Constraint Satisfaction (CSP) and Scheduling Problems

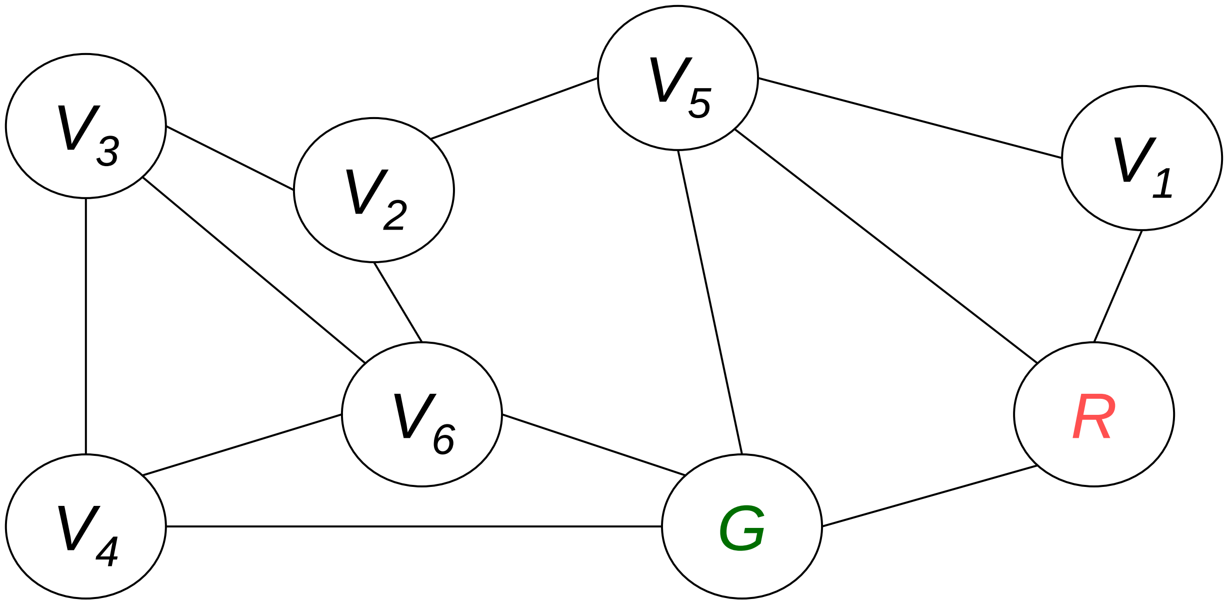
Department of Electrical and Electronics Engineering
Spring 2005
Dr. Afşar Saranlı

Thanks to Professor Andrew W. Moore (Carnegie Mellon University) http://www.cs.cmu.edu/~awm/tutorials
Also: Artificial Intelligence: A Modern Approach, 2nd Ed., Russel & Norvig

Overview

- CSPs defined
- Using standard search for CSPs
- Blindingly obvious improvements
  - Backtracking search
  - Forward Checking
  - Constraint Propagation
- Some example CSP applications
  - Overview
  - Waltz Algorithm
  - Job Shop Scheduling
- Variable ordering heuristic
- Value ordering heuristic
- Discussion

A Constraint Satisfaction Problem



- Inside each circle marked V_1 .. V_6 we must assign: *R*, *G* or *B*.
- No two connected circles may be assigned the same symbol.
- Notice that two circles have already been given an assignment.

Formal Constraint Satisfaction 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:

$V = \{ V_1 , V_2 , V_3 , V_4 , V_5 , V_6 \}$

$D = \{ R , G , B \}$

$C = \{ (V_1,V_2) : \{ (R,G), (R,B), (G,R), (G,B), (B,R)\ (B,G)\},$

$\{ (V_1,V_3) : \{ (R,G), (R,B), (G,R), (G,B), (B,R)\ (B,G)\},$

:

: }

- Obvious point: Usually *C* isn't represented explicitly, but by a function.

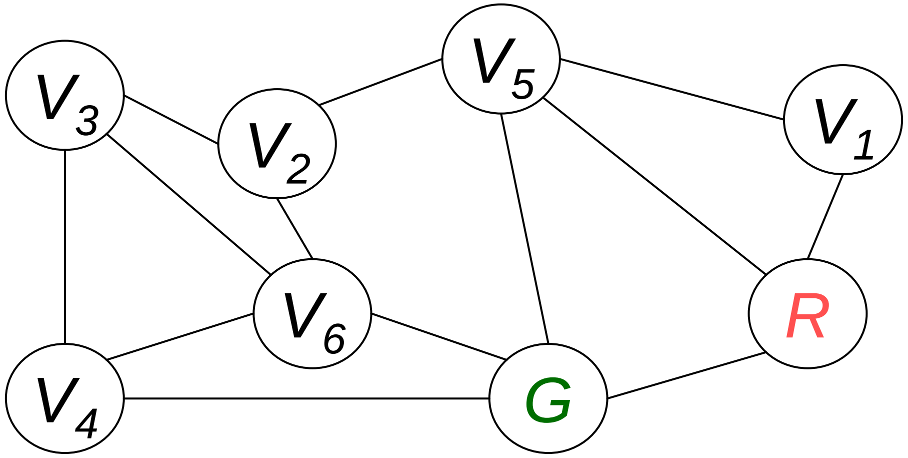
Example: 8-queens problem

- 8 queens in an 8x8 chess board,
- No queens attacking each other
- Let us give it a manual try…

http://www.pvv.org/~hgs/java/queens/simple.html

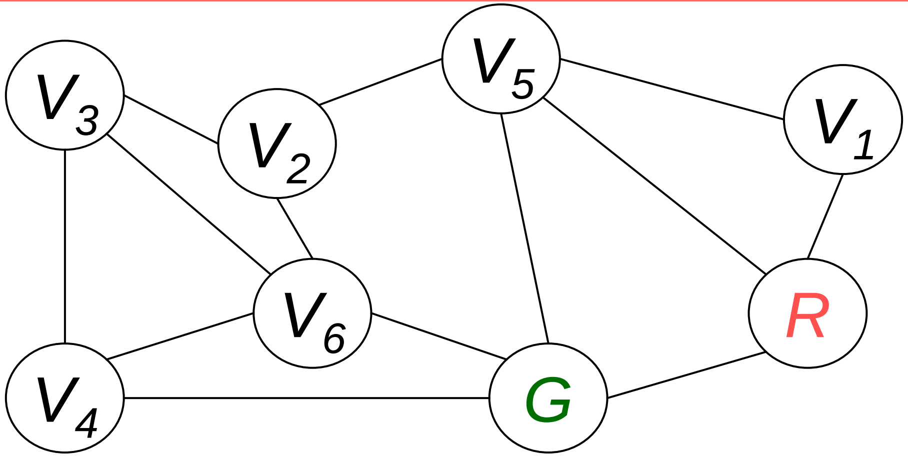
Can it be formulated as a CSP? Yes!

How to Solve our CSP?



- How about using a search algorithm?
- Define: a search state has variables 1 … *k* assigned. Values *k+1* … *n*, as yet unassigned.
- Start state: All unassigned.
- Goal state: All assigned, and all constraints satisfied.
- Successors of a stated with V_1 … V_k assigned and rest unassigned are all states (with V_1 … V_k the same) with V_{k+1} assigned a value from *D*.
- Cost on transitions: 0 is fine. We don't care. We just want any solution.

How to solve our CSP?



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=?)

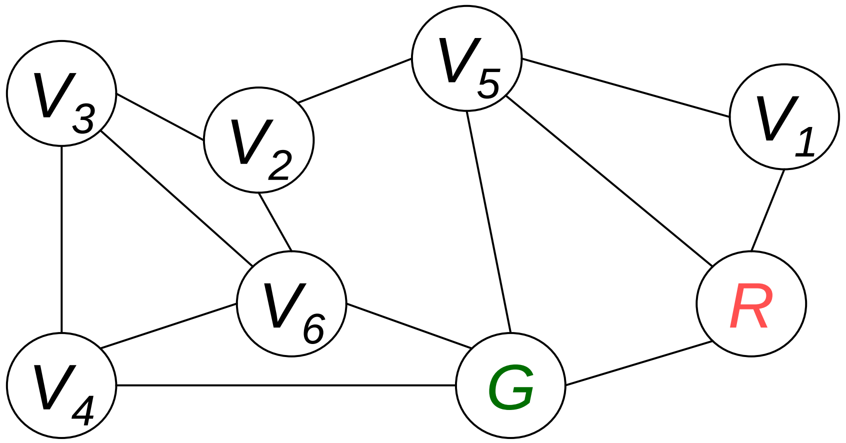
(V_1=G V_2=? V_3=? V_4=? V_5=? V_6=?)

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

What search algorithms could we use?

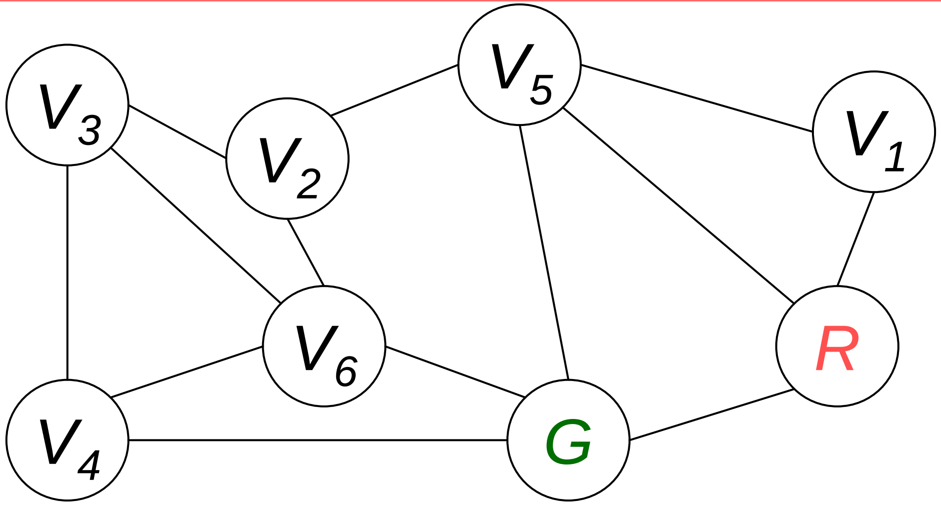
It turns out BFS is not a popular choice. Why not?

Depth-first-search (DFS) for CSPs



- What about DFS?
- Much more popular. At least it has a chance of finding an easy answer quickly.
- What happens if we do DFS with the order of assignments as *B* tried first, then *G* then *R*?
- This makes DFS look very, very stupid!
- Example: http://www.cs.cmu.edu/~awm/animations/constraint/9d.html

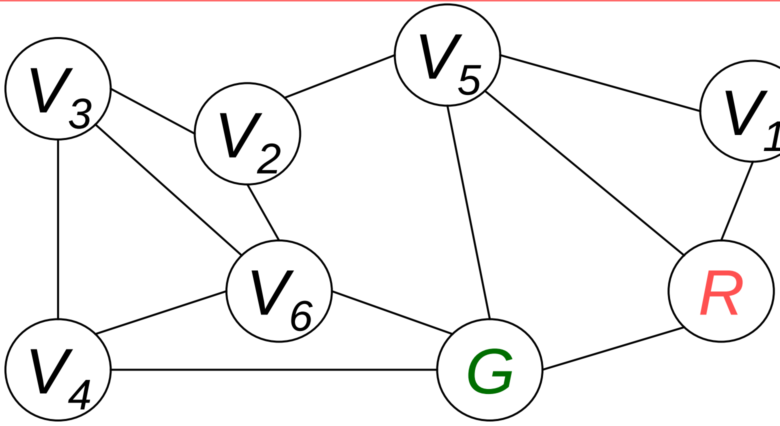
Obvious Improvement: Consistency checking ("Backtracking Search")



Don't ever try successor which causes inconsistency with its neighbors.

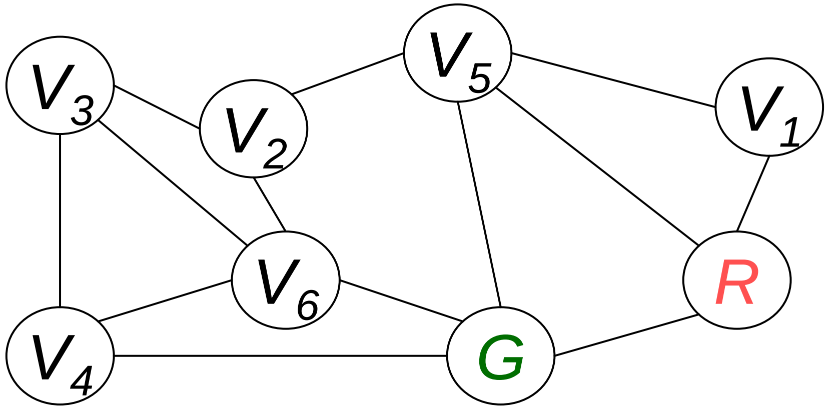
- Again, what happens if we do DFS with the order of assignments as *B* tried first, then *G* then *R*?
- What's the computational overhead for this?
- Backtracking still looks a little stupid!
- Examples: http://www.cs.cmu.edu/~awm/animations/constraint/9b.html and http://www.cs.cmu.edu/~awm/animations/constraint/27b.html

Obvious improvement: Forward checking



- At start, for each variable, record the current set of possible legal values for it.
- When you assign a value in the search, update set of legal values for all variables. Backtrack immediately if you empty a variable's constraint set.
  - Again, what happens if we do DFS with the order of assignments as *B* tried first, then *G* then *R*?
  - Example: http://www.cs.cmu.edu/~awm/animations/constraint/27f.html
  - What's the computational overhead?

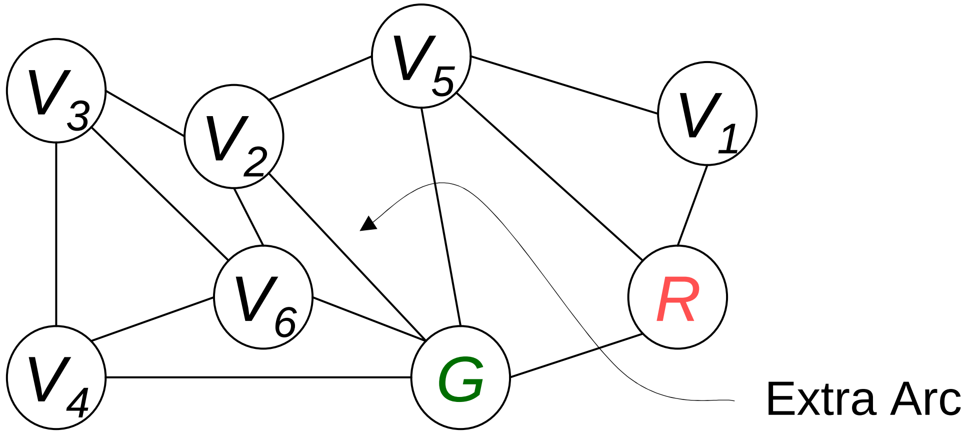
Constraint Propagation



- Forward checking computes the domain of each variable independently at the start, and then only updates these domains when assignments are made in the DFS that are directly relevant to the current variable.
- Constraint Propagation carries this further. When you delete a value from your domain, check all variables connected to you. If any of them change, delete all inconsistent values connected to them, etc…
- In the above example it is useless

Web Example:
  http://www.cs.cmu.edu/~awm/animations/constraint/27p.html

Constraint Propagation being useful



- In this example, constraint propagation solves the problem without search … Not always that lucky!
- Constraint propagation can be done as a preprocessing step. (Cheap).
- Or it can be maintained dynamically during the search. Expensive: when you backtrack, you must undo some of your additional constraints.

Graph-coloring-specific Constraint Propagation

In the case of Graph Coloring, CP looks simple: after we've made a search step (instantiated a node with a color), propagate the color at that node.

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)

Graph-Coloring-specific Constraint Propagation

In the case of Graph Coloring, CP looks simple: after we've made a search step (instantiated a node with a color), propagate the color at that node.

PropagateColorAtNode(node,color)

1. remove color from all of "available lists" of our uninstantiated neighbors.
2. If any of these neig
3. I
   n ..., run
   P

But for General CSP problems, constraint propagation can do much more than only propagating when a node gets a unique value…

A new CSP (where fancier CP is possible)

- The semi magic square
- Each variable can have value 1, 2 or 3

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

General Constraint Propagation

```
Propagate($A_1, A_2, \ldots A_n$)
    finished = FALSE
    while not finished
        finished = TRUE
        foreach constraint C
            Assume C concerns variables $V_1, V_2, \ldots V_k$
            Set $NewA_{V1} = \{\}$ , $NewA_{V2} = \{\}$ , … $NewA_{Vk} = \{\}$
            Foreach assignment $(V_1=x_1, V_2=x_2, \ldots V_k=x_k)$ in C
                If $x_1$ in $A_{V1}$ **and** $x_2$ in $A_{V2}$ **and …** $x_k$ in $A_{Vk}$
                    Add $x_1$ to $NewA_{V1}$ , $x_2$ to $NewA_{V2}$ ,… $x_k$ to $NewA_{Vk}$
            for i = 1 , 2 … k
                $A_{Vi} := A_{Vi}$ intersection $NewA_{Vi}$
                If $A_{Vi}$ was made smaller by that intersection
                    finished = FALSE
                If $A_{Vi}$ 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

General Constraint Propagation

```
Propagate($A_1$, $A_2$ ,… $A_n$)
    finished = FALSE
    while not finished
        finished = TRUE
        foreach constraint C
            Assume C concerns variables $V_1$, $V_2$ ,… $V_k$
            Set NewA$_{V1}$ = {} , NewA$_{V2}$ = {} , … NewA$_{Vk}$ = {}
            Foreach assignment ($V_1$=$x_1$, $V_2$=$x_2$, … $V_k$=$x_k$) in C
                If $x_1$ in $A_{V1}$ **and** $x_2$ in $A_{V2}$ **and …** $x_k$ in $A_{Vk}$
                    Add $x_1$ to NewA$_{V1}$ , $x_2$ to NewA$_{V2}$ ,… $x_k$ to NewA$_{Vk}$
            r i = 1 , 2 … k
                $A_{Vi}$ := $A_{Vi}$ intersection NewA$_{Vi}$
                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.

General Constraint Propagation

```
Propagate($A_1$, $A_2$ ,… $A_n$)
    finished = FALSE
    while not finished
        finished = TRUE
        foreach constraint C
            Assume C concerns variables $V_1$, $V_2$ ,… $V_k$
            Set NewA$_{V1}$ = {} , NewA$_{V2}$ = {} , … NewA$_{Vk}$ = {}
            Foreach assignment ($V_1$=$x_1$, $V_2$=$x_2$, … $V_k$=$x_k$) in C
                If $x_1$ in $A_{V1}$ and $x_2$ in $A_{V2}$ and … $x_k$ in $A_{Vk}$
                    Add $x_1$ to NewA$_{V1}$ , $x_2$ to NewA$_{V2}$ ,… $x_k$ to NewA$_{Vk}$
            for i = 1 , 2
                $A_{Vi}$ := $A_{Vi}$ intersection Ne
                If $A_{Vi}$ was made smaller by that inte
                    finished = FALSE
                If $A_{Vi}$ 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

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.

```
Propagate($A_1$, $A_2$ ,… $A_n$)
    finished = FALSE
    while not finished
        finished = TRUE
        foreach constraint C
            Assume C concerns variable … $V_k$
            Set NewA$_{V1}$ = {} , NewA$_{V2}$ = {} wA$_{Vk}$ = {}
            Foreach assignment ($V_1$=$x_1$, $V_2$= $V_k$=$x_k$) in C
                If $x_1$ in $A_{V1}$ **and** $x_2$ in $A_{V2}$ **and** in $A_{Vk}$
                    Add $x_1$ to NewA$_{V1}$ , $x_2$ to Nev 2 ,… $x_k$ to NewA$_{Vk}$
            for i = 1 , 2 … k
                A$_{Vi}$ := A$_{Vi}$ intersection NewA$_{Vi}$
                If A$_{Vi}$ was made smaller by that intersection
                    finished = FALSE
                If A$_{Vi}$ is empty, we’re toast. Break out with “Backtrack” signal.
```

If A$_{Vi}$ is empty we’ve proved that there are no solutions for the availability-lists that we originally entered the function with

Propagate on Semi-magic square

- The semi magic square
- Each variable can have value 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

Propagate on Sem

- The semi magic square
- Each variable can have val

(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

Propagate on Sem

- $NewAL_{V1}$ = { 1 }
- $NewAL_{V2}$ = { 2 , 3 }
- $NewAL_{V3}$ = { 2 , 3 }

are

- Each varia an have val

(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

After doing 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

And after the 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

CP has now iterated through all constraints once. But does it make further progress when it tries iterating through them again?

And after another round of constraints…

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

YES! And this showed a case of a constraint applying even when none of the variables involved was down to a unique value.

So.. any more changes on the *next* iteration?

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

CSP Sea t Propagation

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

```
CPSearch($A_1$, $A_2$ ,… $A_n$)
    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($A'_1$, $A'_2$ ,… $A'_n$)
        If no "Backtrack" signal
            If $A'_1$, $A'_2$ ,… $A'_n$ are all unique we're done!
            Recursively Call CPSearch($A'_1$, $A'_2$ ,… $A'_n$)
```

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.

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($A'_1, A'_2, \ldots A'_n$)
```

What's the top-level call?

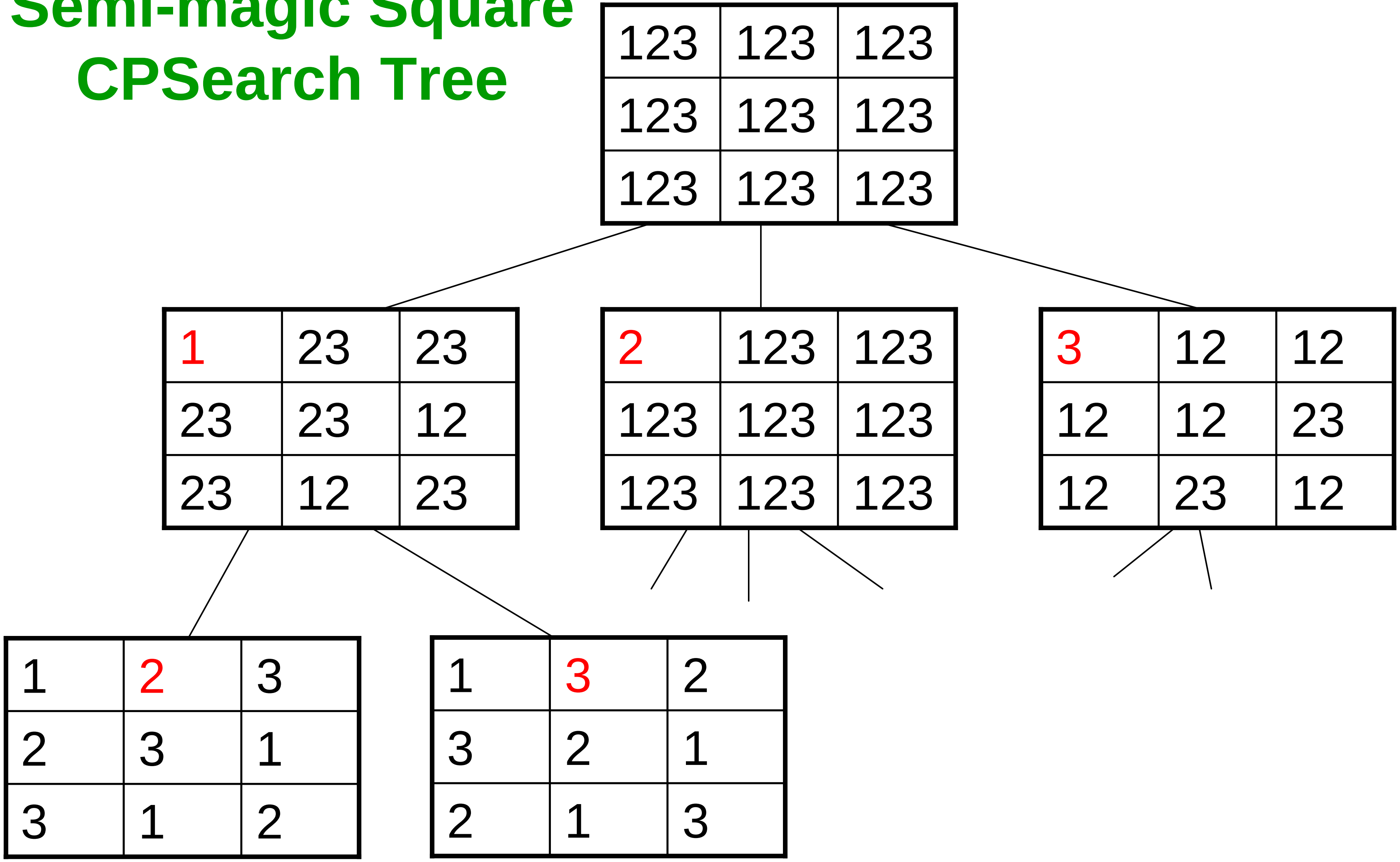
CSP Search with Constraint Propagation

```
CPSearch($A_1, A_2, \dots A_n$)
    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($A'_1, A'_2, \dots A'_n$)
        If no "Backtrack" signal
            If $A'_1, A'_2, \dots A'_n$ are all unique we're done!
            Recursively Call CPSearch($A'_1, A'_2, \dots A'_n$)
```

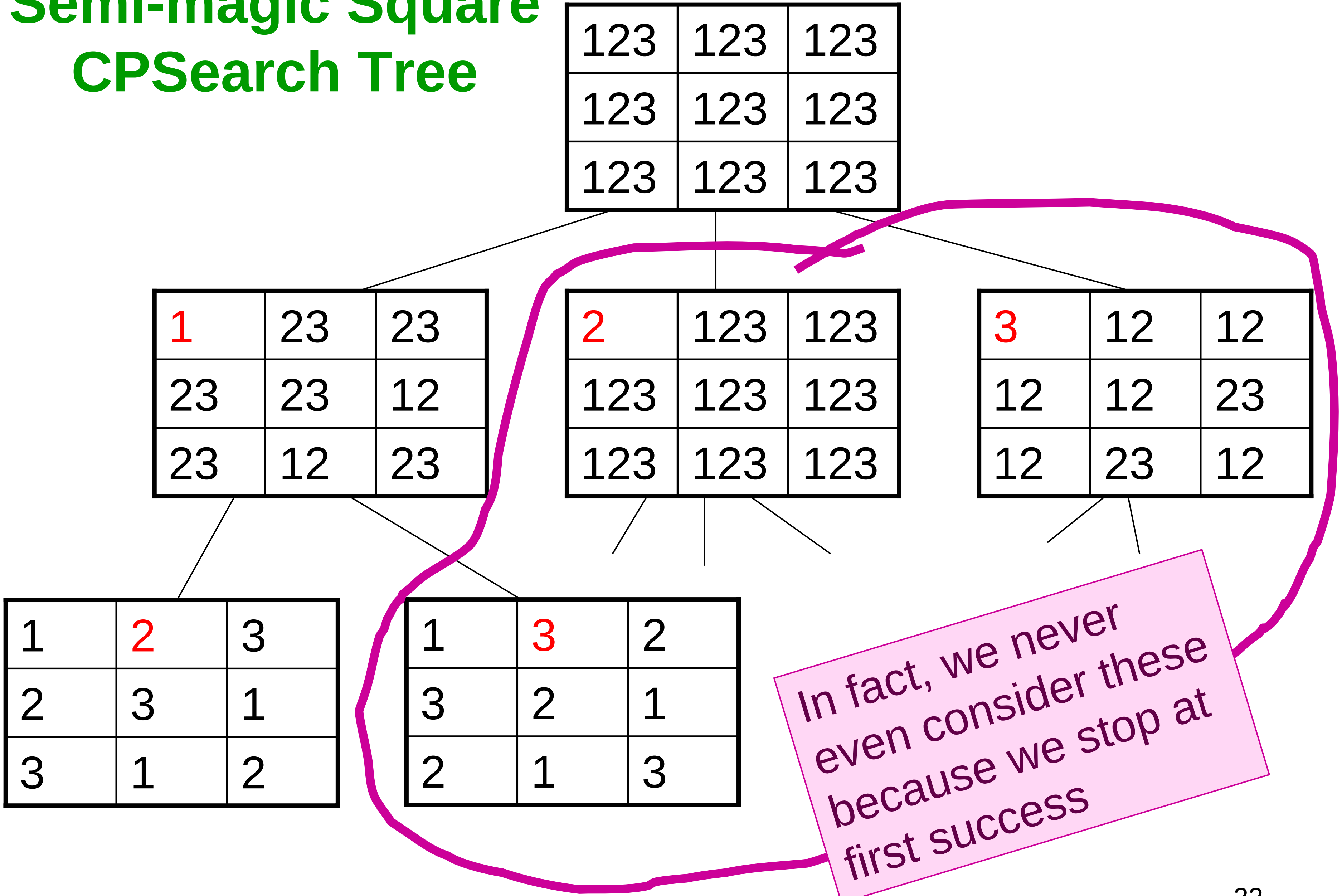
What's the top-level call?

Call with that A_i = complete set of possible values for V_i.

Semi-magic Square CPSearch Tree



Semi-magic Square CPSearch Tree



In fact, we never even consider these because we stop at first success

Some real CSPs

- Graph coloring is a real, and useful, CSP. Applied to problems with many hundreds of thousands of nodes.
- VLSI or PCB board layout.
- Selecting a move in the game of "minesweeper".

0	0	1			
0	0	1			
0	0	1			
1	1	2			

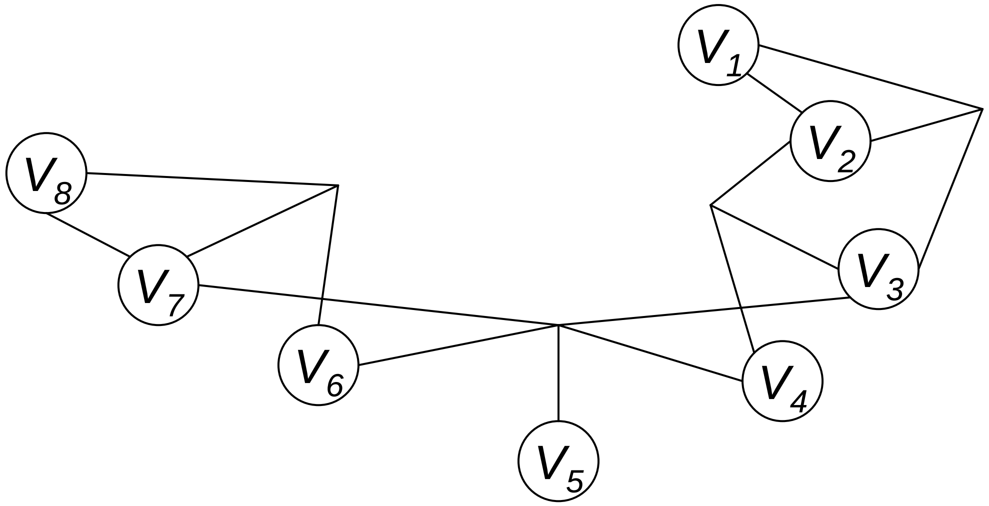
Which squares have a bomb? Squares with numbers don't. Other squares might. Numbers tell how many of the eight adjacent squares have bombs. We want to find out if a given square can possibly have a bomb….

“Minesweeper” CSP

0	0	1	V_1		
0	0	1	V_2		
0	0	1	V_3		
1	1	2	V_4		
V_8	V_7	V_6	V_5		

$V = \{ V_1 , V_2 , V_3 , V_4 , V_5 , V_6 , V_7 , V_8 \}$, $D = \{ B$ (bomb) , S (space) $\}$

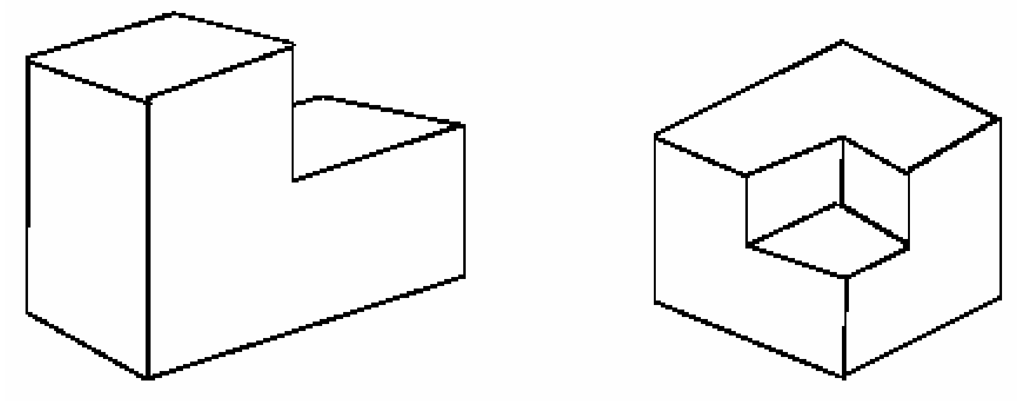
$C = \{ (V_1, V_2) : \{ (B,S) , (S,B) \}, (V_1, V_2, V_3,) : \{ (B,S,S) , (S,B,S) , (S,S,B)\},\ldots\}$



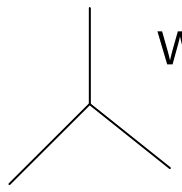
The Waltz algorithm

One of the earliest examples of a computation posed as a CSP.

The Waltz algorithm is for interpreting line drawings of solid polyhedra.



Look at all intersections.



What kind of intersection could this be? A concave intersection of three faces? Or an external convex intersection?

Adjacent intersections impose constraints on each other. Use CSP to find a unique set of labelings. Important step to "understanding" the image.

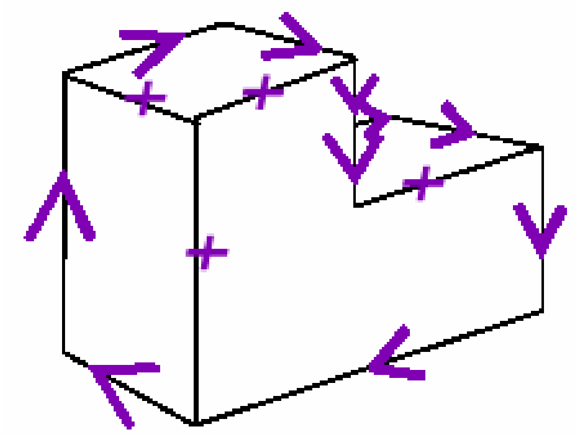
Waltz algorithm on simple scenes

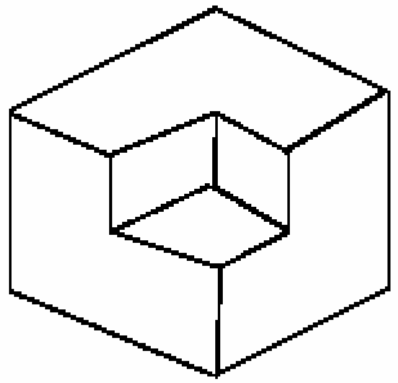
Assume all objects:

- Have no shadows or cracks
- Three-faced vertices
- “General position”: no junctions change with small movements of the eye.

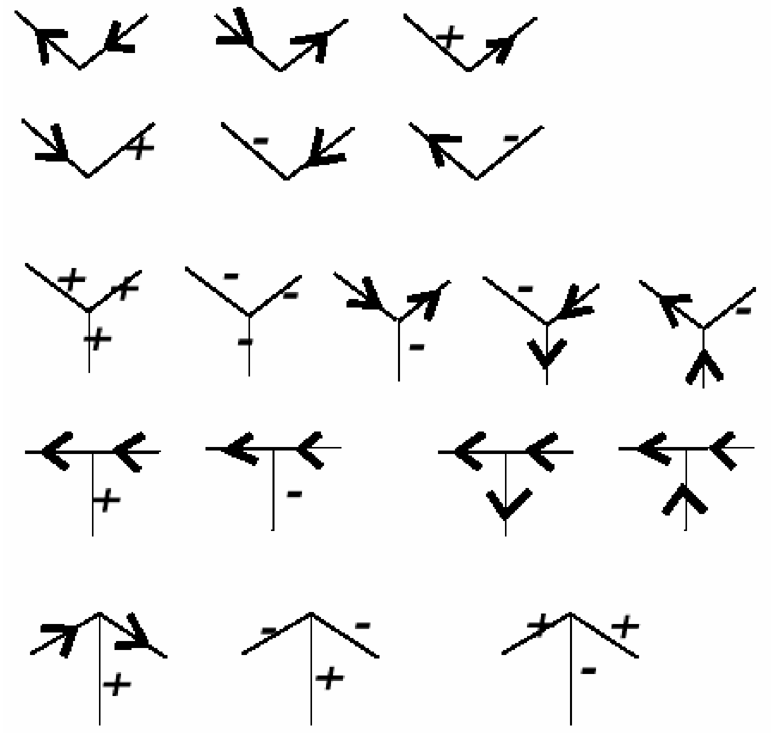
Then each line on image is one of the following:

- Boundary line (edge of an object) (<) with right hand of arrow denoting “solid” and left hand denoting “space”
- Interior convex edge (+)
- Interior concave edge (-)





18 legal kinds of junctions

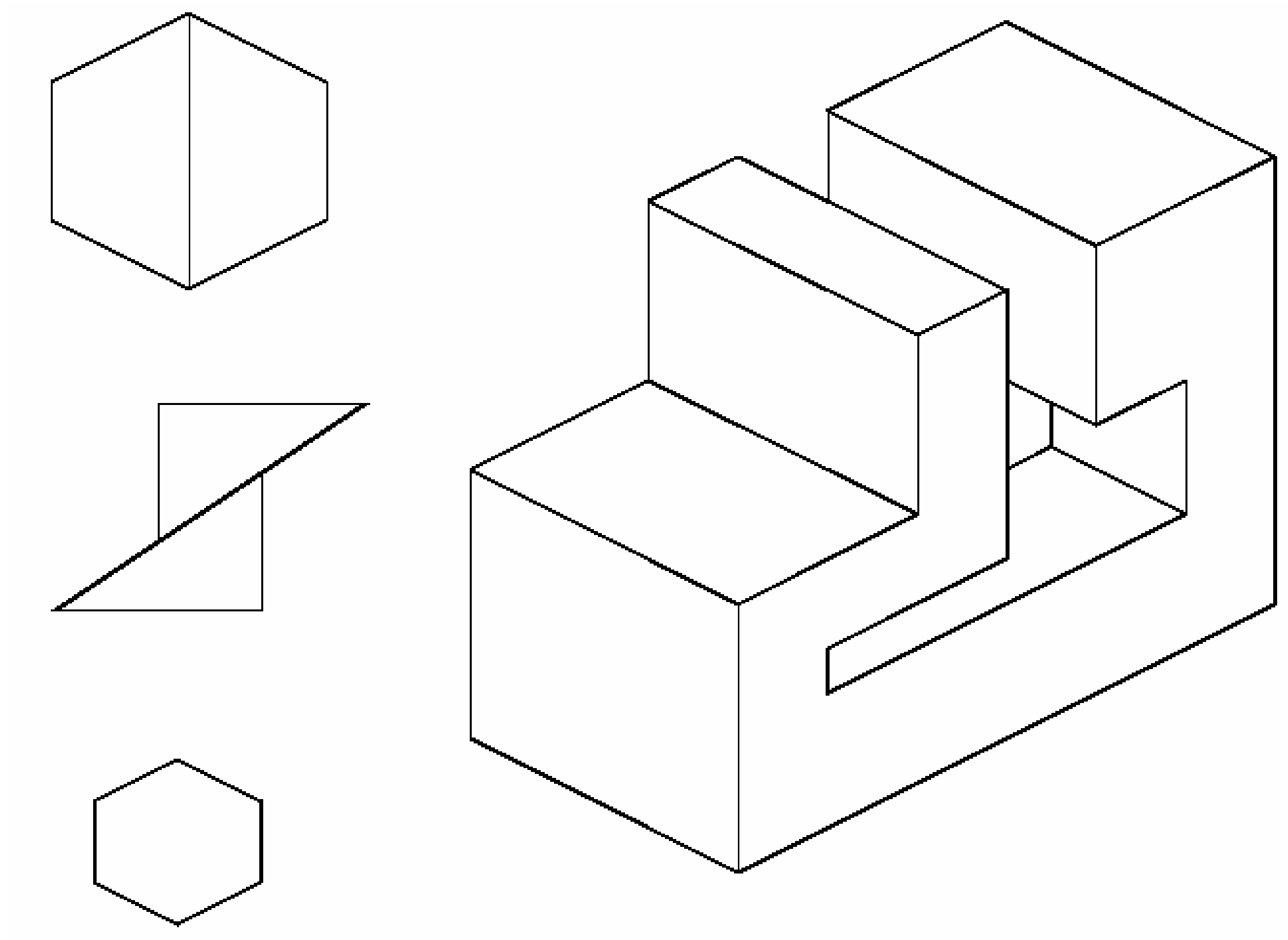


Given a representation of the diagram, label each junction in one of the above manners.

The junctions must be labeled so that lines are labeled consistently at both ends.

Can you formulate that as a CSP? *FACT: Constraint Propagation works perfectly.*

Waltz examples for you



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.

People *really care* about improving scheduling algorithms!

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

At a job-shop you make various products. Each product is a “job” to be done. E.G.

Job_1 = Make a polished-thing-with-a-hole
Job_2 = Paint and drill a hole in a widget

Each job requires several operations. E.G.

Operations for Job_1: Polish, Drill
Operations for Job_2: Paint, Drill

Each operation needs several resources. E.G.

Polishing needs the Polishing machine
Polishing needs Pat (a Polishing expert)
Drilling needs the Drill
Drilling needs Pat (also a Drilling expert)
Or Drilling can be done by Chris

Some operations need to be done in a particular order (e.g. Paint after you’ve Drilled)

Job-Shop: Formal Definition

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: The variables & Constraints

Variables:

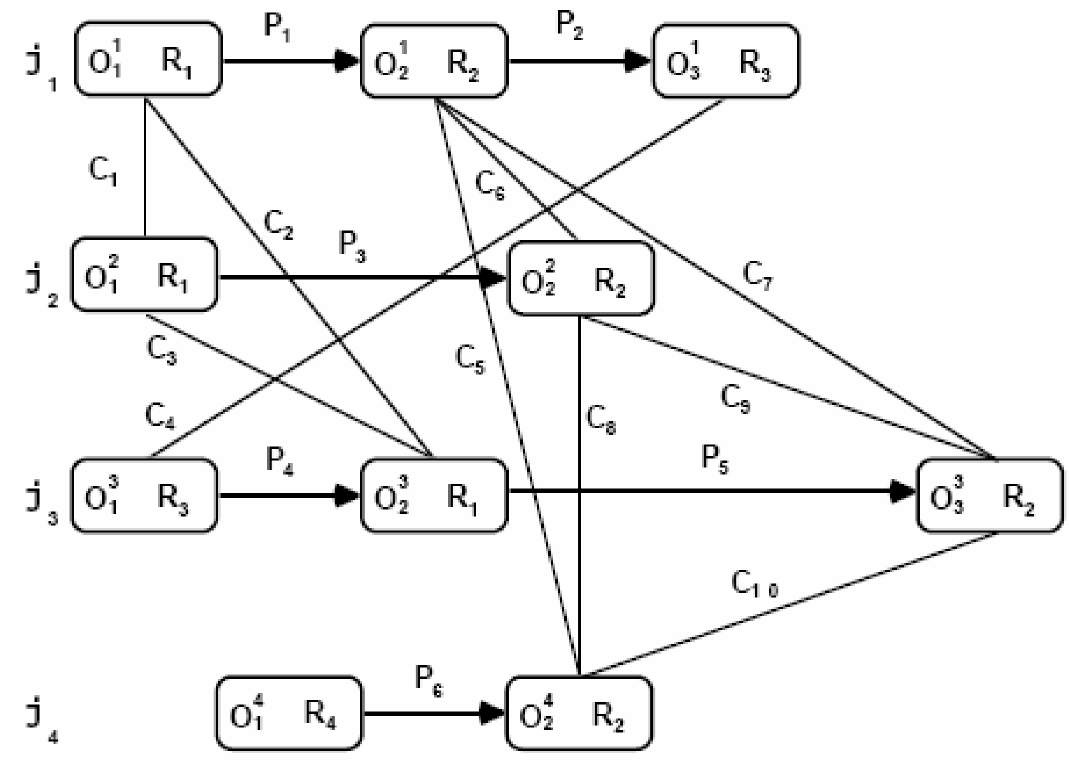
- The operation state times st^l_i
- The resources R^l_{ij} (usually these are obvious from the definition of O^l_i. Only need to be assigned values when there are alternative physical resources available, e.g. *Pat* or *Chris* for operating the *drill*).

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.

Non-challenging question. Can you schedule our Job-shop?

A slightly bigger example



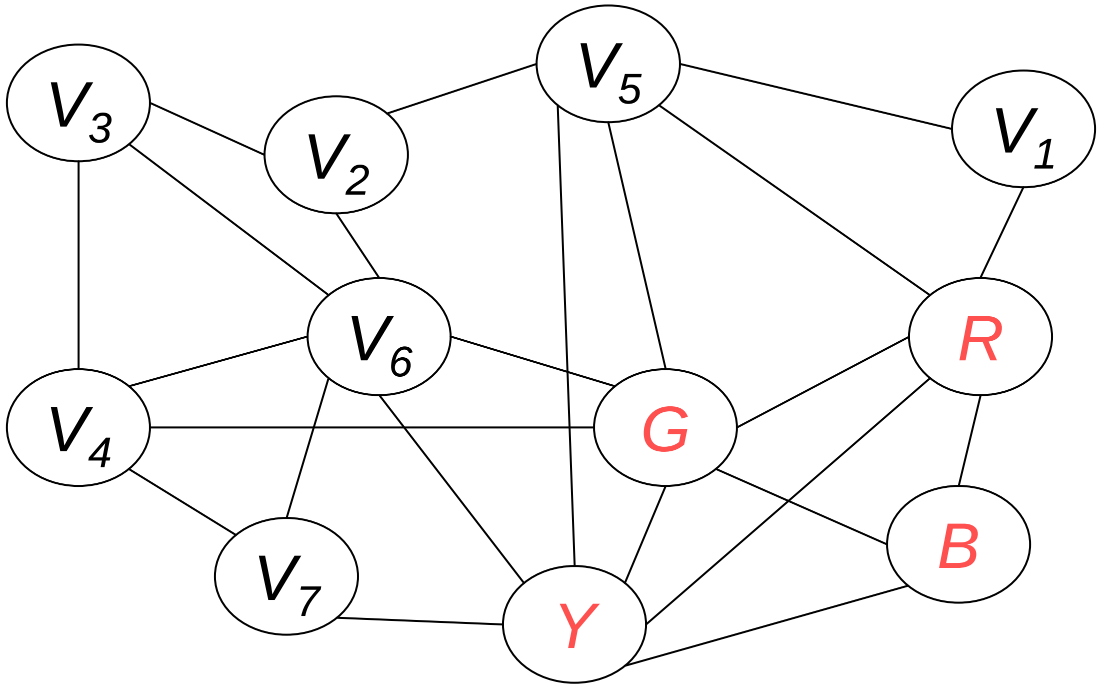
Example from [Sadeh and Fox, 96]: Norman M. Sadeh and Mark S. Fox, Variable and Value Ordering Heuristics for the Job Shop Scheduling Constraint Satisfaction Problem, Artificial Intelligence Journal, Number Vol 86, No1, pages 1-41, 1996.

4 jobs. Each operation 3 units long. All jobs have release date 0 and due date 15. All operations use only one resource each.

Further CSP techniques

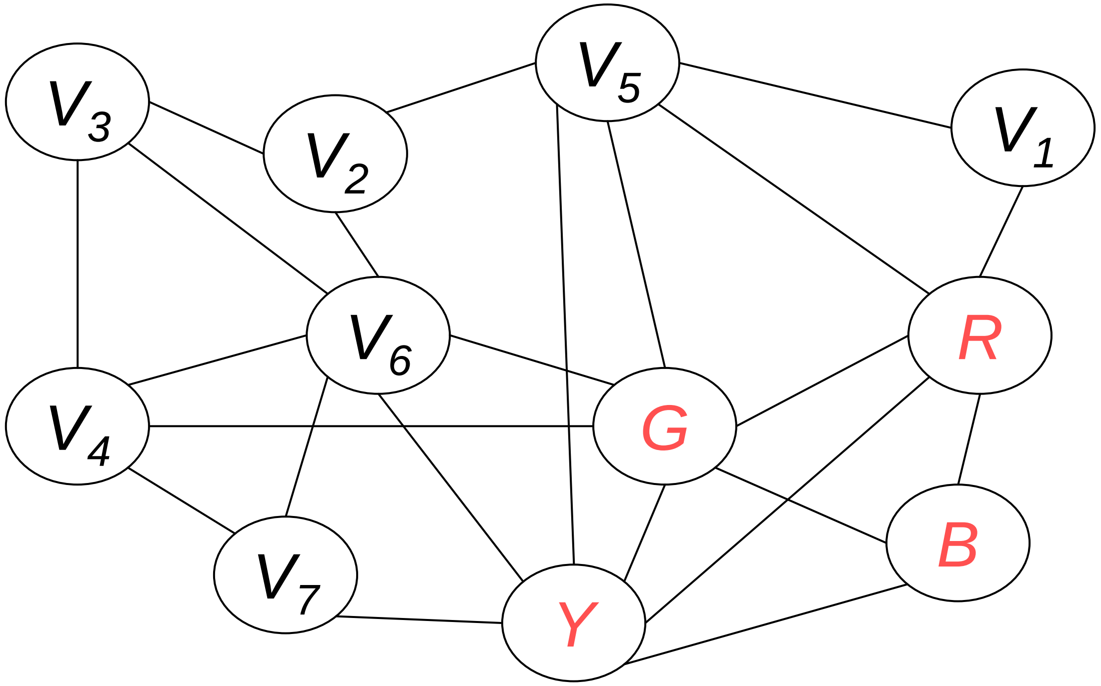
Let’s look at some other important CSP methods. Keep the job-shop example in mind.

Here’s another graph-coloring example (you’re now allowed *R*, *G*, *B* and *Y*)

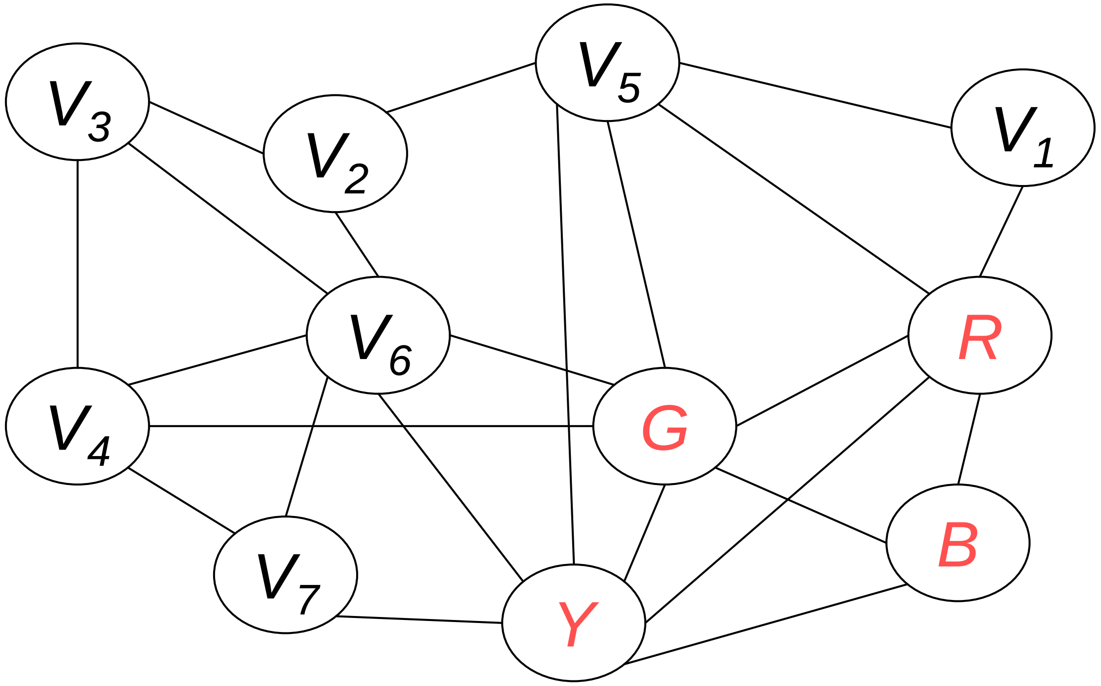


General purpose Variable-Ordering Heuristics

1. **Most constrained variable heuristic.** (or “Minimum remaining values” MRV heuristic)
2. **Most constraining variable heuristic**. (or “Degree heuristic”)



General purpose Value-ordering Heuristics



A good general purpose one is “least-constrained-value”. Choose the value which causes the smallest reduction in number of available values for neighboring variables

General purpose 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.

Reactive CSP solutions

- 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.

Important area of research: efficient schedule repair algorithms.

- Question: If you expect that resources may sometimes break, what could a scheduling program do to take that into account?
- Unrelated Question: Why has none of this lecture used A* Search?

Local Search applied to CSP

- Usually turns-out to be very effective for CSP problems,
- Uses a complete state formulation (i.e., every variable is assigned all the time)
- Usually change only a few (e.g. one) variable at every iteration,
- When choosing a new value for a variable:

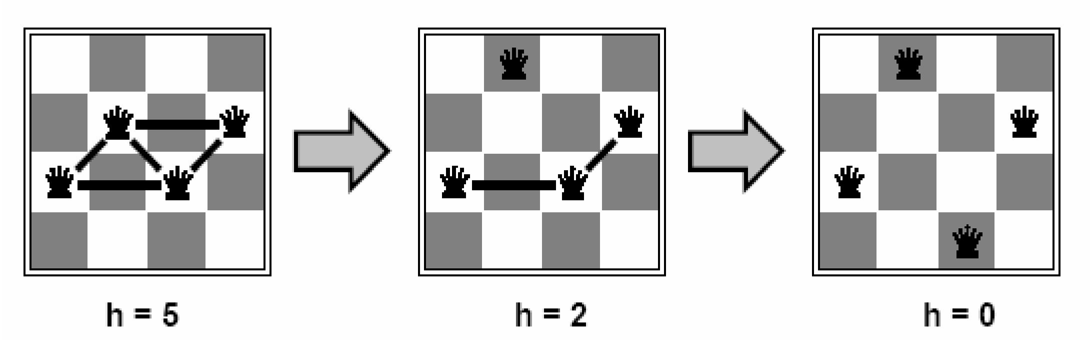
  Use: ***Minimum-conflicts heuristic***
- I.e., choose a value which causes minimum number of conflicts with other variables

Local Search applied to CSP

- Successfully used in large CSPs, including Scheduling problems,
- Used in scheduling the Hubble Space Telescope (scheduling a week of observations was reduced: three weeks!! -> 10 minutes)
- Can be used in on-line scheduling (i.e., schedule repair or re-scheduling under problem change)
- Methods include: Hill Climbing, Tabu search, simulated annealing, genetic algorithms…

Local Search example: 4-queens

- Put 4 queens on the chess board of given size such that:
- Goal test: *No queen attacking each other,*
- 4 queens in 4 columns ($4^4 = 256$ states),
- Evaluation *h*(*n*) = Num of attacks



Let us check again: http://www.pvv.org/~hgs/java/queens/simple.html

Conclusions

- CSPs are a special kind of problem:
  - states defined by values of a xed set of variables
  - goal test defined by constraints on variable values
- Backtracking = depth-first search with one variable assigned per node
- *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 is very useful in analyzing problem structure
- Iterative min-conflicts is usually very effective in practice.