Lecture 8: Search - 7

Victor Lesser

CMPSCI 683
Fall 2004

Lecture 8

- Continuation of Systematic Search for CSPs
- More Complex Search

Solving CSPs using Systematic Search

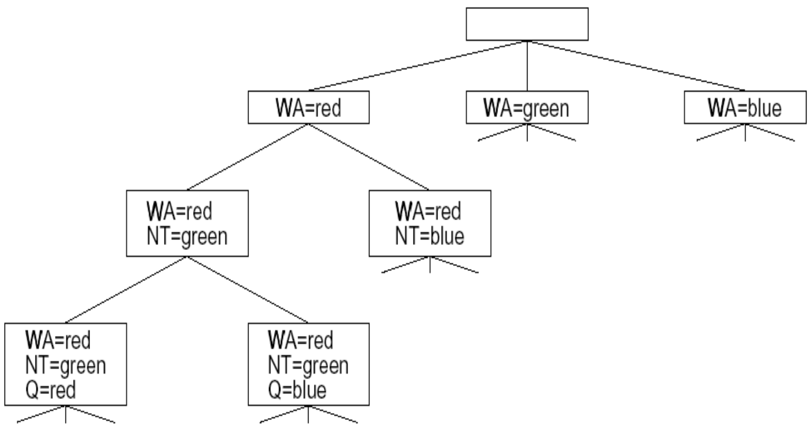
- Initial state: **the empty assignment**
- Successor function: **a value can be assigned to any variable as long as no constraint is violated.**
- Goal test: **the current assignment is complete.**
- Path cost: **a constant cost for every step.**

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
```

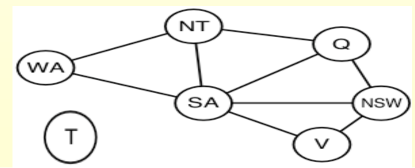
Part of the map-coloring search tree



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

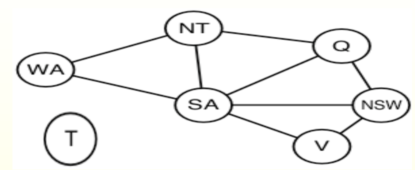
	WA	NT	Q	NSW	V	SA	T
Initial domains	R G B	R G B	R G B	R G B	R G B	R G B	R G B
After WA=red	R	G B	R G B	R G B	R G B	G B	R G B
After Q=green	R	B	G	R B	R G B	B	R G B
After V=blue	R	B	G	R	B		R G B



Arc consistency

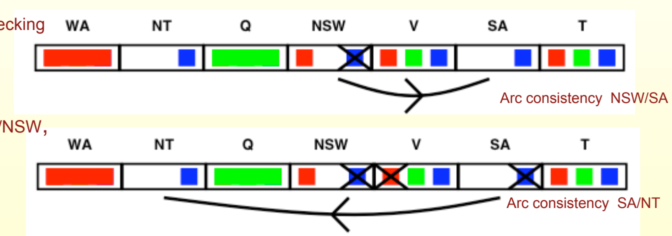
- An arc from *X* to *Y* in the constraint graph is consistent if, for *every* value of *X*, there is *some* value of *Y* that is consistent with *X*.
- *Can detect more inconsistencies than forward checking.*
  - Can be applied as a preprocessing step before search
  - As a propagation step after each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

ARC Consistency Example



Forward checking (WA,Q)

Update V/NSW,



No possible solution with WA=red and Q=green

ARC Consistency Algorithm

```
function AC3( csp) returns the CSP, possibly with reduced domains
   inputs: csp, a binary CSP with variables {X1, X2, ... Xn}
   local variables: queue, a queue of arcs, initially all the arcs in csp

   loop while queue is not empty do
      (Xi, Xj) ← REMOVE-FRONT(queue)
      if REMOVE-INCONSISTENT-VALUES(Xi, Xj) then
         for each Xk in NEIGHBORS[Xi] do          ; propagates effects thru network
            add (Xk, Xi) to queue
end
```

```
function REMOVE-INCONSISTENT-VALUES( Xi, Xj) returns true iff we remove a value
   removed ← false
   loop for each x in DOMAIN[Xi] do
      If (x,y ) arc consistency can not be satisfied with some value y in DOMAIN[Xj]
      then delete x from DOMAIN[Xi]; remove<-true
   end
   return removed
end
```


Complexity of arc consistency

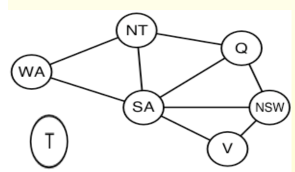
- **A binary CSP has at most O(n^2) arcs**
- **Each arc ($X \rightarrow Y$) can only be inserted on the agenda *d* times because at most *d* values of *Y* can be deleted.**
- **Checking consistency of an arc can be done in O(d^2) time.**
- **Worst case time complexity is: O(n^2d^3).**
- **Does not reveal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



Fixed variable ordering Q,NSW,V,T,SA,WA,NT

{Q,NSW,V,T}, SA=?; backup to T makes no sense

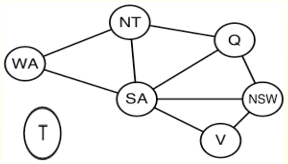
What Variable(s) Caused the Conflict

Backtrack to V, most recent variable set in conflict set

- **Forward Checking can also generate conflict set based on variables that remove elements from domain**

More Advanced Backtracking

- Conflict-directed backjumping: **better definition of conflict sets leads to better performance -- bottom-up/top-down state integration**



WA=red, NSW=red can never be solved

T= red, then assign NT,Q,V,SA (always fails)

How to know that (indirect) conflict set of NT is

WA and NSW since they don't conflict with NT

Conflict set of NT is set of preceding variables that caused NT, together with any subsequent variables, to have no consistent solutions

SA fails conflict {WA,NT,Q} based on forward propagation; backjump to Q

Q absorbs conflict set of SA minus Q {WA,NSW,NT}; backjump to NT

NT absorbs conflict set of Q minus NT {WA,NSW};

Informed-Backtracking Using Min-Conflicts Heuristic

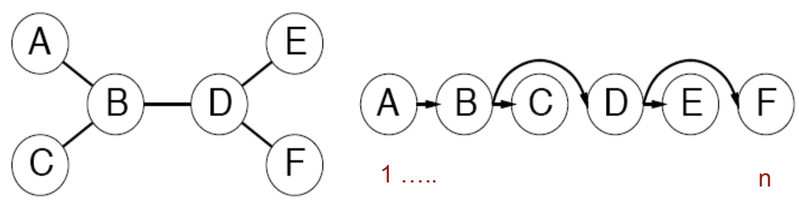
Procedure INFORMED-BACKTRACK (VARS-LEFT VARS-DONE)
  If all variables are consistent, then solution found, STOP.
  Let VAR = a variable in VARS-LEFT that is in conflict (***chosen randomly***).
  Remove VAR from VARS-LEFT.
  Push VAR onto VARS-DONE.
  Let VALUES = list of possible values for VAR ordered in ascending order according to number of conflicts with variables in VARS-LEFT. ; *min-conflict heuristic*
  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

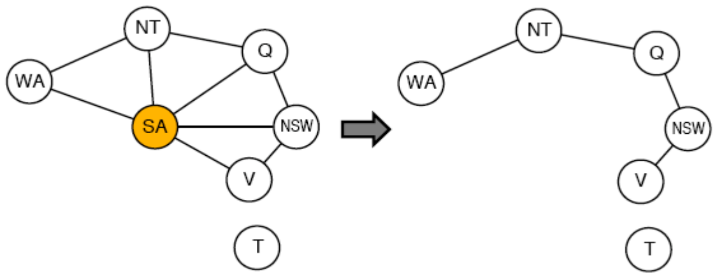
1. Choose a variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering



2. For j from n down to 2, apply $\textsc{RemoveInconsistent}(Parent(X_j), X_j)$
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

More Complex Search

- Multi-Level/Hierarchical Search
- Interdependence of Search Paths
- Non-Monotonic Domain
- Cost of Control
- Non-uniform cost of operator application

Simple Heuristic Search

- **Operators applicable to a search node are not affect by path to node**
  - Markov-like assumption
- **Rating of one search node doesn't affect rating of others nodes on different paths**
  - Near Independence of search Paths

Contrast

- Drilling example -- samples taken from one well may change the likelihood of other well being successful

Crossword Puzzle Search

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

Word List

Aft **Laser** **Ale** **Lee** **Eel** **Line** **Hike** **Sails** **Hoses** **Sheet** **Keel** **Steer** **Knot** **Tie**

1 H	O	2 S	E	3 S
■	■	A	■	T
■	4 H	I	5 K	E
6 A	■	7 L	E	E
8 L	A	S	E	R
E	■	■	L	■

Heuristic Search

- States/Operators?
  - Each numbered row (1,4,7,8) and column (2,3,5,6)
- Independence of States/Operators?
  - If 4="Line" then no way to fill in 5

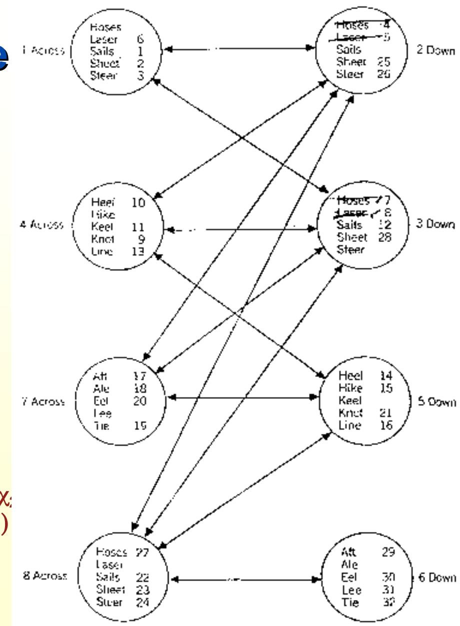
Crossword Puzzle Search as Interacting Subproblems

Waltz Filtering:

Exploiting Pair-Wise Constraints

$(\exists \chi_1)(\exists \chi_2)\ldots(\exists \chi_n)(\chi_1 \in \mathbf{D}_1)(\chi_2 \in \mathbf{D}_2)\ldots(\chi_n \in \mathbf{D}_n)$

$\mathbf{P}_1(\chi_1) \wedge \mathbf{P}_2(\chi_2)\ldots \wedge \mathbf{P}_n(\chi_n) \wedge \mathbf{P}_{12}(\chi_1, \chi$ $\mathbf{P}_{13}(\chi_1, \chi_3) \wedge \ldots \wedge \mathbf{P}_{n-1}, \mathbf{P}_n(\chi_{n-1}, \chi_n)$



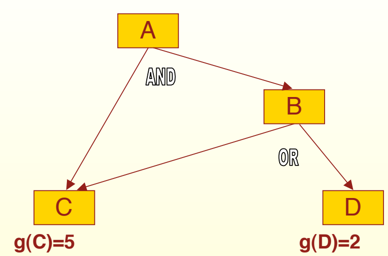
Make the Crossword Puzzle More Complex

- **What happens if you add in more constraints among words?**
  - Grammar/Theme
- **What happens if you add in speech input?**
  - Probablistic knowledge about word likelihood
  - Constraint satisfaction vs constraint optimization
    - Hard and soft constraints

How does the interaction among subproblems change?

Subgoal/Subproblem Interactions

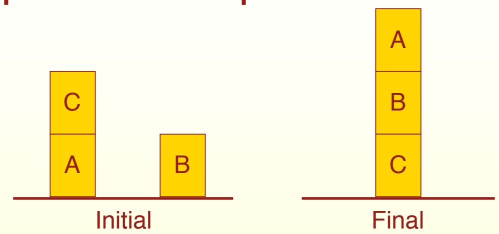
Subgoals B and C cannot be solved independently



From the perspective of subgoal B, subgoal D appears to be the best solution (cost of 2 vs. cost of 5 using subgoal C), but since C must also be satisfied to solve A, the overall best solution is subgoal C.

Example: Blocks World

- **Simple blocks world problem:**



- **Initial state:** ***ON(C,A),CLR(B),CLR(C)***
- **Goal state:** ***ON(B,C),ON(A,B),CLR(A)***

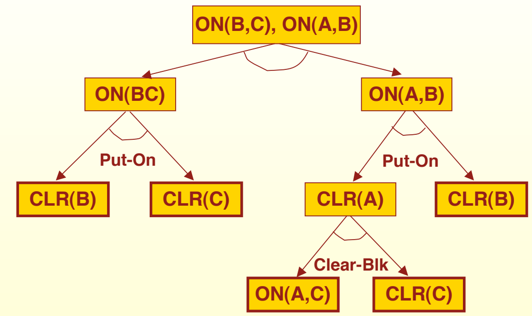
Operators:

1) Clear-Blk: $ON(\chi,y) \wedge CLR(\chi) \rightarrow CLR(y)$

2) Put-On: $CLR(\chi) \wedge CLR(y) \rightarrow ON(\chi,y)$

Example: Blocks World

- **Decomposition with subgoal interactions:**



- **Cannot just combine the solutions to *ON(B,C)* and *ON(A,B)* since each solution makes the other solution invalid.**

Constraints from Subproblem Interaction

- **Take into account the existence of other states or solution to other subproblems**
  - ARC consistency
- **Re-evaluate rating node/operator**
  - Reduce variance, uncertainty in rating
  - Decrease cost of operator application
    - eliminates certain states as infeasible

Nearly Decomposable Problems

- **Some problems are** nearly decomposable:

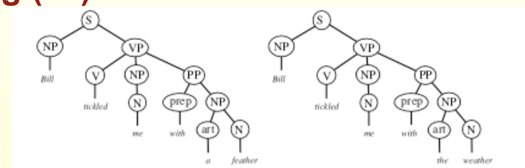
  Their subproblems have only a "small amount" of interaction.
- **A goal that can be decomposed into a set of subgoals is nearly decomposable if:**
  - most of the time, independently considered solutions to the subgoals can be combined into a consistent solution to the goal;
  - Only a subset of the subgoal solutions interact so as to be inconsistent;
  - Consistent solutions can be found without completely re-solving the joint subproblem

Nearly Decomposable Problems (cont'd)

- **Many AI techniques have been developed to handle nearly decomposable problems**
- **Typically, this involves independently solving the subproblems and then "repairing" the solutions to deal with any interactions.**
- **How does it relate to success of Heuristic Repair/Local Search success**
  - Dynamically evolving interacting subproblems

Another Version of Monotone Assumption

- **Two states both apply to same operator&data**
- **Two nodes A & B, rating(A) > rating(B)**
  If extended by same data then rating(A^1) > rating (B^1)



f_S("Bill tickled me with a feather.") > f_S("Bill tickled me with the weather.")

f_{VP}("tickled me with a feather") < f_{VP}("tickled me with the weather")

f_{PP}("with a feather") < f_{PP}("with the weather")

Non-Monotone Example

Next lecture

- **Blackboard Systems**