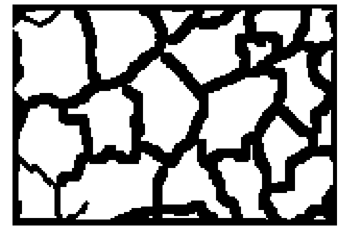
Constraint Satisfaction

Jacky Baltes
Department of Computer Science
University of Manitoba
Email: jacky@cs.umanitoba.ca
WWW:
http://www4.cs.umanitoba.ca/~jacky/Teaching/Courses/COMP_4190-ArtificialIntelligence/current/index.php

Map Colouring

- How many colours does it take to colour this map?
- Adjacent countries must have different colour
- Joined by a line (not a point)

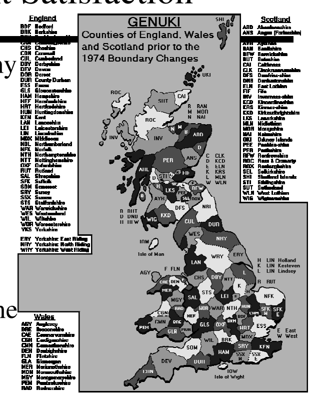


Map Colouring

- Map with four colours

Constraint Satisfaction

- Does this work for any map?
- What is minimum colours needed to colour any map?
- Can we find an algorithm to colour the map?

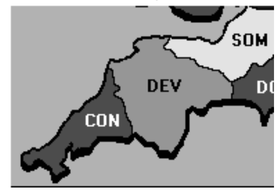


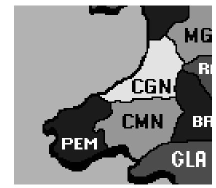
Four Colour Theorem

- Francis Guthrie (1852) posed the question when colouring a map of English counties
- 1878 – 1976: Various “proofs” were published and (much) later proven to be incorrect.
- Appel and Haken (1976)

Proof of the Four Colour Theorem (Outline)

- Reducibility
  - Create a smaller problem and show that if the smaller problem can be solved, so can the larger one.
  - 633 configurations (Robertson et al)





Proof of the Four Colour Theorem (Outline)

- At some point no more countries can be removed
  - Let's call this a canonical map
- Calculate the set of all possible minimal maps
- Show that for any minimal map a four colouring exists

Reducibility

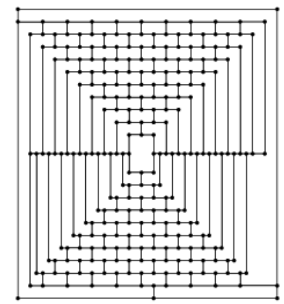
- Can reduce regions iteratively from the map

Proof of the Four Colour Theorem (Outline)

- Part of the Appel-Haken proof uses a computer
  - 1476 different minimal maps/graphs
  - 300 “discharging rules”
- Generation gap
  - This is not a proof!
  - How do we check the computer part?
  - What can we learn from it?

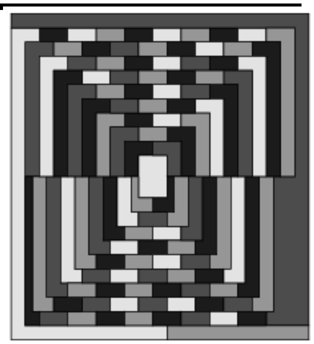
Martin Gardner

- Showed a counterexample to the 4 colour theorem (April 1, 1975)

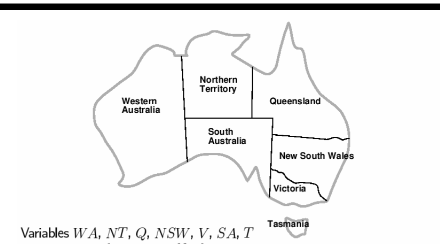


Wagon

- 1998

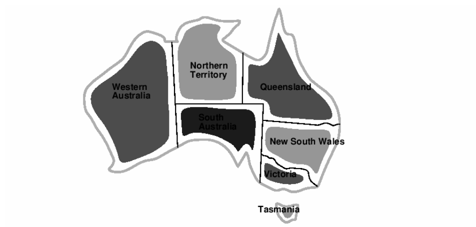


Map Colouring



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

Map Colouring



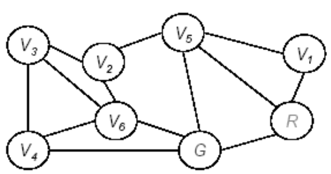
Solutions are assignments satisfying all constraints, e.g.,
$\{WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green\}$

Constraint Satisfaction Problem Map Colouring

- Represent the map as a graph
  - Nodes are regions of the map
  - Edges between nodes indicate that two regions are adjacent
- Find an assignmens of colours to nodes such that no two adjacent nodes have the same colour

Constraint satisfaction problem (CSP)

- A value needs to be assigned to each variable (node)
- No two adjacent nodes can have the same value
- Two nodes already have values



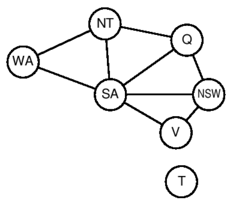
Formal definition of a CSP problem

A CSP is a triplet { V , D , C }. A CSP has a finite set of variables V = { V_1 , V_2 .. V_N }.
Each variable may be assigned a value from a domain D of values.
Each member of C is a pair. The first member of each pair is a set of variables. The second element is a set of legal values which that set may take.
Example:
$V = \{ V_1 , V_2 , V_3 , V_4 , V_5 , V_6 \}$
$D = \{ R , G , B \}$
$C = \{ (V_1,V_2) : \{ (R,G), (R,B), (G,R), (G,B), (B,R) (B,G)\},$
$\{ (V_1,V_3) : \{ (R,G), (R,B), (G,R), (G,B), (B,R) (B,G)\},$
:
: }

- C may be represented explicitly or implicitly

Constraint Graph

- Nodes are variables
- Constraints are edges



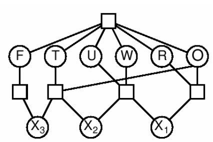
Cryptarithmetic Puzzles

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

Can this be represented as a constraints satisfaction problem?
How?
- variables?
- domains?
- constraints?

Cryptarithmetic Puzzles

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



Variables: $F\ T\ U\ W\ R\ O\ X_1\ X_2\ X_3$
Domains: $\{0,1,2,3,4,5,6,7,8,9\}$
Constraints
- *alldiff*(F,T,U,W,R,O)
- $O+O=R+10\cdot X_1$, etc.

Cryptarithmetic Puzzles

- SEND + MORE = MONEY
- How do humans solve this puzzle?
- M=1->S=8 or 9->

Sudoku

- Each row, column, box must have the numbers from 1..9 in it exactly once
- 9 Rows A..I
- 9 Columns 1..9
- Variables?
- Domains?
- Constraints?

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

Sudoku

- Variables
  - A1,A2,A3,...
- Domains
  - {1..9}
- Constraints
  - A1,A2,A3,...A9=
  - {1,2,3,4,5,6,7,8,9} or
  - {2,1,3,4,5,...}...

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

Varieties of CSP

- Discrete variables
  - Finite domains
  - Infinite domains
    - Linear constraints solvable, non-linear constraints are undecidable
- Continuous variables
  - Linear constraints

Varieties of CSP

- Unary constraints: Only 1 variable affected
  - V1 = red
- A binary CSP: each constraint relates at most two variables
  - V1 != V2
- Higher order (More than two variables)
  - Cryptoarithmetic constraints
  - Map colouring in general
- Preferences (Soft constraints)
  - Red is better than blue
  - Implemented as cost function for value assignment
  - But often more complex. More later

Real World CSP Problems

- Teaching assignments
- Timetabling
- Hardware configuration (VLSI layout)
- Logistics (transport scheduling)
- Job shop scheduling (Operations research)

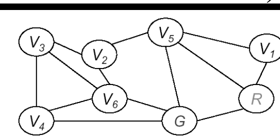
Solving a CSP problem

- A CSP problem can be solved through search
- Definition of a search problem (Search space)
  - States
  - Successor function
  - Goal
- What are these in a CSP problem?

Solving a CSP problem

- Search space
  - States: Partial assignment of values to variables
    - S1={V1=R,V2=G,V3=?,V4=R} ...
  - Initial State: S_Initial = {V1=?,V2=?,...Vn=?}
  - Goal State: All variables are assigned a value and all constraints are satisfied
  - Successor: {...,Vk=?,...} -> {...,Vk=R,...}
  - Cost function (for heuristic search): 0

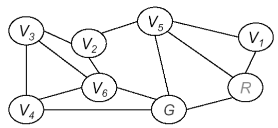
Solving a CSP problem (Exhaustive search)



START =(V_1=? V_2=? V_3=? V_4=? V_5=? V_6=?)
succs(START) =
(V_1=R V_2=? V_3=? V_4=? V_5=? V_6=?)
(V_1=G V_2=? V_3=? V_4=? V_5=? V_6=?)
(V_1=B V_2=? V_3=? V_4=? V_5=? V_6=?)

- How many possible successor states for initial?
- Is the order of the assignment important?
- What is the depth of the search tree?

Solving a CSP problem (Exhaustive search)



START =(V_1=? V_2=? V_3=? V_4=? V_5=? V_6=?)

succs(START) =

(V_1=R V_2=? V_3=? V_4=? V_5=? V_6=?)
(V_1=G V_2=? V_3=? V_4=? V_5=? V_6=?)
(V_1=B V_2=? V_3=? V_4=? V_5=? V_6=?)

- How many possible successor states for initial? 6 * 3 = 18
- Is the order of the assignment important? No
- What is the depth of the search tree? 6

Solving a CSP problem

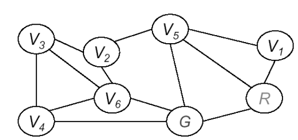
- Total size of the state space: 3^6 = 729
- Exhaustive search space:
  - 18 * 15 * 12 * 9 * 6 * 3 = 524880
  - Extremely inefficient (6!) 720 times larger
- Ordered search space: 729
- BUT: order is important if we use smarter search methods

Solving a CSP problem

- Search strategy:
  - Breadth-first search
  - Depth first search
  - Beam search
  - A* search
  - IDA*

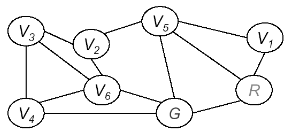
Breadth First Search

- Show the execution of BFS on the sample problem
- Why is BFS not suitable for CSP problems?



Depth First Search

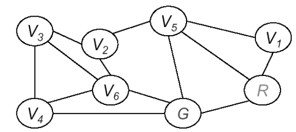
- Possibility of finding a solution quickly
- Trace the DFS algorithm on this problem. Assign values in the order B,G,R
- Looks pretty stupid, because it does not check constraints
- Generate and Test algorithm
- 6109 expanded nodes



- <B,?,?,?,?,?>
- <B,B,?????>
- <B,B,B,???> Aaargghhh!!!
- ...

Backtracking Search Check Constraints

- Obvious improvement
  - Backtrack if constraints are violated
- Trace the execution of Backtracking search
- 15 steps until it finds a solution
- Computational overhead?

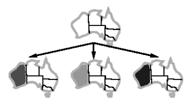


Backtracking Search

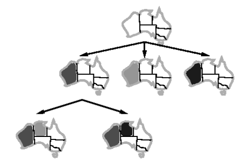
```
function BACKTRACKING-SEARCH(csp) returns solution/failure
   return RECURSIVE-BACKTRACKING([], csp)
function RECURSIVE-BACKTRACKING(assigned, csp) returns solution/failure
   if assigned is complete then return assigned
   var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assigned, csp)
   for each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


Backtracking Example

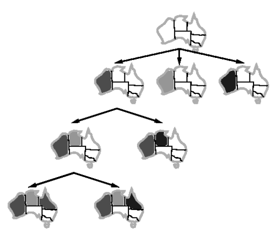
Backtracking Example



Backtracking Example



Backtracking Example



Improving Backtracking

- What information can we use to make backtracking more efficient?
- How to speed up the general algorithm?

Improving Backtracking

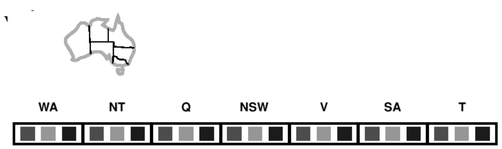
- The problem is when we backtrack!
  - Avoid backtracking
  - Smarter backtracking
- Implementation
  - Order of variables
  - Order of values
- Break up into smaller problems

Forward Checking

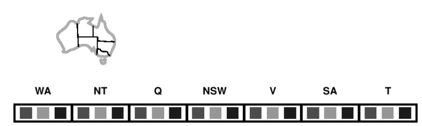
- At the start, record the set of all legal values
- If you assign a variable, remove from all other nodes values that are now not legal anymore
- If a node's set of legal values becomes empty, then backtrack immediately
- Efficient way to check the constraints

Forward Checking

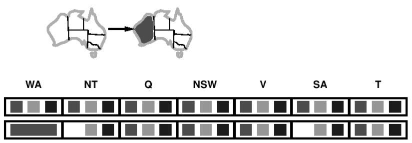
- Keep track of remaining values for all variables
- Backtrack if any variable has no more legal



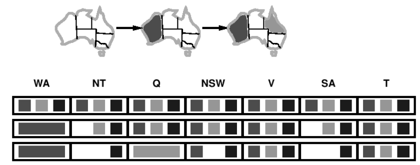
Forward Checking



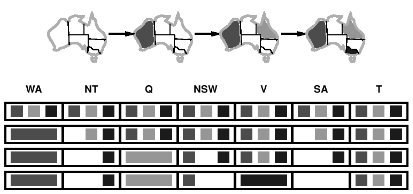
Forward Checking



Forward Checking

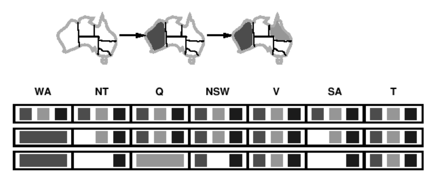


Forward Checking



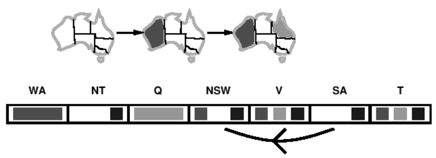
Constraint Propagation

- Forward checking can not detect all failures
- May still need to backtrack
- NT and SA can not both be blue!



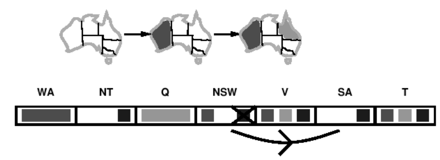
Arc Consistency

- Simplest form makes arcs consistent
- X -> Y is consistent iff
  - for every value of X, there is some value of Y



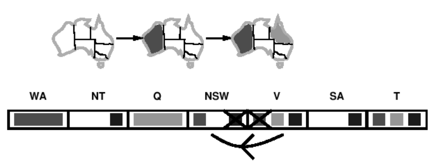
Arc Consistency

- Simplest form makes arcs consistent
- X -> Y is consistent iff
  - for every value of X, there is some value of Y



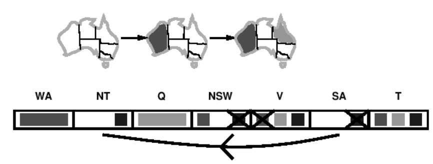
Arc Consistency

- If X looses a value, neighbors of X need to be rechecked



Arc Consistency

- Arc consistency detects failures earlier than forward checking
- Can be run as a preprocessor after each assignment



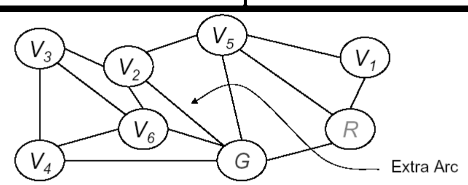
Arc Consistency Algorithm

```
function AC3( csp) returns the CSP, possibly with reduced domains
   local variables: queue, a queue of arcs, initially all the arcs in csp

   loop while queue is not empty do
      (X_i, X_j) ← REMOVE-FRONT(queue)
      if REMOVE-INCONSISTENT(X_i, X_j) then
         for each X_k in NEIGHBORS[X_i] do
            add (X_k, X_i) to queue

function REMOVE-INCONSISTENT( 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]
         then delete x from DOMAIN[X_i]; removed ← true
   return removed
```


Constraint Propagation

- Forward checking creates the set of legal values only at the start
- Domain of variables are only updated if it is mentioned in a constraint directly
- Constraint propagation carries this further:
  - If you delete a value from the domain of a variable
  - Then propagate the change to all other variables

Constraint Propagation Example 2



- In this case, no backtracking. Not always this good
- Constraint propagation can be done
  - Preprocessing
  - Dynamically, more expensive when backtracking

Graph Colouring and Constraint Propagation

- In graph colouring problems, CP is simple
- If a node has only one colour left, propagate this colour to all neighbors

PropagateColorAtNode(node,color)

1. remove color from all of “available lists” of our uninstantiated neighbors.
2. If any of these neighbors gets the empty set, it’s time to backtrack.
3. Foreach n in these neighbors: if n previously had two or more available colors but now has only one color c, run PropagateColorAtNode(n,c)

Constraint Propagation

- In general CSP problems, CP is more useful than just propagating if a variable was assigned a specific value

General Constraint Propagation Algorithm

General Constraint Propagation

```
Propagate(A1, A2 ,… An)
    finished = FALSE
    while not finished
        finished = TRUE
        foreach constraint C
            Assume C concerns variables V1, V2 ,… Vk
            Set NewAV1 = {} , NewAV2 = {} , … NewAVk = {}
            Foreach assignment (V1=x1, V2=x2, … Vk=xk) in C
                If x1 in AV1 and x2 in AV2 and … xk in AVk
                    Add x1 to NewAV1 , x2 to NewAV2 ,… xk to NewAVk
            for i = 1 , 2 … k
                AVi := AVi intersection NewAVi
                If AVi was made smaller by that intersection
                    finished = FALSE
                If AVi is empty, we're toast. Break out with "Backtrack" signal.
```

Specification: Takes a set of availability-lists for each and every node and uses all the constraints to filter out impossible values that are currently in availability lists

Details on next slide

Slide 15

General Constraint Propagation Algorithm

General Constraint Propagation


```
Propagate(A1, A2 ,… An)
    finished = FALSE
    while not finished
        finished = TRUE
        foreach constraint C
            Assume C concerns variables V1, V2 ,… Vk
            Set NewAV1 = {} , NewAV2 = {} , … NewAVk = {}
            Foreach assignment (V1=x1, V2=x2, … Vk=xk) in C
                If x1 in AV1 and x2 in AV2 and … xk in AVk
                    Add x1 to NewAV1 , x2 to NewAV2 ,… xk to NewAVk
            i = 1 , 2 … k
                AVi := AVi intersection NewAVi
                naller by that intersection
                LSE
                're toast. Break out with "Backtrack" signal.
```


A_i denotes the current set of possible values for variable i. This is call-by-reference. Some of the A_i sets may be changed by this call (they'll have one or more elements removed)

We'll keep iterating until we do a full iteration in which none of the availability lists change. The "finished" flag is just to record whether a change took place.

Slide 16

General Constraint Propagation Algorithm

General Constraint Propagation


```
Propagate(A1, A2 ,… An)
    finished = FALSE
    while not finished
        finished = TRUE
        foreach constraint C
            Assume C concerns variables V1, V2 ,… Vk
            Set NewAV1 = {} , NewAV2 = {} , … NewAVk = {}
            Foreach assignment (V1=x1, V2=x2, … Vk=xk) in C
                If x1 in AV1 and x2 in AV2 and … xk in AVk
                    Add x1 to NewAV1 , x2 to NewAV2 ,… xk to NewAVk
            for i =
                AVi := AVi intersection
                If AVi was made smaller by that inte
                    finished = FALSE
                If AVi is empty, we're toast. Break
```


NewA$_i$ is going to be filled up with the possible values for variable V_i taking into account the effects of constraint C

After we've finished all the iterations of the foreach loop, *NewA*$_i$ contains the full set of possible values of variable V_i taking into account the effects of constraint C.

General Constraint Propagation Algorithm

General Constraint Propagation

```
Propagate(A1, A2 ,… An)
    finished = FALSE
    while not finished
        finished = TRUE
        foreach constraint C
            Assume C concerns variable … Vk
            Set NewAV1 = {} , NewAV2 = {}  wAVk = {}
            Foreach assignment (V1=x1, V2  Vk=xk) in C
                If x1 in AV1 and x2 in AV2 and  in AVk
                    Add x1 to NewAV1 , x2 to Ne  2 ,… xk to NewAVk
            for i = 1 , 2 … k
                AVi := AVi intersection NewAVi
                If AVi was made smaller by that intersection
                    finished = FALSE
                If AVi  is empty, we're toast. Break out with "Backtrack" signal.
```

If this test is satisfied that means that there's at least one value q such that we originally thought q was an available value for V_i but we now know q is impossible.

If A_{Vi} is empty we've proved that there are no solutions for the availability-lists that we originally entered the function with

Slide 18

Semi-Magical Squares

- Rows and columns sum up to 6
- One diagonal sums up to 6
- How to represent the constraints?

V_1	V_2	V_3	This row must sum to 6
V_4	V_5	V_6	This row must sum to 6
V_7	V_8	V_9	This row must sum to 6
This column must sum to 6	This column must sum to 6	This column must sum to 6	This diagonal must sum to 6

CP Example: Semi-Magic Square

- V1 is fixed at 1
- Each variable can have values 1, 2, or 3

1	123	123	This row must sum to 6
123	123	123	This row must sum to 6
123	123	123	This row must sum to 6
This column must sum to 6	This column must sum to 6	This column must sum to 6	This diagonal must sum to 6

CP: Semi Magic Square

Propagate on Se

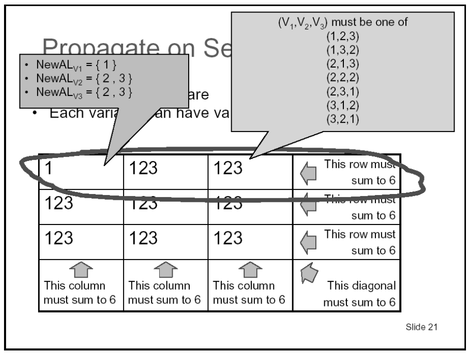
- The semi magic square
- Each variable can have va

(V_1,V_2,V_3) must be one of
(1,2,3)
(1,3,2)
(2,1,3)
(2,2,2)
(2,3,1)
(3,1,2)
(3,2,1)

1	123	123	This row must sum to 6
123	123	123	This row must sum to 6
123	123	123	This row must sum to 6
This column must sum to 6	This column must sum to 6	This column must sum to 6	This diagonal must sum to 6

Slide 20

CP: Semi Magic Squares



After first row constraint

1	23	23	This row must sum to 6
123	123	123	This row must sum to 6
123	123	123	This row must sum to 6
This column must sum to 6	This column must sum to 6	This column must sum to 6	This diagonal must sum to 6

After all row and column constraints

1	23	23	This row must sum to 6
23	123	123	This row must sum to 6
23	123	123	This row must sum to 6
This column must sum to 6	This column must sum to 6	This column must sum to 6	This diagonal must sum to 6

After diagonal constraint

1	23	23	This row must sum to 6
23	23	123	This row must sum to 6
23	123	23	This row must sum to 6
This column must sum to 6	This column must sum to 6	This column must sum to 6	This diagonal must sum to 6

- Iterated through all constraints once
- What happens in the next iteration?

Next Iteration

1	23	23	This row must sum to 6
23	23	12	This row must sum to 6
23	12	23	This row must sum to 6
This column must sum to 6	This column must sum to 6	This column must sum to 6	This diagonal must sum to 6

- Constraints apply even if no variable is down to a single value
- Next iteration?

CSP Search with Constraint Propagation

CSP Search with Constraint Propagation

CPSearch($A_1, A_2, \ldots A_n$)
Let i = lowest index such that A_i has more than one value
foreach available value x in A_i
foreach k in 1, 2.. n
Define $A'_k := A_k$
$A'_i := \{ x \}$
Call Propagate($A'_1, A'_2, \ldots A'_n$)
If no "Backtrack" signal
If $A'_1, A'_2, \ldots A'_n$ are all unique we're done!
Recursively Call CPSearch($A'_1, A'_2, \ldots A'_n$)

Details on next slide

Slide 26

CSP Search with Constraint Propagation

CSP Search with Constraint Propagation

Specification: Find out if there's any combination of values in the combination of the given availability lists that satisifes all constraints.

CPSearch($A_1, A_2, \ldots A_n$)
Let i = lowest index such that A_i has more than one value
foreach available value x in A_i
foreach k in 1, 2.. n
Define $A'_k := A_k$
$A'_i := \{ x \}$
Call Propagate($A'_1, A'_2, \ldots A'_n$)
If no "Backtrack" signal
If $A'_1, A'_2, \ldots A'_n$ are all unique we're done!
Recursively Call CPSearch($A'_1, A'_2, \ldots A'_n$)

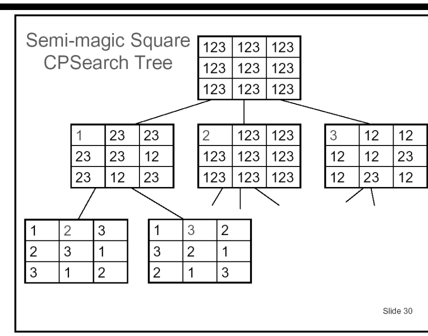
At this point the A-primes are a copy of the original availability lists except A'_i has committed to value x.

This call may prune away some values in some of the copied availability lists

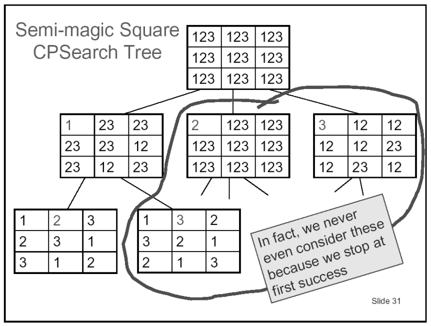
Assuming that we terminate deep in the recursion if we find a solution, the CPSeach function only terminates normally if no solution is found.

Slide 27

Example

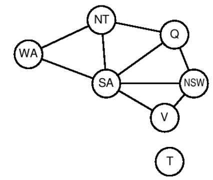


Example



Problem Structure

- Tasmania and mainland are separate problems
- Identifiable as connected components of the constraint graph



Problem Structure

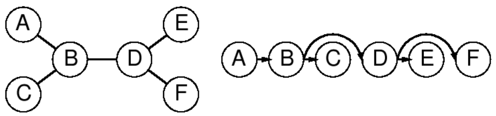
- Suppose each problem has c variables out of n total
- Worst case solution time is n/c * d^c
- Linear in n, very good
- E.g., n = 80, d = 2, c = 20
  - 2^80 = 4 billion years at 10 million nodes/sec.
  - 4 * 2^20 = 0.4 secs. at 10 million nodes/sec.

Tree Structured CSPs

- Theorem: If the constraint graph has no loops then the CSP can be solved in O(n d^2) time
- General CSP: Worst case is O(d^n)
- This property also applies to logical and probabilistic reasoning: syntactic restrictions and the complexity of reasoning

Tree Structured CSP

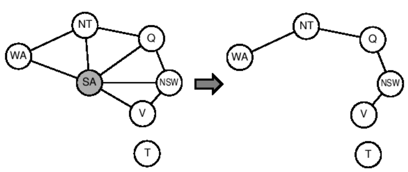
- Algorithm
  - Choose a variable as root
  - Order variables from root to leaves such that every nodes parent precedes it in the ordering
  - For j = n to 2, apply RemoveInconsistent(Parent(Xj), Xj)
  - For j = 1 to n, assign Xj consistent with Parent(Xj)



Nearly Tree Structured CSP

- Conditioning: Instance a variable, prune its neighbors domains
- Cutset conditioning: Instantiate (in all ways) a set of variables in such a way that the remaining constraint graph is a tree
- Cutset size c -> runtime O(d^c * (n – c)d^2)
- Very fast for small c

Nearly Tree Structured CSPs



Scheduling

- A very big, important use of CSP methods.
  - Used in many industries. Makes many multi-million dollar decisions.
  - Used extensively for space mission planning.
  - Military uses
- Problems with phenomenally huge state spaces. But for which solutions are needed very quickly.
- Many kinds of scheduling problems e.g.:
  - Job shop: Discrete time; weird ordering of operations possible; set of separate jobs.
  - Batch shop: Discrete or continuous time; restricted operation of ordering; grouping is important.
  - Manufacturing cell: Discrete, automated version of open job shop.

Job Shop Scheduling

- Make various products. Each product is a job
  - Job1: Make a polished thing with a hole.
  - Job2: Paint and drill a hole in a widget
- Each job requires several operations
  - Operations for Job1: polish, drill
  - Operations for Job2: paint, drill

Job Shop Scheduling

- Each operation needs several resources
- Polishing needs
  - Polishing machine
  - Polishing expert Pat
- Drilling
  - Drilling machine
  - Pat or drill expert Dave

Job Shop Scheduling

- Order constraints
- Some of the operations have to be done in a specific order
  - Drill before you paint

Formal Definition of a Job Shop

A Job Shop problem is a pair (J , RES)
J is a set of jobs $J = \{j_1, j_2, \ldots j_n\}$
RES is a set of resources $RES = \{R_1 .. R_m\}$

Each job j_l is specified by:

- a set of operations $O^l = \{O^l_1\ O^l_2 \ldots O^l_{n(l)}\}$
- and must be carried out between release-date rd_l and due-date dd_l.
- and a partial order of operations: (O^l_i before O^l_j), ($O^l_{i'}$ before $O^l_{j'}$), etc…

Each operation O^l_i has a variable start time st^l_i and a fixed duration du^l_i and requires a set of resources. e.g.: O^l_i requires $\{ R^l_{i1}, R^l_{i2} \ldots \}$.

Each resource can be accomplished by one of several possible physical resources, e.g. R^l_{i1} might be accomplished by any one of $\{r^l_{ij1}, r^l_{ij2}, \ldots\}$. Each of the r^l_{ijk}s are a member of RES.

Job Shop Example

j_1 = *polished-hole-thing* = { O^1_1 , O^1_2 }
j_2 = *painted-hole-widget* = { O^2_1 , O^2_2 }
RES = { Pat,Chris,Drill,Paint,Drill,Polisher }
O^1_1 = *polish-thing: need resources…*
  { R^1_{11} = *Pat* , R^1_{12} = *Polisher* }
O^1_2 = *drill-thing: need resources…*
  { R^1_{21} = (r^1_{211}=*Pat* or r^1_{212}=*Chris*), R^1_{22} = *Drill* }
O^2_1 = *paint-widget: need resources…*
  { R^2_{11} = *Paint* }
O^2_2 = *drill-widget : need resources…*
  { R^2_{21} = (r^2_{211}=*Pat* or r^2_{212}=*Chris*), R^2_{22} = *Drill* }
Precedence constraints : O^2_2 before O^2_1. All operations take one time unit du^l_i = 1 forall *i*,*l*. Both jobs have release-date rd^l = 0 and due-date dd^l = 1.

Job Shop as a Constraint Satisfaction Problem

- How do we solve a JS problem?
- How do we represent it as a CSP?

Variables
- The operation state times st^l_i
- The resources R^l_{ij} (usually these are obvious from the definition of O^l_i. Only need to be assigned values when there are alternative physical resources available, e.g. *Pat* or *Chris* for operating the *drill*).

Constraints:
- Precedence constraints. (Some O^l_is must be before some other O^l_js).
- Capacity constraints. There must never be a pair of operations with overlapping periods of operation that use the same resources.

JS Example

$O^1_1\ R_1 \xrightarrow{\text{before}} O^1_2\ R_2 \xrightarrow{\text{before}} O^1_3\ R_3$

$O^2_1\ R_1 \xrightarrow{\text{before}} O^2_2\ R_2$

$O^3_1\ R_3 \xrightarrow{\text{before}} O^3_2\ R_1 \xrightarrow{\text{before}} O^3_3\ R_2$

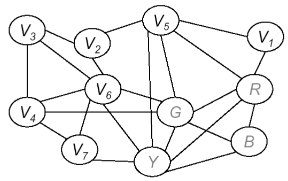
$O^4_1\ R_4 \xrightarrow{\text{before}} O^4_2\ R_2$

Example from [Sadeh and Fox, 96]: Norman M. Sadeh and Mark S. Fox, Variable and Value Ordering Heuristics for the Job Shop Scheduling Constraint Satisfaction Problem, Artificial Intelligence Journal, Number Vol 86, No1, pages 1-41, 1996. Available from citeseer.nj.nec.com/sadeh96variable.html

- 4 jobs. 3 units each.
- Release date =0, Due date = 15

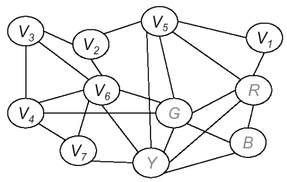
CSP Heuristics

- Graph colouring with four colours
- Which node to colour first?



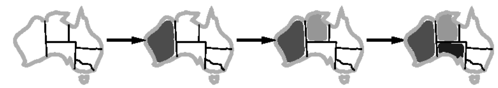
CSP Heuristics: Variable ordering

- Most constrained variable first
- Most constraining variable first

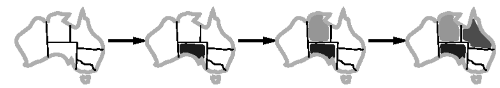


Variable Ordering

- Most constrained variable

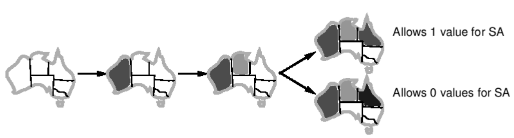


- Most constraining variable (tie-breaker)



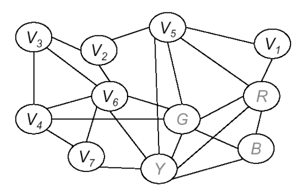
Variable Ordering

- Least constraining variable



CSP Heuristics: Value ordering

- Least constrained value :- choose the value that causes the smallest reduction in the number of connected variables



Sadeh and Fox
General CSP Algorithm

(From Sadeh+Fox)

1. If all values have been successfully assigned then stop, else go on to 2.
2. Apply the consistency enforcing procedure (e.g. forward-checking if feeling computationally mean, or constraint propagation if extravagant. There are other possibilities, too.)
3. If a deadend is detected then backtrack (simplest case: DFS-type backtrack. Other options can be tried, too). Else go on to step 4.
4. Select the next variable to be assigned (using your variable ordering heuristic).
5. Select a promising value (using your value ordering heuristic).
6. Create a new search state. Cache the info you need for backtracking. And go back to 1.

- Best methods for steps 2,3,4,5, and 6 depend on the domain

Job Shop Example
Consistency Enforcement

- Sadeh claims that generally forward-checking is better, computationally, than full constraint propagation. But it can be supplemented with a Job-shop specific trick.
- The precedence constraints (i.e. the available times for the operations to start due to the ordering of operations) can be computed exactly, given a partial schedule, very efficiently.

CSP and Reactive Methods

- Say you have built a large schedule.
- Disaster! Halfway through execution, one of the resources breaks down. We have to reschedule!
- Bad to have to wait 15 minutes for the scheduler to make a new suggestion.
- Efficient schedule repair methods
- Take possibility of breakdown into consideration from the start
  - Plans that are easy to fix
  - Soft constraints (Preferences)

Other Approaches

- Local methods (e.g., Hill climbing, Tabu search)
- Genetic algorithms, ANN, Simulated Annealing

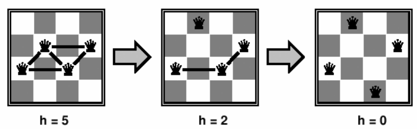
4 Queens Problem

States: 4 queens in 4 columns ($4^4 = 256$ states)

Operators: move queen in column

Goal test: no attacks

Evaluation: $h(n)$ = number of attacks

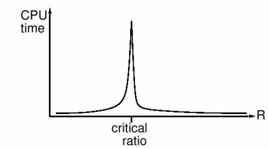


Performance of Min-conflicts

Given random initial state, can solve n-queens in almost constant time for arbitrary n with high probability (e.g., n = 10,000,000)

The same appears to be true for any randomly-generated CSP except in a narrow range of the ratio

$$R = \frac{\text{number of constraints}}{\text{number of variables}}$$



Summary

- Map colouring and the four colour theorem
- Formal and informal definition of CSP problems
  - How to convert a problem into a CSP problem
- Backtracking search, forward checking, constraint propagation
- Job Shop scheduling as a CSP problem

References

- Four Colour Theorem:
  - http://www.math.gatech.edu/~thomas/FC/fourcolor.html
  - http://mathworld.wolfram.com/Four-ColorTheorem.html
- Constraint satisfaction
  - Russel and Norvig, AI a Modern Approach, Chapter 5
  - Andrew Moore slides, www.cs.cmu.edu/~awm/tutorials (Many slides taken from his CSP presentation)