CS60045 Artificial Intelligence Autumn 2023

Constraint Satisfaction Problems (CSP)

CSP: A new type of search problems

$\mathcal{V} = \{ x_1, x_2, \ldots, x_n \}$ = a set of variables

$\mathcal{D}_i$ = the domain of the variable x_i

$\mathcal{C}$ = a set of constraints each involving one or more variables from $\mathcal{V}$

Basic goal: To find an assignment of all the variables in $\mathcal{V}$ from their respective domains such that all the constraints in $\mathcal{C}$ are satisfied.

Such an assignment is called **consistent**.

Partial assignment is an assignment of some (not necessarily all) of the variables of $\mathcal{V}$.

If a partial assignment does not violate any constraint in $\mathcal{C}$, it is called **consistent**.

Optimization goal: Minimize or maximize a function $f(x_1, x_2, \ldots, x_n)$ over all consistent (complete) assignment of the variables in $\mathcal{V}$. **[Constrained optimization problem (COP)]**

Example 1: Set of linear equations

Variables: $x_1, x_2, \ldots, x_n$

Domains: The set of real numbers or integers or positive integers or …

Constraints:

$$a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1$$
$$a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2$$
$$\ldots$$
$$a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n = b_m$$

Standard solution: Gaussian Elimination

Restricted domains may be an issue (find positive integer solutions for a less-than-full-rank system).

Example 2: Boolean satisfiability (SAT)

Variables: $x_1, x_2, \ldots, x_n$

Domains: The domain of each variable is { 0, 1 }

Constraint:

$$\Phi(x_1, x_2, \ldots, x_n) = 1.$$

CNFSAT: Φ is presented in the CNF (product of sum). Each sum in the product is called a **clause**. Let $\Phi = C_1 C_2 \ldots C_m$. Then the constraint $\Phi = 1$ is equivalent to the set as the set of m constraints

$$C_i = 1 \text{ for all } i = 1, 2, \ldots, m.$$

Both SAT and CNFSAT are NP-Complete.

There are many SAT solvers.

The problem is different from AND-OR search, because clauses cannot be solved independently.

Example 3: Linear Programming (LP)

Minimize/Maximize a linear function of the variables (the objective function) subject to a set of linear equality and inequality constraints.

The output is the optimum value of the objective function along with an assignment of the variables where the optimum is obtained.

If only a CSP is to be solved, take 0 as the objective function.

Domains

- Real-valued LP: Efficient algorithms exist
- Integer LP: Each variable must be given an integer value (difficult to solve in general).
- Binary LP: Each variable must be given a value from { 0, 1 } (difficult to solve in general)

Example 4: Solving Sudoku [Linear equation formulation]

Let d_{rc} be the digit at row r and column c. The domain of each variable is { 1, 2, 3, 4, 5, 6, 7, 8, 9 }.

The sum of the entries in each row/column/block must be 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 = 45.

These constraints give us 27 linear equations.

Some d_{rc} values are available immediately from the initial board.

A typical board may contain 50 or more blank cells. Let b be the count of the blank cells.

So the system generated is of less than full rank.

We want a solution (unique for good puzzles) from a set of 9^b assignments.

The sums do not reflect the constraints fully. For example, 2 + 2 + 2 + 4 + 5 + 6 + 7 + 8 + 9 = 45 is a solution for a row/column/block but is not consistent with the rules of the game.

Example 4: Sudoku [LP formulation]

Introduce variables x_{drc}. Domain of each variable is { 0, 1 }. We have the interpretation $x_{drc} = 1$ if the (r,c)-th entry is to be filled by the digit d. x_{drc} is 0 otherwise.

Constraint: Each entry must be filled by only one digit.

$$\sum_d x_{drc} = 1 \text{ for all } r, c.$$

Constraint: Each row has exactly one occurrence of each digit d.

$$\sum_c x_{drc} = 1 \text{ for all } d, r.$$

Constraint: Each column has exactly one occurrence of each digit d.

$$\sum_r x_{drc} = 1 \text{ for all } d, c.$$

Constraint: Each block has exactly one occurrence of each digit d.

$$\sum_{r,c} x_{drc} = 1 \text{ for all } d \text{ and for all block.}$$

If b is the number of blank cells, the number of unknown variables is $9b$. The number of equations is $b + 243$.

We can invoke an **LP solver**: Minimize 0 subject to the $9b$ linear constraints.

Example 4: Sudoku (SAT formulation)

Let $Y_1, Y_2, \ldots, Y_9$ be 0,1-valued variables.

The equation $Y_1 + Y_2 + \ldots + Y_9 = 1$ is equivalent to the Boolean condition

$$\begin{aligned} & Y_1\bar{Y}_2\bar{Y}_3\bar{Y}_4\bar{Y}_5\bar{Y}_6\bar{Y}_7\bar{Y}_8\bar{Y}_9 + \bar{Y}_1Y_2\bar{Y}_3\bar{Y}_4\bar{Y}_5\bar{Y}_6\bar{Y}_7\bar{Y}_8\bar{Y}_9 + \bar{Y}_1\bar{Y}_2Y_3\bar{Y}_4\bar{Y}_5\bar{Y}_6\bar{Y}_7\bar{Y}_8\bar{Y}_9 + \\ & \bar{Y}_1\bar{Y}_2\bar{Y}_3Y_4\bar{Y}_5\bar{Y}_6\bar{Y}_7\bar{Y}_8\bar{Y}_9 + \bar{Y}_1\bar{Y}_2\bar{Y}_3\bar{Y}_4Y_5\bar{Y}_6\bar{Y}_7\bar{Y}_8\bar{Y}_9 + \bar{Y}_1\bar{Y}_2\bar{Y}_3\bar{Y}_4\bar{Y}_5Y_6\bar{Y}_7\bar{Y}_8\bar{Y}_9 + \\ & \bar{Y}_1\bar{Y}_2\bar{Y}_3\bar{Y}_4\bar{Y}_5\bar{Y}_6Y_7\bar{Y}_8\bar{Y}_9 + \bar{Y}_1\bar{Y}_2\bar{Y}_3\bar{Y}_4\bar{Y}_5\bar{Y}_6\bar{Y}_7Y_8\bar{Y}_9 + \bar{Y}_1\bar{Y}_2\bar{Y}_3\bar{Y}_4\bar{Y}_5\bar{Y}_6\bar{Y}_7\bar{Y}_8Y_9 \\ = \; & 1 \end{aligned}$$

We seek for a 0,1-valued solution for $Y_1, Y_2, \ldots, Y_9$ from the $b + 243$ constraints.

We can invoke a **SAT solver**.

Example 4: Sudoku [A human-like formulation]

Let d_{rc} be the digit at row r and column c. The domain of each variable is { 1, 2, 3, 4, 5, 6, 7, 8, 9 }.

Constraints

For each row r

AllDiff(d_{r1}, d_{r2}, d_{r3}, d_{r4}, d_{r5}, d_{r6}, d_{r7}, d_{r8}, d_{r9})

For each column c

Alldiff(d_{1c}, d_{2c}, d_{3c}, d_{4c}, d_{5c}, d_{6c}, d_{7c}, d_{8c}, d_{9c})

For each block with the top-left cell at (r, c)

AllDiff($d_{r,c}$, $d_{r,c+1}$, $d_{r,c+2}$, $d_{r+1,c}$, $d_{r+1,c+1}$, $d_{r+1,c+2}$, $d_{r+2,c}$, $d_{r+2,c+1}$, $d_{r+2,c+2}$)

Note: You can replace each AllDiff constraint by several two-variable inequalities.

Example 5: n-Queens problem (Solved by local search earlier)

Variables: $r_1, r_2, r_3, \ldots, r_n$ (the row numbers where the queens appear in the n columns).

Domains: { 1, 2, 3, …, n } for each variable.

Constraints: For each $1 \leqslant i < j \leqslant n$, we have three constraints:

$$r_j \neq r_i$$

$$r_j \neq r_i + (j - i)$$

$$r_j \neq r_i - (j - i)$$

Example 6: TSP (Solved by local search earlier)

Introduce **0,1-valued variables** x_{ij}. We have $x_{ij} = 1$ if City j is visited immediately after City i, $x_{ij} = 0$ otherwise.

Constraints

Unique city after each City i: $\sum_j x_{ij} = 1$ for all i.

Unique city before each City j: $\sum_i x_{ij} = 1$ for all j.

No collection of disjoint cycles: $\sum_{i \in S} \sum_{j \notin S} x_{ij} \geqslant 1$ for all non-empty proper subsets S of cities.

Optimization

Minimize $\sum_i \sum_{j \neq i} c_{ij} x_{ij}$.

Note: Exponentially many constraints.

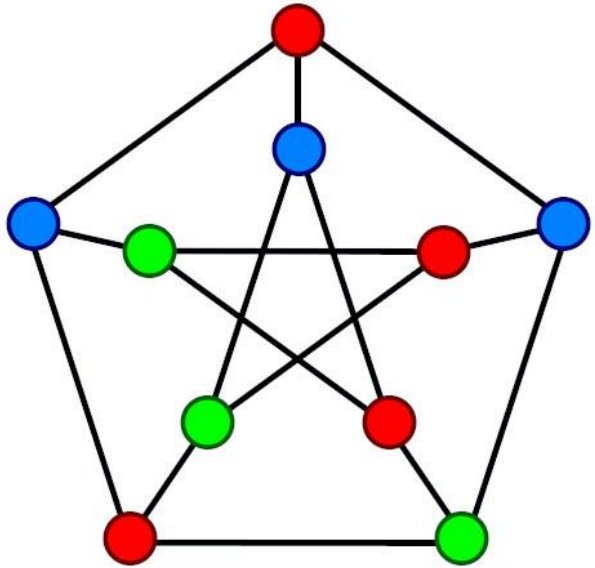
Example 7: Graph (Map) coloring (Non-numeric variables)

Variables: The color c_v of each vertex v.

Domain: A set of colors like { R, G, B } as in the example.

Constraints:

$c_u \neq c_v$ for every edge (u, v) of the graph.



A 3-coloring of the Petersen graph
(Source: Wikipedia)

Example 7: Map coloring

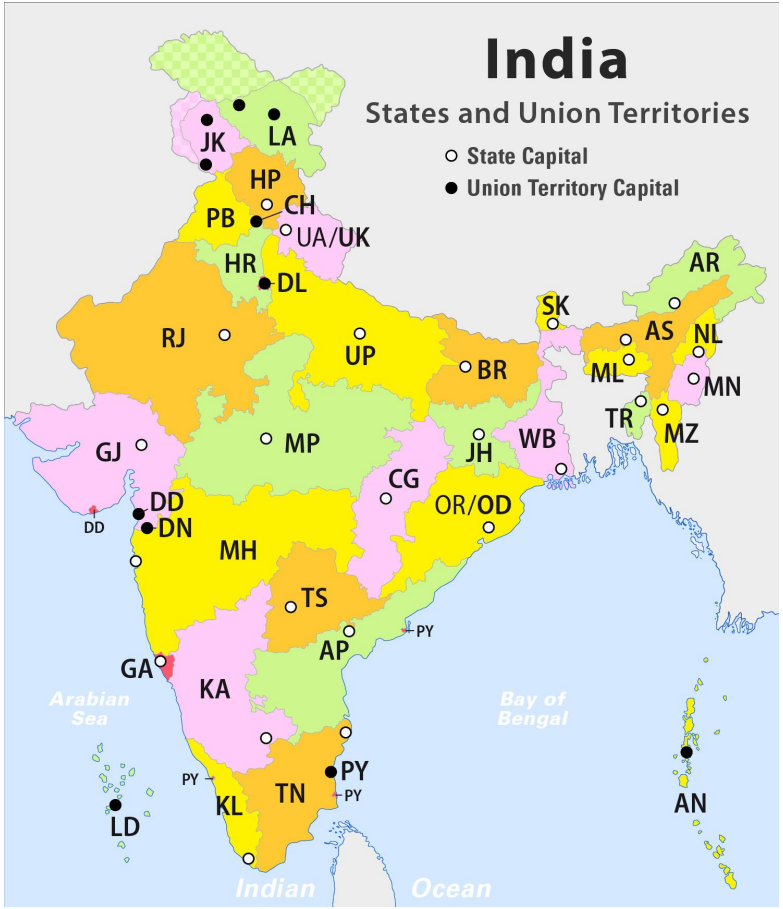
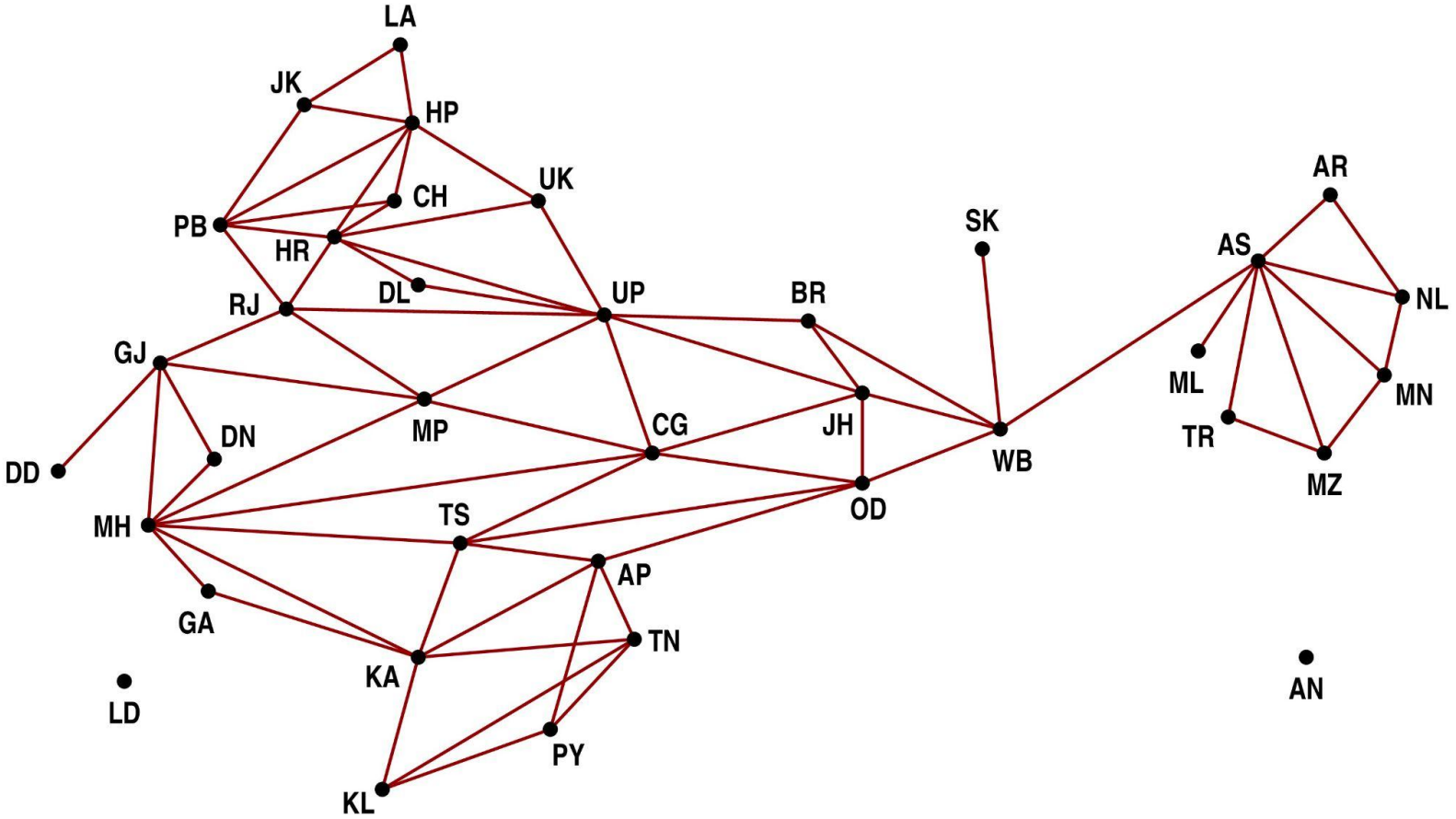
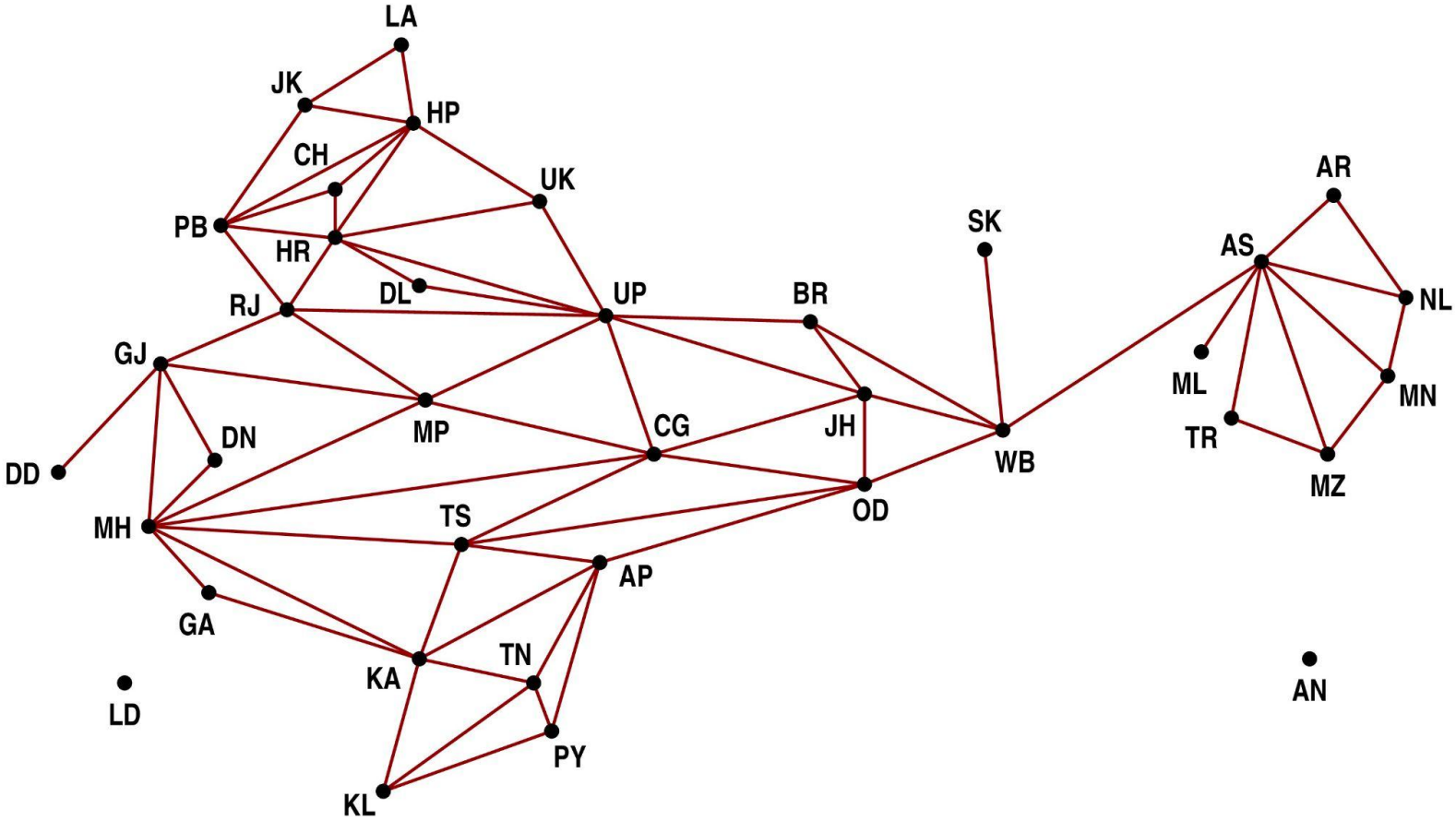


Image source: Wikipedia

Example 7: Map coloring



Example 7: Map coloring



Example 8: Job-shop scheduling

A machine is assembled in a workshop from the components C_1, C_2, …, C_n.

Fitting C_i takes T_i time.

Variables: S_i to denote the start time of fitting C_i.

Constraints

[C_j cannot be fitted before C_i] $S_i + T_i \geqslant S_j$.

[Two components C_i and C_j need a specific tool, and cannot be fitted in parallel]

$$S_i + T_i \geqslant S_j \text{ } \textbf{or} \text{ } S_j + T_j \geqslant S_i.$$

Objective

[CSP]	**Schedule within a total time of *T***	Domain of S_i will be [0, $T - T_i$].
[COP]	**Finish as soon as possible**	Minimize $\max_i (S_i + T_i)$.

Types of constraints

Unary constraint: Involves a single variable (Examples: $0 \leqslant x \leqslant 100$, $x \neq U$)

Binary constraint: Involves two variables (Examples: $x + y \geqslant 10$, $x \neq y$)

Binary CSP: A CSP with only unary and binary constraints

Global constraint: Involves any number of variables (Examples: $x + y \geqslant z + 10$, AllDiff(w, x, y, z))

Binarization of a global constraint

For each global constraint, introduce a new variable whose domain consists of all pairs satisfying the constraint. Also introduce a binary constraint indicating the position each original variable in the new variable.

Example: Convert $x + y \geqslant z + 10$ where x, y, z have domains $\mathcal{D}_X$, $\mathcal{D}_Y$, and $\mathcal{D}_Z$.
Introduce a new variable U with domain $\{(a, b, c) \mid a \in \mathcal{D}_X, b \in \mathcal{D}_Y, c \in \mathcal{D}_Z, a + b \geqslant c + 10\}$, and the new constraints $X = \text{first}(U)$, $Y = \text{second}(U)$, and $Z = \text{third}(U)$.

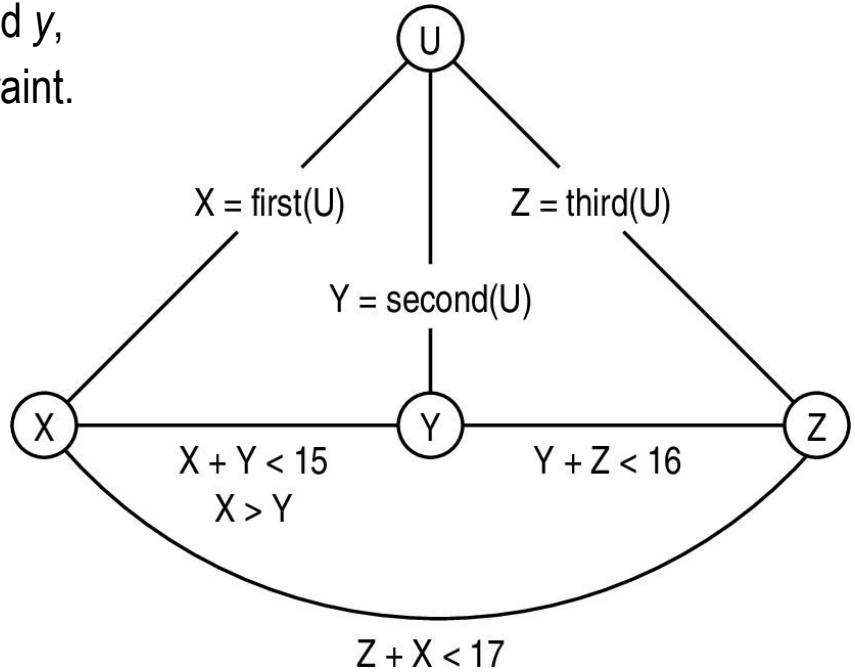
Constraint graph (undirected) for a binary CSP

Introduce a node for each variable.

Each unary constraint restricts the domain of the variable involved, and is not needed to be shown.

For each binary constraint involving the variables *x* and *y*, create an edge between *x* and *y* marked by the constraint.

Example: Let X, Y, Z be variables each with domain {1, 2, 3, …, 10} along with the constraints X > Y, NotAllSame(X, Y, Z), X + Y < 15, Y + Z < 16, and Z + X < 17. Introduce a new variable U with domain consisting of all triples (*a*, *b*, *c*) with each component being an integer in the range [1, 10] and with the exception of the triples of the form (*a*, *a*, *a*).



Domain reduction using consistency checking

CSP solvers have two types of moves:

- Move to a neighbor by assigning value to an unassigned variable
- Check consistency

Consistency checking can significantly cut down the domain (and may even eliminate the necessity of any search).

Example (Sudoku)

Choose a yet unfilled cell *C*.
Find out which values are still allowed for *C*.
For each allowed digit *d* in *C* repeat:
    Put *d* in *C*.
    Update the allowed domains of other blank cells, as caused by this placement.
    If the domain of some cell reduces to empty, stop further exploration along this line, otherwise continue the search.

Types of consistency

A *partial assignment* is **consistent** if all the unassigned variables can be given at least one set of values from their respective domains so that all the constraints are satisfied.

Node consistency

For each assigned variable v, the value of v satisfies all the unary constraints on v.

Arc (or Edge) consistency

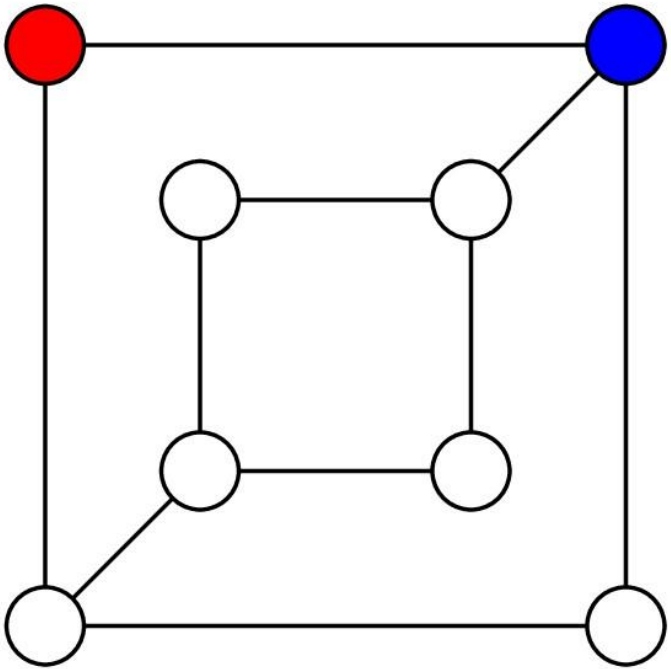
For each pair (v, w) of assigned variables, the values assigned to v and w satisfy all the constraints against the edge (v, w).

Path consistency

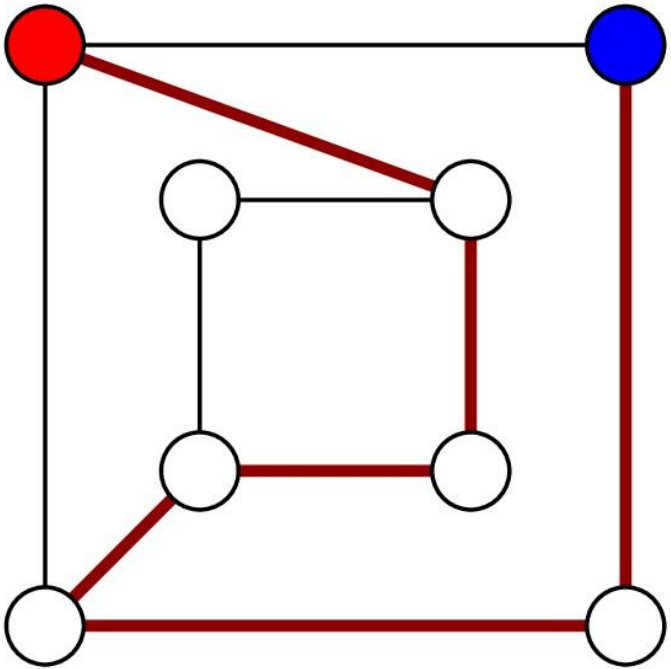
Let v and w be two assigned variables. Path consistency holds for the pair (v, w) if each $v \longrightarrow w$ path can be consistently assigned values using the allowed domains of all variables on the path. The partial assignment is called path-consistent if it is so for all pairs (v, w) of vertices.

Example: 2-coloring a graph

Both these assignments are edge–consistent



This assignment is path–consistent



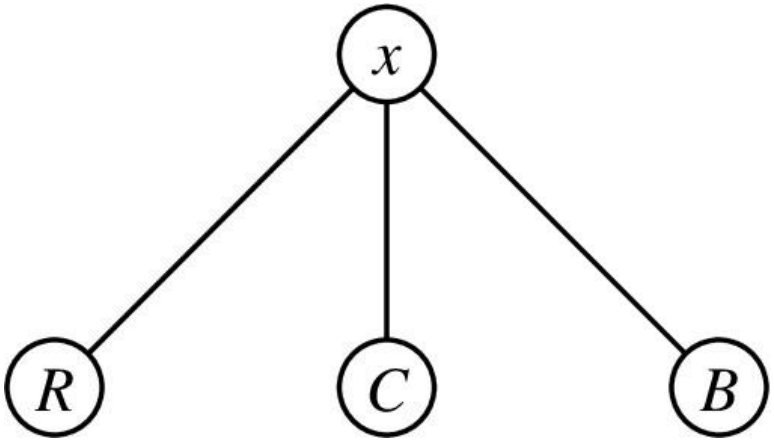
This assignment is not path–consistent

Example: Sudoku

Binarization of the AllDiff constraints on rows, columns, and blocks gives three edges connecting a cell *x* to its three constraint vertices *R*, *C*, and *B*.

The links between *x* and its constraint vertices Indicate that the current domain of *x* must be the intersection of the projected values.

This is the same as saying that *x* cannot take a value already occupied by a filled cell in the row or the column or the block of *x*.

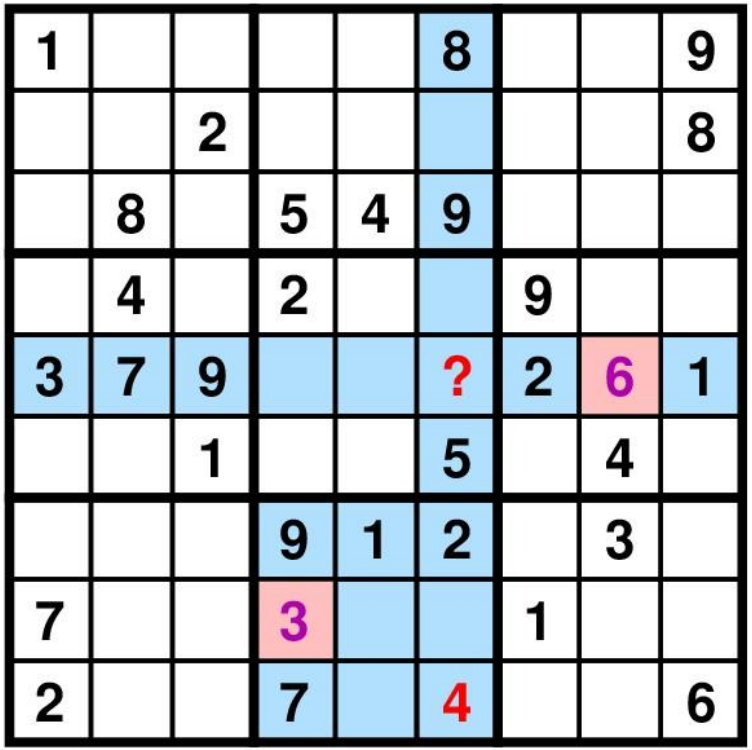


Edge consistency means no row or column or block constraint is violated.

Path consistency extends beyond a single row or column or block.

Path inconsistency in Sudoku

1					8			9
		2						8
	8		5	4	9			
	4		2			9		
3	7	9				2	6	1
		1			5		4	
			9	1	2		3	
7			3			1		
2			7					6



Path of inconsistency: $x_{84} - B_{32} - x_{96} - C_{6} - x_{56} - R_{5} - x_{58}$

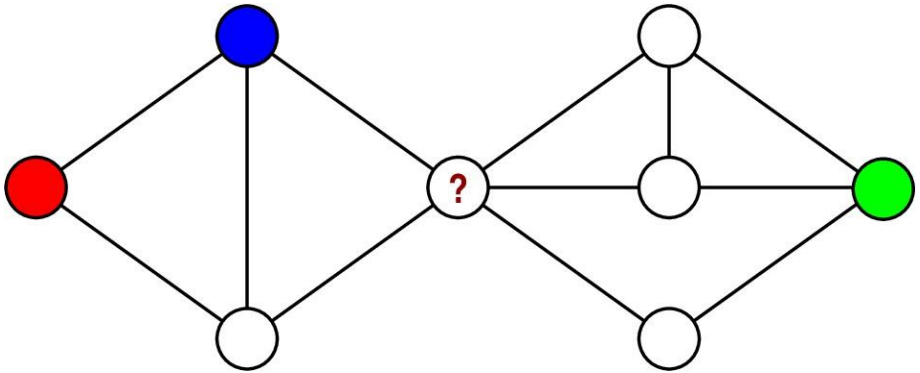
K-consistency

A generalization of path consistency

Given assignments of $k - 1$ variables $v_1, v_2, \ldots, v_{k-1}$, a k-th variable v_k can be consistently assigned.

Strongly k-consistent means i-consistent for all $i = 1, 2, 3, \ldots, k$.

Path consistency is the same as 3-consistency if all relations are binary.



4-inconsistency for 3-coloring

How easy is it to check (in)consistency

Let a CSP have *n* variables Any feasible solution for the CSP must be strongly *n*-consistent.

But checking for strong *n*-consistency may take time and space exponential in *n*.

Most commonly, we maintain 2-consistency.

Much less commonly, 3-consistency is also maintained.

Maintaining higher-order consistency is very time-consuming. Moreover, nothing less than strong *n*-consistency is a complete solution.

The AC-3 algorithm ensures edge consistency in the entire graph.

We assume that all variables have finite domains.

The algorithm uses a queue *Q* of edges.

Q may be the set of all edges in the constraint graph (assumed to be binary).

Constraint propagation by the AC-3 Algorithm

AC3 (*Q*)

While *Q* is not empty, repeat:
- Take any (*i*, *j*) from *Q*.
- Set changemade = *false*.
- For each *x* in $\mathcal{D}_i$, repeat:
  - If there exists no *y* in $\mathcal{D}_j$ for which the constraint (*i*, *j*) is satisfied, then:
    - Remove *x* from $\mathcal{D}_i$.
    - Set changemade = *true*.
- If changemade is true, then:
  - If $\mathcal{D}_i$ is reduced to the empty set, then return *failure*.
  - For each neighbor $k \neq j$ of *i* in the constraint graph, do:
    - Add (*k*, *i*) to *Q* (unless it is already there).

Return *success*.

Performance of AC-3

Let the number of edges in the constraint graph be c.

Let d be the maximum domain size of a variable.

Each edge (i, j) can be introduced in Q at most d times.

We can check the consistency of an edge in $O(d^2)$ time.

The worst-case running time is therefore $O(cd^3)$.

How to solve a CSP?

- Convert the instance to a SAT or LP instance, and invoke a SAT or LP solver.
- Do your own search.
  - We will first look at a backtracking search for CSP.
  - Let there be n variables, each with a domain of size at most d (assumed finite).
  - We assume that the order in which the variables are assigned is not important.
  - Then, there are d^n possible assignments for all variables.
  - We search for all possibilities using a DFS fashion (low memory requirement).
  - We do pruning based upon consistency checking.
  - We will later look at a local search algorithm for CSP.

Backtracking search for CSP

Backtrack(*A*)

If *A* is a complete assignment, return *A*.

Choose a variable *X* which is unassigned in *A*.
For each value *a* in the domain $\mathcal{D}_X$, repeat:
Add the assignment (*X* = *a*) to *A*.
Check consistency of *A*.
If the consistency check returns *success*, then:
Recursively call Backtrack() on the updated *A*.
If the recursive call returns a complete assignment *C*, return *C*.
Remove the assignment (*X* = *a*) from *A*.

Return *failure*.

Variable ordering: Which unassigned variable is to be chosen first?

Static (or fixed) ordering

Example: Color the states of India in alphabetical order: AN, AP, AR, AS, BR, CG, CH, …, WB.

Random ordering

Degree Heuristics

Relevant at the beginning or in case of ties.

Choose the unassigned variable which has the largest number of constraints with the remaining unassigned variables.

Example: Start with UP which has the largest number (eight) of neighbors.

Variable ordering: Which unassigned variable is to be chosen first?

Minimum-remaining-values (MRV) heuristic

Also called most-constrained-variable or fail-safe heuristic.

Choose an unassigned variable with the smallest reduced domain.

Such a variable is likely to lead to a failure soon. This implies good pruning possibilities.

For example, if such a variable has no values left in its domain, then failure will happen immediately. This removes the effort of assigning values to other variables.

The MRV heuristic is in general the best in the lot.

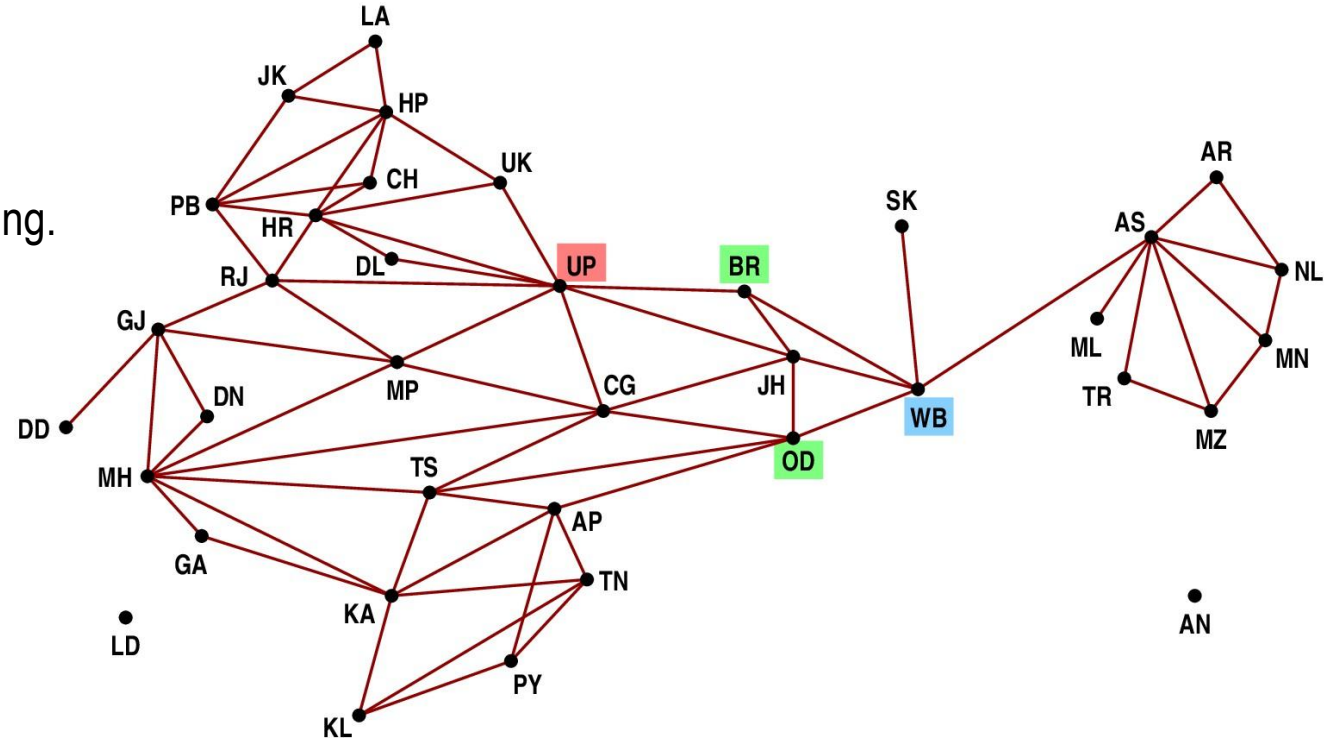
Value ordering: Which value of the chosen variable to be taken first?

Least-constraining-value heuristic

Choose the value which eliminates the fewest of the values from the neighbors.

Example:

Use 4 colors **R**, **G**, **B**, **Y**.
CG is chosen next for coloring.
Options for CG: **B**, **Y**
Choosing **Y** for CG will eliminate the only option **Y** for JH.
So choose **B** for CG.



Looking forward

Suppose that X and a are chosen using good heuristics.

The introduction of the new assignment $X = a$ may create inconsistencies with the remaining unassigned variables.

Checking these inconsistencies may restrict the domains of the remaining unassigned variables, making the future search more efficient.

Forward checking

> Look at all unassigned neighbors Y of X, and reduce the domain of each neighbor (whenever possible) caused by the assignment $X = a$.

Maintaining arc consistency (MAC)

> Run the AC-3 algorithm by setting $\mathcal{D}_X = \{x\}$, and with Q initialized by the edges (Y, X) for all unassigned neighbors Y of X.

If the assignment $X = a$ eventually leads to failure, we need to restore the domains of X and the neighbors Y.

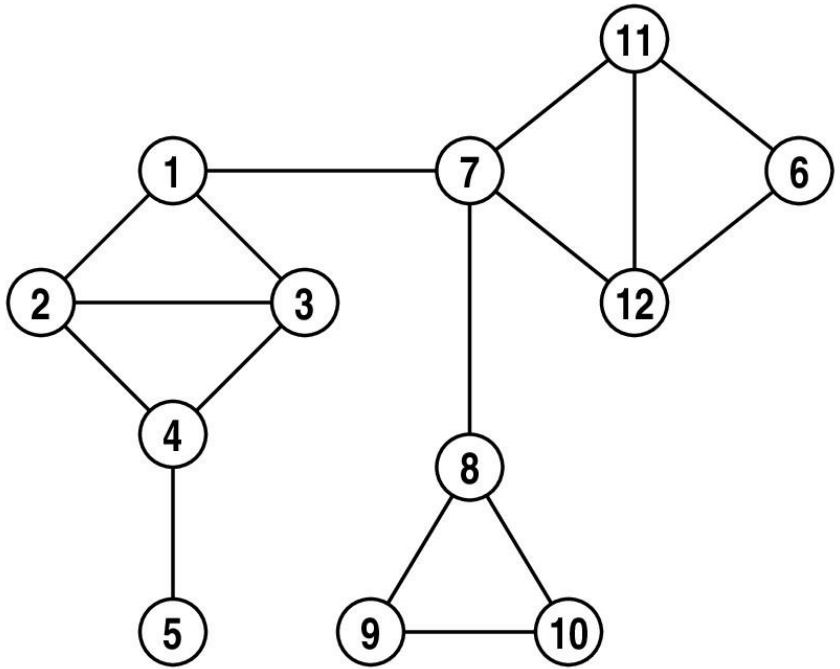
Looking backward

Chronological backtracking

- Exploration down the tree fails when the domain of a node *u* in the search path becomes empty.
- Go to the parent *v* of *u*.
- If *v* has more values of explore, proceed forward.
- Otherwise, go to the parent *w* of *v*, and so on.

Example

We will color the adjacent graph with three colors. The vertices are chosen in the fixed order 1, 2, 3, …, 12, and colors are chosen in the sequence R, G, B. Only forward checking is done (no MAC).



Chronological backtracking

Search fails at 12.

Go back to 11. No further options.

Go back to 10: No further options.

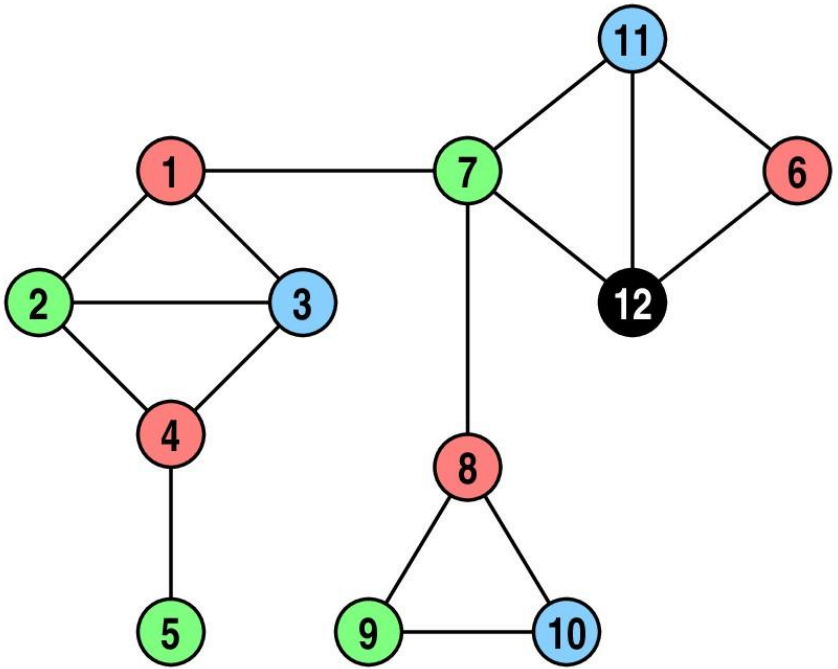
Go back to 9: Recolor 9 by B. Proceed.

The same inconsistency is reached.

Backtrack to 8. Change its color to G. Continue.

Observation

> The coloring of 12 failed, because its three neighbors 6, 7, and 11 shrunk its domain to empty. Recoloring 8, 9, and 10 cannot repair the problem.



Conflict sets

Eech variable starts with an empty conflict set.

Whenever an assignment $X = a$ causes the domain of a neighbor Y of X to shrink, add X to the conflict set of Y.

Suppose that the current search path is $v_1, v_2, \ldots, v_k$. Then the search fails at v_{k+1}.

We look at the conflict set $C(v_{k+1})$. This is a subset of $\{ v_1, v_2, \ldots, v_k \}$.

Choose the largest j for which v_j is in $C(v_{k+1})$.

Changing the assignments of $v_{j+1}, \ldots, v_k$ cannot repair the problem faced by v_{k+1}.

Skip all these intermediate nodes, and backtrack directly to v_j.

This is called **backjumping**.

Backjumping

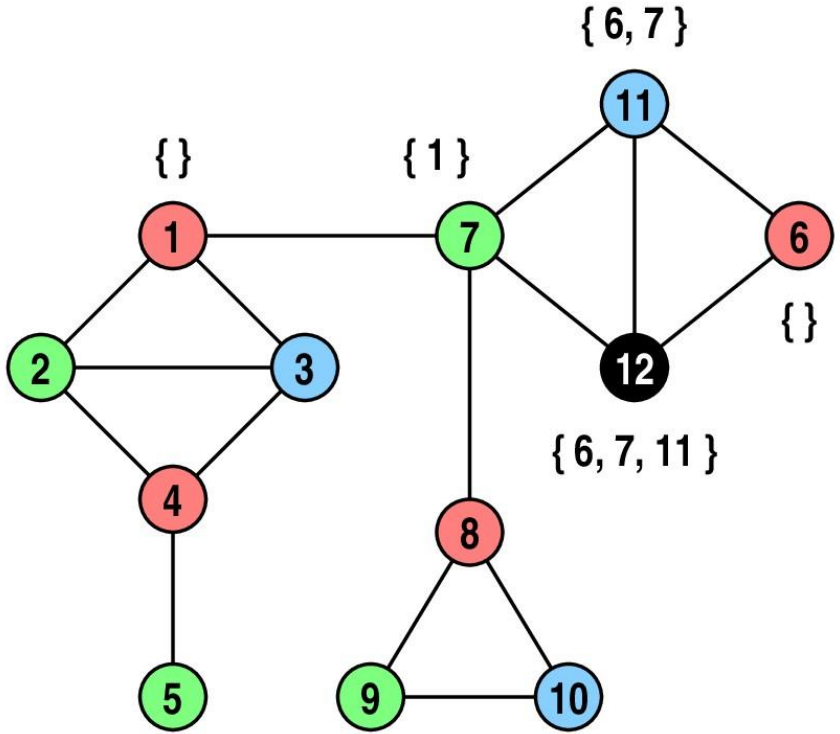
Search fails at 12.

Backjump to 11. No color left for 11.

Backjump to 7 (skipping 8, 9, 10).

Color 7 by B.

Restart coloring the remaining nodes.



Backjumping

This does not repair the problem.

Search fails again at 12.

Backjump to 11. No color left for 11.

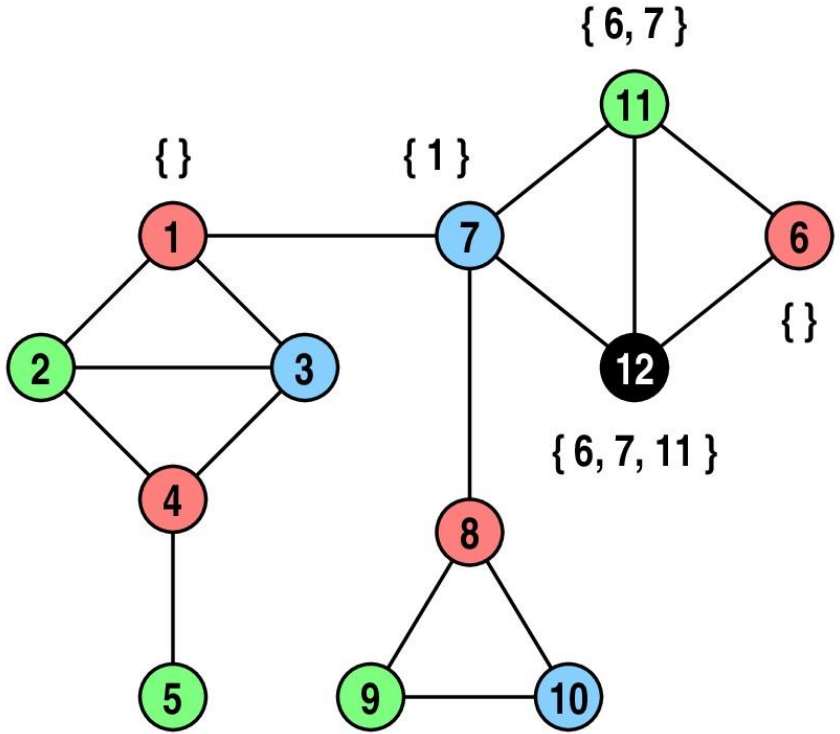
Backjump to 7 (skipping 8, 9, 10).

No color left for 7.

Backjump to 1 (skipping 2, 3, 4, 5).

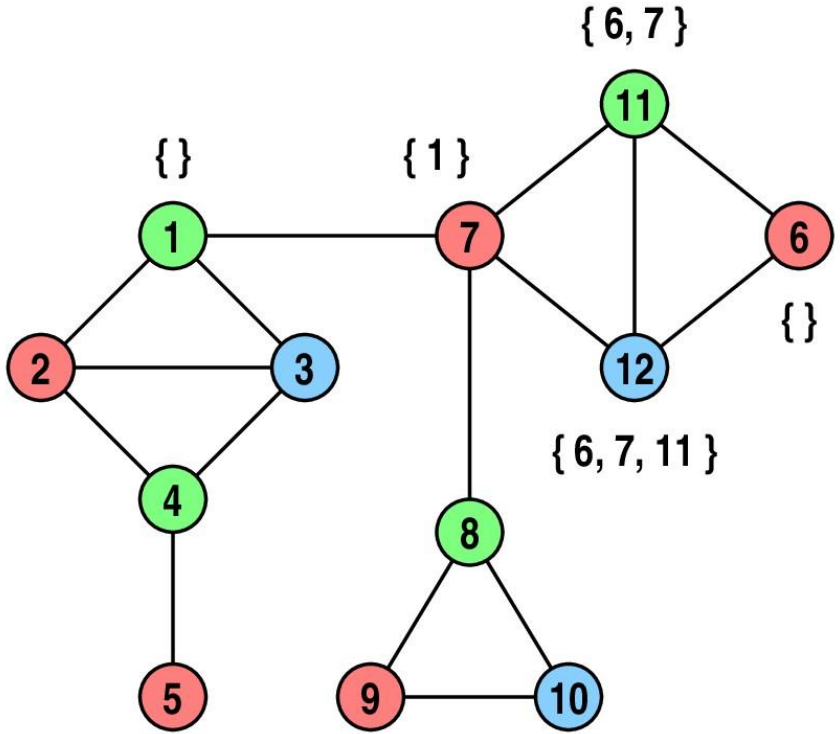
Recolor 1 by G.

Restart coloring the remaining nodes.



Backjumping

Problem solved at last.



A relook into the first failure situation

1 is not the culprit.

When backjumping comes to 7, the real problem is 6. But 7 has no direct conflict with 6.

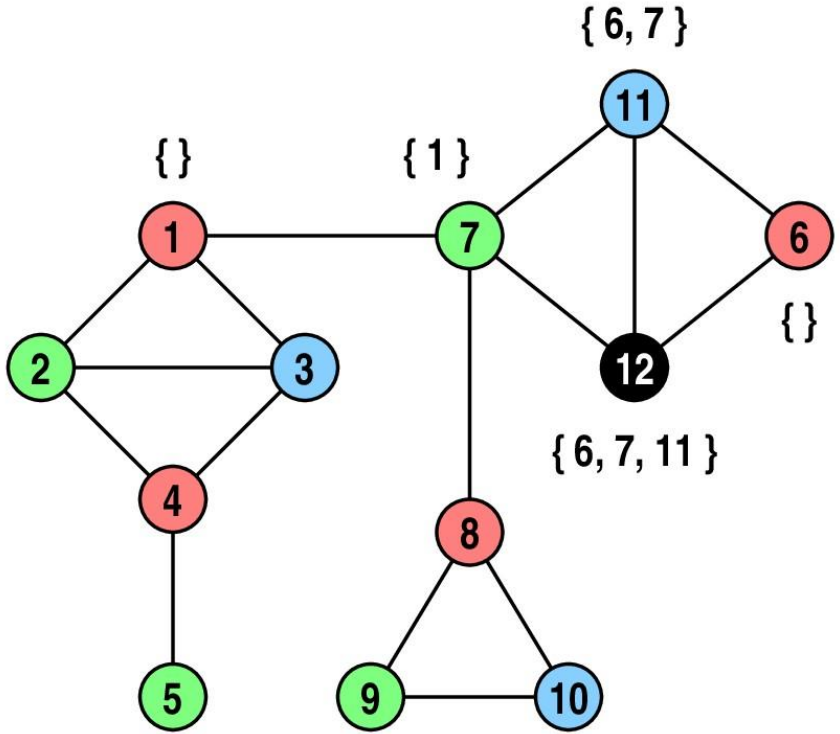
However, 7 has a conflict with the triangle formed by 6, 11, 12.

The nodes must pass on their personal conflicts to the nodes where they backjump to.

Conflict-directed backjumping (CBJ)

If we make a backjump from Y to X, then we update $C(X) = \big(C(X) \cup C(Y) \big) - \{ X \}$.

X absorbs the conflict set from Y. This is a kind of **learning while searching**.



Conflict-directed backjumping (CBJ)

Backjump to 11: Update

$C(11) = \{ 6, 7 \} \cup \{ 6, 7, 11 \} - \{ 11 \} = \{ 6, 7 \}$

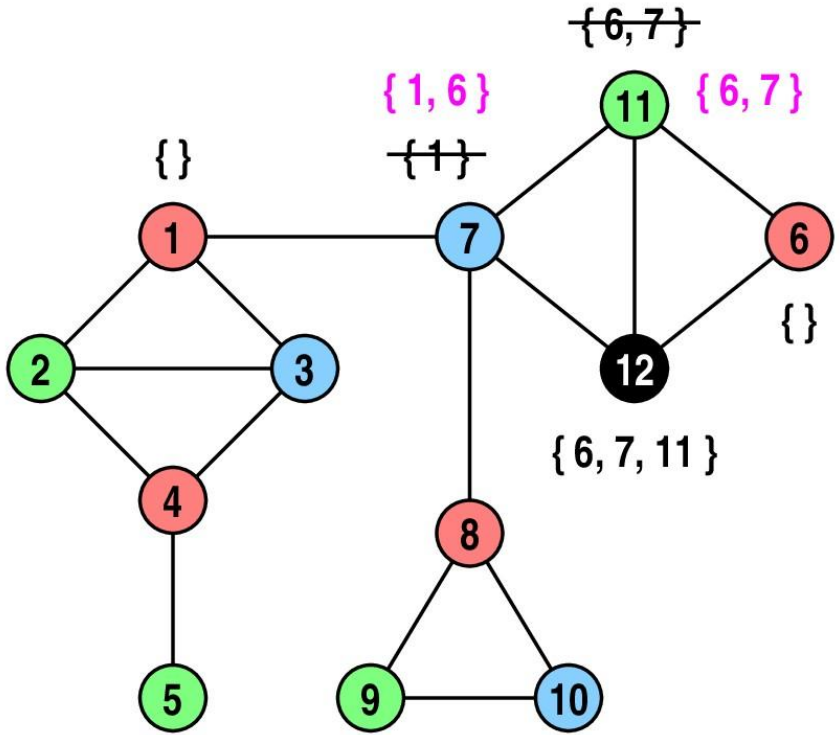
Backjump to 7: Update

$C(7) = \{ 1 \} \cup \{ 6, 7 \} - \{ 7 \} = \{ 1, 6 \}.$

7 has one color left. Color 7 by B.

Restart coloring the remaining nodes.

Search fails again at 12.



Conflict-directed backjumping (CBJ)

Backjump first to 11 and then to 7, and update $C(11)$ and $C(7)$ as before.

Now, 7 has no more options left.

Backjump to 6. Update

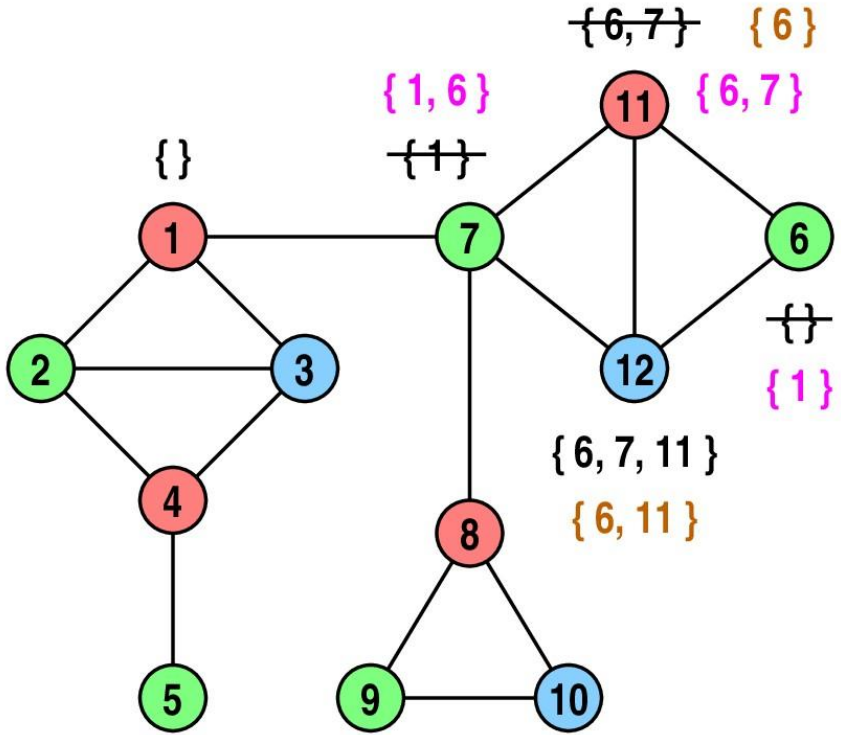
$C(1) = \{\} \cup \{1, 6\} - \{6\} = \{1\}$.

Recolor 6 by G.

Restart numbering the remaining nodes.

Done.

There was no need to backjump to 1.



Local Search for CSP

Min-conflicts heuristic

Start with a complete random assignment *A* of all the variables.

While (terminal condition is not reached), repeat:

> If *A* is a consistent assignment, return *A*.
> Randomly choose a conflicted variable *X*. Let its current assignment be $X = a$.
> Find the value *b* from the domain of *X*, which has the minimum conflict with other variables.
> If $b \neq a$, change the assignment of *X* to $X = b$.

This algorithm works quite well for many CSPs

Variants

> **Sideways movement:** Prevent cycling by remembering recently visited states (**tabu search**)
> **Simulated annealing**
> **Constraints weighting:** Assign weight to constraints.