Five men who are originally from different countries live in consecutive houses on a street. **Given** are each man's **job**, the colors their houses are **painted**, the favorite **drink** of each household, and the **pets** they keep.
We want to know: Who owns a zebra? And whose favorite drink is mineral water?
The following rules (constraints) are provided:

- The English man lives in a red house
- The Spaniard owns a dog
- The Japanese man is a painter
- The Italian drinks tea
- The Norwegian lives in the first house on the left
- The green house immediately to the right of the white one
- The photographer breeds snails
- The diplomat lives in the yellow house
- Milk is drunk in the middle house
- The owner of the green house drinks coffee
- The Norwegian's house is next to the blue one
- The violinist drinks orange juice
- The fox is in a house that is next to that of the physician
- The horse is in a house next to that of the diplomat

House	**FL**: Far Left	**L**: Left	**M**: Middle	**R**: Right	**FR**: Far Right
Nation (N)	**N1**: English	**N2**: Spanish	**N3**: Japanese	**N4**: Italian	**N5**: Norwegian
Color (C)	**C1**: Red	**C2**: Green	**C3**: White	**C4**: Yellow	**C5**: Blue
Drink (D)	**D1**: Tea	**D2**: Orange Juice	**D3**: Milk	**D4**: Coffee	**D5**: Mineral Water
Pet (P)	**P1**: Dog	**P2**: Fox	**P3**: Horse	**P4**: Snails	**P5**: Zebra
Job (J)	**J1**: Painter	**J2**: Diplomat	**J3**: Physician	**J4**: Violinist	**J5**: Photographer

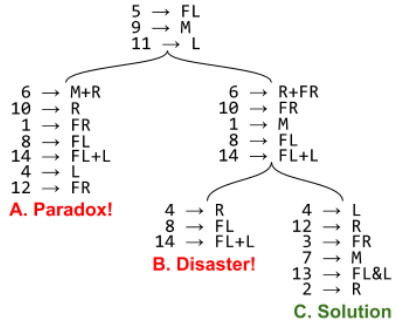
House	FL	L	M	R	FR
Nation	N5	N4	N1	N2	N3
Color	C4	C3	C1	C2	C5
Drink	D5	D1	D3	D2	D4
Pet	P2	P5	P4	P1	P3
Job	J2	J5	J5	J3	J1

The English man lives in a red house	N1 = C1
The Spaniard owns a dog	N2 = P1
The Japanese man is a painter	N3 = J1
The Italian drinks tea	N4 = D1
The Norwegian lives in the first house on the left	H1 = N5
The green house immediately to the right of the white one	C2 = Hx & C3 = Hx+1
The photographer breeds snails	J5 = P4
The diplomat lives in the yellow house	C4 = J2
Milk is drunk in the middle house	H3 = D3
The owner of the green house drinks coffee	C2 = D4
The Norwegian's house is next to the blue one	N5 = Hx & C5 = Hx±1
The violinist drinks orange juice	J4 = D2
The fox is in a house that is next to that of the physician	P2 = Hx & J3 = Hx±1
The horse is in a house next to that of the diplomat	P3 = Hx & J2 = Hx±1

Rule	1	2	3	4	5	6		7	8	9	10	11		12	13		14	
House					FL					M								
Nation	N1	N2	N3	N4	N5							N5						
Color	C1					C3	C2		C4		C2		C5					
Drink				D1						D3	D4			D2				
Pet		P1						P4							P2		P3	
Job			J1					J5	J2					J4		J3		J2

Constraint Satisfaction

Rule	1	2	3	4	5	6		7	8	9	10	11		12	13		14	
House					FL					M								
Nation	N1	N2	N3	N4	N5							N5						
Color	C1					C3	C2		C4		C2		C5					
Drink				D1						D3	D4			D2				
Pet		P1						P4							P2		P3	
Job			J1					J5	J2					J4		J3		J2



Zebra Puzzle

- Another application of search is **constraint satisfaction**.
- Rather than searching through states, we search through possible assignments of variables
- More formally:
  - Given a collection of **n variables**, with each variable Vn (V1, V2, etc) having a domain of **possible values** D1, D2, D3, etc., it can take on, and a set of **k** constraints on those possible values C1, C2, … Ck:
  - A *solution* is an **assignment** of values to **variables**, with each value **assigned** being from the correct domain for that variable
  - A solution that assigns a value to all variables is *complete (*otherwise it is *partial)*.
  - A solution is *consistent* **iff** all constraints are satisfied by it
  - A solution is *correct* **iff** it is complete and consistent.
- An advantage of constraint satisfaction is that it's **general**; we can write a general constraint-satisfaction solver and turn it loose on ANY problem that can be formulated appropriately
  - We construct a **constraint graph**, with each node a variable, *connected if they share a constraint*.
  - We can then apply search techniques to that graph
  - We can also go beyond "this isn't a solution, go on to the next state" by **identifying which constraints are violated** & focusing on **assignments** that **satisfy** them
- Many real-world problems fall into this category; for example, observation time on the Hubble telescope involves starting & ending times & telescope orientations, subject to a variety of priority & power constraints

Constraints

- There are several types of constraints:
  - **Unary** constraints apply to 1 variable
  - **Binary** constraints apply to 2 variables
  - **N-ary**, etc., …(follow from here)
  - **Global** constraints apply universally
    - ALLDIFF: each variable must be assigned a *unique* value.
  - There are also softer *preference constraints* in which we have a **preferred** state and apply a **penalty (cost)** to other options
    - I'd prefer to fly in the **afternoon**, but will fly in the **morning** if it'll save me $100 or more.
    - Prof. Smith prefers to teach in the **morning**, Prof. Jones in the **afternoon**. A schedule with both of them teaching in the afternoon is valid but ***not optimal***.

Constraint Propagation & Inference

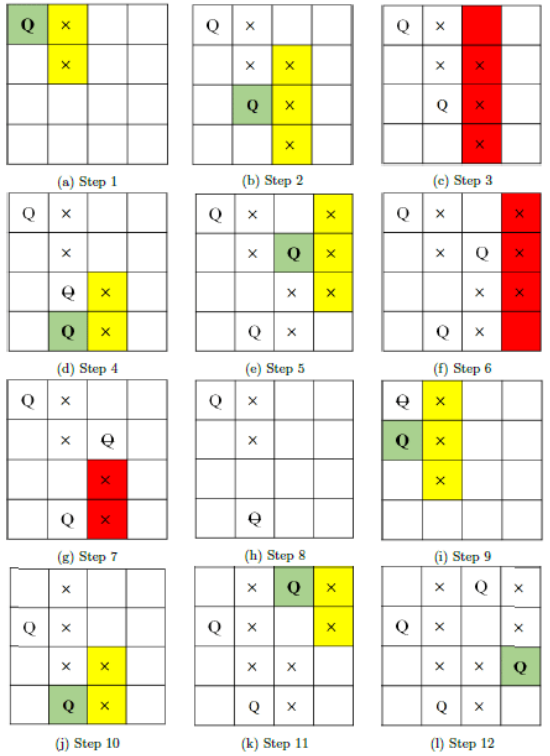
- In *Constraint Satisfaction Problems* (CSPs), an algorithm can search (*try* another assignment of values to variables) or do a type of **inference** called **constraint propagation**
  - Using the constraints ***already known*** to reduce the number of legal values for a variable, which in turn might limit possible values for other variables.
- Key idea is **local consistency**
  - Each variable is a node on a constraint graph.
  - Enforcing local consistency throughout the constraint graph causes inconsistent values to be eliminated
- A variable is **node-consistent** if all values in that variable's **domain** satisfy the variable's unary constraints
  - Node consistency can be enforced **one variable at a time**.
  - N-ary constraints can be made into (decomposed)**binary** constraints
  - Therefore most CSP solvers only work with binary constraints
  - But we can generalize this concept to **hyperarc consistency** or **generalized arc consistency** to deal with multiple values
- A variable is **arc-consistent** if every value in that variable's domain satisfies the variable's binary constraints
  - Arc consistency **eliminates** values of each variable domain that can ***never satisfy a particular constraint*** (an **arc**).
  - More formally, X_i is **arc**-consistent with respect to X_j if **for every value in D_i** there **exists some value** D_j that **satisfies** the binary constraint
- A network is arc-consistent if every variable is arc-consistent with respect to every other variable
  - For example, suppose X and Y are integers in {1,2,3…9,10} and we add the constraint $Y = X^2$.
  - We can make this arc-consistent if we reduce the range of X to {1, 2, 3} (to make X arc-consistent with Y) and Y to {1, 4, 9} (to make Y arc-consistent with X).

AC-3

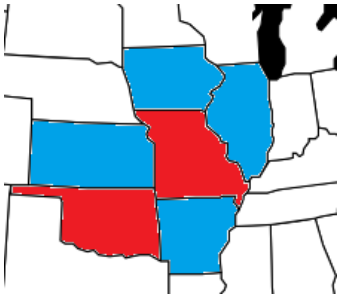
- The most popular algorithm for arc consistency is AC 3
  - Maintain a **queue** (or set) of **arcs** to consider;
    (initially this is all arcs in the CSP)
    - Binary constraints become *two* arcs, one in each direction

Take an arbitrary arc (X_i, X_j) from the queue
  make X_i arc-consistent with X_j
  If this has no effect on D_i
    go on to the next arc
  But if this reduces D_i,
    add to the queue all arcs (X_k, X_i) [where X_k is a neighbor of X_i]
    (The reduction in D_i might affect D_k even if we've already processed D_k)
  If D_k is reduced down to nothing
    CSP has no consistent solution
      return **failure**
Otherwise continue (until no more arcs are in queue)

- At this point we have a CSP equivalent to the original—it has the same solutions—but it has smaller domains and so can be searched faster
- If variables all have domains of size at most *d* and there are *c* binary constraints, checking consistency of an arc takes at most time $O(d^2)$ and so we get total worst-case time of $O(cd^3)$



Path consistency



- Arc consistency can go a long way (sometimes all the way) to a solution
  - Solution, by reducing every **domain** size to **1**
  - OR -- Verifying that there is **no solution**, by reducing some **domain** size to **0**
- But sometimes not enough
- Consider the map coloring with only R, B
- Every variable can be arc-consistent (R at one end, B at the other) but still no solution
  - Since MO, IL, KY all touch each other, we need 3 colors just for them
- Path consistency **tightens** binary constraints by looking at **triples**
  - A 2-variable set $\{X_i, X_j\}$ is path-consistent with respect to a third variable X_m
    - if for every assignment $\{X_i = a, X_j = b\}$ consistent with binary constraints on X_i and X_j
      - there is assignment to X_m that satisfies binary constraints on (X_i, X_m) and (X_m, X_j)
  - That is, there is an assignment for all variables that makes a path from X_i to X_j passing through X_m consistent with all binary constraints on all 3 variables
- So, for example, we try the 2-color map problem for MO, IL, IN, KY.
- Possible solutions are {MO = R, IL = B} and {IN = R, IL = B} So far so good.
- But with both of those assignments, KY can be neither R nor B because it would conflict with one or the other
- Therefore there is no valid option for KY. The domain is of size 0, which tells us this problem has no solution.
- The PC-2 algorithm applies the same basic idea as the AC-3 algorithm.

K-consistency

- Taking this further, a CSP is **K-consistent** if for any set of K-1 variables and for any consistent assignment to those variables, a consistent value can be assigned to the **k**th variable.
  - 1-consistency says given the empty set we can make any one variable consistent (node consistency)
  - 2-consistency is the same as **arc** consistency
  - **3**-consistency is the same as **path** consistency
- A CSP is **strongly K-consistent** if it is k-consistent and also (k-1) consistent, (k-2) consistent, etc all the way down to node consistent.
- If we have a graph of N nodes and make it strongly N consistent, we have a solution
  - Choose any consistent value for X1.
  - We are guaranteed to be able to choose a consistent value for X2 because it is 2-consistent
  - We can also choose a consistent value for X3 because it is 3-consistent
  - And so on to Xn. Worst case time is $O(n^2d)$
- No free lunch! Establishing strong N-consistency takes exponential time and space in worst case
- In practice, we commonly compute 2-consistency & sometimes 3-consistency, then fall back on search

Global constraints

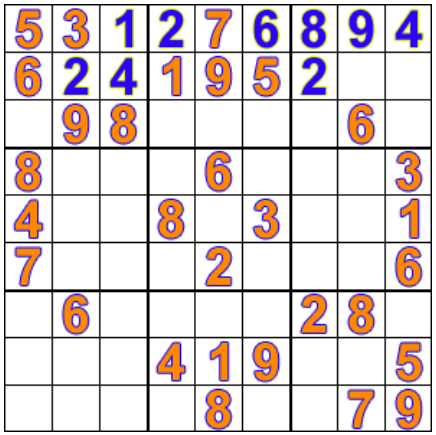
- Recall that a global constraint is one involving an arbitrary number of variables (not necessarily all)
- Special algorithms can be used to reduce the search space or detect inconsistency
  - Consider the Alldiff constraint. If we have **M** values involved in the constraint, and there are a total of **N** possible values among all of them, and **M** > **N**, the constraint cannot be satisfied
  - This leads to a simple algorithm:
    - **If any variable has a singleton domain** (1 possible value), assign **that** *value* to **that** *variable*, and *remove that value from the domain of the remaining variables*
    - Repeat as long as there are singletons.
    - If an empty domain ever occurs or there are more variables than domain values, an inconsistency has been detected

Other constraints

- We also have the **resource constraint**, also known as the **atmost** constraint.
  - We have **4** tasks {P1,P2,P3,P4} and 10 employees.
  - We can specify this as **atmost**(10, P1, P2, P3, P4)
  - Check the sum of the minimum values for each job
    - If each variable has the domain {3, 4, 5, 6} (need **at least** 3 employees for each job), **the constraint cannot be satisfied**
  - We can also enforce **consistency** by deleting **maximum values** if they are inconsistent with **minima**
    - If each variable has range {2, 3, 4, 5, 6}, we can delete values 5 & 6 (after assigning minimum numbers, there aren't enough workers left to put 5 or more on 1 job)

- A CSP is **bounds consistent** if for every variable X and both the lower and upper bounds of X, there exists some value of Y that satisfies the constraint between X and Y for every variable Y.

Backtracking search for CSPs



- Some problems (e.g. Sudoku) can be solved entirely via constraint satisfaction
- But sometimes after satisfying all constraints we do not have a unique solution
- Could apply depth-limited search
  - A state is a partial assignment
  - A step is adding var = value
- But this is problematic
  - The branching factor at top level is nd (assign any of d values to any of n variables)
  - At the next level it's (n-1)d, and so on for n levels
  - Thus we generate n! * d^n leaves even though there are only d^n possible assignments
- But this ignores commutativity
  - A problem is **commutative** if the order of application of any given set of actions has no effect on the outcome
- CSP problems are commutative because when assigning values to variables we reach the same partial assignments regardless of application order
- So we only need to consider *one* variable at each node in the search tree
  - We might choose between KY = red, KY = green, and KY = blue, but not between KY = red and MO = blue
- With this restriction, the number of leaves is d^n.

Backtracking search

- A **backtracking search** is a *depth-first search* that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign
  - Choose a variable, then try all values in the domain of that variable in turn
  - If an inconsistency is found, then it returns a failure, causing the previous call to try another value
- With uninformed depth-first search we could speed up search by applying problem-specific heuristics. CSPs can be sped up without problem-specific heuristics, by addressing these questions:
  - Which variable should be assigned next?
  - In what order should values be tried?
  - What inferences should be performed at each step in the search?
  - When the search arrives at an assignment that violates a constraint, can the search avoid repeating this failure?

Variable Ordering

- Simplest method is to choose next unassigned variable
  - But this seldom leads to the most efficient search
  - For example, in our map coloring, after MO = red and IL = green, there is only 1 possible value for KY, so it makes sense to assign KY = blue immediately
    - And after that, assignment for IN is also forced, with only 1 possible value
- This is the **minimum-remaining-values** heuristic: Choosing the variable with the fewest legal values
  - Also known as “most constrained variable” or “fail-first” heuristic
  - If some variable has 0 values left, it will be picked next and the failure detected immediately
- Performs better than a random or static ordering, sometimes by a factor of 1000 or more, but it depends quite a bit on the problem
- But MRV is no help choosing the *first* variable. In this case we use the **degree heuristic**, choosing the variable involved in the most binary constraints
  - Again, the idea is to reduce possible values in other variables as quickly as possible
- MRV is more powerful; degree heuristic is a useful tie-breaker

Value ordering

- Once we've selected a variable, what order do we try to assign its values?
- The **least-constraining-value** heuristic prefers to use the value that rules out the fewest values in surrounding variables in the constraint graph
  - In general, we want the maximum flexibility for future assignments
  - Of course, if we're looking for ALL solutions, not just the first, this doesn't matter since we have to try them all eventually anyway
  - Likewise if there are no solutions
- *Choosing variables is fail-first (most constrained variable), choosing values is fail-last (least constraining value)*
  - Choosing variables fail-first prunes off entire branches of the search tree earlier
  - For values, we only need to find one solution, so we look for the most likely

Local search for CSP

- Local search techniques can be very effective for CSPs
- Use a complete-state formulation, and start in a random state
- Select a variable and search for a value that reduces the number of conflicts and constraint violations
  - The obvious heuristic is **min-conflicts**, the value that results in the minimum number of conflicts. If more than one value gives same minimum number of conflicts, choose arbitrarily
- For n-queens, the solution is approximately independent of problem size, solving the 1,000,000 queens problem in 50 steps after initial assignment.
  - For this problem, solutions are dense throughout the search space
- This has also been used in practical problems.
  - Scheduling observation time for the Hubble telescope used to take 3 weeks to schedule 1 week's observations; now takes 10 minutes

Local search for CSP

- Most of the local search methods we looked at can be used for CSP
- CSP problems often have a series of plateaus, states 1 conflict away from a solution
  - Allowing sideways movement to other states with the same score allows exploring this space
  - Maintaining a **taboo list** of recently-visited states & not allowing return to them can prevent aimless wandering
  - Simulated annealing can also get a search off a plateau
- Another approach is **constraint weighting**
  - Start with all constraints having a weight of 1
  - At each step, choose variable/value that minimizes total weights of violated constraints
  - Increment the weight of all constraints still violated
  - This adds topography, ensuring progress is possible, and adds weight to constraints that are proving difficult to satisfy
- Another advantage of local search is that it minimizes the number of changes, useful in online or dynamic situations
  - Airline scheduling involves elaborate constraints of time, personnel, equipment, etc.
  - If something changes (bad weather at an airport, crew member ill, etc) we must find another solution
  - Local search will find something requiring only a few changes; re-starting the assignments from scratch might find a valid solution requiring dozens (or hundreds!) of changes

AND-OR trees

- For some problems, we have choices which might be equally valid, or some things that must all be done, and other choices of which any one would work
  - We want dinner AND a movie; for dinner we can have pizza OR middle eastern OR Thai; what's within walking distance?
    - Any of the restaurant options can be chosen as part of a solution
    - If a restaurant AND movie theater aren't within the specified range, there is no solution to the overall problem no matter how good the restaurant is, even if there's a pizza place, a middle eastern place, and a Thai place all right across the street
- It can be represented with an AND-OR tree
- A node in such a tree is solvable if
  - It is a terminal node
  - It is nonterminal whose successors are AND nodes that are all solvable or
  - It is nonterminal whose successors are OR nodes and at least one of them is solvable
- Sample AND-OR tree, p. 101

Heuristic Bidirectional Search

- A bidirectional search can be more time effective, if the two parallel searches actually intersect
- A naïve bidirectional search can suffer from the missile metaphor problem: A missile and anti-missile can be targeted toward each other, and both miss
- However, this is not a general problem; as it turns out, wave-shaping algorithms can be used to direct the two searches toward each other (diagram, p. 102)
- Rather than front-to-end heuristic estimates, this method uses front to front estimates—heuristic cost of a path from some node in the "source" front to some node in the "goal" front
- This helps address the frontiers problem—as each search front grows, storage space becomes a problem as the two fronts try to find each other
- Or a frontier search is carried out
  - A node is selected and a BFS out a certain distance from it is carried out, and all nodes stored (usually in a hash table)
  - A forward search then starts from source S, targeting the perimeter nodes, using A* or ID-DFS
  - Likewise, a backward search from goal G, targeting the perimeter, can also be carried out
  - This results in opening about ¼ the nodes of a unidirectional search
  - While there is no connection between the S and G searches, their intersection is the empty set.
  - Once they collide, a search is carried out to find the optimal path within that (smaller) intersection set