Lecture 7: Search - 6

Victor Lesser

CMPSCI 683
Fall 2004

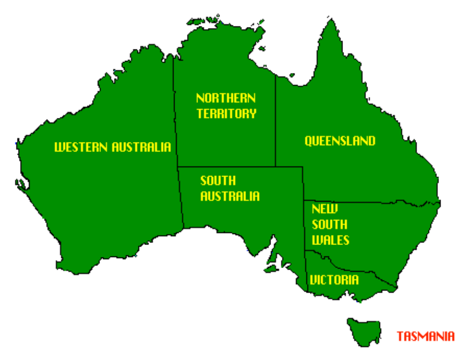
Today's lecture

- **Local Search**
  - Heuristic Repair
    - CSP and 3-SAT
- **Solving CSPs using Systematic Search.**
- **The relationship between problem structure and complexity.**

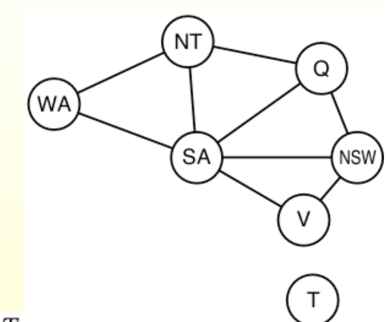
Constraint Satisfaction Problems (CSP)

- **A set of variables $X_1 \ldots X_n$, and a set of constraints $C_1 \ldots C_m$. Each variable X_i has a domain D_i of possible values.**
- **A solution to a CSP: a complete assignment to all variables that satisfies all the constraints.**
- **Representation of constraints as predicates.**
- **Visualizing a CSP as a constraint graph.**

Example: Map coloring

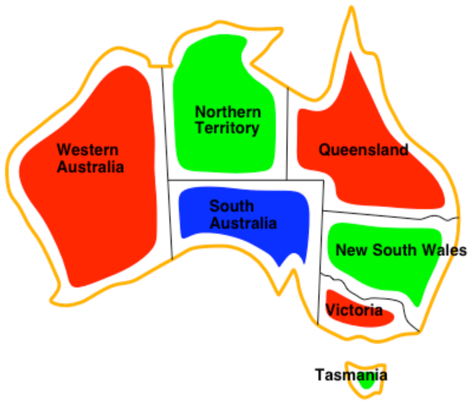


Constraint graph: nodes are variables
arcs show constraints



Variables WA, NT, Q, NSW, V, SA, T
Domains $D_i = \{red, green, blue\}$
Constraints: adjacent regions must have different colors
e.g., $WA \neq NT$ (if the language allows this), or
$(WA, NT) \in \{(red, green), (red, blue), (green, red), (green, blue), \ldots\}$

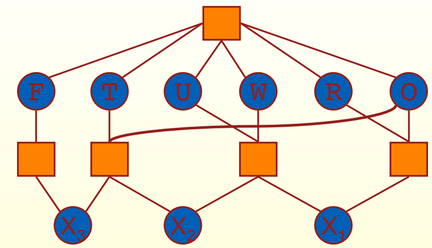
A Valid Map Assignment



Solutions are assignments satisfying all constraints, e.g.,
$\{WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green\}$

High-order constraints

```
  T W O
+ T W O
F O U R
```



- $O + O = R + 10 \cdot X1$
- $X_1 + W + W = U + 10 \cdot X_2$
- $X_1 + T + T = O + 10 \cdot X_3$
- $X_3 = F$
- *alldiff*(F,T,U,W,R,O)
- Between0-9(F,T,U,W,R,O)
- Between0-1 (X_1, X_2, X_3)

3 or more variables constraints

Finite vs. infinite domains

- **Finite domains: 8-queens, matching, cryptarithmetic, job assignment**
  - Finite-domain ⊂ Boolean ⊂ 3SAT (NP-complete)
- **Infinite domains: job scheduling**
  - Cannot enumerate all possibilities
    - Over the range of integers
  - Need a constraint language:
    - $StartJob_1 + 5 \leq StartJob_3$
    - Bound range

Constraint optimization

- **Representing *preferences* versus absolute constraints.**
  - Weighted by constraints violated/satisfied
- **Constraint optimization is generally more complicated.**
- **Can be solved using local search techniques.**
- **Hard to find optimal solutions.**

Local search for CSPs: Heuristic Repair

- **Start state is some assignment of values to variables that may violate some constraints.**
  - Create a complete but inconsistent assignment
- **Successor state: change value of one variable.**
- **Use** heuristic repair **methods to reduce the number of conflicts (iterative improvement).**
  - **The** min-conflicts heuristic**: choose a value for a variable that minimizes the number of remaining conflicts.**
  - **Hill climbing on the number of violated constraints**
- **Repair constraint violations until a consistent assignment is achieved.**
- **Can solve the *million*-queens problem in an average of 50 steps!**

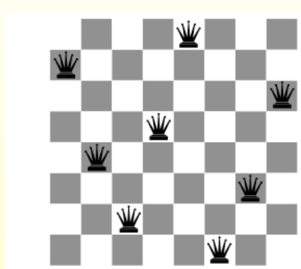
Heuristic Repair Algorithm

function MIN-CONFLICTS(*csp*, *max-steps*) **returns** a solution or failure
  inputs: *csp*, a constraint satisfaction problem
    max-steps, the number of steps allowed before giving up
  local variables: *current*, a complete assignment
    var, a variable
    value, a value for a variable

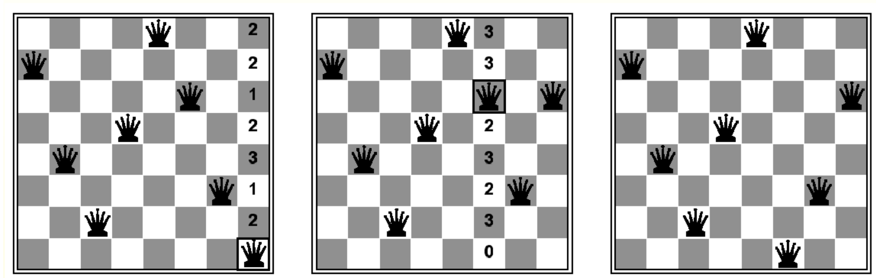
  current ← an initial complete assignment for *csp*
  for *i* = 1 to *max-steps* **do**
    var ← a randomly chosen, conflicted variable from VARIABLES[*csp*]
    value ← the value *v* for *var* that minimizes CONFLICTS(*var*, *v*, *current*, *csp*)
    set *var*=*value* in *current*
    if *current* is a solution for *csp* **then return** *current*
  end
  return *failure*

N-Queens Heuristic Repair

- **Pre-processing phase to generate initial assignment**
  - Greedy algorithm that iterates through rows placing each queen on the column where it conflicts with the fewest previously placed queens
- **Repair phase**
  - Select (randomly) a queen in a specific row that is in conflict and moves it to the column (within the same row) where it conflicts with the fewest other queens



Example of min-conflicts: N-Queens Problem



A two-step solution of an 8-queens problem. The number of remaining conflicts for each new position of the selected queen is shown. Algorithm moves the queen to the min-conflict square, breaking ties randomly.

Number of backtracks/repairs for *N*-Queens algorithms *(S. Minton et al.)*

n	Constructive: Standard backtrack	Constructive: Most constrained backtrack¨	Repair-based: Min-conflicts hill-climbing	Repair-based: Min-conflicts backtrack
$n = 10^1$	53.8	17.4	57.0	46.8
$n = 10^2$	4473 (70%)	687 (96%)	55.6	25.0
$n = 10^3$	88650 (13%)	22150 (81%)	48.8	30.7
$n = 10^4$	*	*	48.5	27.5
$n = 10^5$	*	*	52.8	27.8
$n = 10^6$	*	*	48.3	26.4

* = exceeded computation resources

Potential Reasons for Heuristic Repair to be Advantageous

- **Nonsystematic search hypothesis**
  - Depth-first search badly organized
  - Poorer choices are explored first at each branch point
  - More solutions with first queen placed in center of first row
    - Takes a very long time to recover from bad decision made early in search
  - Backtracking program that randomly orders rows (and columns within rows) still performs poorly
- **Distribution of solutions**
  - Depth first does not perform well where solutions clustered in tree
  - Random backtracking (Las Vegas algorithm) does better but still problem

Potential Reasons for Heuristic Repair to be Advantageous *(cont'd)*

- **Informedness hypothesis**
  - Heuristic repair is better because it has more information that is not available to a constructive backtracking (more encompassing view of search space)
  - Mini-conflict heuristic — select a variable that is in conflict and assign it a value that minimizes the number of conflicts (number of other variables that will need to be repaired)

SAT- Satisfiability Problem

Given a propositional sentence, determine if it is satisfiable, and if it is, show which propositions have to be true to make the sentence true. 3SAT is the problem of finding a satisfying truth assignment for a sentence in a special format

Definition of 3SAT

- **A** literal **is a proposition symbol or its negation (e.g., *P* or ¬ *P*).**
- **A** clause **is a disjunction of literals; a 3-clause is a disjunction of exactly 3 literals (e.g., *P* ∨ *Q* ∨ ¬ *R*).**
- **A sentence in CNF or** conjunctive normal form **is a conjunction of clauses; a 3-CNF sentence is a conjunction of 3-clauses.**
- **For example,**

$(P \lor Q \lor \neg S) \land (\neg P \lor Q \lor R) \land (\neg P \lor \neg R \lor \neg S) \land (P \lor \neg S \lor T)$

Is a 3-CNF sentence with four clauses and five proposition symbols.

Converting N-SAT into 3-SAT

$A \lor B \lor C \lor D$

$\equiv$

$(A \lor B \lor E) \land (\sim E \lor C \lor D)$

A = T	A = F	A = F	
B = F	B = T	B = F	
C = F	C = F	C = T	L L
<u>D = F</u>	<u>D = F</u>	<u>D = F</u>	
E = F	E = F	E = T	

2 - SAT polynomial time but can' t
map all problem into 2 - SAT

Mapping 3-Queens into 3SAT

At least 1 has a Q   not exactly 2 have Q's   not all 3 have Q's

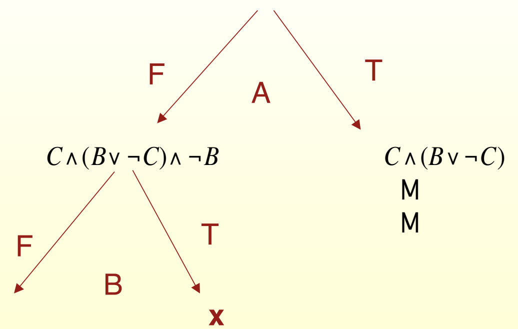
$(Q_{1,1} \lor Q_{1,2} \lor Q_{1,3}) \land (Q_{1,1} \lor \neg Q_{1,2} \lor \neg Q_{1,3})$
$\land (\neg Q_{1,1} \lor Q_{1,2} \lor \neg Q_{1,3})$
$\land (\neg Q_{1,1} \lor \neg Q_{1,2} \lor Q_{1,3}) \quad \land (\neg Q_{1,1} \lor \neg Q_{1,2} \lor \neg Q_{1,3})$

Do the same for each row, the same for each column, the same for each diagonal, and'ing them all together.

$\land$
$(Q_{2,1} \lor Q_{2,2} \lor Q_{2,3}) \land (Q_{2,1} \lor \neg Q_{2,2} \lor \neg Q_{2,3})$
$\land (\neg Q_{2,1} \lor Q_{2,2} \lor \neg Q_{2,3}) \land (\neg Q_{2,1} \lor \neg Q_{2,2} \lor Q_{2,3}) \land (\neg Q_{2,1} \lor \neg Q_{2,2} \lor \neg Q_{2,3})$
$\land$
$(Q_{1,1} \lor Q_{2,2} \lor Q_{3,3}) \land (Q_{1,1} \lor \neg Q_{2,2} \lor \neg Q_{3,3}) \land (\neg Q_{1,1} \lor Q_{2,2} \lor \neg Q_{3,3})$
$\land (\neg Q_{1,1} \lor \neg Q_{2,2} \lor Q_{3,3}) \land (\neg Q_{1,1} \lor \neg Q_{2,2} \lor \neg Q_{3,3})$
M
etc.

Davis-Putnam Algorithm (Depth-First Search)

$(A \lor C) \land (\neg A \lor C) \land (B \lor \neg C)$
$\land\ (A \lor \neg B)$



GSAT Algorithm

Problem: **Given a formula of the propositional calculus, find an interpretation of the variables under which the formula comes out true, or report that none exists.**

procedure GSAT

Input: **a set of clauses ∝, MAX-FLIPS, and MAX-TRIES**

Output: **a satisfying truth assignments of ∝, if found**

begin

for *i* **:= 1** to **MAX-TRIES**

T **:= a randomly generated truth assignment**

for *j* **:= 1** to **MAX-FLIPS**

if ***T*** **satisfies ∝** then return ***T***

p **:= a propositional variable such that a change in its truth assignment gives the largest increase in total number of clauses of ∝ that are satisfied by *T*.**

T **:= *T* with the truth assignment of *p* reversed**

end for

end for

return **"no satisfying assignment found"**

end

GSAT Performance

formulas		GSAT			DP		
vars	clauses	M-FLIPS	tries	time	choices	depth	time
50	215	250	6.4	0.4s	77	11	1.4s
70	301	350	11.4	0.9s	42	15	15s
100	430	500	42.5	6s	84×10^3	19	2.8m
120	516	600	81.6	14s	0.5×10^6	22	18m
140	602	700	52.6	14s	2.2×10^6	27	4.7h
150	645	1500	100.5	45s	—	—	—
200	860	2000	248.5	2.8m	—	—	—
250	1062	2500	268.6	4.1m	—	—	—
300	1275	6000	231.8	12m	—	—	—
400	1700	8000	440.9	34m	—	—	—
500	2150	10000	995.8	1.6h	—	—	—

GSAT versus Davis-Putnam

(a backtracking style algorithm)

Domain: ***hard*** **random 3CNF formulas, all satisfiable (hard means chosen from a region in which about 50% of problems are unsolvable)**

Domain: *n*-queens

formulas			GSAT		
Queens	vars	clauses	flips	tries	time
8	64	736	105	2	0.1s
20	400	12560	319	2	0.9s
30	900	43240	549	1	2.5s
50	2500	203400	1329	1	17s
100	10000	1.6×10^6	5076	1	195s

GSAT Performance *(cont'd)*

- ***Biased Random Walk***
- **With probability *p*, follow the standard GSAT scheme,**
  - *i.e.*, make the best possible flip.
- **With probability 1 - *p*, pick a variable occurring in some unsatisfied clause and flip its truth assignment. (Note: a possible uphill move.)**

formula		GSAT						Simul. Ann.	
		basic		walk		noise			
vars	clauses	time	flips	time	flips	time	flips	time	flips
100	430	.4	7554	.2	2385	.6	9975	.6	4748
200	860	22	284693	4	27654	47	396534	21	106643
400	1700	122	2.6×10^6	7	59744	95	892048	75	552433
600	2550	1471	30×10^6	35	241651	929	7.8×10^6	427	2.7×10^6
800	3400	*	*	286	1.8×10^6	*	*	*	*
1000	4250	*	*	1095	5.8×10^6	*	*	*	*
2000	8480	*	*	3255	23×10^6	*	*		

Comparing noise strategies on hard random 3CNF formulas. (Time in seconds on an SGI Challenge)

3SAT Phase Transition

- **Easy -- Sastifiable problems where many solutions**
- **Hard -- Sastifiable problems where few solutions**
- **Easy -- Few Satisfiable problems**

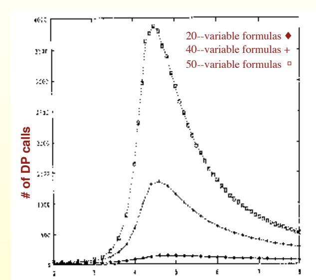


Fig. 1 Solving 3SAT problems.

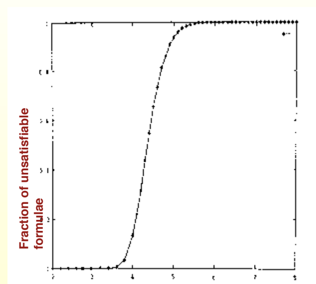


Fig. 2 Fraction of unsatisfiable 3SAT problems.

- **Assumes concurrent search in the satisfiable space and the non-satisfiable space (negation of proposition)**

Commutativity

- **Naïve application of search to CSPs:**
  - Branching factor is $n \cdot d$ at the top level, then $(n-1)d$, and so on for n levels.
  - The tree has $n! \cdot d^n$ leaves, even though there are only d^n possible complete assignments!
- **Naïve formulation ignores** commutativity **of all CSPs.**
  - Solution: **consider a single variable at each depth of the tree**.

Solving CSPs using Systematic Search

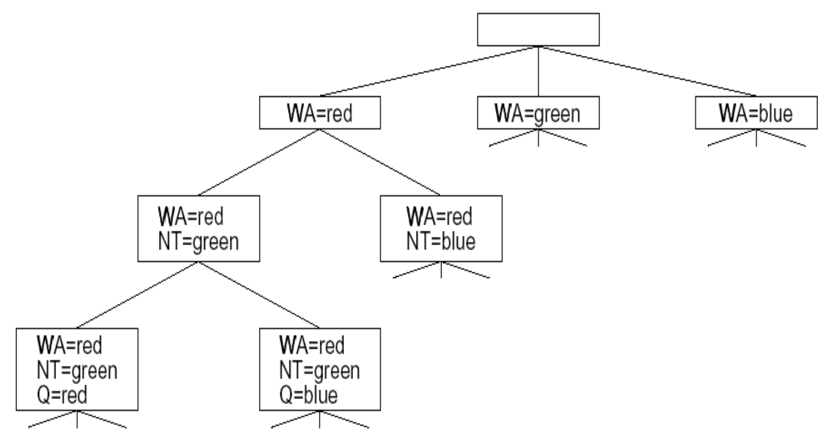
- Initial state: **the empty assignment**
- Successor function: **a value can be assigned to any variable as long as no constraint is violated.**
- Goal test: **the current assignment is complete.**
- Path cost: **a constant cost for every step.**

Simple backtracking

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return RECURSIVE-BACKTRACKING([], csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
     if value is consistent with assignment according to CONSTRAINTS[csp] then
        result ← RECURSIVE-BACKTRACKING([var = value—assignment], csp)
        if result ≠ failure then return result
  end
  return failure
```


Part of the map-coloring search tree



Intelligent Backtracking for CSPs

- **CSP search complexity may be affected by:**
  - The order in which variables are assigned values;
  - The domain values chosen for assignment.
- **Variable-ordering heuristics reduce the bushiness of the search tree by moving failures to upper levels.**
- **Value-ordering heuristics move solutions to the "left" of the search tree so they are found more quickly by backtracking search.**
- **Good heuristics can reduce search complexity by nearly an order of magnitude.**

Heuristics that can help

Key questions:

1. **Which variable should be assigned next and in what order should the values be tried?**
2. **What are the implications of the current variable assignments for the other unassigned variables?**
3. **When a path fails, can the search avoid repeating this failure in subsequent paths?**

Problem Textures

- **Relate decisions about search control to characteristics of the problem space**
- **Characterize the problem topology by a set of texture measures**
- **Static and Dynamic Meta-Level Information**

Variable and Value ordering

Variable ordering

- **The most-constrained-variable heuri**
  - has the fewest "legal" values
  - reduce branching factor

- **The most-constraining-variable heuristic**
  - involved in largest number of constraints
  - likely reduce future branching factors

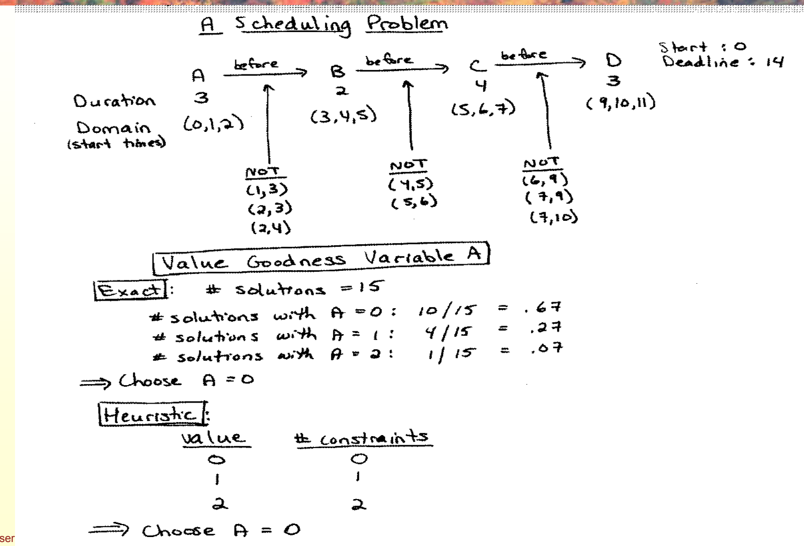
Value ordering

- **The least-constraining-value heuristic**
  - rules out the fewest choices for neighboring vars
  - reduce likelihood of backtracking

(Variable) Value Goodness

- **Define: The probability that the assignment of a particular value to a particular variable leads to an overall solution to the problem**
- **Compute: The ratio of complete assignments that are solutions to the problem and have that value for the variable over the total number of possible assignments**
- **Heuristic: The number of constraints on the variable involving that value**

VVG-examples



Variable Tightness

- **Define: The probability that an assignment consistent with all the problem constraints that do not involve a given variable does not result in a solution. Variable tightness is the backtracking probability when the variable in question is the last one instantiated.**
- **Compute: The ratio of the number of solutions to the problem with constraints on the variable in question removed that could not be solutions to the fully-constrained problem to the total number of solutions to the problem with constraints on the variable removed.**
  - Let c' = the set of constraints involving v
  - Let B = the problem without c' in A
    - (solutions to B not solutions to A) / (solutions B)
    - High means variable should be bound early
- **Heuristic: The number of constraints on the variable**

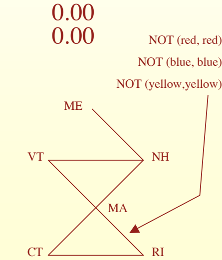
Variable Tightness - Example

Exact Variable Tightness Textures Measures

State	Deconstrained Solutions	Non-Solutions	Variable Tightness
CT	24	0	0.00
MA	72	48	0.67
ME	12	0	0.00
NH	36	12	0.33
RI	12	0	0.00
VT	12	0	0.00

Heuristic Variable Tightness Texture Measures

State	Number of Constraints
CT	6
MA	12
ME	3
NH	9
RI	6
VT	6



3-color map coloring New England

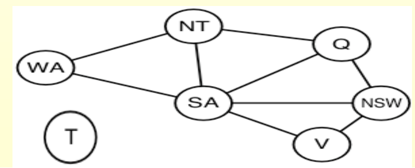
Summary of Heuristics for CSPs

- **Most-constraining variable**
  - Select for assignment the variable that is involved in the largest number of constraints on unassigned variables;
  - Also called the *search-rearrangement method*;
- **Least-constraining value**
  - Select a value for the variable that eliminates the smallest number of values for variables connected with the variable by constraints;
  - i.e., maximize the number of assignment options still open.

Constraint propagation

- **Reduce the branching factor by deleting values that are not consistent with the values of the assigned variables.**
- **Forward checking: a simple kind of propagation**

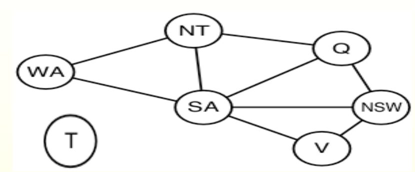
	WA	NT	Q	NSW	V	SA	T
Initial domains	R G B	R G B	R G B	R G B	R G B	R G B	R G B
After WA=red	(R)	G B	R G B	R G B	R G B	G B	R G B
After Q=green	(R)	B	(G)	R B	R G B	B	R G B
After V=blue	(R)	B	(G)	R	(B)		R G B

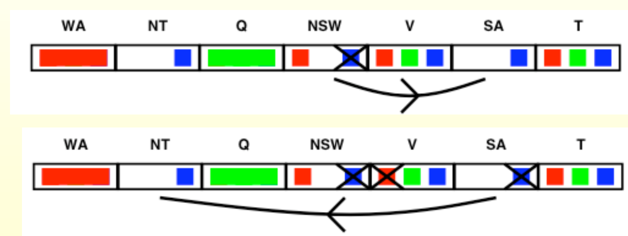


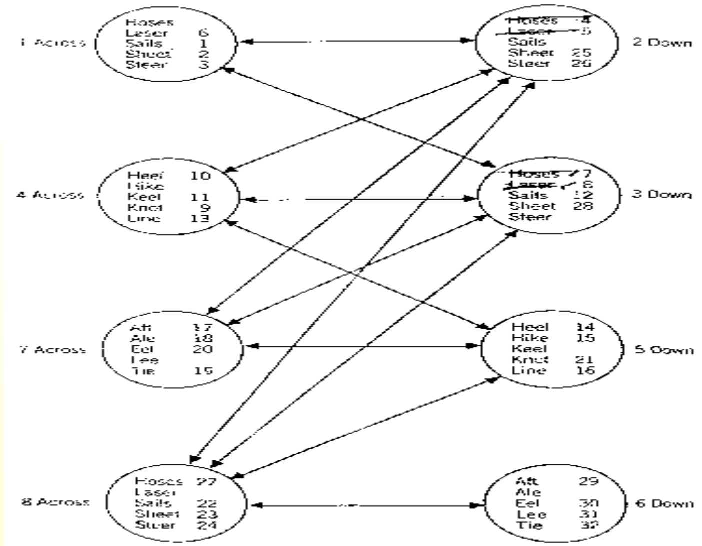
Arc consistency

- **An arc from *X* to *Y* in the constraint graph is consistent if, for *every* value of *X*, there is *some* value of *Y* that is consistent with *X*.**
- **Can detect more inconsistencies than forward checking.**
- **Can be applied as a preprocessing step before search or as a propagation step after each assignment during search.**
- **Process must be applied *repeatedly* until no more inconsistencies remain. Why?**

ARC Consistency Example







Waltz Filtering: Exploiting Pair-Wise Constraints

$(\exists\chi_1)(\exists\chi_2)\ldots(\exists\chi_n)(\chi_1 \in D_1)(\chi_2 \in D_2)\ldots(\chi_n \in D_n)\ P_1(\chi_1) \wedge P_2(\chi_2)\ldots \wedge P_n(\chi_n) \wedge P_{12}(\chi_1, \chi_2) \wedge P_{13}(\chi_1, \chi_3) \wedge \ \ldots \wedge P_{n-1}, P_n(\chi_{n-1}, \chi_n)$

ARC Consistency Algorithm

function AC3(*csp*) **returns** the CSP, possibly with reduced domains
  inputs: *csp*, a binary CSP with variables $\{X_1, X_2, \ldots X_n\}$
  local variables: *queue*, a queue of arcs, initially all the arcs in *csp*

  loop while *queue* is not empty **do**
    $(X_i, X_j) \leftarrow$ REMOVE-FRONT(*queue*)
    if REMOVE-INCONSISTENT-VALUES(X_i, X_j) **then**
      for each X_k **in** NEIGHBORS[X_i] **do**
        add (X_k, X_i) to *queue*
end

---

function REMOVE-INCONSISTENT-VALUES(X_i, X_j) **returns** true iff we remove a value
  removed ← *false*
  loop for each x **in** DOMAIN[X_i] **do**
    if (x,y) satisfies the constraint for some value y in DOMAIN[X_j]   Wrong
      then delete x from DOMAIN[X_i]; *removed* ← *true*
  end
  return *removed*
end

if(x,y) can not be satisfied with some value y in DOMAIN[Xj] then delete x from DOMAIN[Xi]; remove<-true

Complexity of arc consistency

- **A binary CSP has at most O(n^2) arcs**
- **Each arc ($X \rightarrow Y$) can only be inserted on the agenda *d* times because at most *d* values of *Y* can be deleted.**
- **Checking consistency of an arc can be done in O(d^2) time.**
- **Worst case time complexity is: O(n^2d^3).**
- **Does not reveal every possible inconsistency!**

K-consistency

- **A graph is *k*-consistent if, for any set of *k* variables, there is always a consistent value for the *k*th variable given any consistent partial assignment for the other *k*-1 variables.**
  - A graph is strongly k-consistent if it is *i*-consistent for *i* = 1..*k*.
  - IF k=number of nodes than no backtracking
- **Higher forms of consistency offer stronger forms of constraint propagation.**
  - Reduce amount of backtracking
  - Reduce effective branching factor
  - Detecting inconsistent partial assignments
- **Balance of how much pre-processing to get graph to be k consistent versus more search**

Intelligent backtracking

- Chronological backtracking: **always backtrack to most recent assignment. Not efficient!**
- Conflict set: **A set of variables that caused the failure.**
- Backjumping: **backtrack to the most recent variable assignment in the conflict set.**
- **Simple modification of BACKTRACKING-SEARCH.**
- **Every branch pruned by backjumping is also pruned by forward checking!**
- Conflict-directed backjumping: **better definition of conflict sets leads to better performance.**

Informed-Backtracking Using Min-Conflicts Heuristic

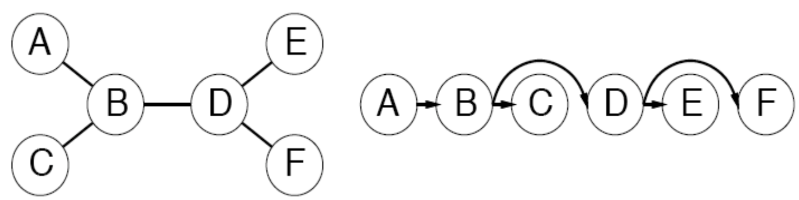
```
Procedure INFORMED-BACKTRACK (VARS-LEFT VARS-DONE)
    If all variables are consistent, then solution found, STOP.
    Let VAR = a variable in VARS-LEFT that is in conflict.
    Remove VAR from VARS-LEFT.
    Push VAR onto VARS-DONE.
    Let VALUES = list of possible values for VAR ordered in ascending
                     order according to number of conflicts with variables
                     in VARS-LEFT.
    For each VALUE in VALUES, until solution found:
        If VALUE does not conflict with any variable that is in VARS-DONE,
        then Assign VALUE to VAR.
            Call INFORMED-BACKTRACK(VARS-LEFT VARS-DONE)
        end if
    end for
end procedure
Begin program
    Let VARS-LEFT = list of all variables, each assigned an initial state
    Let VARS-DONE = nil
    Call INFORMED-BACKTRACK(VARS-LEFT VARS-DONE)
End program
```


Complexity and problem structure

- **The complexity of solving a CSP is strongly related to the** structure **of its constraint graph.**
- **Decomposition into** independent subproblems **yields substantial savings: O(d^n) → O($d^c \cdot n/c$)**
-
- Tree-structured problems **can be solved in linear time O($n \cdot d^2$)**
- Cutset conditioning **can reduce a general CSP to a tree-structured one, and is very efficient if a small cutset can be found.**

Algorithm for Tree Structured CSPs

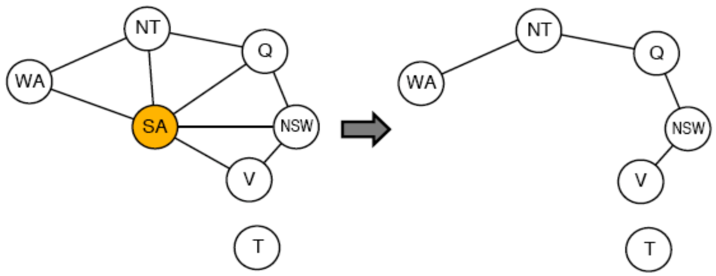
1. Choose a variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering



2. For j from n down to 2, apply $\text{REMOVEINCONSISTENT}(Parent(X_j), X_j)$
3. For j from 1 to n, assign X_j consistently with $Parent(X_j)$

Algorithm for Nearly-Tree Structured CSPs

Conditioning: instantiate a variable, 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 $c \Rightarrow$ runtime $O(d^c \cdot (n-c)d^2)$, very fast for small c

Summary

CSPs are a special kind of problem:
states defined by values of a fixed set of variables
goal test defined by *constraints* on variable values

Backtracking = depth-first search with one variable assigned per nod

Variable ordering and value selection heuristics help significantly

Forward checking prevents assignments that guarantee later failure

Constraint propagation (e.g., arc consistency) does additional work to constrain values and detect inconsistencies

The CSP representation allows analysis of problem structure

Tree-structured CSPs can be solved in linear time

Iterative min-conflicts is usually effective in practice

Next lecture

- **Interacting Subproblems**
- **Multi-level Search**
  - blackboard