Chapter 6: Constraint Satisfaction Problems

These slides are adopted from Berkeley course materials and Russell and Norvig textbook

Announcement

- The midterm exam will be on Thursday 1 December, at 11 am.

Constraint Satisfaction Problems

- Standard search problems studied so far use atomic states (a black box with no internal structure.)
- Constraint satisfaction problems are a special kind of search problems that use a factored representation for the state space.
  - For each state: there is a set of variables, each of which has a value.
  - A problem is solved when each variable has a value that satisfies all the constraints on the variables.

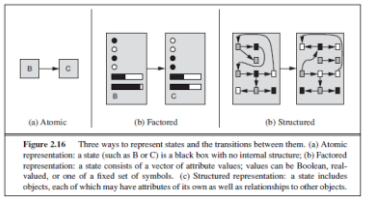


Figure 2.16 Three ways to represent states and the transitions between them. (a) Atomic representation: a state (such as B or C) is a black box with no internal structure; (b) Factored representation: a state consists of a vector of attribute values; values can be Boolean, real-valued, or one of a fixed set of symbols. (c) Structured representation: a state includes objects, each of which may have attributes of its own as well as relationships to other objects.

kol state 3ndha set of variables, w kol variable leh value.
dayman bn7awl enna n7ot values t7a2a2 kol el shrot m3 kol el variable el tanya

Constraint Satisfaction Problems

A constraint satisfaction problem consists of three components: X,D, and C:

- X is a set of variables, {X1, . . . ,Xn}.
- D is a set of domains, {D1, . . . ,Dn}, one for each variable.
- C is a set of constraints that specify allowable combinations of values.

- Allows useful general-purpose algorithms with more power than standard search algorithms.
- The main idea is to eliminate large portions of the search space all at once by identifying variable/value combinations that violate the constraints.

Goal test has structure, it ‘s more like a manual describing a set of constraints that have to hold true

Constraint Satisfaction Problems

- Each state in a CSP is defined by an **assignment** of values to some or all of the variables, {Xi =vi,Xj = vj , . . .}.
- A **consistent** assignment is an assignment that does not violate any constraints.
- A **complete assignment** is one in which every variable is assigned.
- A **solution** to a CSP is a consistent, complete assignment.
- A **partial assignment** is one that assigns values to only some of the variables.

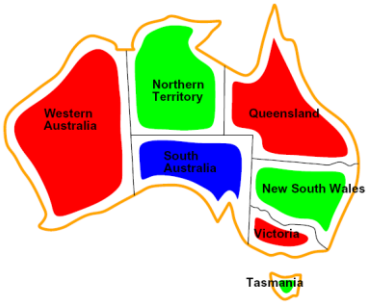
ex: $x^2 + y^2 = 16$

x:1, y:2 => complete assignment

x:1 -> partial assignment

x: 4, y:0 -> solution

CSP Examples: Map coloring



Example: Map Coloring

Variables: WA, NT, Q, NSW, V, SA, T

Domains: $D = \{red, green, blue\}$

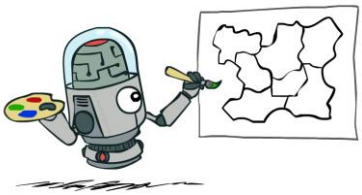
Constraints: adjacent regions must have different colors

Implicit: $WA \neq NT$

Explicit: $(WA, NT) \in \{(red, green), (red, blue), \ldots\}$

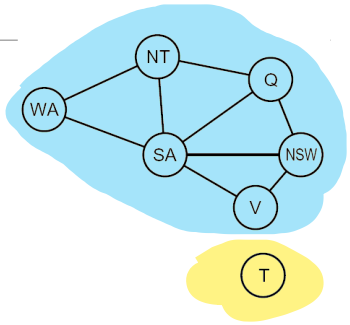
Solutions are assignments satisfying all constraints, e.g.:

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



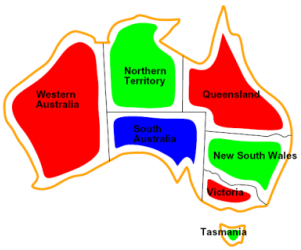
Constraint Graphs

- Binary CSP: each constraint relates (at most) two variables.
- Binary constraint graph: nodes are variables, arcs show constraints.
- General-purpose CSP algorithms use the graph structure to speed up search. E.g., Tasmania is an independent subproblem!



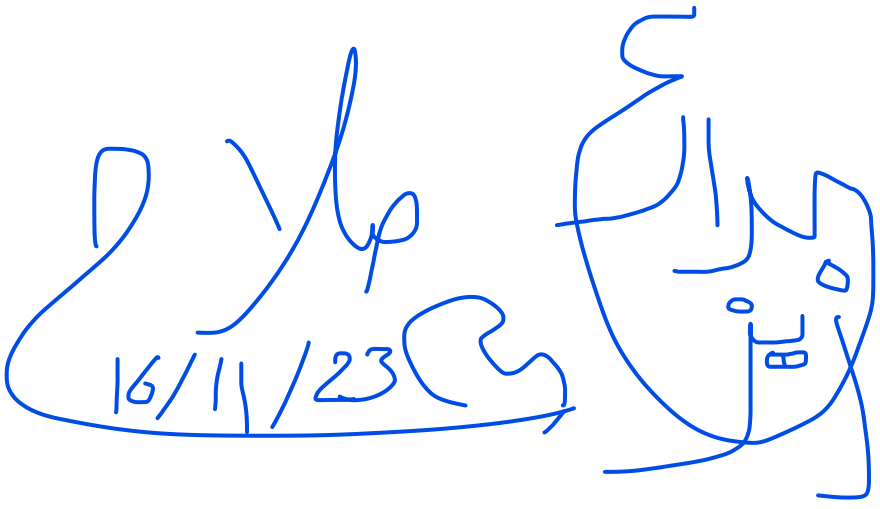
Why formulate the problem as a CSP?

- CSP solvers can be faster than state-space searchers because the CSP solver can quickly eliminate large portions of the search space.
- For example, assume {SA=blue} in the Australia map coloring problem, we can conclude that none of the five neighboring variables can take on the value blue.
- Without taking advantage of constraint propagation, a search procedure would have to consider $3^5 = 243$ assignments for the five neighboring variables.
- With constraint propagation we never have to consider blue as a value, so we have only $2^5 = 32$ assignments to look at, a reduction of 87%.



el fekra enk lama byb2a 3ndk relation
bt2dr te3ml eliminate le invalde choices

3la 3ks lw mkontsh 3arf el aslun en feh 3laka, w tgrb blindly.



Why formulate the problem as a CSP? (cont.)

- With CSPs, once we find out that a partial assignment is not a solution, we can immediately discard further refinements of the partial assignment.

Example: Cryptarithmetic

Variables:

carry

$$F \; T \; U \; W \; R \; O \; X_1 \; X_2 \; X_3$$

Domains:

$$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

Constraints:

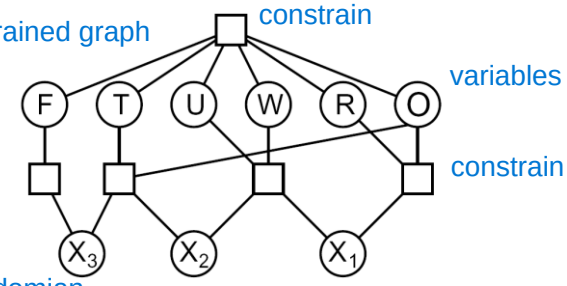
$$\text{alldiff}(F, T, U, W, R, O)$$

$$\ldots \quad O + O = R + 10 \cdot X_1$$

ana de el 7aga elly 3auz awslha,
msh m3naha en 5 + 5 = R + 10x1
kol 7rf ye2dr y5ud kema mokhtlfa mn el domian

$$\begin{array}{r} \text{T W O} \\ + \; \text{T W O} \\ \hline \text{F O U R} \end{array}$$

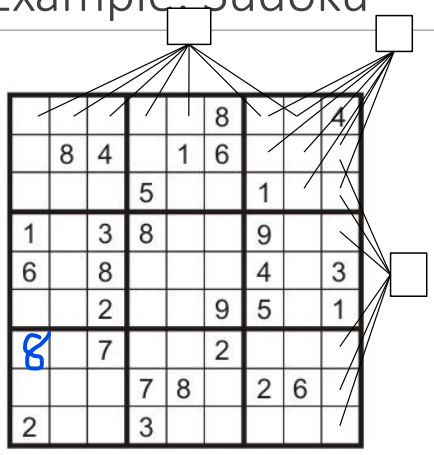
constrained graph



0=7, R=4, W=6, U=2, T=8, F=1; 867 + 867 = 1734

efhmo kwys, 34an hyegy fl emt7an

Example: Sudoku



- Variables:
  - Each (open) square
- Domains:
  - {1,2,…,9}
- Constraints:

kol mkan fady hwa variable.

de el arkam elly a2dr a3wd beha.

9-way alldiff for each column

9-way alldiff for each row

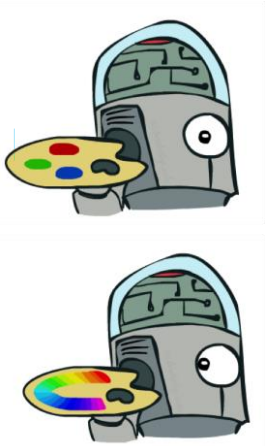
9-way alldiff for each region

(or can have a bunch of pairwise inequality constraints)

Varieties of CSPs

Discrete Variables
- Finite domains
  - Size d means $O(d^n)$ complete assignments
  - E.g., Boolean CSPs, including Boolean satisfiability (NP-complete)
- Infinite domains (integers, strings, etc.)
  - E.g., job scheduling, variables are start/end times for each job
  - Linear constraints solvable, nonlinear undecidable

Continuous variables
- E.g., start/end times for Hubble Telescope observations
- Linear constraints solvable in polynomial time by Linear Programming methods



linear programming problems, where constraints must be linear equalities or inequalities. Linear programming problems can be solved in time polynomial in the number of variables

Varieties of Constraints

Varieties of Constraints
- Unary constraints involve a single variable (equivalent to reducing domains), e.g.:

$$SA \neq green$$

- Binary constraints involve pairs of variables, e.g.:

$$SA \neq WA$$

- Higher-order constraints involve 3 or more variables:
  e.g., cryptarithmetic column constraints

Preferences (soft constraints):
- E.g., red is better than green for a certain city in Australia map coloring or Prof. A prefers his lecture in the morning.
- Often representable by a cost for each variable assignment

Real-World CSPs

Assignment problems: e.g., who teaches what class

Timetabling problems: e.g., which class is offered when and where?

Hardware configuration

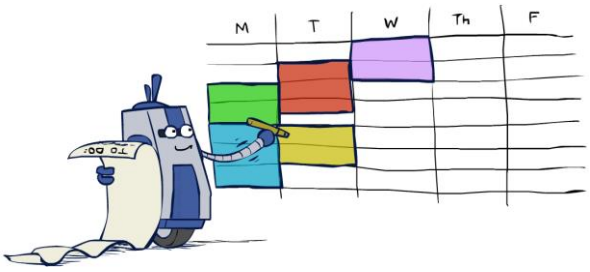
Transportation scheduling

Factory scheduling

Circuit layout

Fault diagnosis

… lots more!

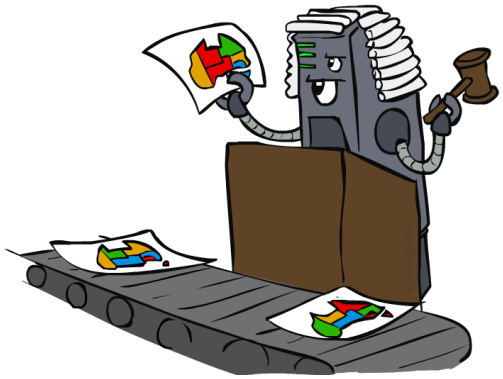


Many real-world problems involve real-valued variables…

Standard Search Formulation

- Standard search formulation of CSPs
- States defined by the values assigned so far (partial assignments)
- Initial state: the empty assignment, {}
- Successor function: assign a value to an unassigned variable.
- Goal test: the current assignment is complete and satisfies all constraints.

- We'll start with the straightforward, naïve approach, then improve it.



Standard Search Algorithms

- Assume a CSP problem with n variables, each of domain size d.
- The branching factor at the top level is nd because any of d values can be assigned to any of n variables.
- At the next level, the branching factor is (n − 1)d, and so on for n levels.
- We generate a tree with $n!d^n$ leaves, even though there are only d^n possible complete assignments!
- Can we improve this?
  - Yes, exploit CSP **commutativity**.
  - CSPs are commutative because the order of assigning values to variables does not change the outcome (the partial assignment).
  - For example, [WA = red then NT = green] is the same as [NT = green then WA = red]
  - Therefore, we need only consider a *single* variable at each node in the search tree (number of leaves would be d^n).

Backtracking Search for CSP

- Shall you choose BFS or DFS for solving a CSP problem?
- **Backtracking search** is a depth-first search with two improvements for using in CSP:
  - It chooses values for one variable at a time and backtracks when a variable has no legal values left to assign.
  - It checks the constraints by considering only values which do not conflict previous assignments
    - Might have to do some computation to check the constraints
    - "Incremental goal test"

Backtracking Search Algorithm

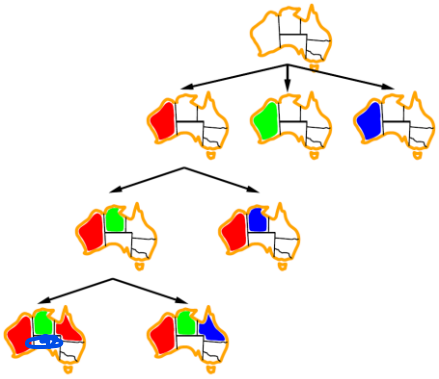
```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return BACKTRACK({}, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment then
      add {var = value} to assignment
      inferences ← INFERENCE(csp, var, value)
      if inferences ≠ failure then
        add inferences to assignment
        result ← BACKTRACK(assignment, csp)
        if result ≠ failure then
          return result
    remove {var = value} and inferences from assignment
  return failure
```

by7awl yshof hl feh 7aga momken n3mlha assign mn gher search.

It repeatedly chooses an unassigned variable, and then tries all values in the domain of that variable in turn, trying to find a solution. If an inconsistency is detected, then BACKTRACK returns failure, causing the previous call to try another value

Backtracking- map coloring example



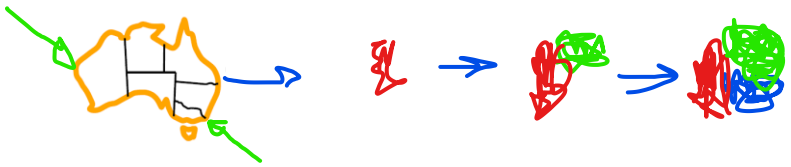
Improving Backtracking

- Ordering
  - Which variable should be assigned next ?
  - What order should its values be tried?
- Inference
  - Can we detect failures earlier?

In Chapter 3 we improved the poor performance of uninformed search algorithms by supplying them with domain-specific heuristic functions derived from our knowledge of the problem. It turns out that we can solve CSPs efficiently *without* such domain-specific knowledge.

Variable Ordering

- Variable Ordering: Minimum remaining values (MRV):
  - Choose the variable with the fewest legal left values in its domain



- Why min rather than max?
- Also called "most constrained variable"
- "Fail-fast" ordering as it picks a variable that is most likely to cause a failure soon, thereby pruning the search tree.
- The MRV heuristic usually performs better than a random or static ordering.

If some variable X has no legal values left, the MRV heuristic will select X and failure will be detected immediately—avoiding pointless searches through other variables.

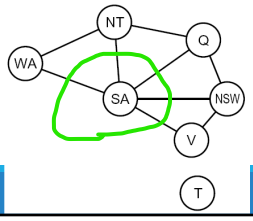
Variable Ordering

- The MRV heuristic doesn't help in choosing the first region to color in Australia (each region has three possible colors).
- **Degree Heuristic:**

Choose the variable with the highest degree in the constraint search graph

- The degree heuristic selects the variable involved in the largest number of constraints on other unassigned variables.
- In the map coloring example, which variable to choose at first according to the degree heuristic?

aktur wa7ed 3ndo edges.



, SA is the variable with highest degree, 5; the other variables have degree 2 or 3, except for T, which has degree 0

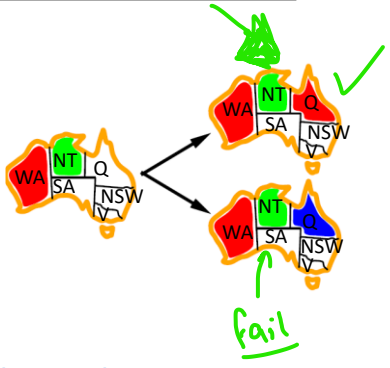
Value Ordering

Which value shall we choose for coloring Q red or blue? Why?

Value Ordering: Least Constraining Value
- Given a choice of variable, choose the *least constraining value*
- I.e., the one that rules out the fewest values in the remaining variables
- Note that it may take some computation to determine this!

Why least rather than most?
- To leave the maximum flexibility for subsequent variable assignments.

Combining these ordering ideas makes
1000 queens feasible



e5tar el kema elly tseb beha akbur options le gerank.
3n tre2 enk takhud el kema w tro7 ts2l gerank, lw khdt de ento hytfdlko ad a

For example, suppose that in Figure 6.1 we have generated the partial assignment with WA=red and NT =green and that our next choice is for Q. Blue would be a bad choice because it eliminates the last legal value left for Q's neighbor, SA. The least-constraining-value heuristic therefore prefers red to blue.

Constraint Propagation

- In regular state-space search, an algorithm can do only one thing: search.
- In CSPs, an algorithm can search (choose a new variable assignment from several possibilities) or do a specific type of **inference** called **constraint propagation**.
- **Constraint propagation** is using the constraints to reduce the number of legal values for a variable, which in turn can reduce the legal values for another variable, and so on.
- Constraint propagation may be done with search, or as a preprocessing step, before the search.
- Sometimes this preprocessing can solve the whole problem, so no search is required at all!

Inference: Forward Checking

- Constraint Propagation

kol ma bashel element mn domain, bla2y eny sghr el domain bta3 element tany, b3dha el tany ntegt el tghyer bta3o, asar 3la variable talet, w hakaza.

Using the constraints to reduce the number of legal values for a variable, which in turn can reduce the legal values for another variable, and so on.

- Forward checking:

For each unassigned variable Y that is connected to X by a constraint, delete from Y 's domain any value that is inconsistent with the value chosen for X.

lama t3ml assignment, bos 3la el neigbours bto3o howa bs, w shel mn 3ndhom el option da. w eb2a estfed mno b3den.

da asr3 bkter mn el constraint propagation.

of a value for a variable, we have a brand-new opportunity to infer new domain reductions on the neighboring variables

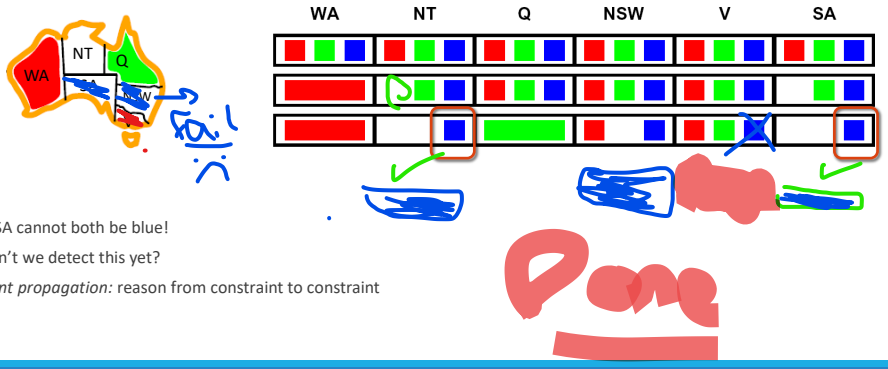
I am trying so hard to succed... but actually I am not tired yet. hope to achieve what I dream at the end...

Inference: Forward Checking Example

[Demo: coloring -- forward checking]

Forward Checking Example (cont.)

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



NT and SA cannot both be blue!

Why didn't we detect this yet?

Constraint propagation: reason from constraint to constraint

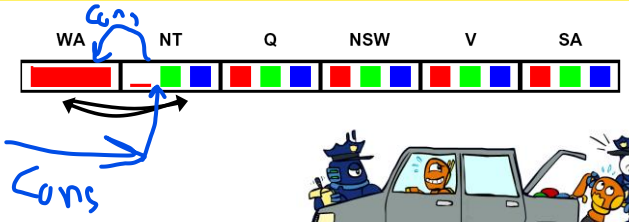
We've already failed, but forward checking does not detect that.

Forward checking shows that when WA is red and Q is green, both NT and SA are forced to be blue. Forward
checking does not look far enough ahead to notice that this is an inconsistency: NT and SA
are adjacent and so cannot have the same value.

lw 3ndy arc mn x l y
yeb2a lazm kol element fe x, ykon fe possible option y5do mn y.

Constraint Propagation: Arc Consistency

An arc X → Y is consistent iff for *every* x in the tail there is *some* y in the head which could be assigned without violating a constraint.



Forward checking?

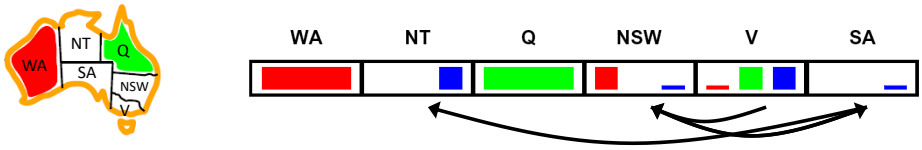
Delete from the tail!

Enforcing consistency of arcs pointing to each new assignment (Y->X) if Y is an unassigned variable, and X is the newly assigned variable.

Mnemonic – csp police checks my trunk

Arc Consistency of an Entire CSP

A simple form of propagation makes sure all arcs are consistent:



Important: If X loses a value, neighbors of X need to be rechecked!

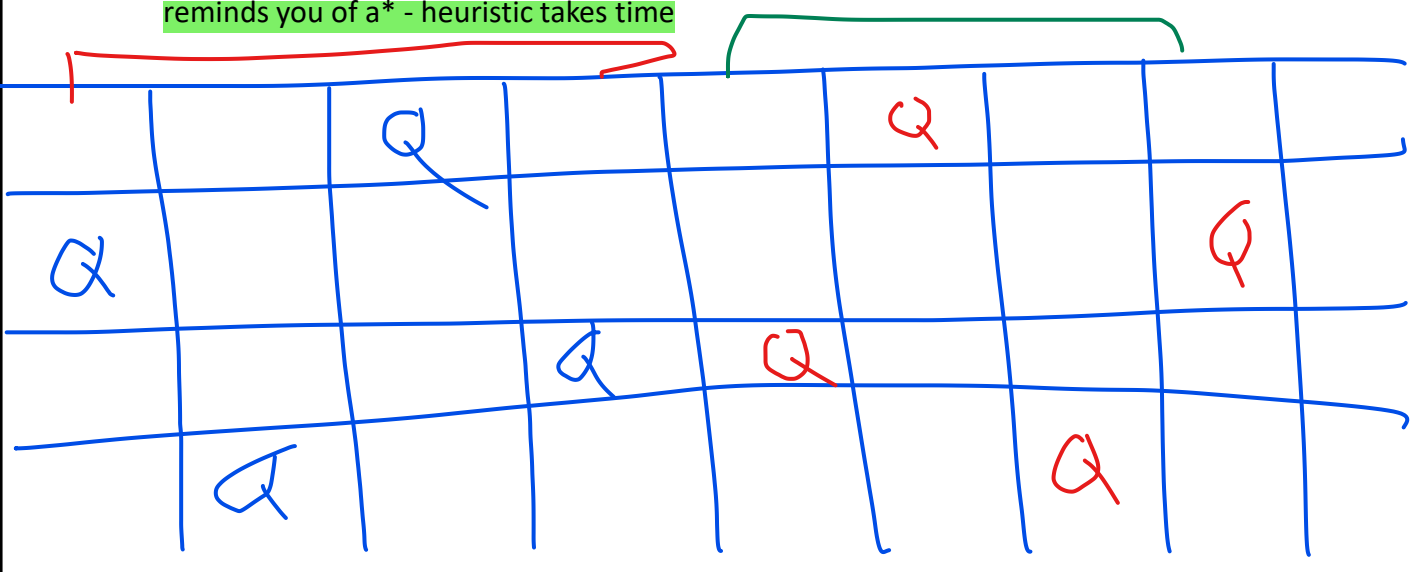
Arc consistency detects failure earlier than forward checking.

Can run as a preprocessor (before the search) or after each assignment.

What's the downside of enforcing arc consistency?

Remember: Delete from the tail!

reminds you of a* - heuristic takes time



Arc Consistency Algorithm in CSP

```
function AC-3(csp) returns false if an inconsistency is found and true otherwise
  inputs: csp, a binary CSP with components (X, D, C)
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (Xi, Xj) ← REMOVE-FIRST(queue)
    if REVISE(csp, Xi, Xj) then
      if size of Di = 0 then return false
      for each Xk in Xi.NEIGHBORS - {Xj} do
        add (Xk, Xi) to queue
  return true
```

```
function REVISE(csp, Xi, Xj) returns true iff we revise the domain of Xi
  revised ← false
  for each x in Di do
    if no value y in Dj allows (x,y) to satisfy the constraint between Xi and Xj then
      delete x from Di
      revised ← true
  return revised
```

Di is ith domain

Runtime: $O(cd^3)$, c is number of constraints and d is the domain size.

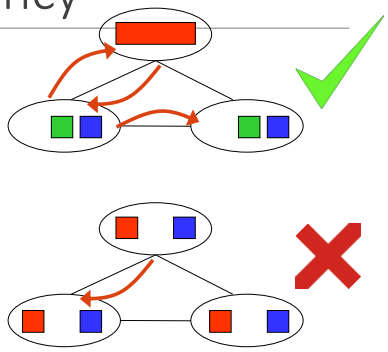
Auto-consistency (note initially nothing gets deleted/pruned since there are consistent options for all constraints for initial variable domains)

Arc can be inserted d times
Each arc (Xk,Xi) can be inserted in the queue only d times because Xi has at most d values to delete
Checking consistency of a single arc d^2

Limitations of Arc Consistency

After enforcing arc consistency:

- Can have one solution left
- Can have multiple solutions left
- Can have no solutions left (and not know it)

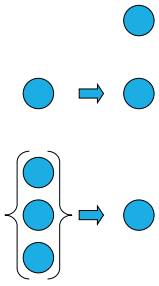


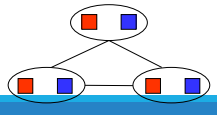
K-Consistency

Increasing degrees of consistency

- 1-Consistency (Node Consistency): Each single node's domain has a value which meets that node's unary constraints
- 2-Consistency (Arc Consistency): For each pair of nodes, any consistent assignment to one can be extended to the other
- K-Consistency: For each k nodes, any consistent assignment to k-1 can be extended to the k^{th} node.
  For any set of k – 1 variables and for any consistent assignment to those variables, a consistent value can always be assigned to any k^{th} variable
- When k=3, It's called path consistency.

- Higher k is more expensive to compute





is **local consistency**. If we treat each variable as a node in a graph (see CONSISTENCY
Figure 6.1(b)) and each binary constraint as an arc, then the process of enforcing local consistency
in each part of the graph causes inconsistent values to be eliminated throughout the graph.

Strong K-Consistency

Strong k-consistency: also k-1, k-2, … 1 consistent

Claim: strong n-consistency means we can solve without backtracking!

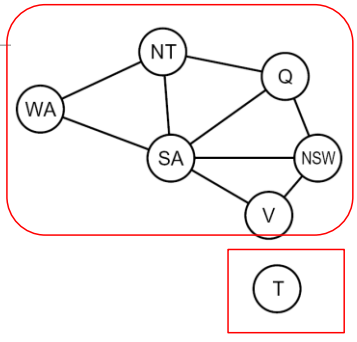
Why?

- Choose any assignment to any variable
- Choose a new variable
- By 2-consistency, there is a choice consistent with the first
- Choose a new variable
- By 3-consistency, there is a choice consistent with the first 2
- …

Problem Structure

Extreme case: independent subproblems
- Example: Tasmania and mainland do not interact

Independent sub problems are identifiable as connected components of constraint graph

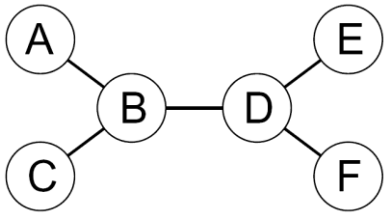


Suppose a graph of n variables can be broken into sub problems of only c variables:
- Worst-case solution cost is $O((n/c)(d^c))$, linear in n
- E.g., n = 80, d = 2, c =20
- 2^{80} = 4 billion years at 10 million nodes/sec
- $(4)(2^{20})$ = 0.4 seconds at 10 million nodes/sec

Tree-Structured CSPs

- A constraint graph is a tree when any two variables are connected by only one path.

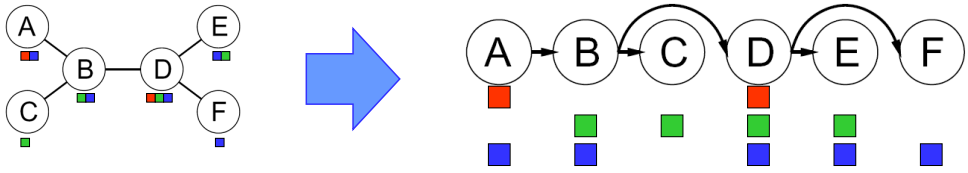
- **Directed arc consistency** (DAC)
  A CSP is defined to be directed arc-consistent under an ordering of variables X1,X2, . . . ,Xn if and only if every Xi is arc-consistent with each Xj for j > i.



Tree-Structured CSPs

Algorithm for tree-structured CSPs:

- Order: Choose a root variable, order variables so that parents precede children (topological sort)



- Remove backward: For i = n : 2, apply RemoveInconsistent(Parent(X_i),X_i)
- Assign forward: For i = 1 : n, assign X_i consistently with Parent(X_i)

Runtime: $O(n\ d^2)$ (why?)

n-1 arcs

d values for the two variables of each arc.

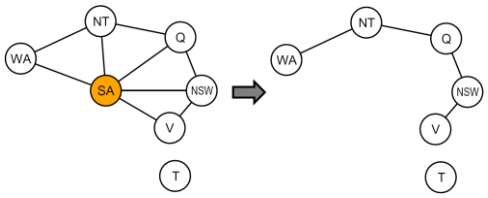
Tree CSP Solver

```
function TREE-CSP-SOLVER(csp) returns a solution, or failure
  inputs: csp, a CSP with components X, D, C

  n ← number of variables in X
  assignment ← an empty assignment
  root ← any variable in X
  X ← TOPOLOGICALSORT(X, root)
  for j = n down to 2 do
    MAKE-ARC-CONSISTENT(PARENT(X_j), X_j)
    if it cannot be made consistent then return failure
  for i = 1 to n do
    assignment[X_i] ← any consistent value from D_i
    if there is no consistent value then return failure
  return assignment
```

No backtracking.
Since each link from a parent to its child is arc consistent, we know that for any value we choose for
the parent, there will be a valid value left to choose for the child. That means we won't have
to backtrack; we can move linearly through the variables.

Nearly Tree-Structured CSPs



Conditioning: Assign a value to a variable (SA), prune its neighbors' domains

Cutset conditioning: instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree

Cutset size c gives runtime $O((d^c) (n-c) d^2)$, very fast for small c

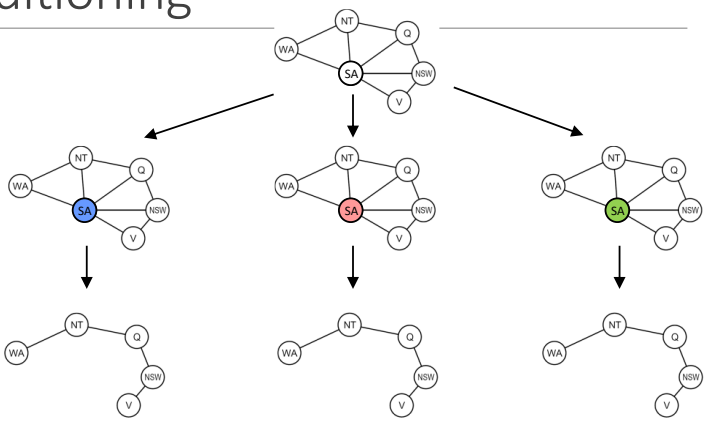
Cutset Conditioning

Choose a cutset

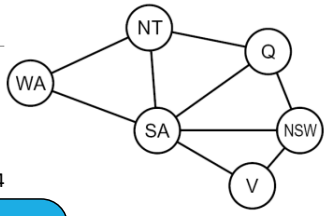
Instantiate the cutset (all possible ways)

Compute residual CSP for each assignment

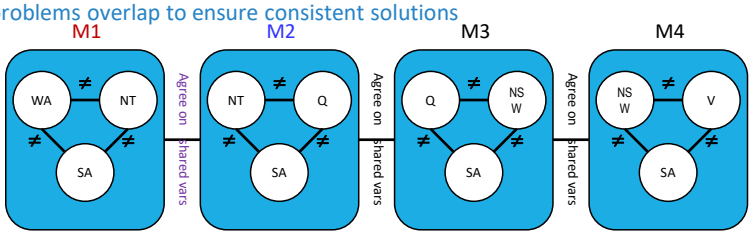
Solve the residual CSPs (tree structured)



Tree Decomposition



- Idea: create a tree-structured graph of mega-variables
- Each mega-variable encodes part of the original CSP
- Subproblems overlap to ensure consistent solutions



{(WA=r,SA=g,NT=b), (WA=b,SA=r,NT=g), …}

{(NT=r,SA=g,Q=b), (NT=b,SA=g,Q=r), …}

Agree: (M1,M2) ∈ {((WA=g,SA=g,NT=g), (NT=g,SA=g,Q=g)), …}

Tree Decomposition

- A tree decomposition must satisfy the following three requirements:
  - Every variable in the original problem appears in at least one of the subproblems.
  - If two variables are connected by a constraint in the original problem, they must appear together (along with the constraint) in at least one of the subproblems.
  - If a variable appears in two subproblems in the tree, it must appear in every subproblem along the path connecting those subproblems.

- A given constraint graph admits many tree decompositions; in choosing a decomposition, the aim is to make the subproblems as small as possible.

Tree Decomposition

- Each subproblem is solved independently; if any one has no solution, we know the entire problem has no solution.

If we can solve all the subproblems, then we attempt to construct a global solution as follows:

1. We view each subproblem as a “mega-variable” whose domain is the set of all solutions for the subproblem

2. We solve the constraints connecting the subproblems, using the efficient algorithm for trees given earlier.

The constraints between subproblems simply insist that the subproblem solutions agree on their shared variables.

For example, given the solution {WA = red ,SA = blue,NT = green} for the first subproblem, the only consistent solution for the next subproblem is {SA = blue,NT = green,Q = red}.

Local Search for CSP

- Local search algorithms turn are effective in solving many CSPs.
- They use a complete-state formulation: the initial state assigns a value to every variable, and the search changes the value of one variable at a time.

- The initial state violates several constraints, so local search tries to eliminate the violated constraints.
- Local search uses the min-conflicts heuristic for choosing a new value for a variable.
- The min-conflicts heuristic results in the minimum number of conflicts with other variables.

Local Search for CSP

- **Constraint Weighting** can help concentrate the search on the important constraints.
  - Each constraint is given a numeric weight, Wi, initially all 1.
  - At each step of the search, the algorithm chooses a variable/value pair to change that will result in the lowest total weight of all violated constraints.
  - The weights are then adjusted by incrementing the weight of each constraint that is violated by the current assignment.
  - This adds weight to the constraints that are proving difficult to solve.

Local Search for CSP

- Local search can be used in an online setting when the problem changes.
- For example, a week's airline schedule may involve thousands of flights, but bad weather at one airport can render the schedule infeasible.
- We would like to repair the schedule with a minimum number of changes.
- This can be easily done with a local search algorithm starting from the current schedule.
- On the other hand, a backtracking search with the new set of constraints usually requires much more time and might find a solution with many changes from the current schedule.

Summary

- CSP Formulation
- Backtracking Search
- Improving backtracking search
- Variable and value ordering
- Forward checking
- Constraint Propagation
- Arc Consistency
- Problem Structure
- Local Search for CSP