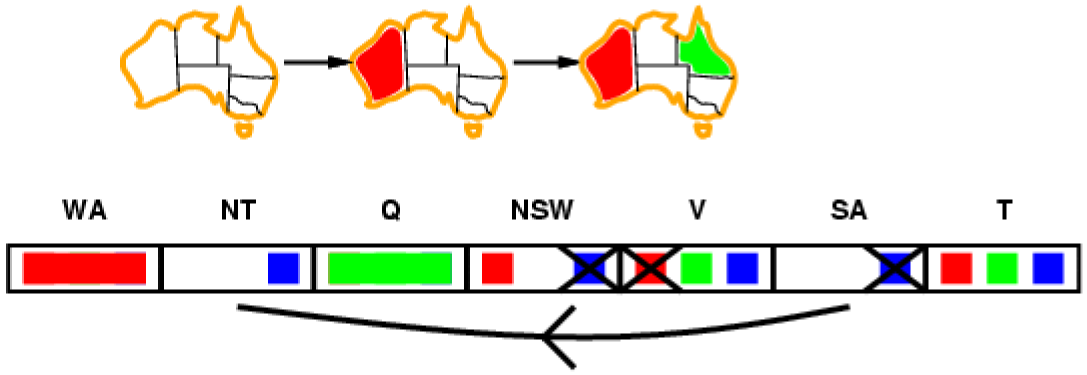


- If X loses a value, neighbors of X need to be rechecked

Arc consistency

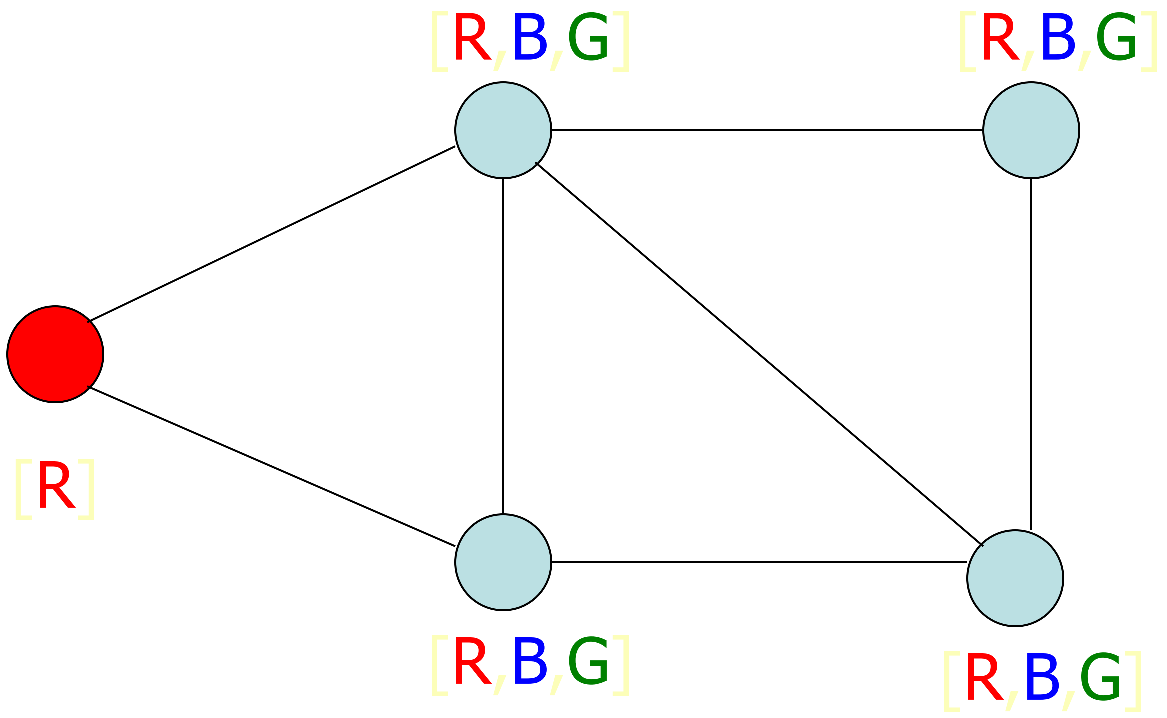
- Simplest form of propagation makes each arc consistent
- X →Y is consistent iff

  for every value x of X there is some allowed y

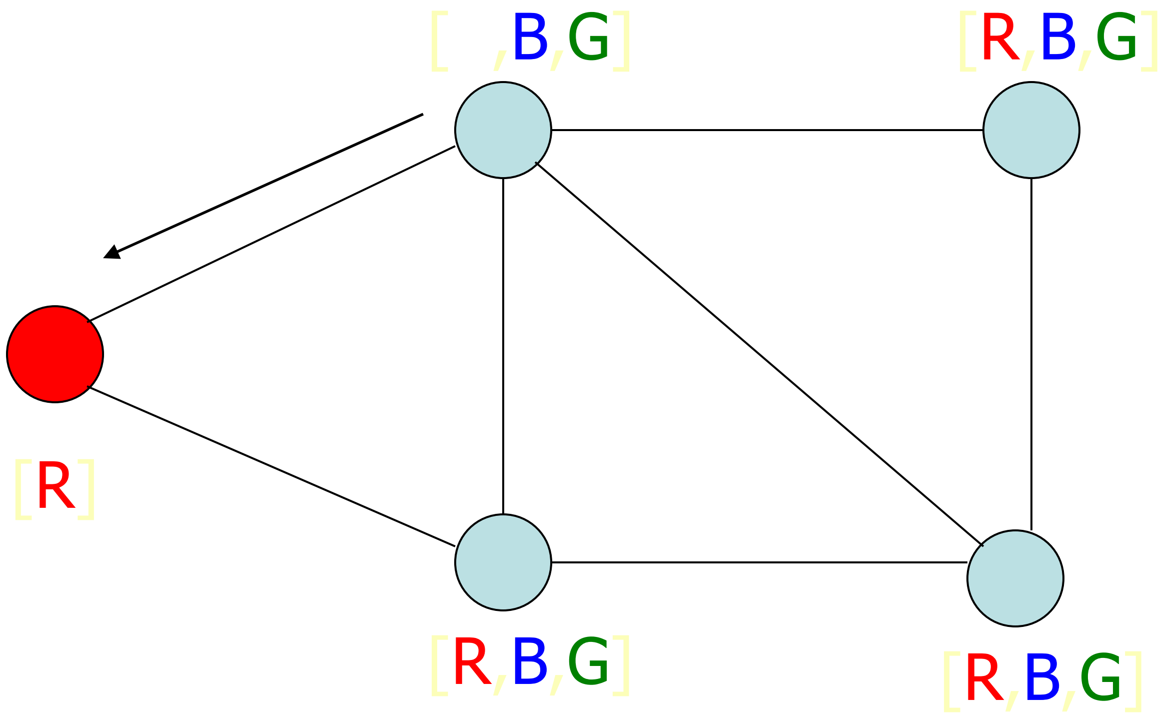


- If X loses a value, neighbors of X need to be rechecked
- Arc consistency detects failure earlier than forward checking
- Can be run as a preprocessor or after each assignment

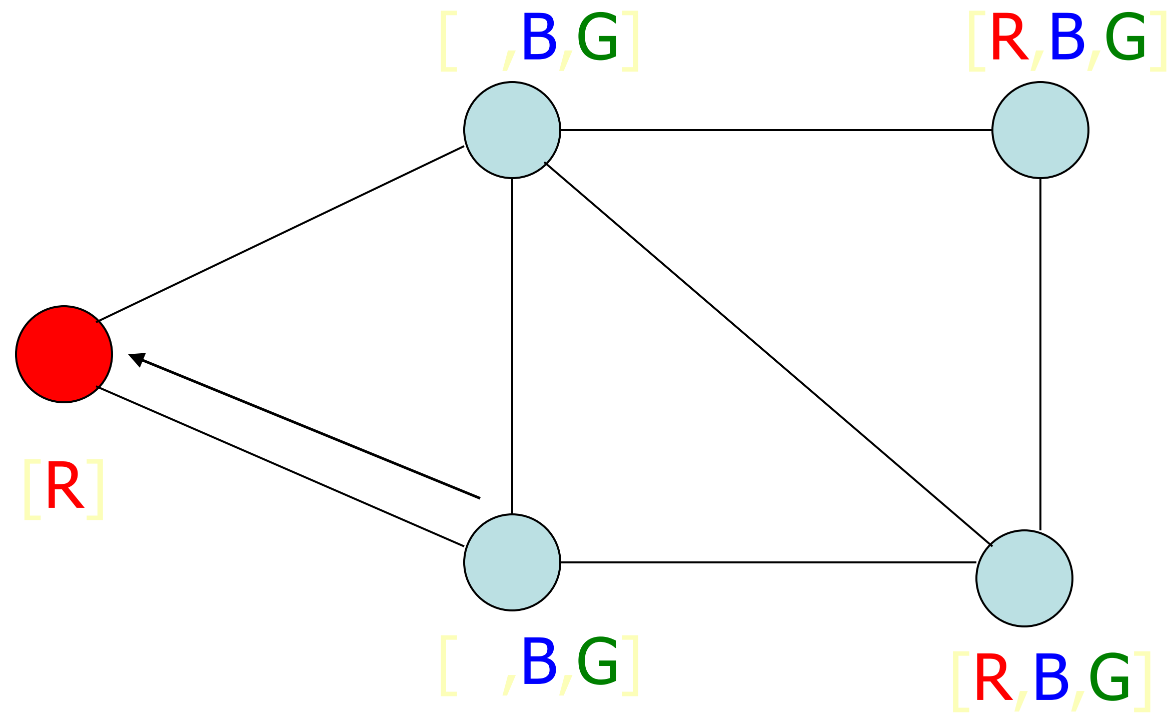
Arc Consistency: AC3



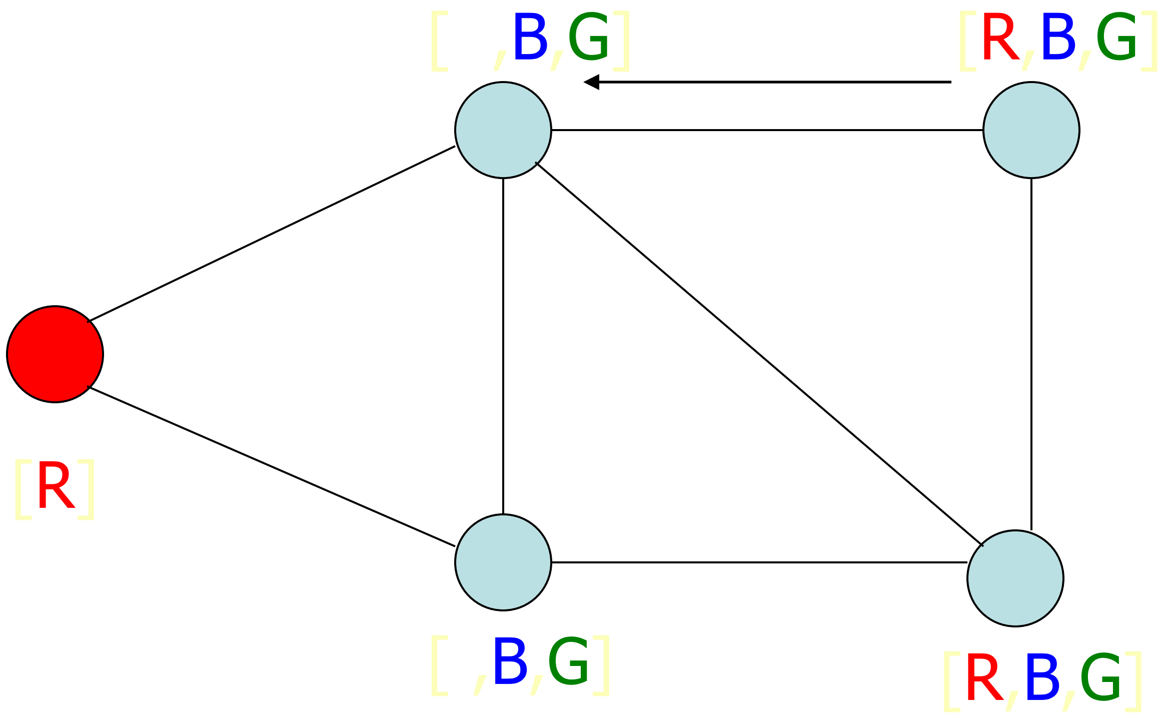
Arc Consistency: AC3



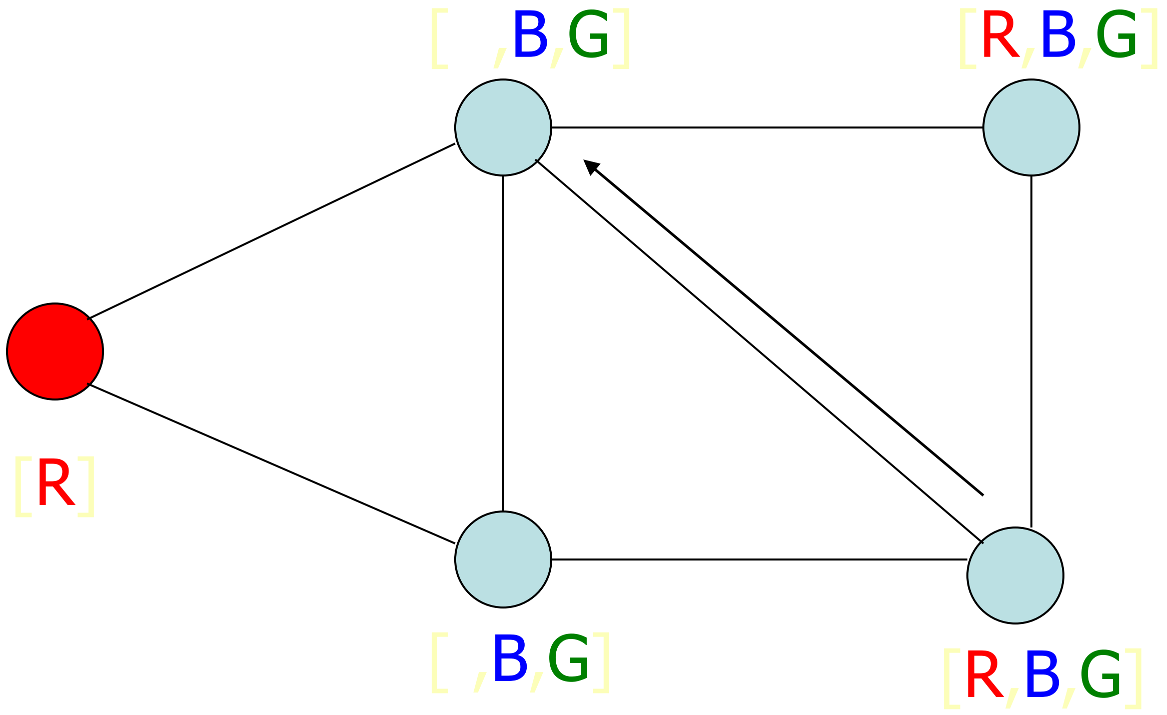
Arc Consistency: AC3



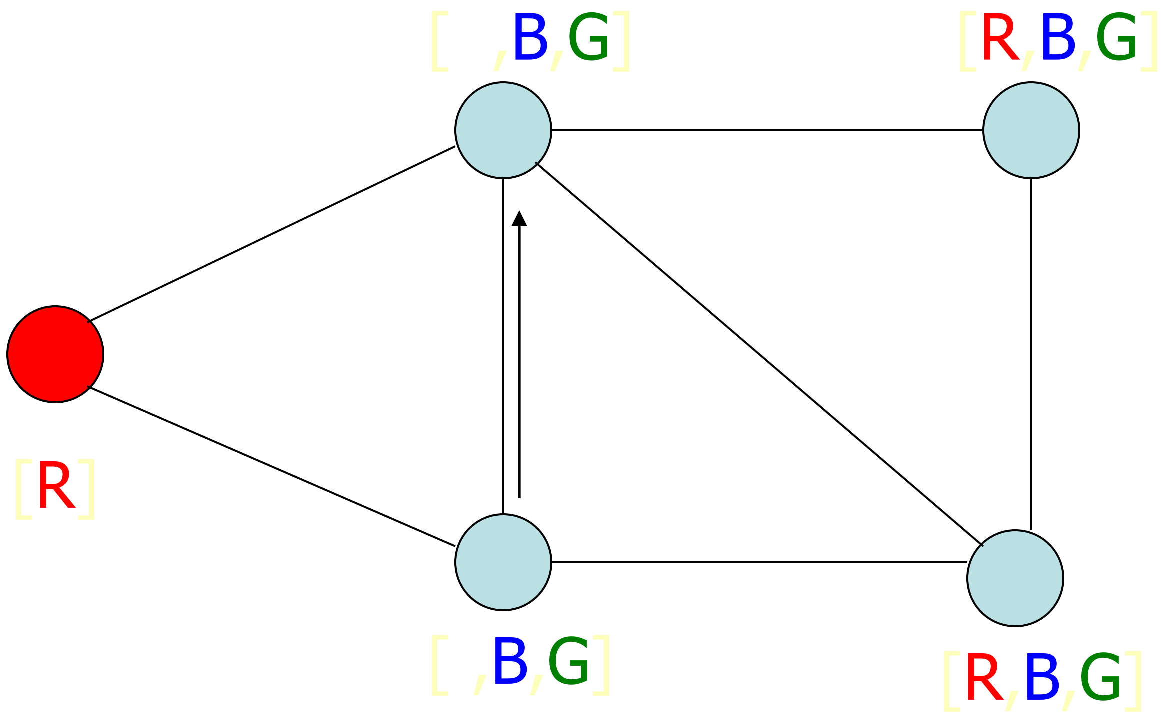
Arc Consistency: AC3



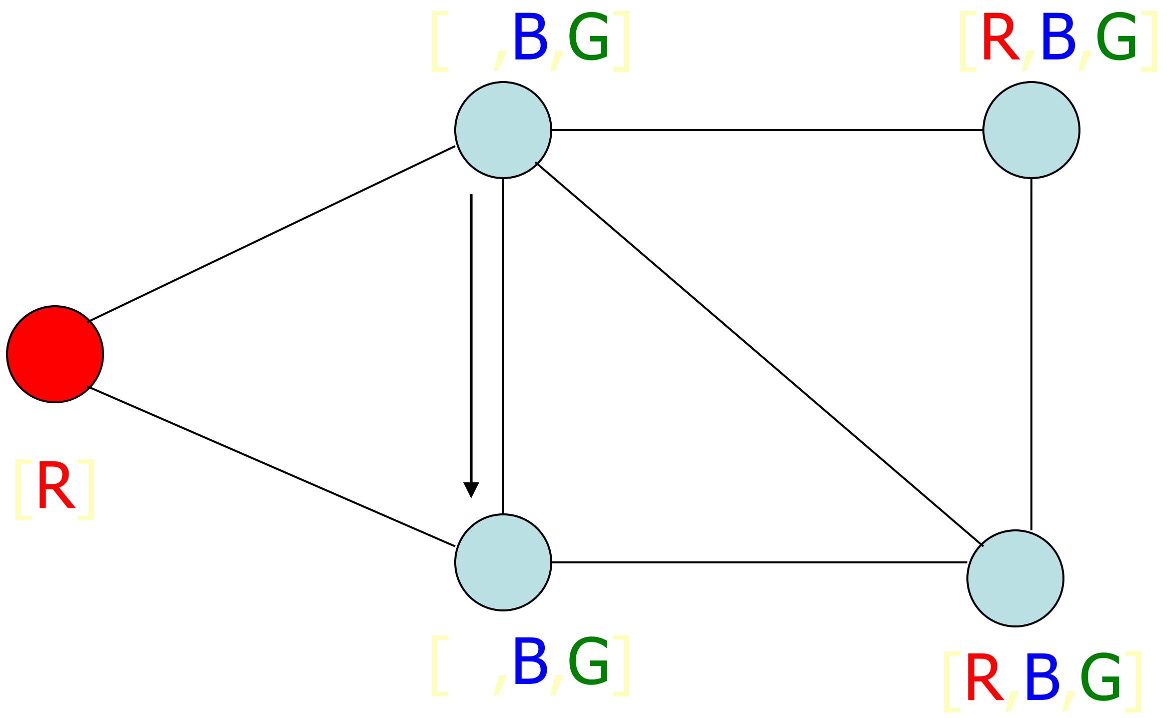
Arc Consistency: AC3



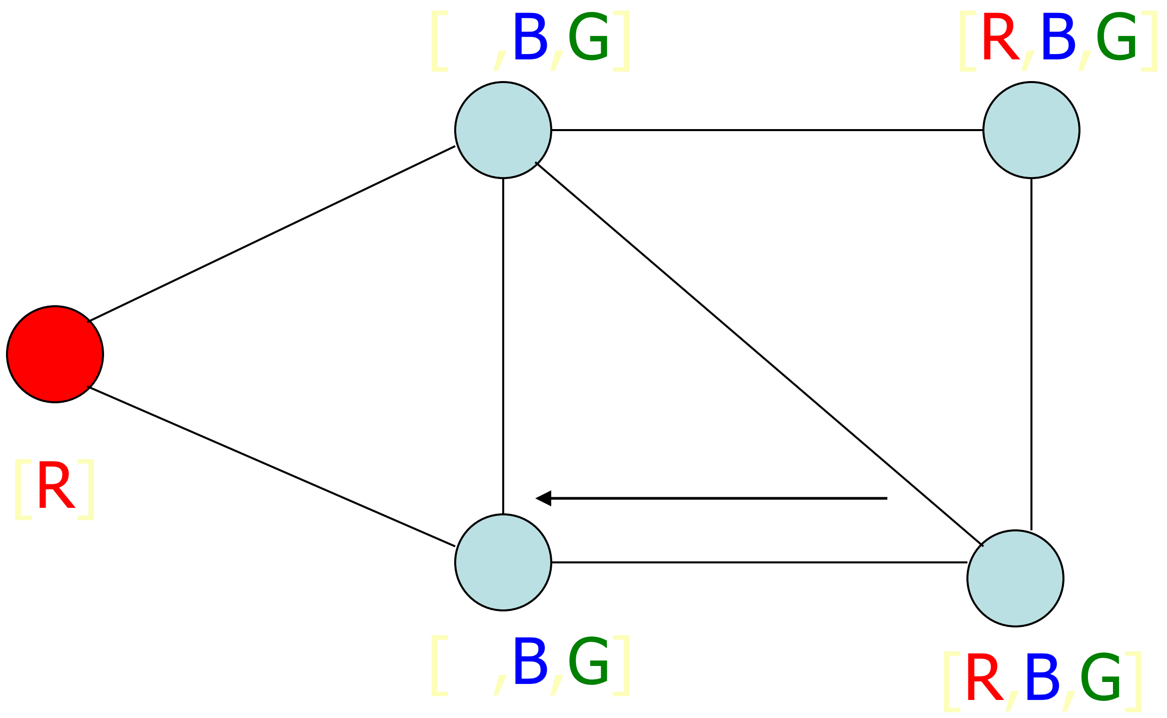
Arc Consistency: AC3



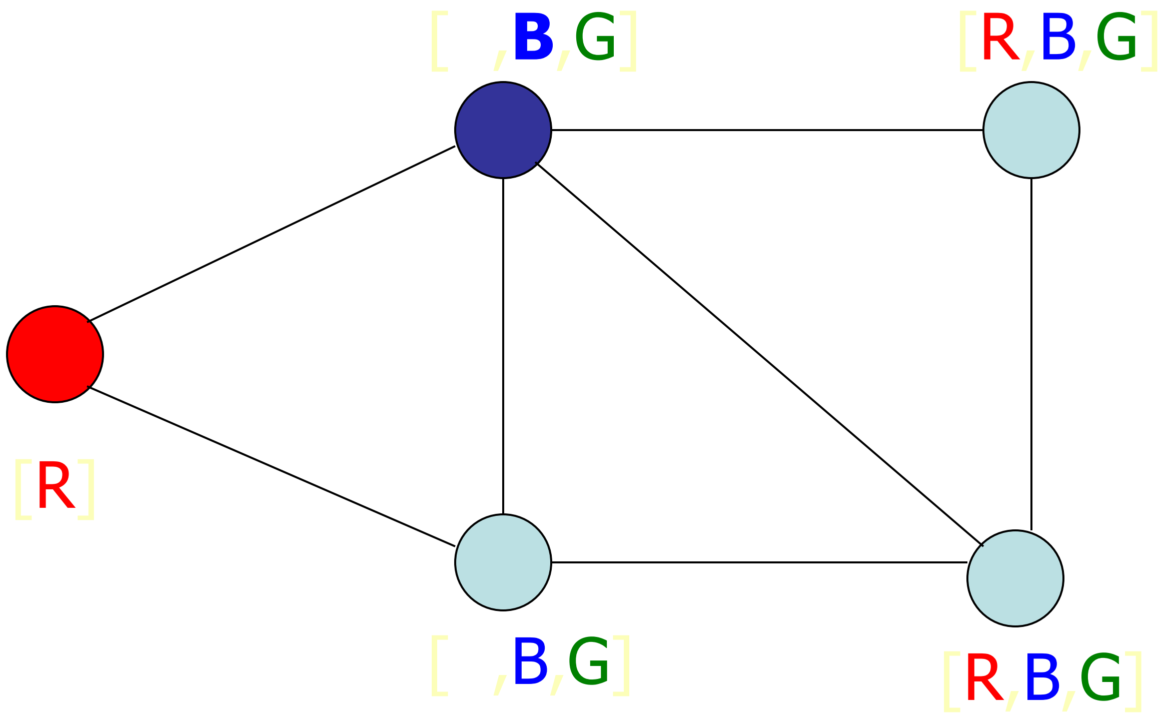
Arc Consistency: AC3



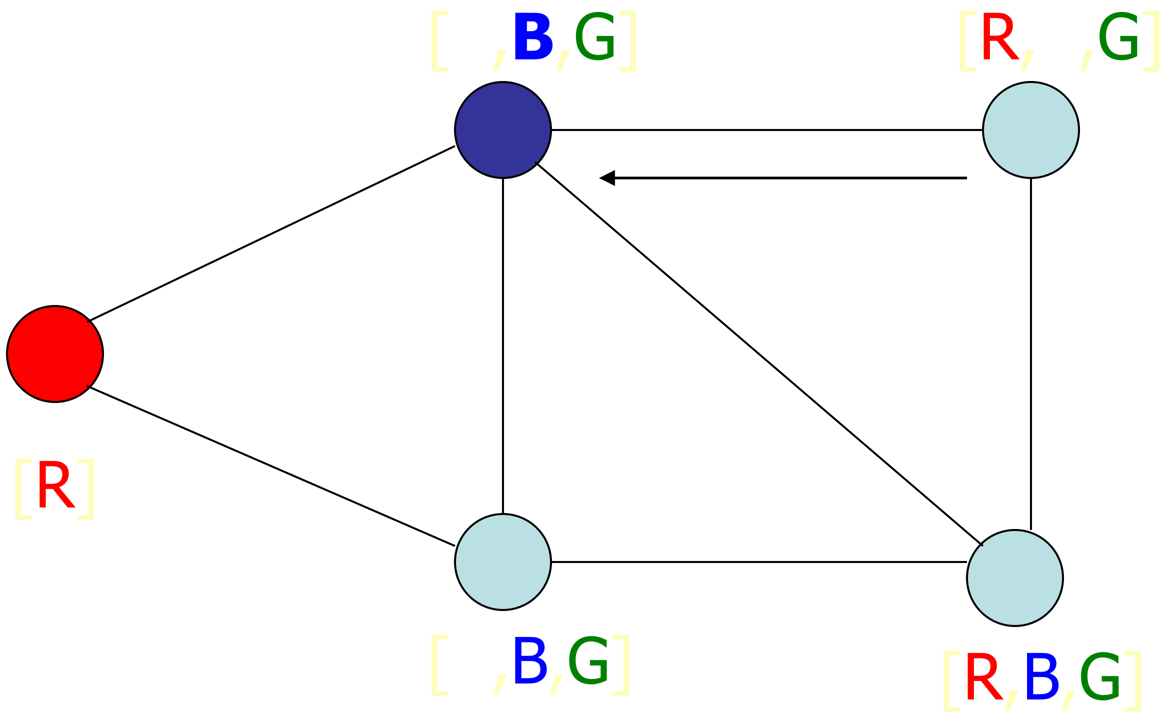
Arc Consistency: AC3



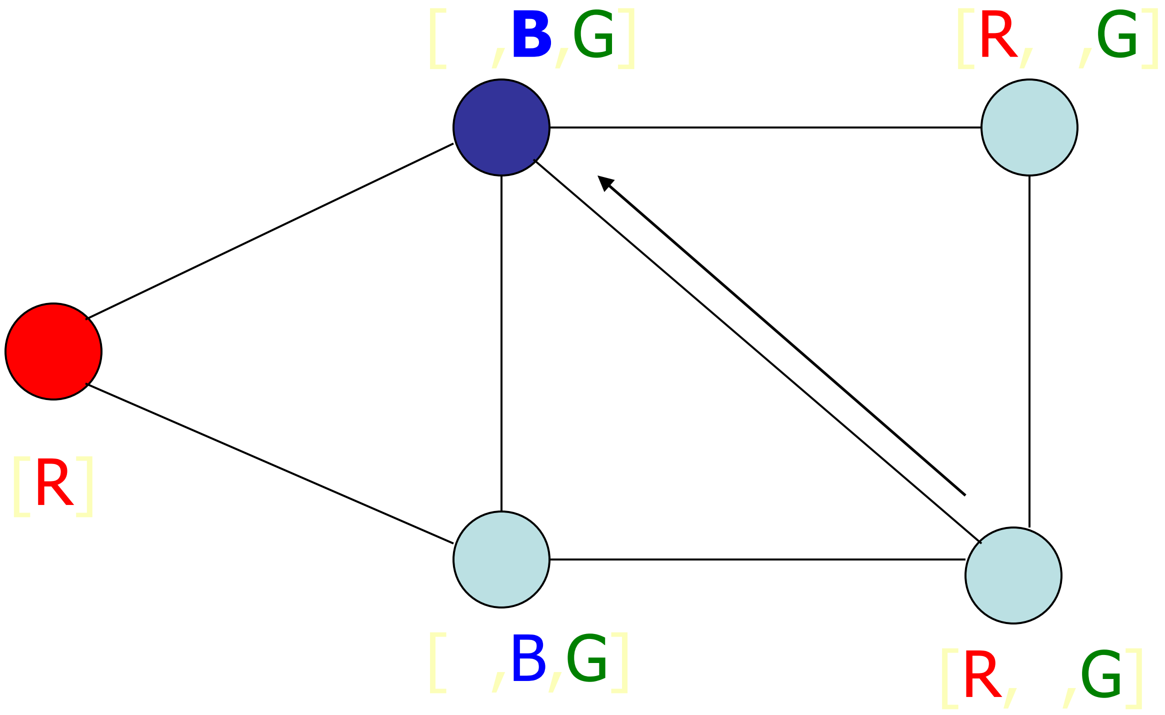
Arc Consistency: AC3



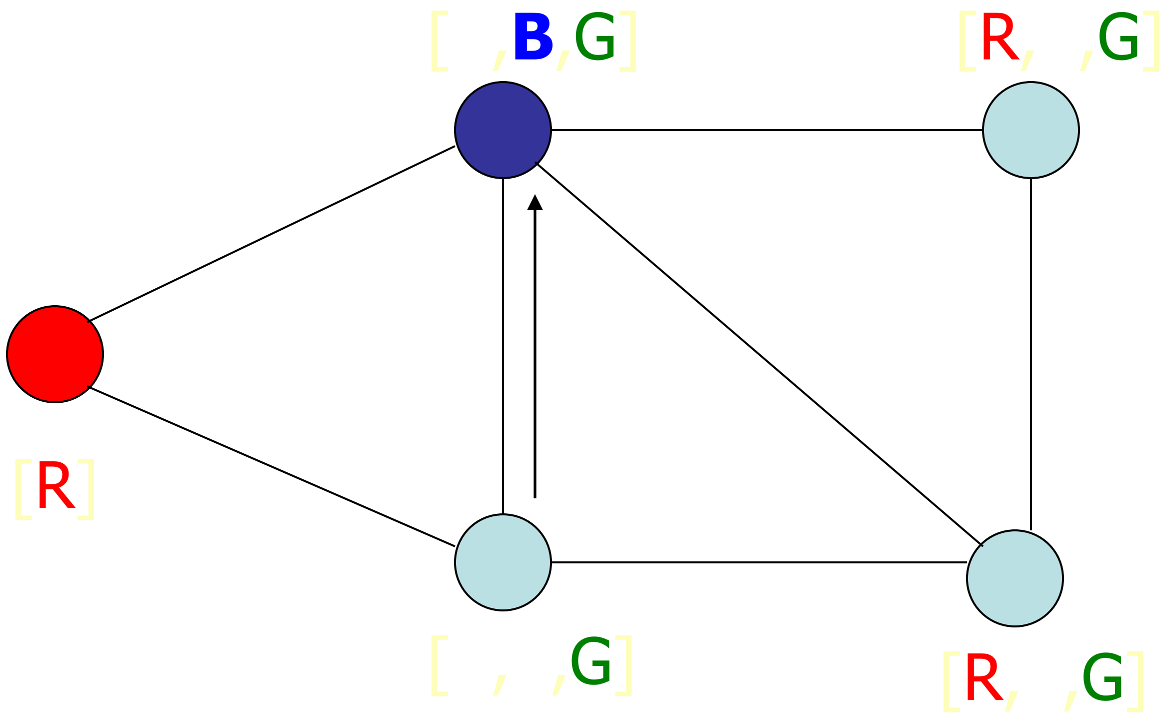
Arc Consistency: AC3



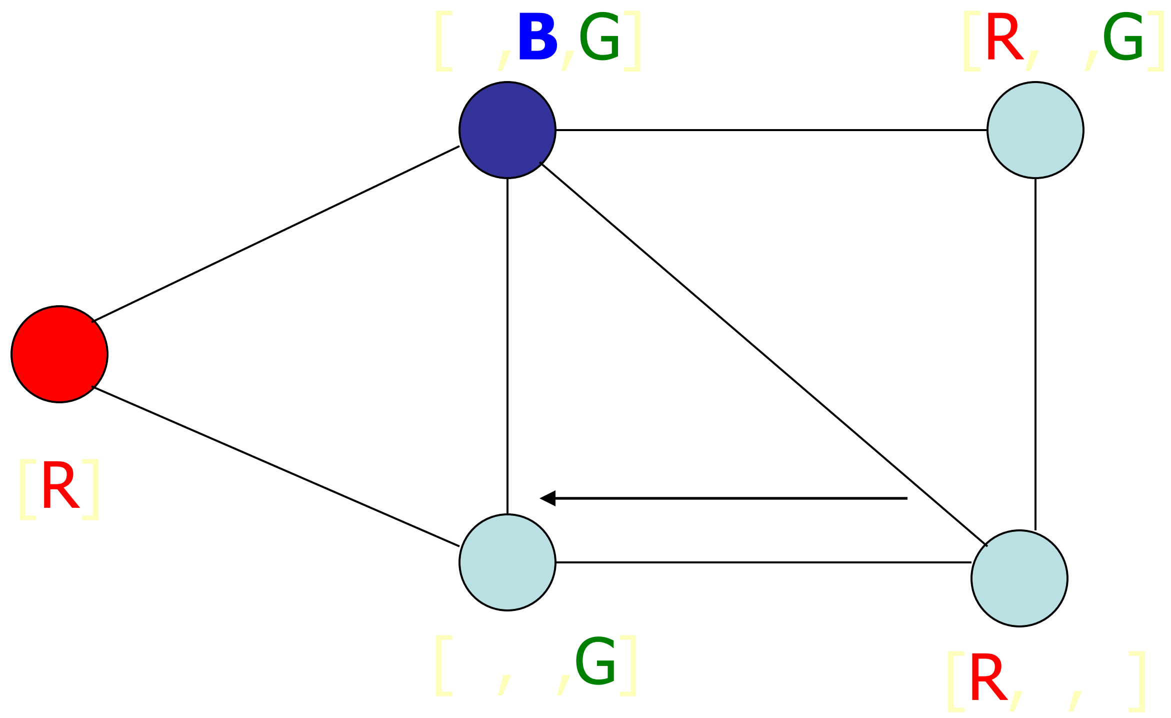
Arc Consistency: AC3



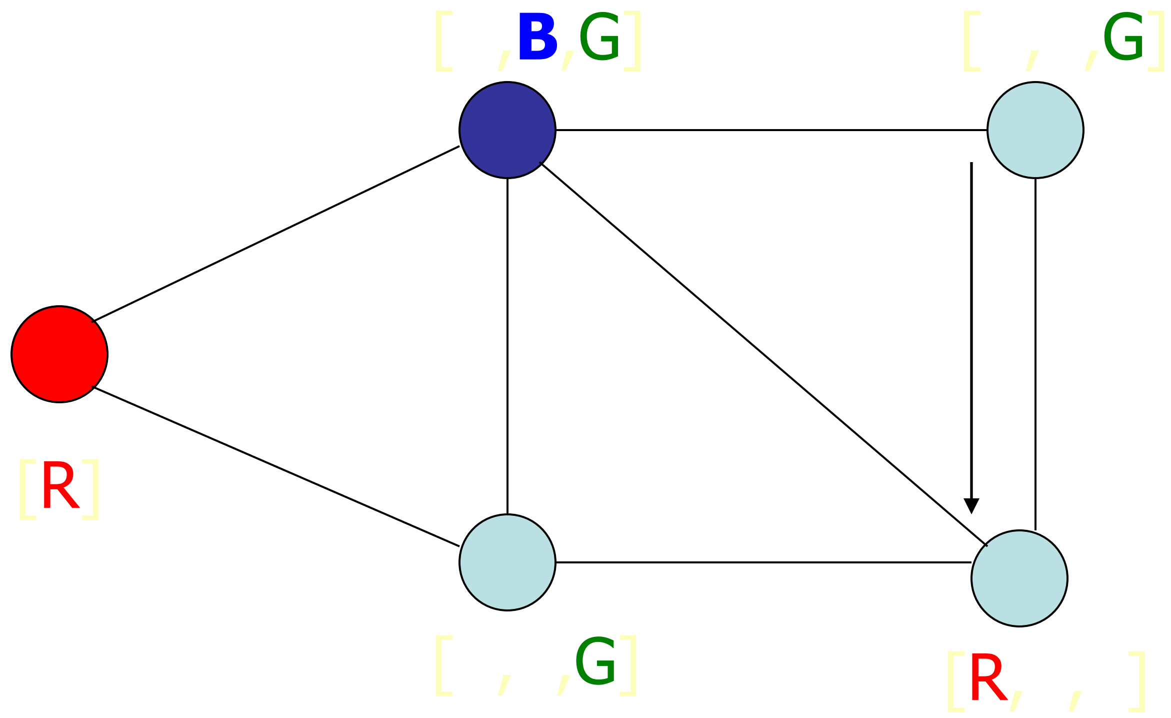
Arc Consistency: AC3



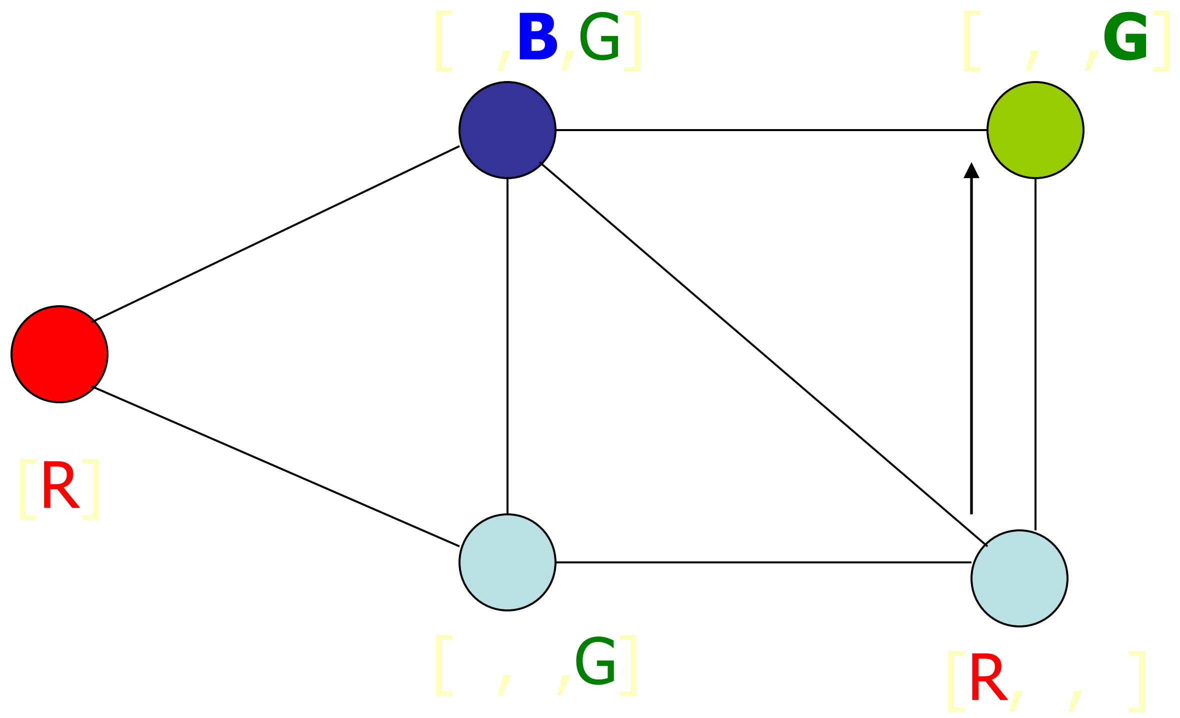
Arc Consistency: AC3



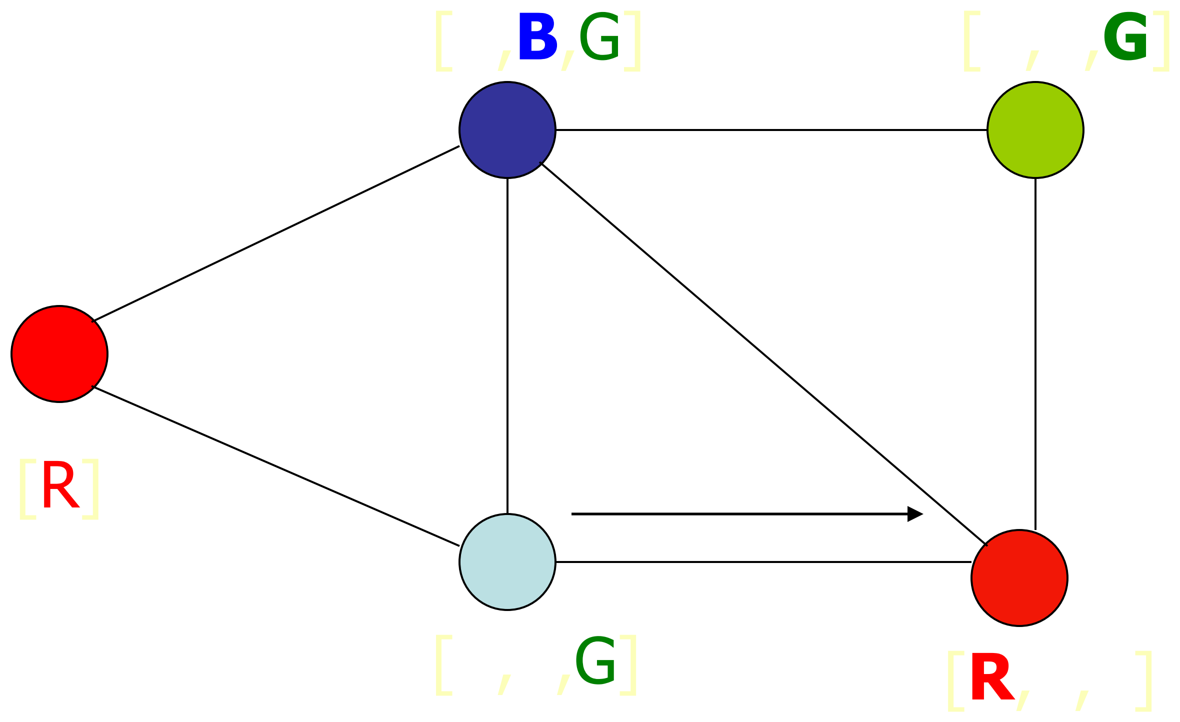
Arc Consistency: AC3



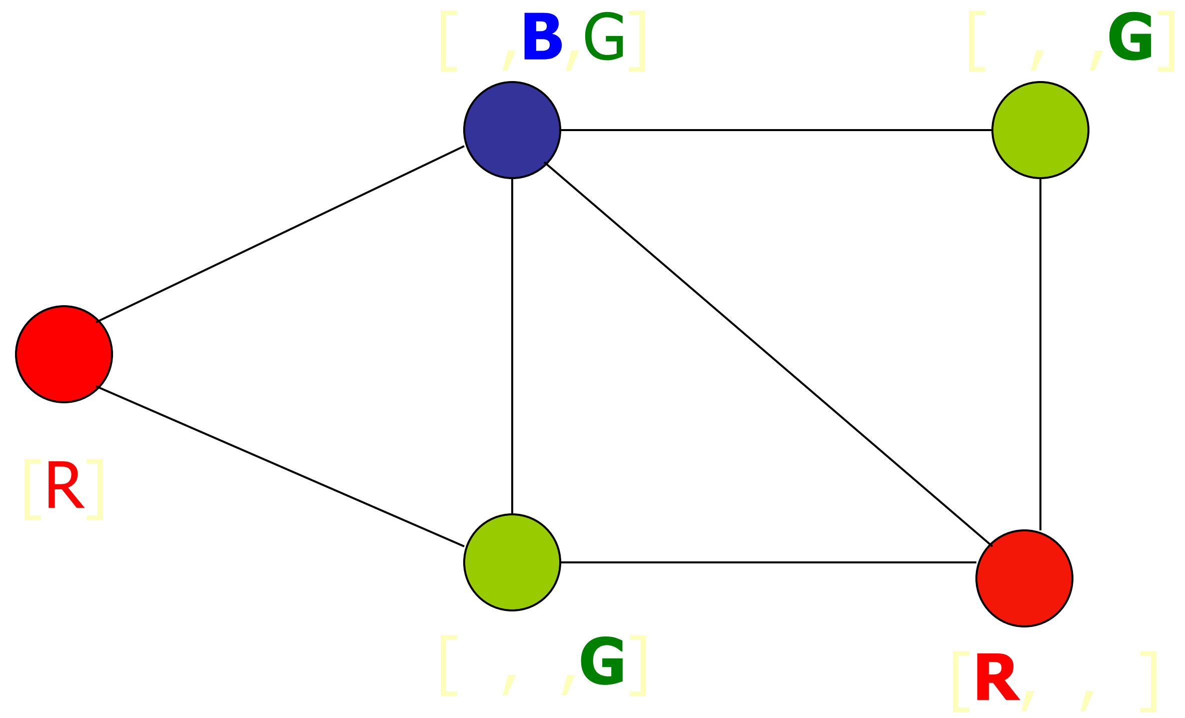
Arc Consistency: AC3



Arc Consistency: AC3



Arc Consistency: AC3



Solution !!!

Local Search and CSP

- local search (iterative improvement) is frequently used for constraint satisfaction problems
  - values are assigned to all variables
  - modification operators move the configuration towards a solution
- often called heuristic repair methods
  - repair inconsistencies in the current configuration
- simple strategy: min-conflicts
  - minimizes the number of conflicts with other variables
  - solves many problems very quickly
    - million-queens problem in less than 50 steps
- can be run as *online* algorithm
  - use the current state as new initial state

Local search for CSPs

- Hill-climbing, simulated annealing typically work with "complete" states, i.e., all variables assigned

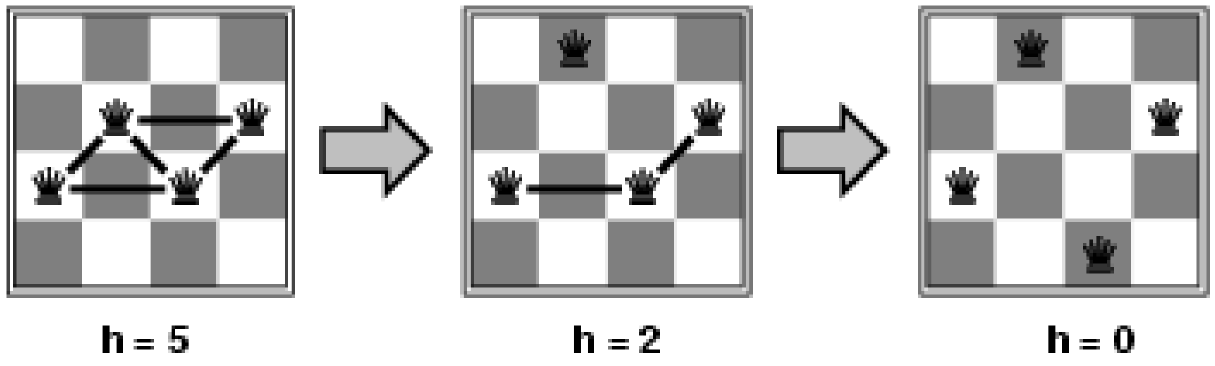
- To apply to CSPs:
  - allow states with unsatisfied constraints
  - operators reassign variable values

- Variable selection: randomly select any conflicted variable

- Value selection by min-conflicts heuristic:
  - choose value that violates the fewest constraints
  - i.e., hill-climb with $h(n)$ = total number of violated constraints

Example: 4-Queens

- States: 4 queens in 4 columns ($4^4 = 256$ states)
- Actions: move queen in column
- Goal test: no attacks
- Evaluation: $h(n)$ = number of attacks



- Given random initial state, can solve n-queens in almost constant time for arbitrary n with high probability (e.g., n = 10,000,000)