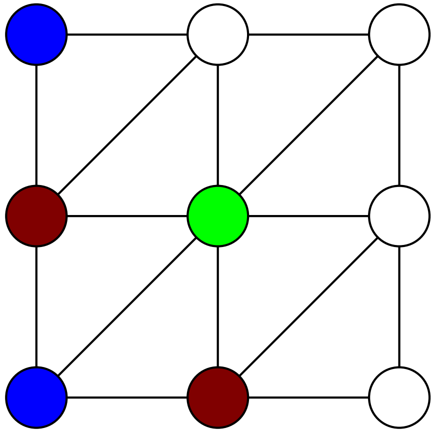
Knowledge Representation and Reasoning: Solving Constraint Satisfaction Problems

Chapter 6

Constraint Satisfaction Problems



◇ Binary constraint network $\gamma = \langle V, D, C \rangle$

— V a finite set of variables $v_1, \ldots, v_n$

— D a set of [finite] sets $D_{v_1}, \ldots, D_{v_n}$

— C a set of binary relations $\{C_{u,v} \mid u, v \in V, u \neq v\}$

$C_{u,v} \subseteq D_u \times D_v$

Outline of the lecture

- ◇ Recall constraint networks and backtracking search
- ◇ Tightening CSPs by learning from mistakes
- ◇ Problem structure: constraint graphs
- ◇ Symmetry
- ◇ CSPs and optimisation
- ◇ Summary

Recall

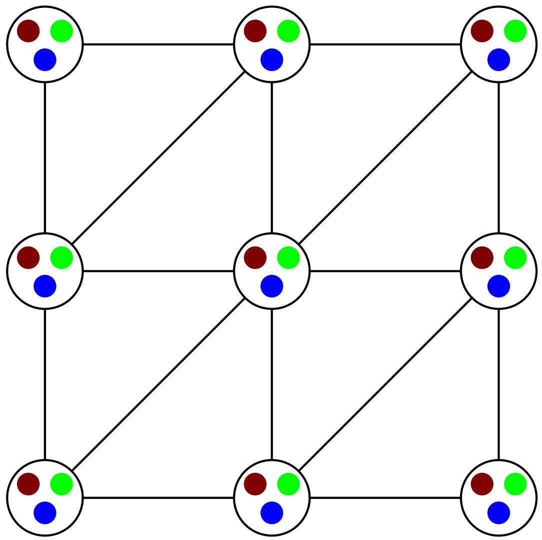
◊ Pure backtracking
— If the current partial assignment is consistent
— Choose a variable, assign each value from its domain in turn
— Search the resulting sub-tree

◊ Forward checking
— Prune values from neighbour variables if they are not supported by the assigned one

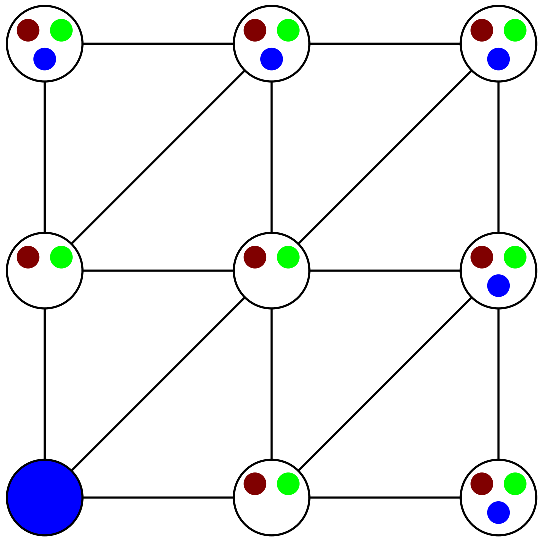
◊ Arc consistency
— Prune similarly for all pairs of values related by a constraint

◊ Variable ordering and value ordering heuristics important for real efficiancy

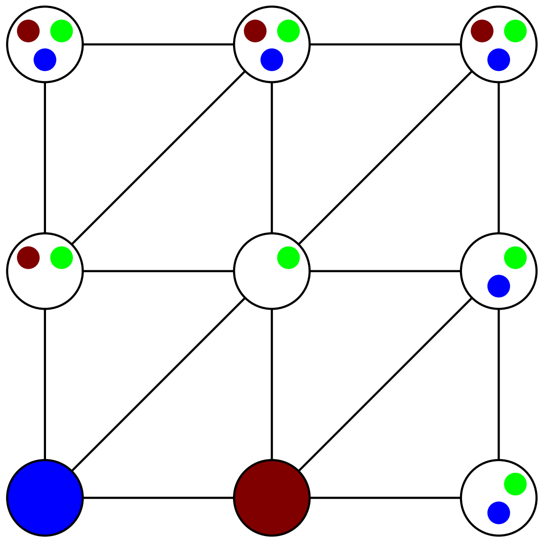
Learning from mistakes



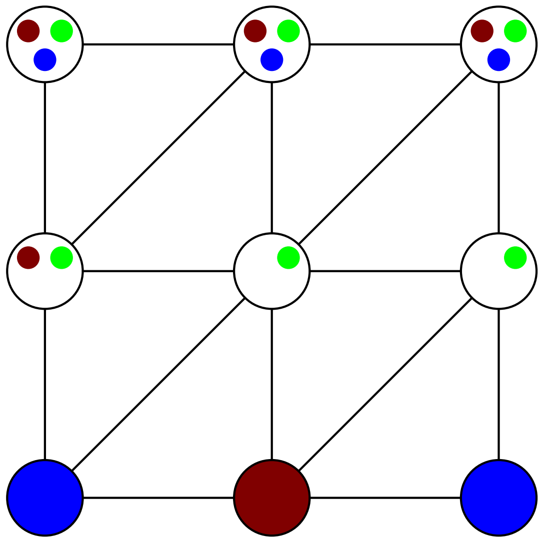
Learning from mistakes



Learning from mistakes

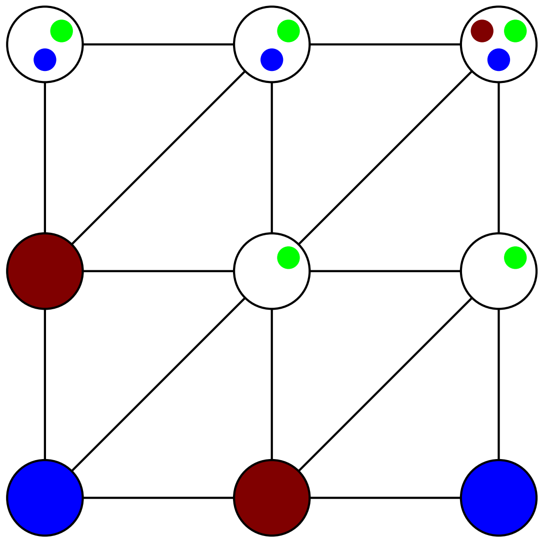


Learning from mistakes



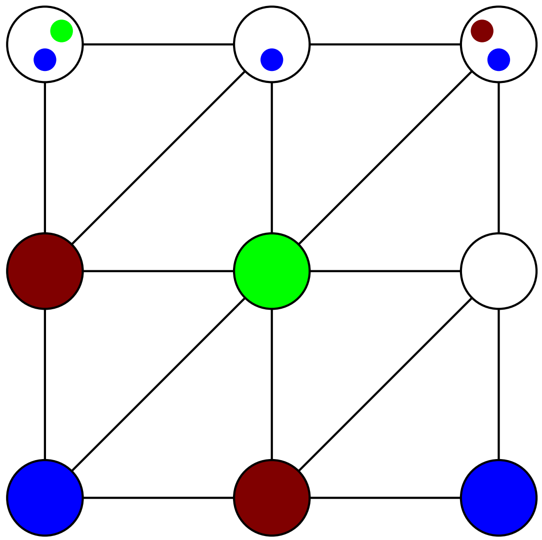
♢ This assignment is consistent but can't be extended to a solution

Learning from mistakes



♢ This assignment is consistent but can't be extended to a solution

Learning from mistakes

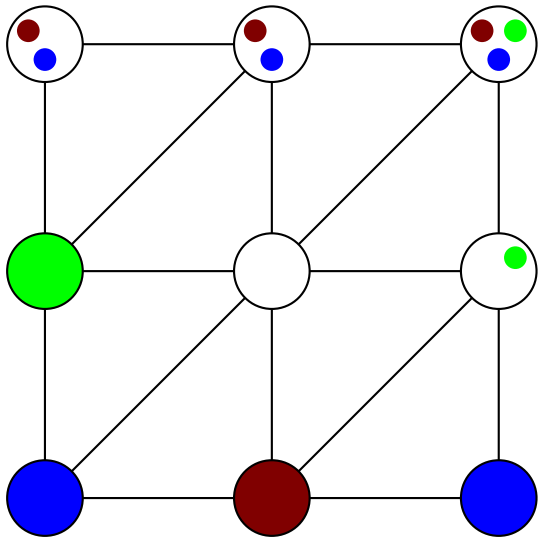


◇ The previous assignment must be wrong

not counting the last green one, which was forced

so remember the earlier choices, and don't do it again!

Learning from mistakes



◇ Actually, we're going to backtrack further

so the bottom line was no good.

Remember that combination $(v_1\!:\!b,\ v_2\!:\!r,\ v_3\!:\!b)$ as a disallowed triple of a (3-ary) constraint

◇ Never repeat a mistake: don't backtrack twice for the same reason

Constraint learning: notes

◇ Learned constraints may be added to the network or kept separately

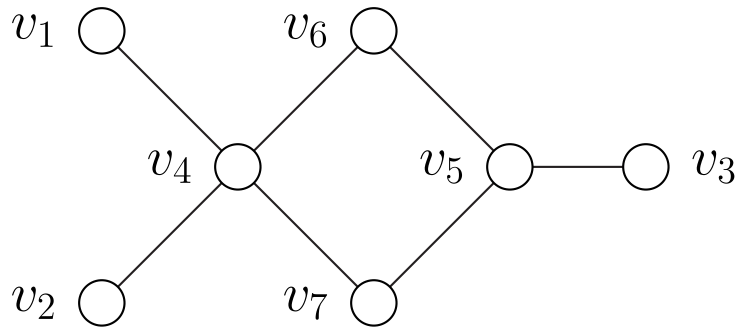
◇ A separate store of nogoods is usual, as they are usually large
— May add binary ones to the network and store the rest
— Data structures matter: indexing for rapid inference is important

◇ Every branch may add another nogood, so there are too many of them
— Storage requires exponential space

◇ Hence common to have a strategy for forgetting them
— e.g. let the longest ones lapse after a while
— or just keep the "tail" and discard when backtracking leaves the region where it applies

◇ Constraint learning useful for CSP solvers; essential for SAT solvers

Constraint graphs



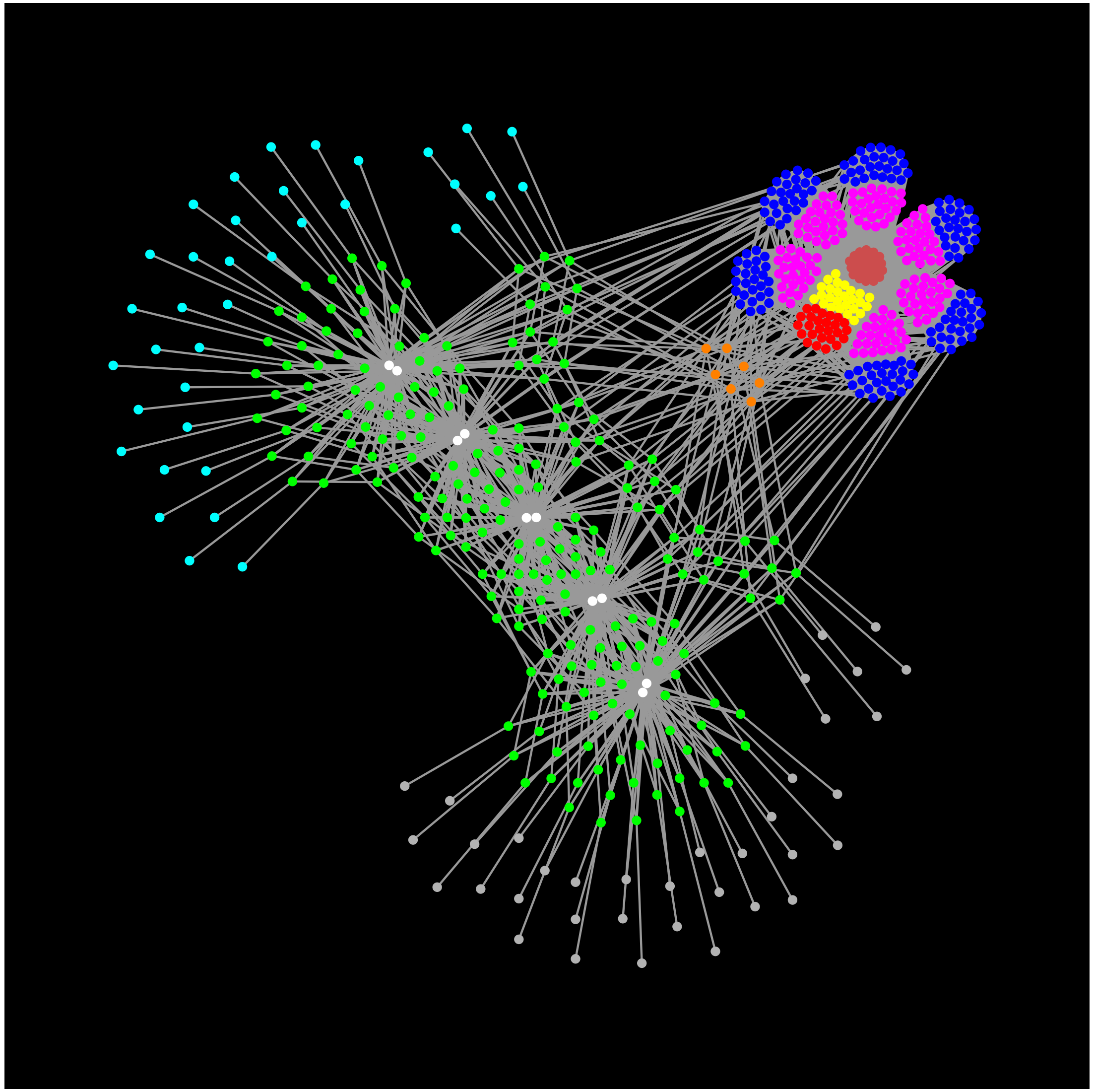
♢ Some decision variables are related by constraints, some are not

♢ Hence we may consider the graph where
— vertices are decision variables
— edges are constraints

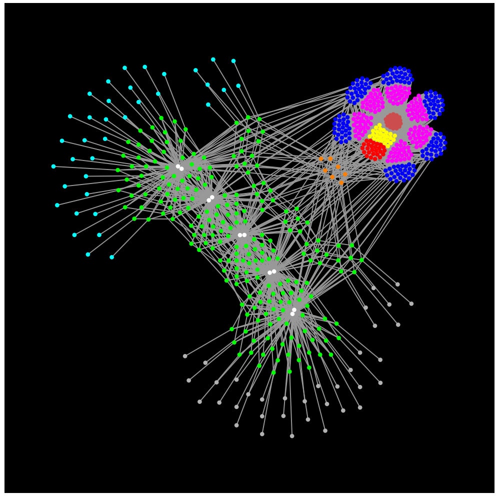
♢ Graph contains information about the structure of the problem

♢ Great for visualisation, as well as automated reasoning

PSR Constraint Graph



PSR Constraint Graph



Can observe properties of the encoding from the graph:

— The plan (white) only loosely communicates with the calculations (blue and magenta)

— There is no direct information flow from the calculations at one step to the calculations at the next, even though most of the distribution grid is the same

— The first line of switches (green), next to the circuit breakers (orange) has a special status in the CSP. This may be worth investigating.

Constraint graphs: notes

- The examples are static views. Dynamic ones animated to show the search can also be very useful.

  by swapping vertices & edges.

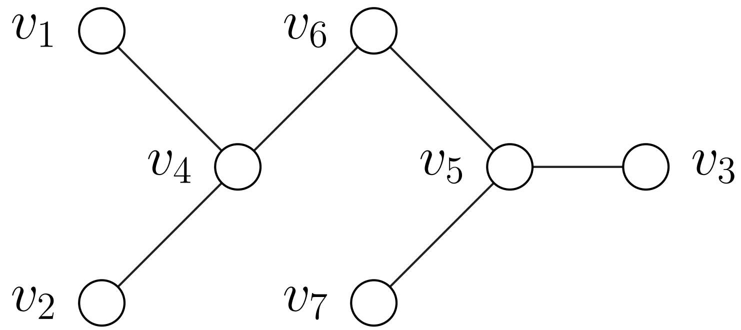
- The dual graph, where the vertices are the constraints and an edge between two constraints means they share a variable, gives yet another view.

- So does the bipartite graph with variable nodes and constraint nodes.

- Constraint graphs are not specific to binary CSPs: they can be useful in analysing logical descriptions of given domains, in SAT solving or in automated reasoning generally.

- Note: the constraint graph only shows which decision variables are connected. It is not affected by whether the problem has solutions or not.

Tree-like constraint graphs



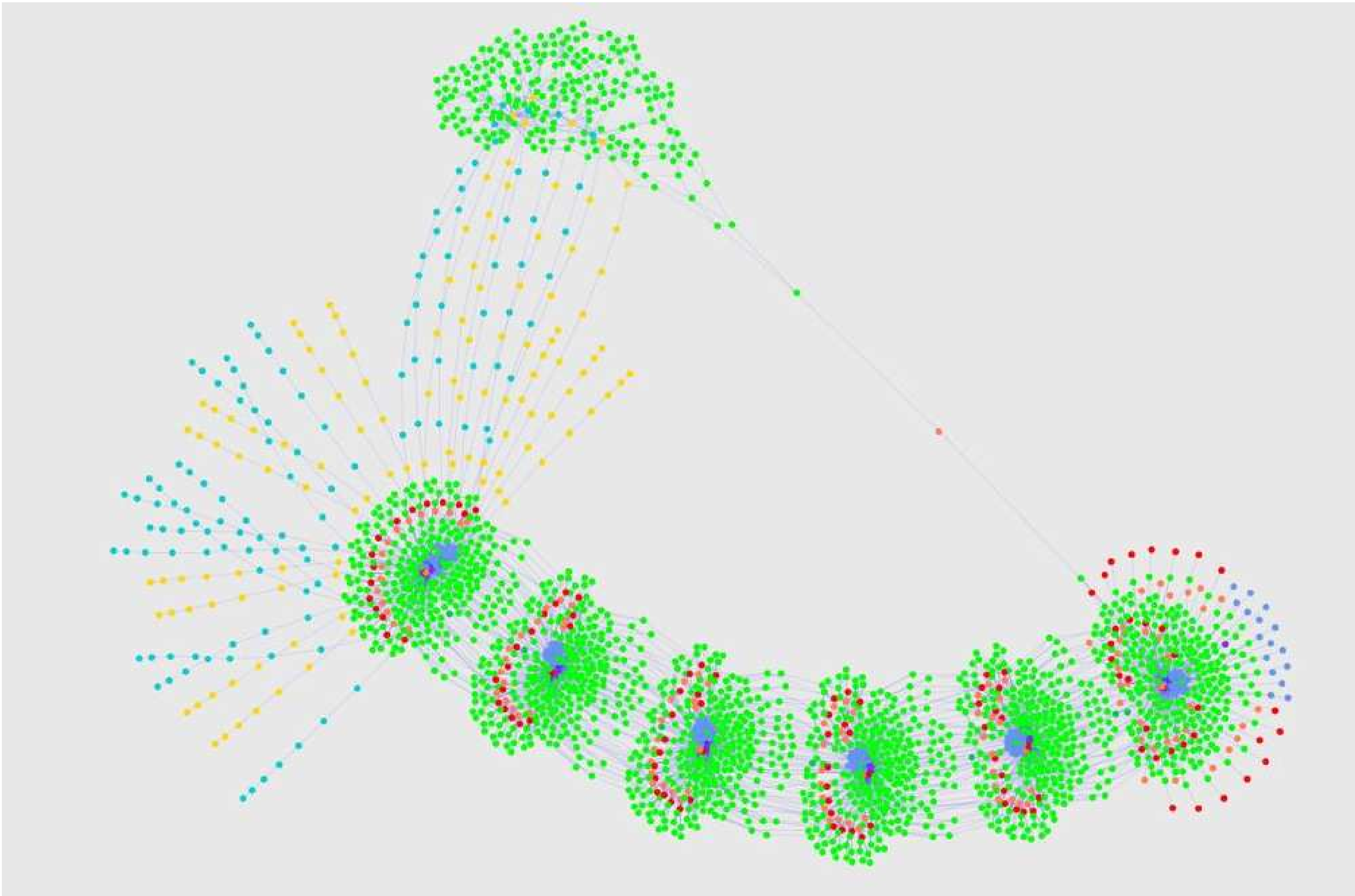
If the constraint graph is a tree, this is always good news!

We can always solve such a CSP efficiently:

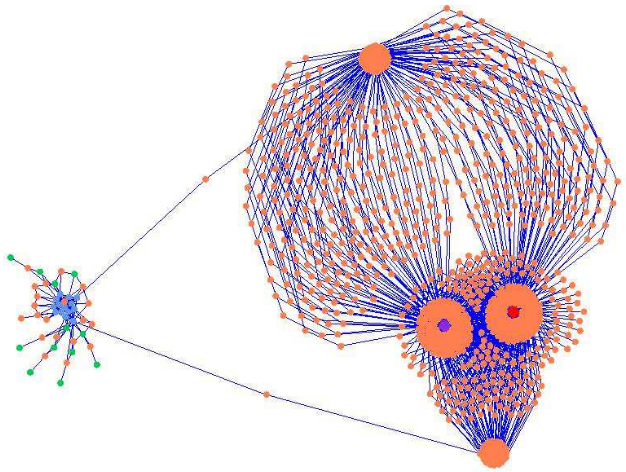
Enforce arc consistency: if wiped out, you're done
Choose a vertex to be the root of the tree
Start assigning values at the root
Don't assign a value to a variable before its parent in the tree
Do forward checking at each step

The search will be backtrack-free.

Constraint graph: Longmult (SAT)

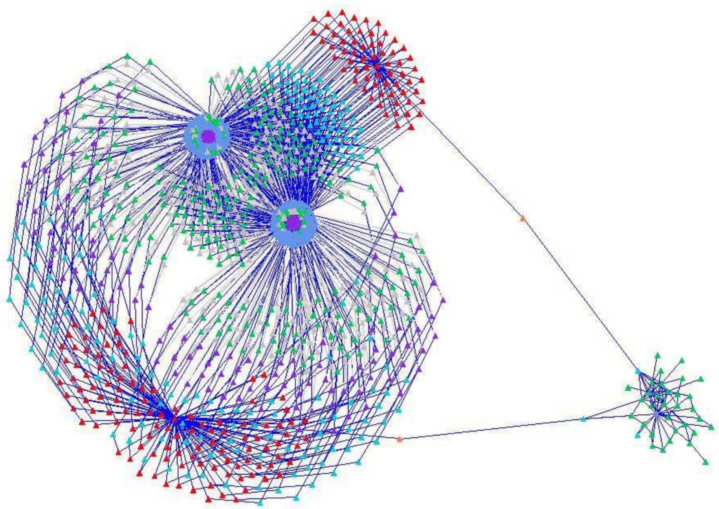


Constraint graph: logical calculus tester



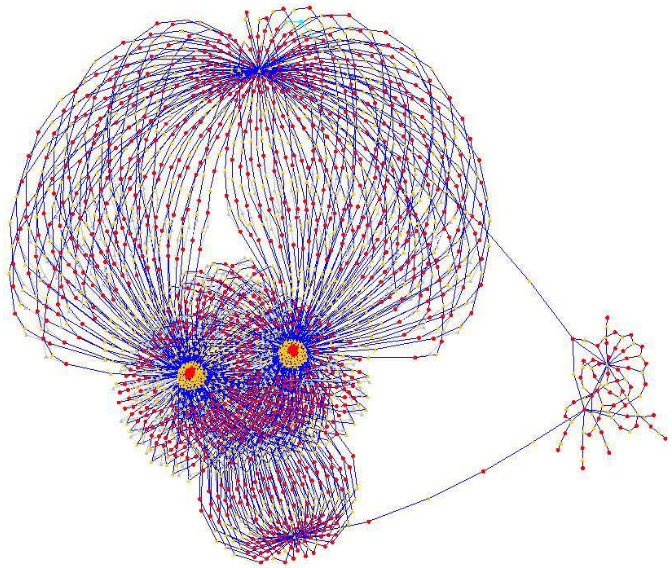
Normal view: variables as vertices

Constraint graph: logical calculus tester



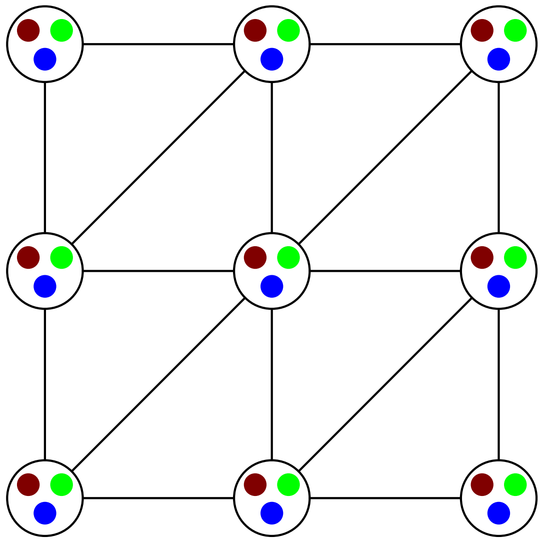
Dual view: constraints as vertices

Constraint graph: logical calculus tester

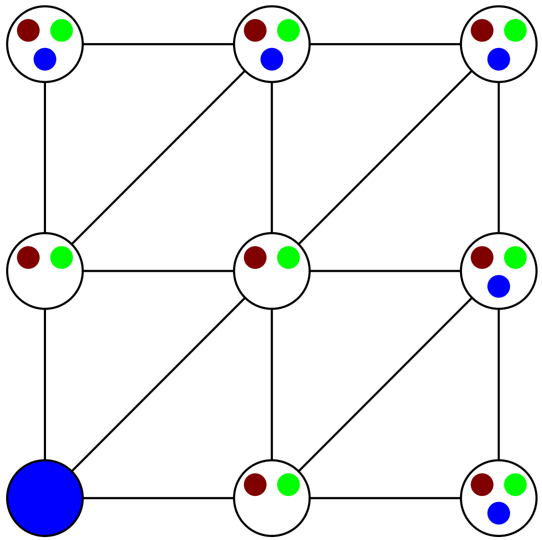


Bipartite view: variables and constraints as vertices

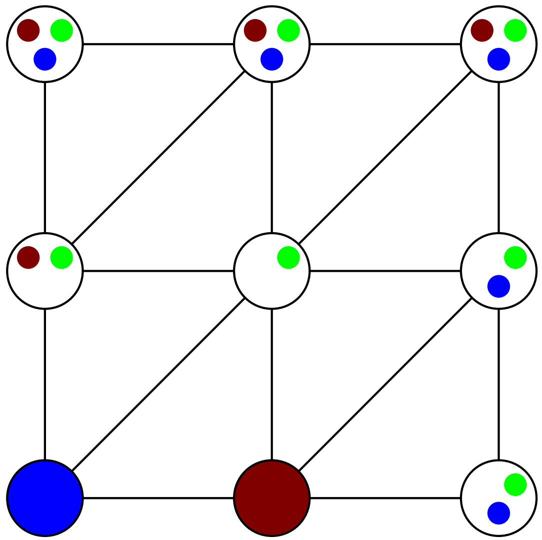
Symmetry



Symmetry

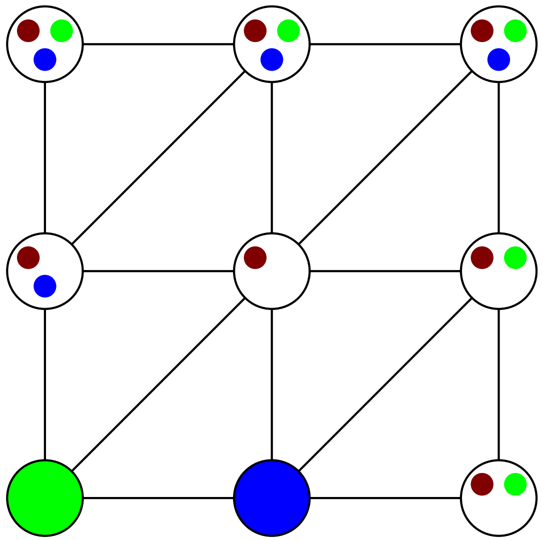


Symmetry



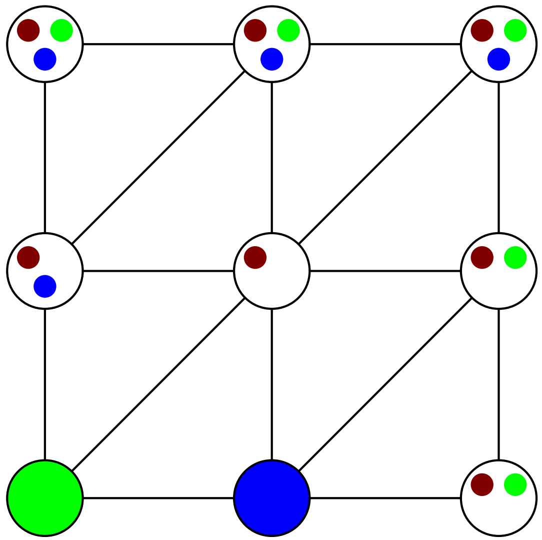
◇ What would happen if we started with a different choice of colours?

Symmetry



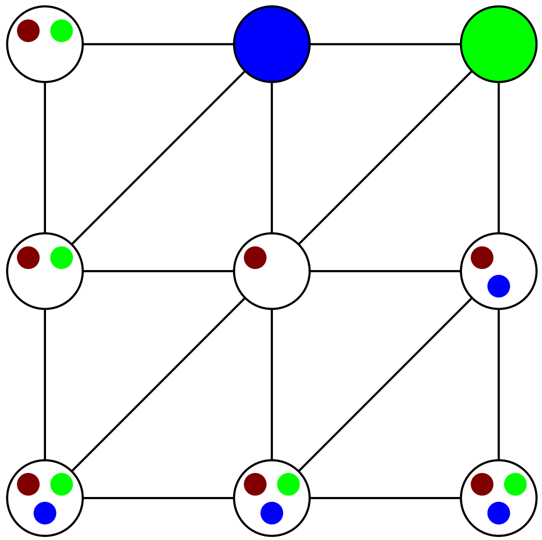
- What would happen if we started with a different choice of colours?
- Exactly the same, but with the colours interchanged.
- So any solution to graph colouring with k colours can be re-labelled to give $k!$ solutions with the same colours in different orders.
- We say that the values in this problem are symmetric.

Symmetry



- ♢ What would happen if we started at the top right?

Symmetry



- What would happen if we started at the top right?
- Exactly the same, but rotated 180°.
- Any solution can be rotated or reflected in a a diagonal to give an equivalent solution with variables interchanged.
- We say that this problem has a variable symmetry.

Using symmetry

- ♢ Is symmetry on our side or against us?
- ♢ It's our friend if we know about it and use it, but our enemy otherwise!
- ♢ The bad part: if a problem has lots of symmetries, we can waste huge amounts of time searching symmetric (and equally empty) sub-spaces, or generating solutions that tell us nothing really new.
- ♢ The good part: if we explore one of these sub-spaces, we know we can prune all of the others without losing anything essential.
- ♢ Unlike arc consistency, etc, symmetry pruning can delete solutions, but it can never delete **all** of them.

Symmetry: how it's done

◇ Note that if solutions are symmetric, partial assignments have (at least) the same symmetries, so early pruning may be possible

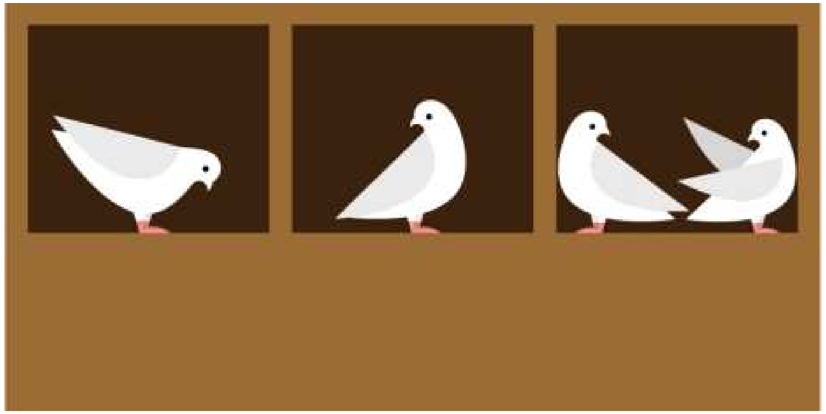
◇ The usual method for removing symmetric sub-problems is to add symmetry-breaking constraints

◇ Formulae true of one (or some) of the symmetric solutions but false of the others.

◇ E.g. we could add a constraint saying $v_1 = \text{blue} \wedge v_2 = \text{red}$.
— safe addition: if there are solutions, there's one satisfying this
— reduces two of the domains to singletons
— rules out 5 of the 6 symmetric solutions.

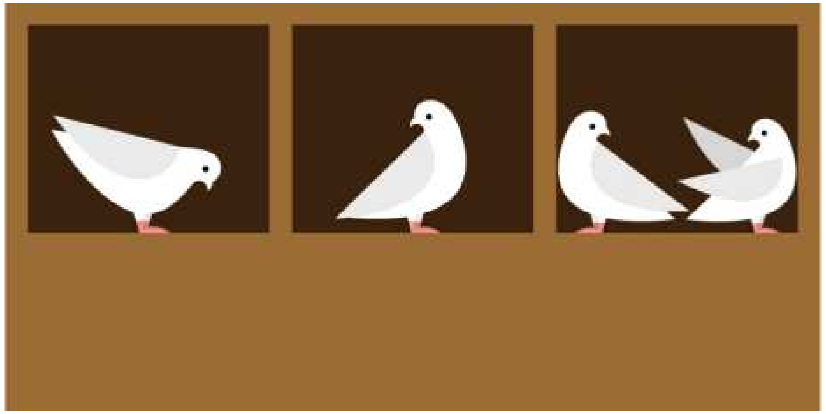
◇ If we want to recover the missing solutions, that's possible without search.

Symmetry: an extreme case



- Suppose we have a pigeonhole problem: show that it's impossible to fit 10 pigeons in 9 pigeonholes (without overcrowding)
- A backtracking search will start assigning holes to pigeons, house 9 of them and discover that the tenth has nowhere to go.
- Then it will backtrack, try a different ninth pigeon, and find that there is still one left over ... etc.
- 9! backtracks, even with arc consistency; no solution.

Symmetry: an extreme case



- But one pigeon looks just like another (to a CSP solver), and one hole looks just like another as well.
- So pigeon number 1 goes in hole number 1, without loss of generality.
- Assign hole 2 to pigeon 2, etc. Then pigeon 10 is homeless. The end!
- A good symmetry-breaker is $\forall x \forall y ((x < y) \rightarrow (\text{hole}(x) < \text{hole}(y)))$.
  —would be true of 1 solution if there were just enough holes
- 9! branches reduced to 1.

Optimal Solutions

◇ Constraint solvers often asked to produce optimal solutions
– though in practice, suboptimal but good solutions suffice

◇ Optimisation not treated (much) in this course
— worth a course on its own

◇ However, we should note it, so:

Optimal Solutions

- Constraint solvers often asked to produce optimal solutions
  – in practice, suboptimal but good solutions usually suffice

- Optimisation not treated (much) in this course
  — worth a course on its own

- However, we should note it, so

- Two common ways of defining “better” or “worse” solutions:

1. via an objective function: a quantity to be minimised (or maximised)
2. via soft constraints: can be violated, but as little as possible

- The sum of soft constraint violations behaves as an objective function.

Optimal Solving

There are **many** techniques for solving problems optimally. The only one to be noted here is Depth First Branch and Bound (DFBB)

- ♢ The default search algorithm used by most FD solvers
- ♢ Easy to implement, generally applicable, complete
- ♢ Also functions well as an anytime method

Branch and Bound

- ♢ Use lower bound estimate L of the cost of solutions extending the current partial assignment
  — underestimates the objective function at each node
- ♢ Also use a bound B
  — strictly overestimates the objective function (globally)
  — initialise to infinity (or a known overestimate)
- ♢ Traverse the search tree e.g. depth first
- ♢ Backtrack if $L \geq B$
- ♢ Each time a solution is found, set B to its objective value
- ♢ B is monotone decreasing as solutions are found
- ♢ So search tree branches tend to get shorter towards the end

DFBB: Intermediate solutions

◊ First solution is at the bottom of the leftmost (complete) branch
— Fast: Likely to be found quickly
— Dirty: Likely to be of low quality

◊ Always trying to improve on the best so far
— Any improvement will do

◊ So DFBB produces a sequence of (strictly) improving solutions

◊ We can interrupt the search at any time
— when the current solution is good enough
— when a time limit expires
— when the next process needs to start
— when we just get fed up with waiting

◊ Intermediate solutions are valuable, because optimal ones can be very expensive to compute (and proofs of optimality even more expensive).

Summary

◇ Constraint (nogood) learning from wipeouts usually improves efficiency
— Space (memory) is a limitation for nogood learning, so forgetting is also important

◇ Constraint graphs give information about problem structure
— Certain constraint graphs (e.g. trees) indicate that problems are easy

◇ Value symmetry and variable symmetry are frequently present in CSPs
— Pruning symmetric sub-spaces is a big winner where there is extensive symmetry

◇ Optimisation (minimising a cost or objective function) is usual for CSPs

◇ Depth First Branch and Bound is commonly used in FD solvers
— Conveniently provides intermediate solutions of increasing goodness