



Search

AC algorithms on bounded variables

- AC3 and AC2001 don't make any use of the meaning of the constraints or of any pattern in the allowed pairs
- For arithmetic constraints, e.g. $X < Y$, this may involve repeated wasted computation
 - e.g. $X < Y$, $D_X = \{1, 2, 3, \dots, 99, 100\}$, $D_Y = \{1, 2, \dots, 10\}$
- In most solvers, we can represent integer variables by their upper and lower bounds
- Can we do something better than AC3 for these types of variables and constraints?

AC5 (and related algorithms)

- each variable and constraint is associated with a class in OOP style
- the constraint classes have methods to revise their variables' domains
 - revise as in AC3
 - revise only when an upper bound is reduced
 - revise only when a lower bound is increased
 - revise when a variable is assigned a value
 - ...
- the AC algorithm maintains a queue of triples: the constraint to be revised, and the variable and value deletion that triggered it
- much more efficient in practice.

Other forms of consistency

- *Node* consistency
 - if there are unary constraints, ensure every value in the domain satisfies the constraint
- *Bounds* consistency
 - when representing integer variable domains by their lower and upper bounds, ensure that those bounds have supports in the constraint
- *Path* consistency
 - if we have three constraints C_{XY}, C_{XZ}, C_{YZ} , for every pair $(v, w) \in D_X \times D_Y$, ensure there is a value $u \in D_Z$ s.t. $(v, u) \in C_{XZ}$ and $(w, u) \in C_{YZ}$
- Consistency over non-binary and global constraints
 - we will see later ...

Exercise

- Model the following problem as a CSP, and then apply AC3. Count how many times you checked a pair of values.

A robot delivery drone must plan a schedule for visiting 5 customers: {A,B,C,D,E}. The only available delivery slots are at hours {1,2,3,4} in the afternoon. For various reasons, the following constraints apply:

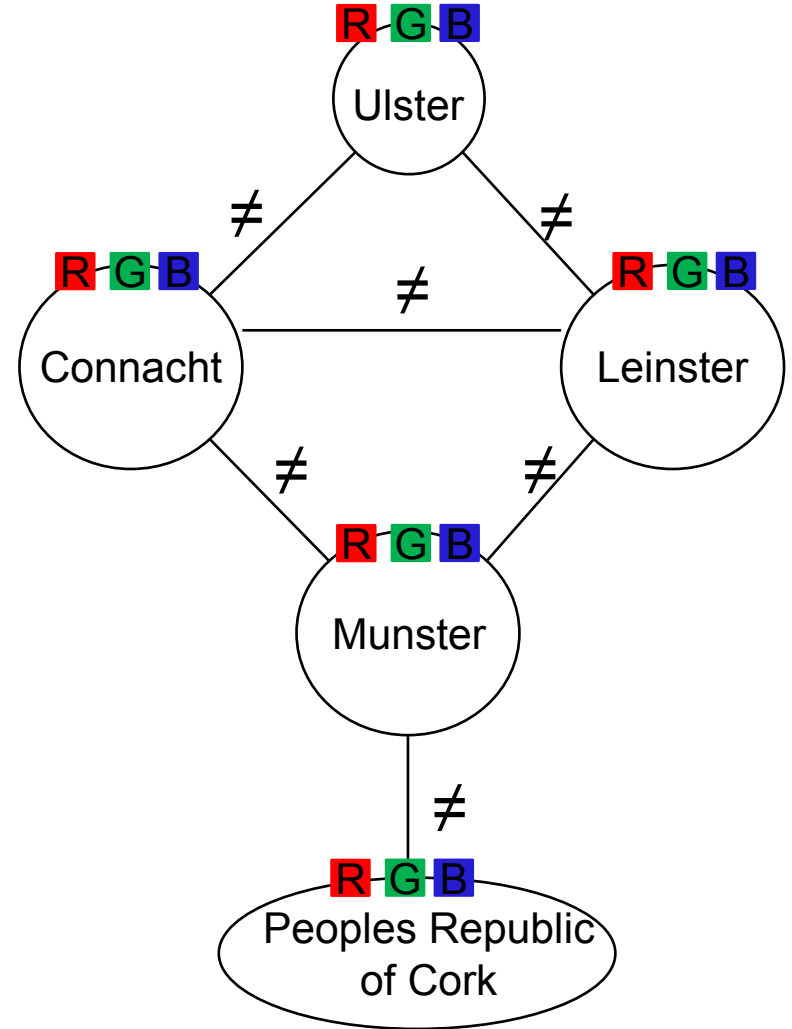
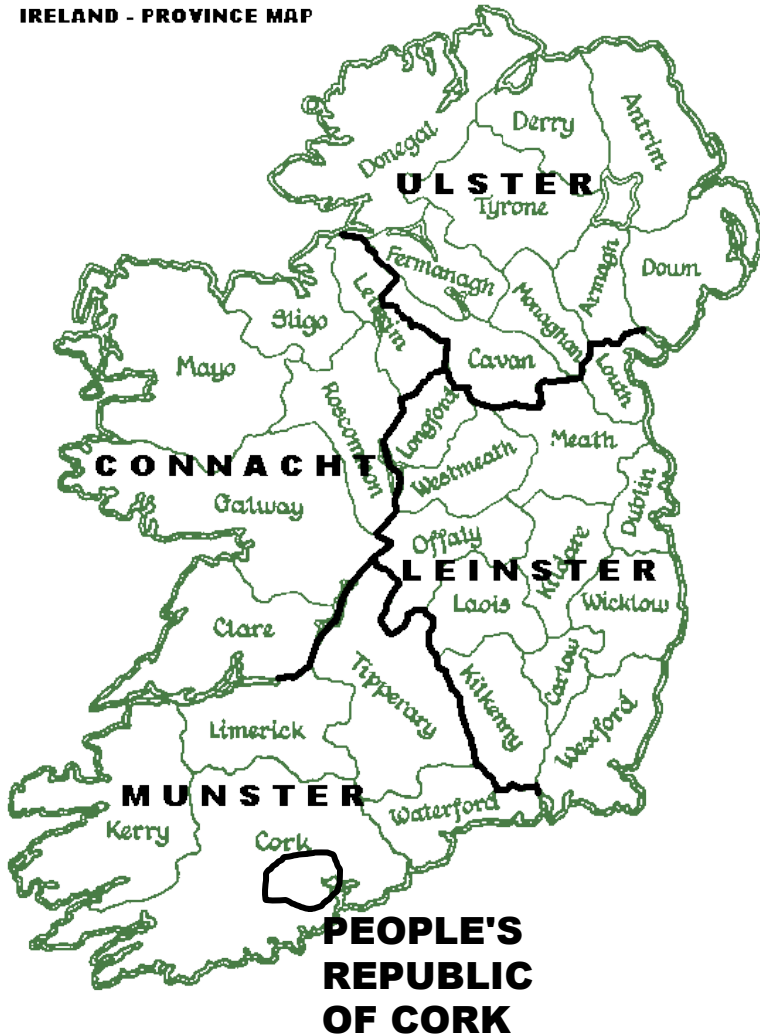
- E must be before all other deliveries
- C must receive its delivery before D
- A and D must get their delivery in the same time slot
- B cannot receive a delivery in time slot 3
- C cannot receive a delivery in time slot 2
- A and B cannot get deliveries in the same slot
- B and C cannot get deliveries in the same slot
- B and D cannot get deliveries in the same slot

You are not being asked to model this in Choco

Exercise adapted from D. Poole and A. Mackworth, *Artificial Intelligence: Foundations of Computational Agents*, Cambridge University Press, 2010.

A colouring problem

IRELAND - PROVINCE MAP



Ireland graph colouring as a CSP

5 variables, 3 values each: 3^5 possible colourings

Pre-processing for AC does nothing for us

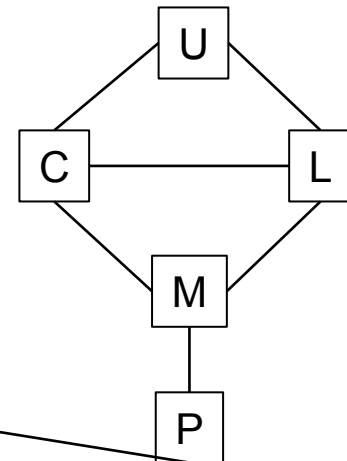
The aim is to find a systematic procedure which will

- find a solution if there is one
- find ***all*** solutions if needed
- never return a colouring that violates a constraint

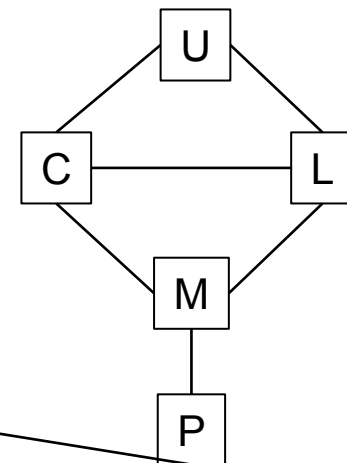
Simplest approach: "generate and test"

```
for each value x for Ulster
  for each value y for Connacht
    for each value z for Leinster
      for each value w for People's Republic of Cork
        for each value t for Munster
          check to see if (x,y,z,w,t) is a solution
```

Fixed value ordering: R, G, B

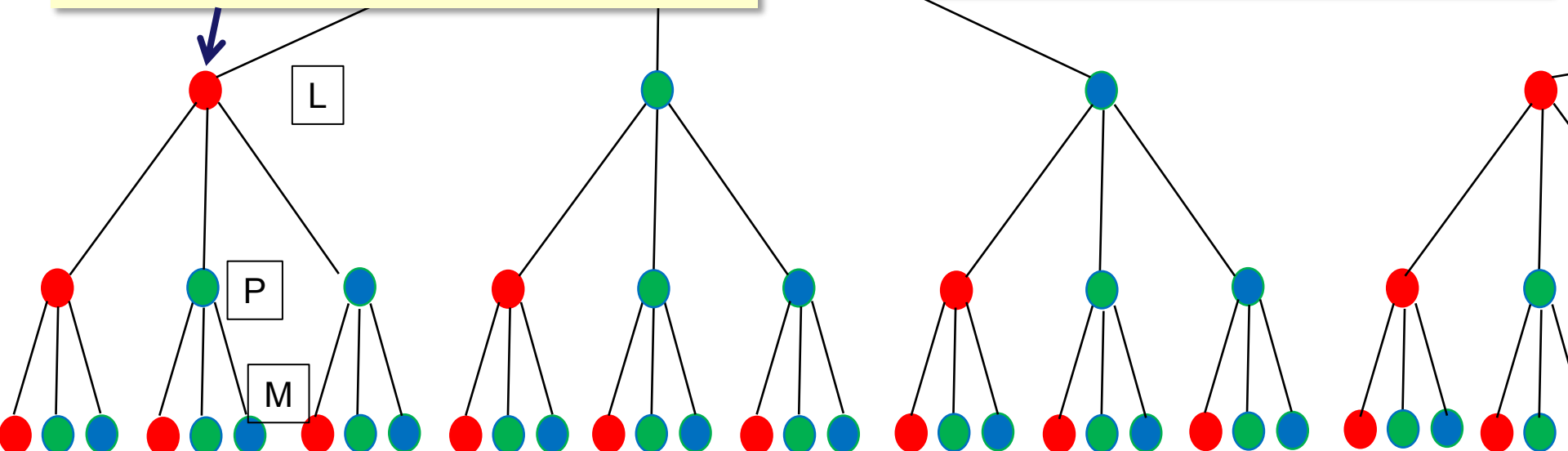


Fixed value ordering: R, G, B



But we know no assignment below here cannot work

So why bother continuing to the leaf nodes?

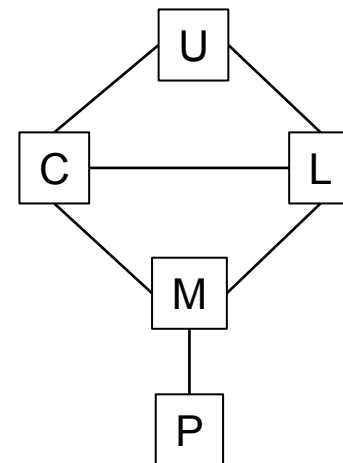


Backtracking

- as we are iterating through the variables and values, each time we try a value for a variable, check back up the tree to see if it is consistent with previous choices
- if not, try a different value
- if no values left, *backtrack* to the previous variable, and try the next value there ...
- In the worst case, we still might have to generate every possible assignment, and so still $O(d^n)$.
- same algorithm as backtracking in AI 1, but:
 - we know exactly how many decisions must be made
 - no risk of looping
 - we know the maximum number of choices per node
- Can we take advantage in our implementation?

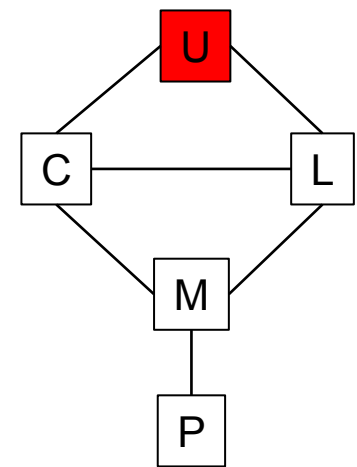
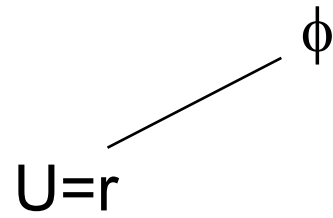
variable order: U-C-L-P-M; value order: r-g-b

ϕ



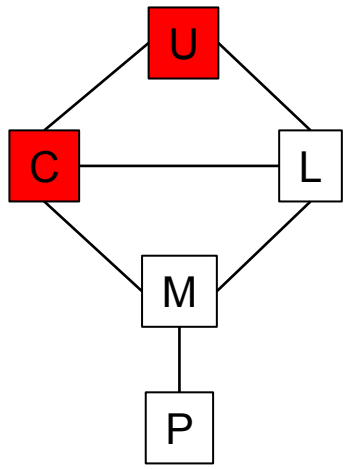
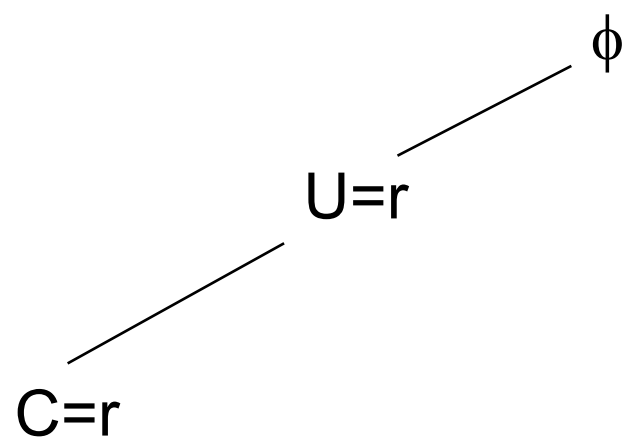
U C L P M

variable order: U-C-L-P-M; value order: r-g-b



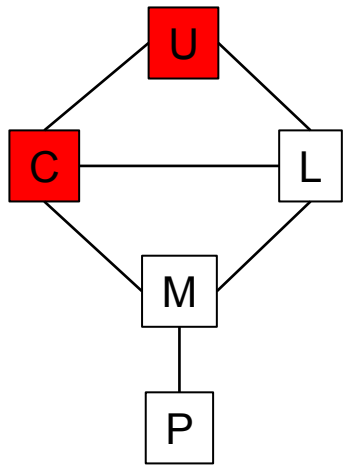
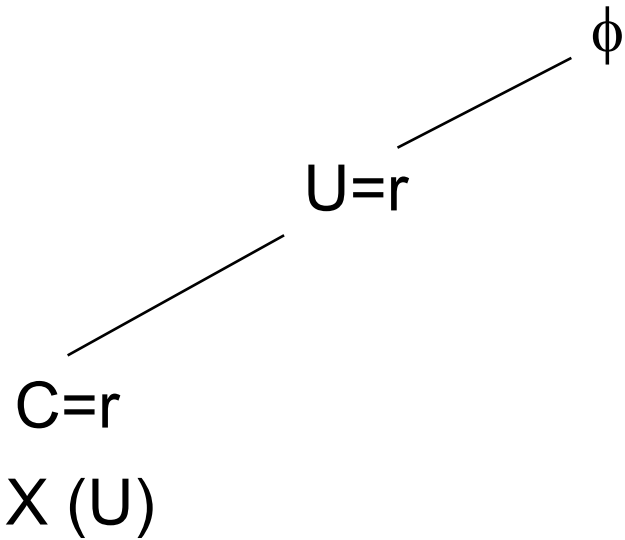
U C L P M
r

variable order: U-C-L-P-M; value order: r-g-b



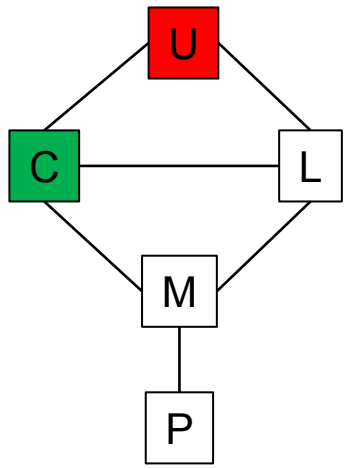
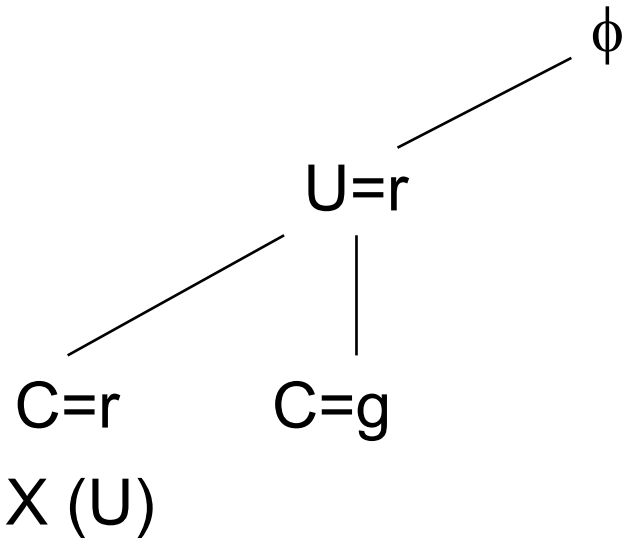
U	C	L	P	M
r	r			

variable order: U-C-L-P-M; value order: r-g-b



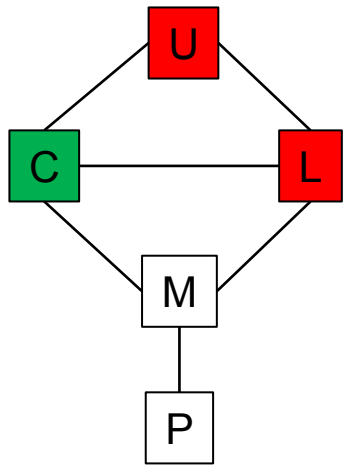
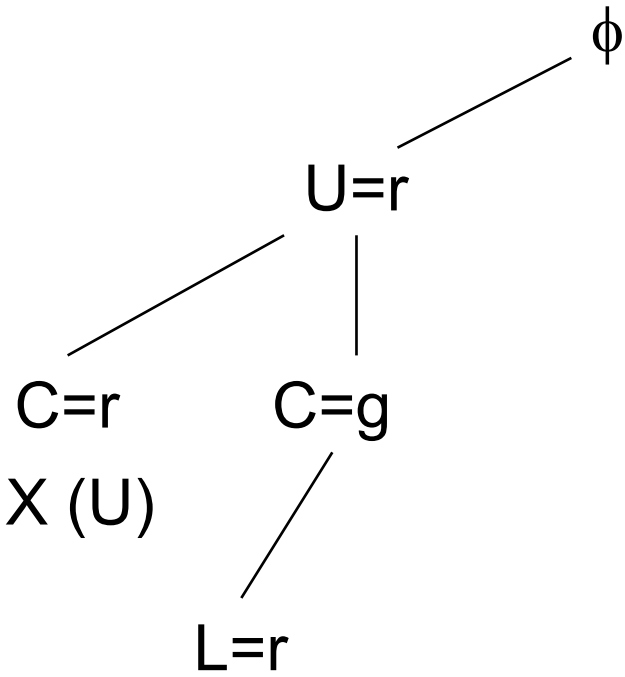
U	C	L	P	M
r	r			

variable order: U-C-L-P-M; value order: r-g-b



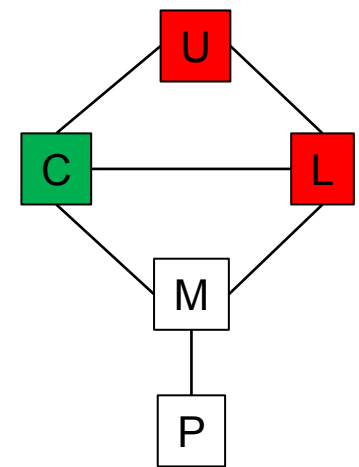
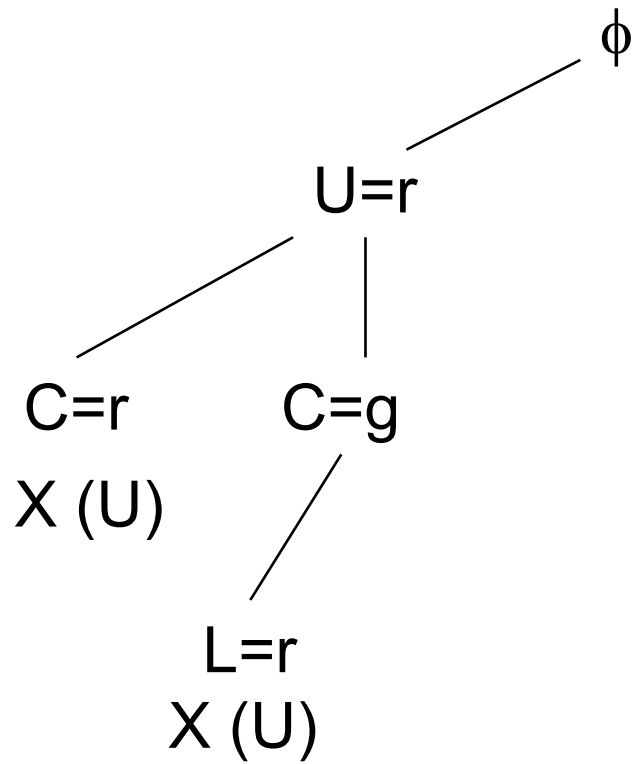
U C L P M
r g

variable order: U-C-L-P-M; value order: r-g-b



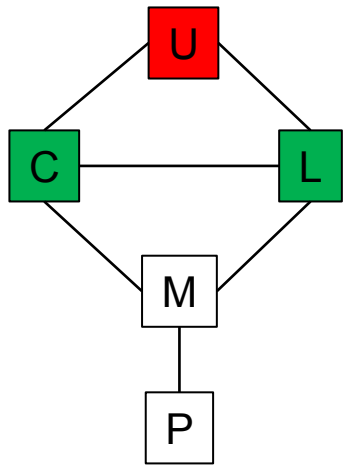
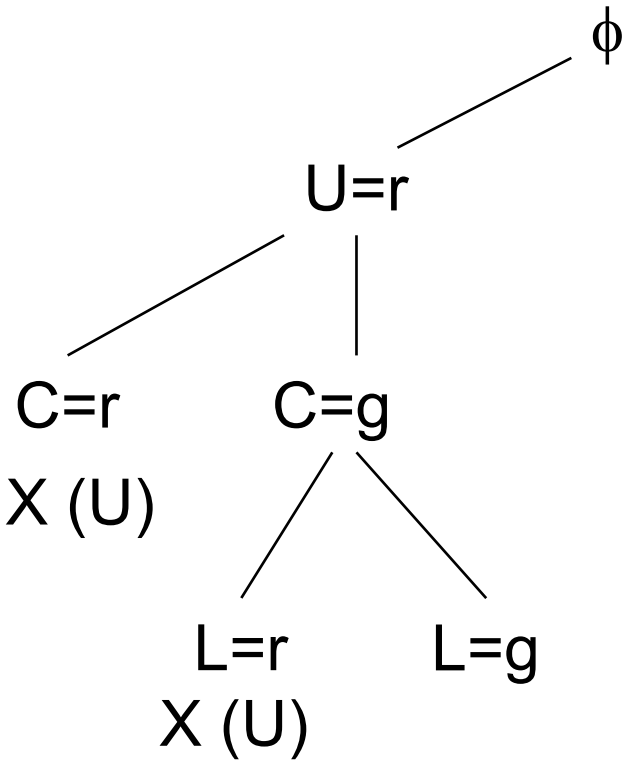
U	C	L	P	M
r	g	r		

variable order: U-C-L-P-M; value order: r-g-b



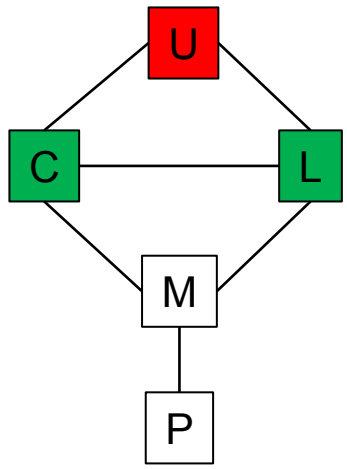
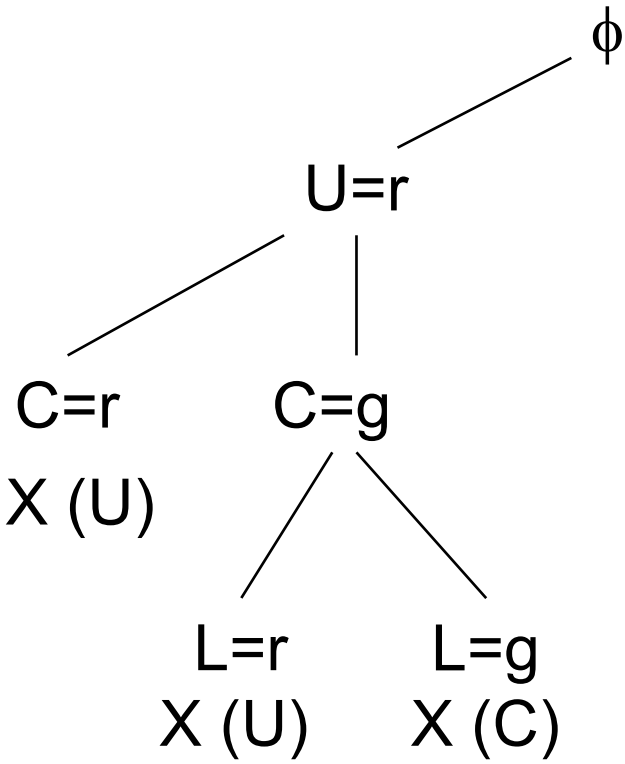
U	C	L	P	M
r	g	r		

variable order: U-C-L-P-M; value order: r-g-b



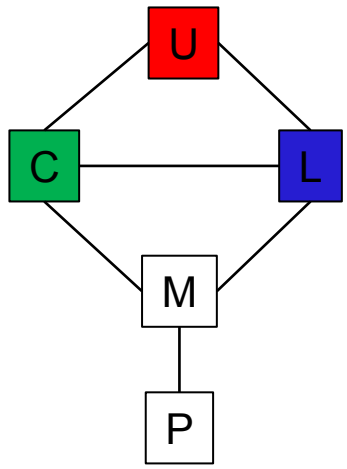
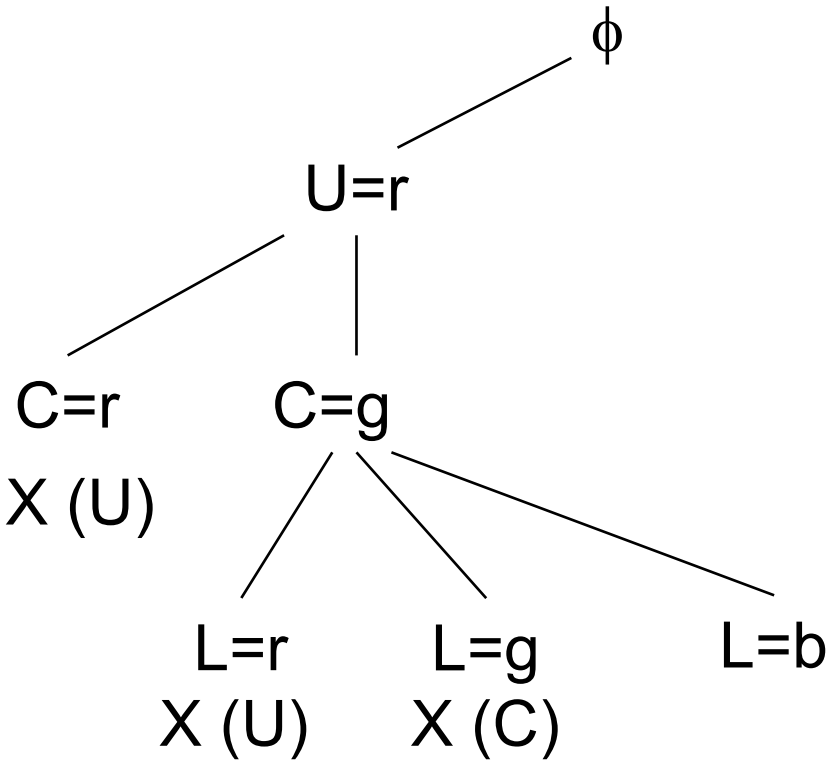
U	C	L	P	M
r	g	g		

variable order: U-C-L-P-M; value order: r-g-b



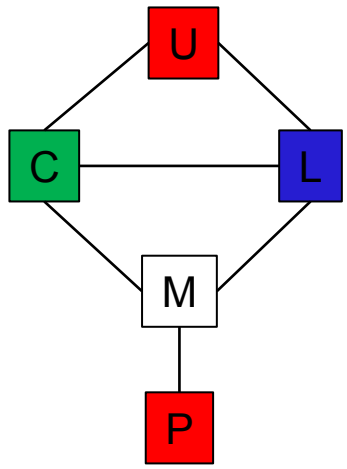
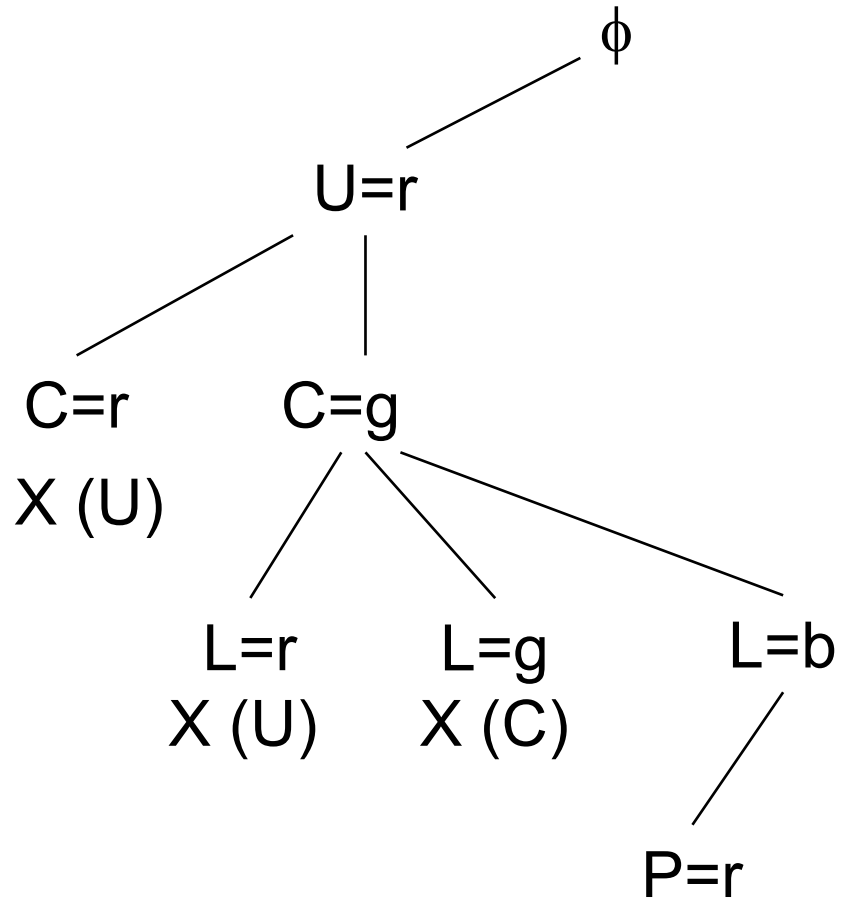
U	C	L	P	M
r	g	g		

variable order: U-C-L-P-M; value order: r-g-b



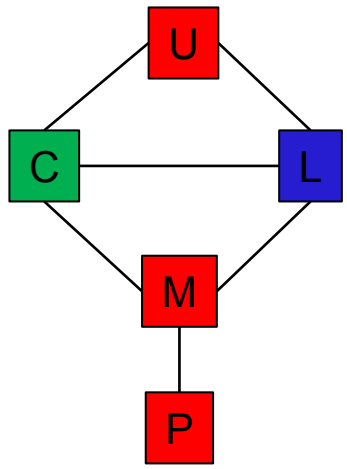
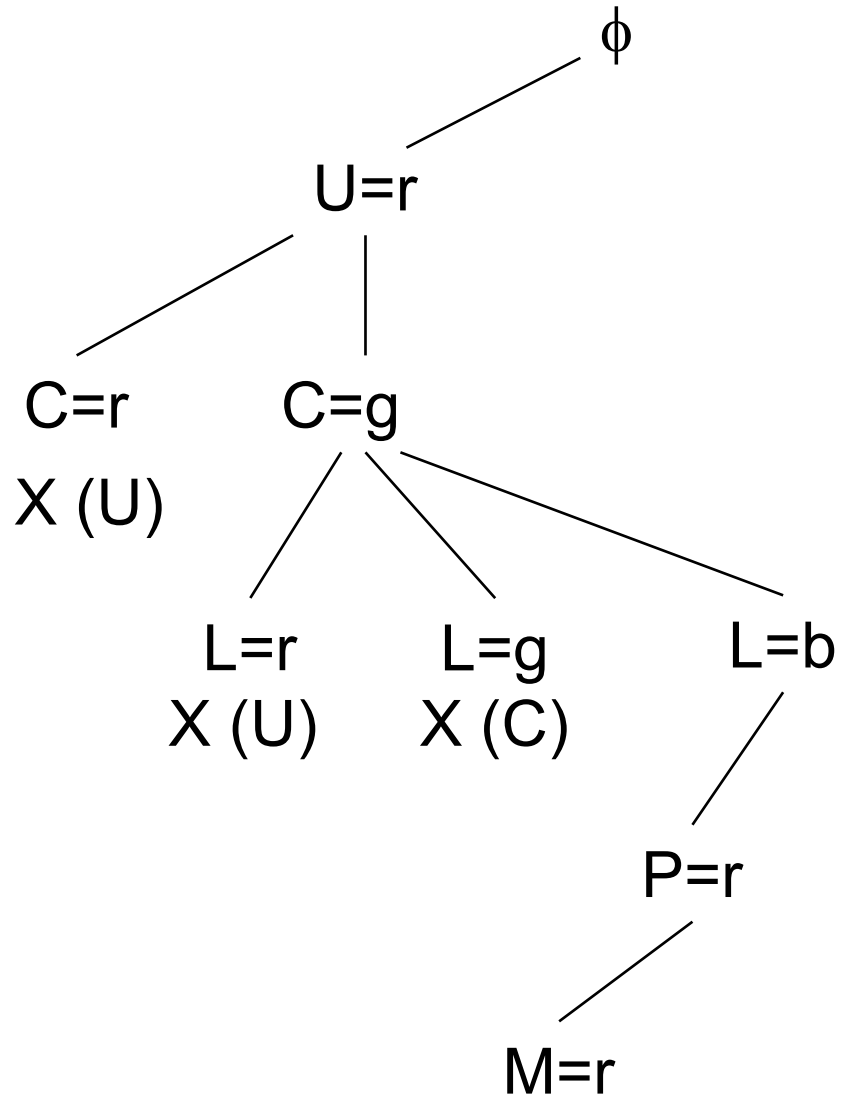
U	C	L	P	M
r	g	b		

variable order: U-C-L-P-M; value order: r-g-b



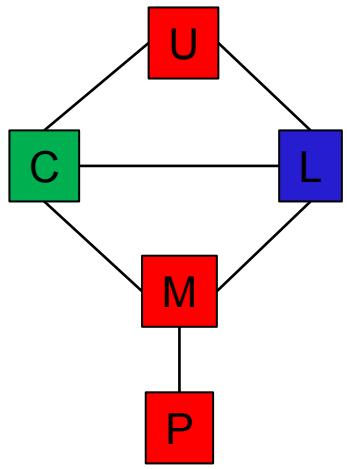
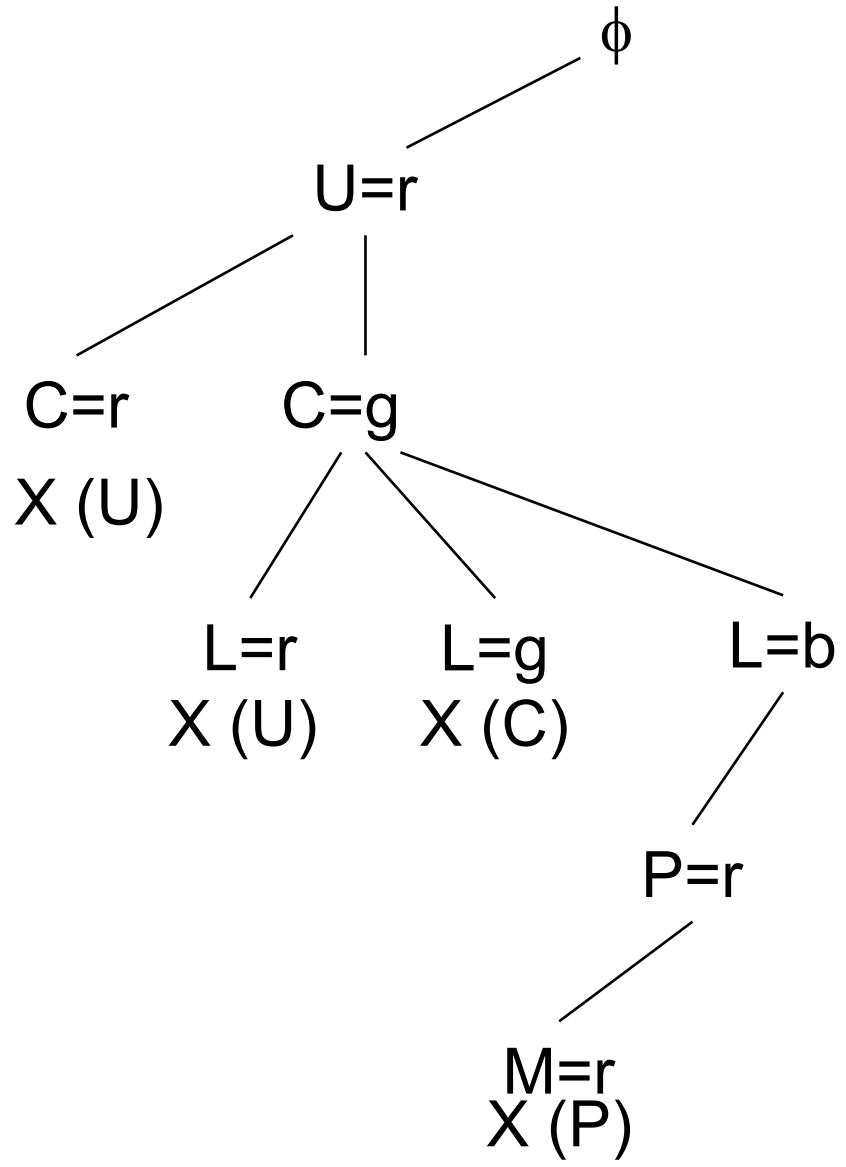
U	C	L	P	M
r	g	b	r	

variable order: U-C-L-P-M; value order: r-g-b



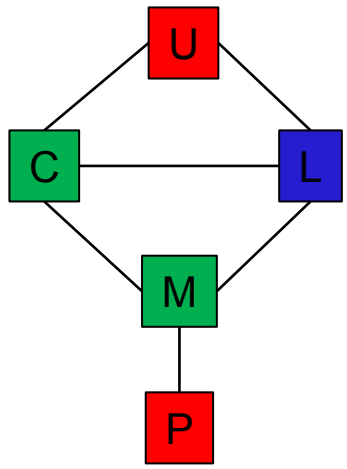
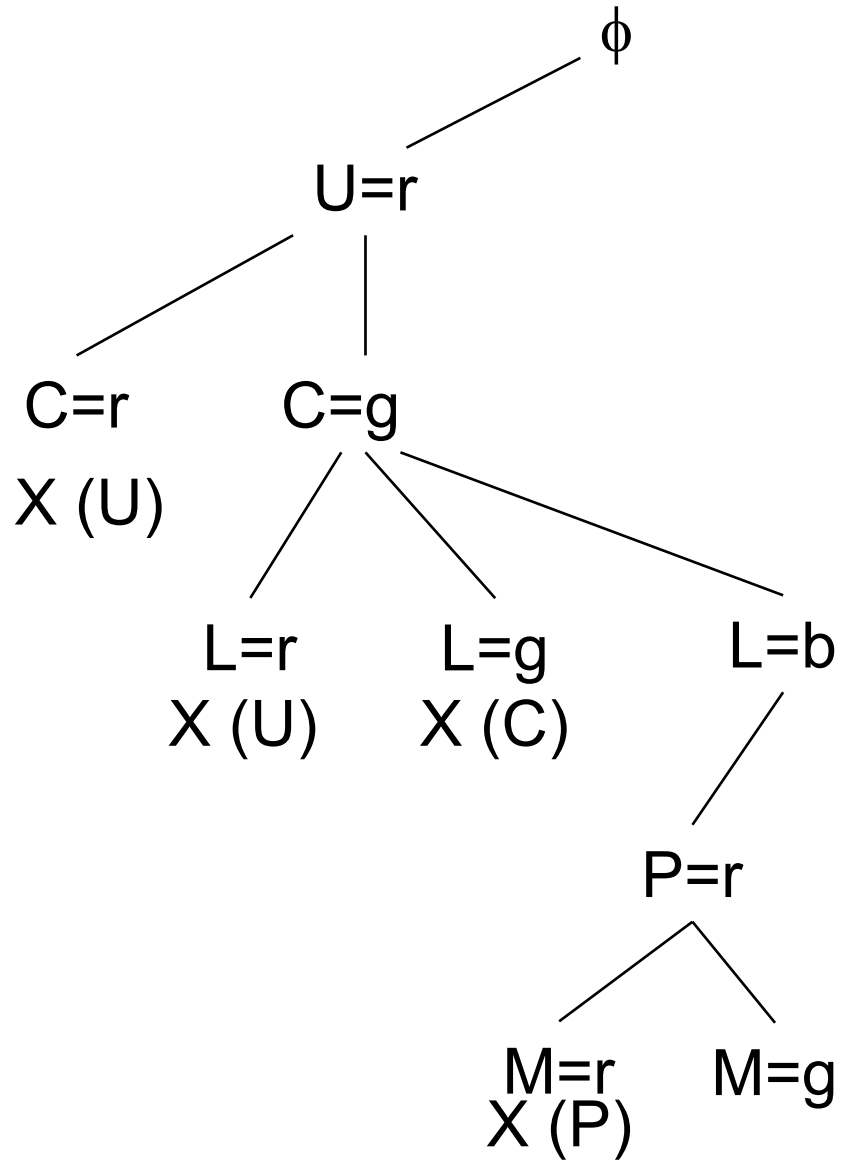
U	C	L	P	M
r	g	b	r	r

variable order: U-C-L-P-M; value order: r-g-b



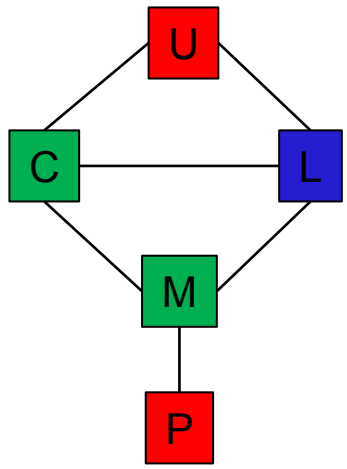
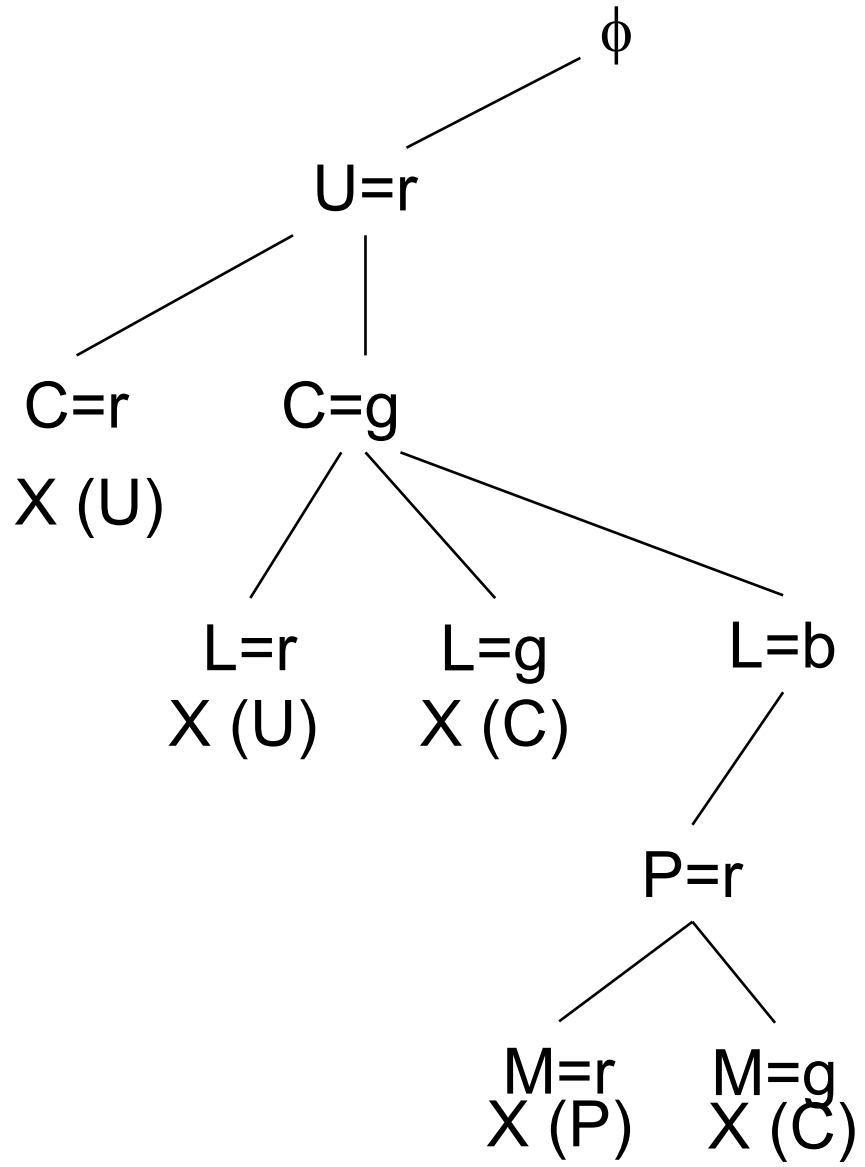
U	C	L	P	M
r	g	b	r	r

variable order: U-C-L-P-M; value order: r-g-b



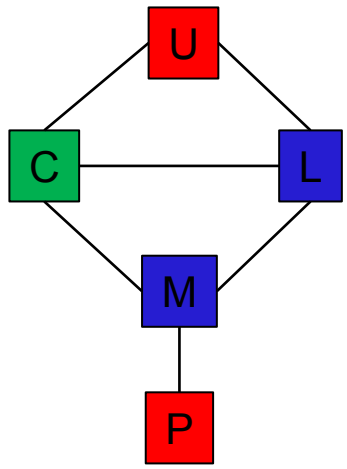
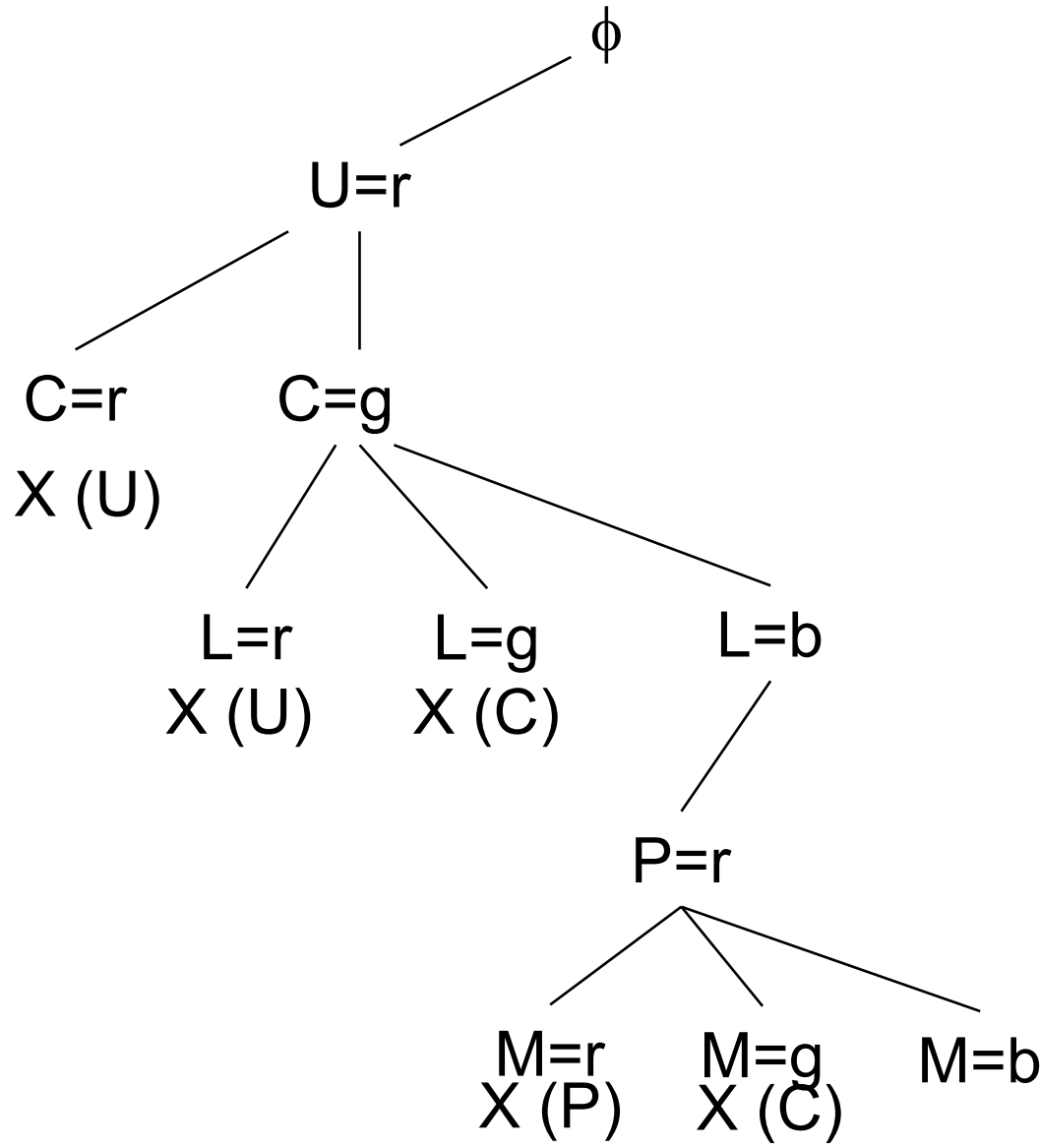
U	C	L	P	M
r	g	b	r	g

variable order: U-C-L-P-M; value order: r-g-b



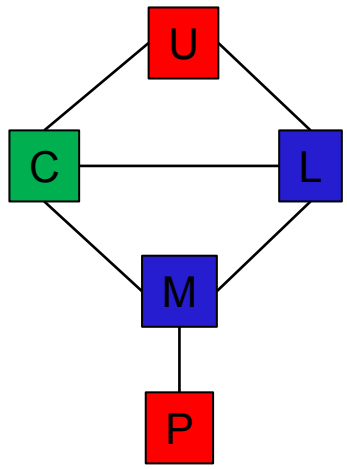
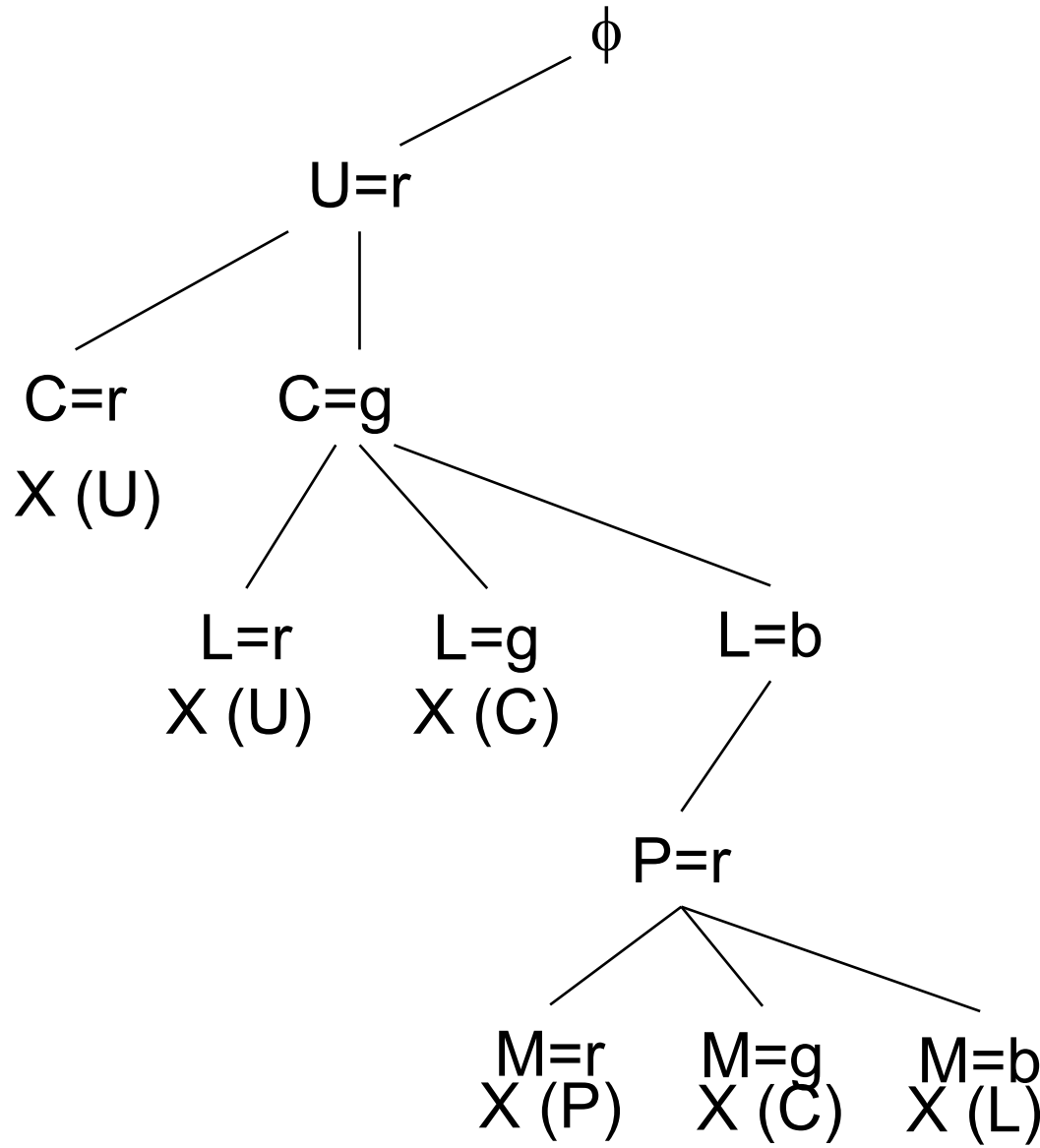
U	C	L	P	M
r	g	b	r	g

variable order: U-C-L-P-M; value order: r-g-b



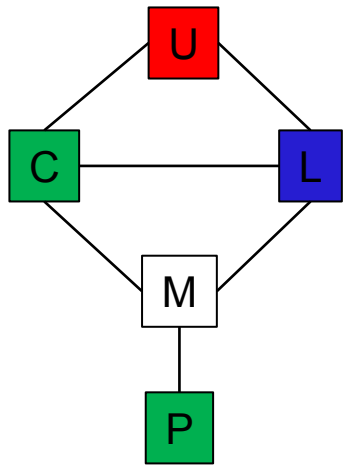
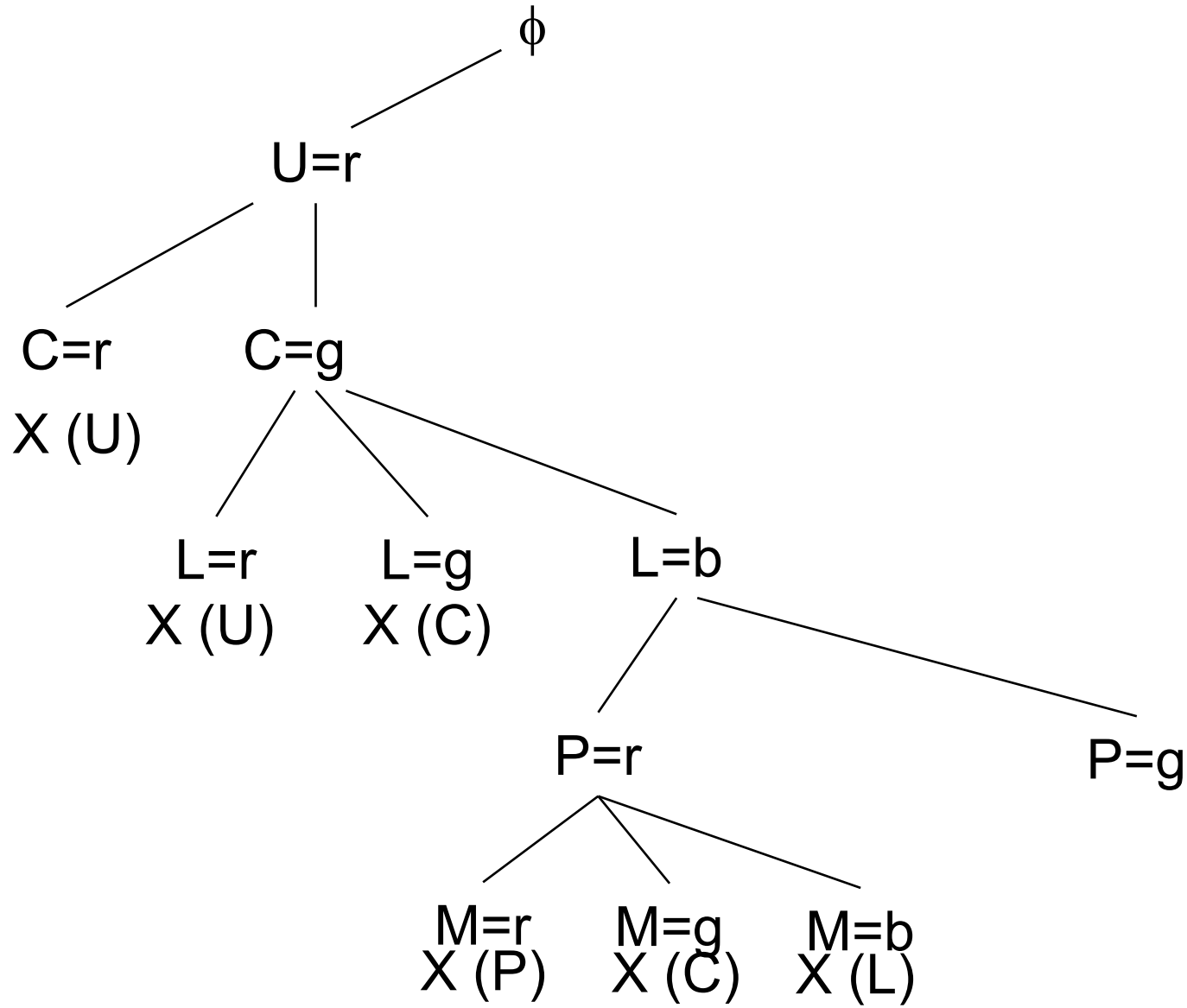
U	C	L	P	M
r	g	b	r	b

variable order: U-C-L-P-M; value order: r-g-b



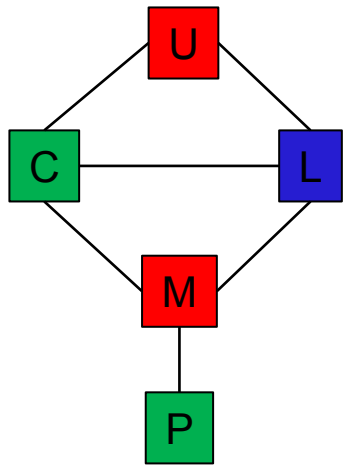
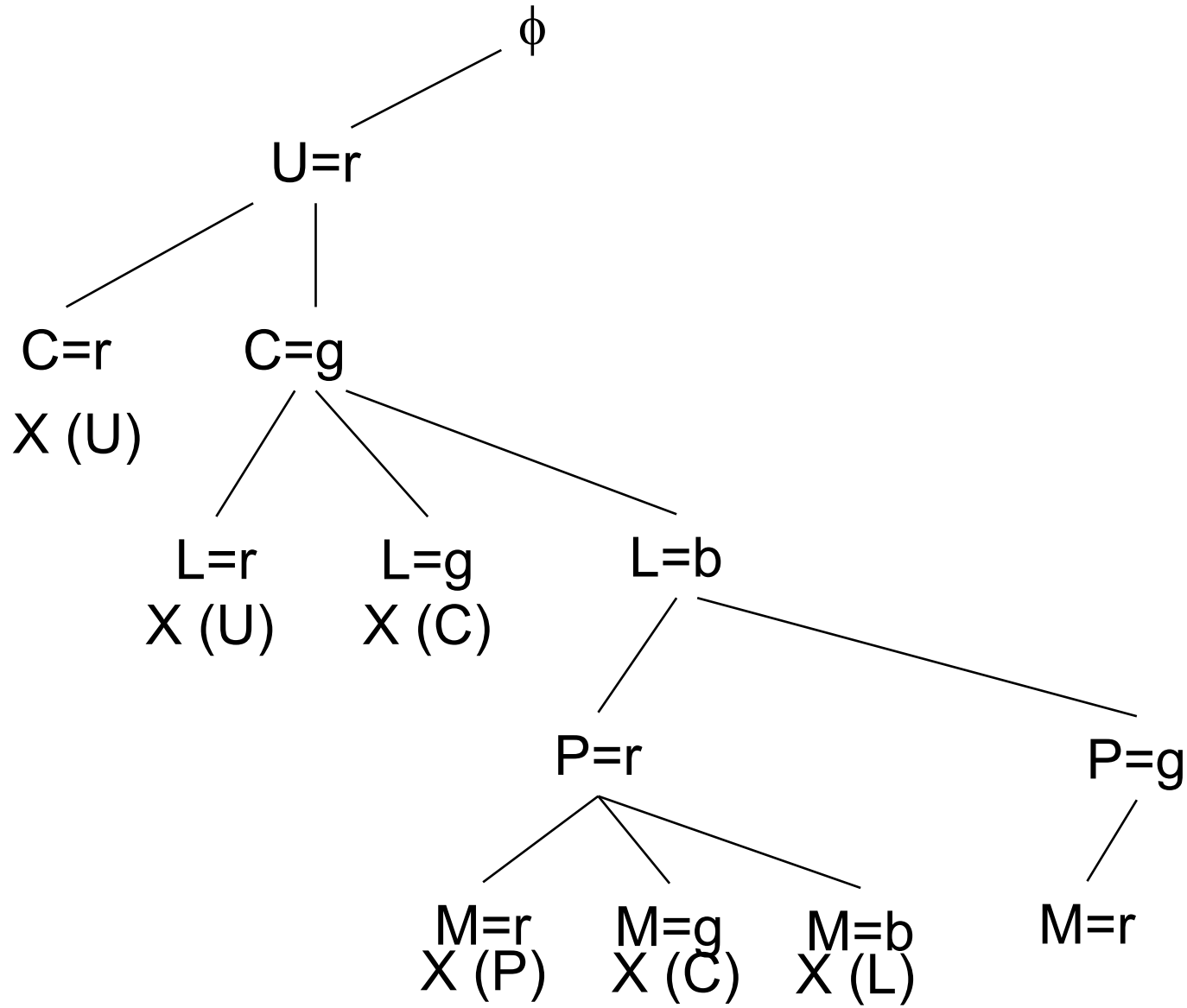
U	C	L	P	M
r	g	b	r	b

variable order: U-C-L-P-M; value order: r-g-b



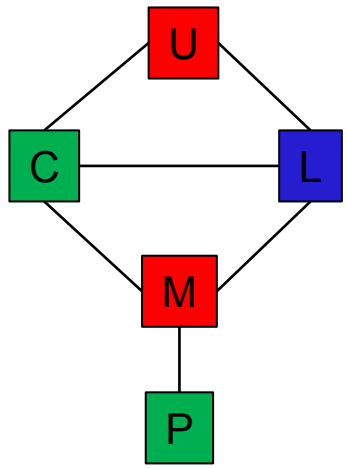
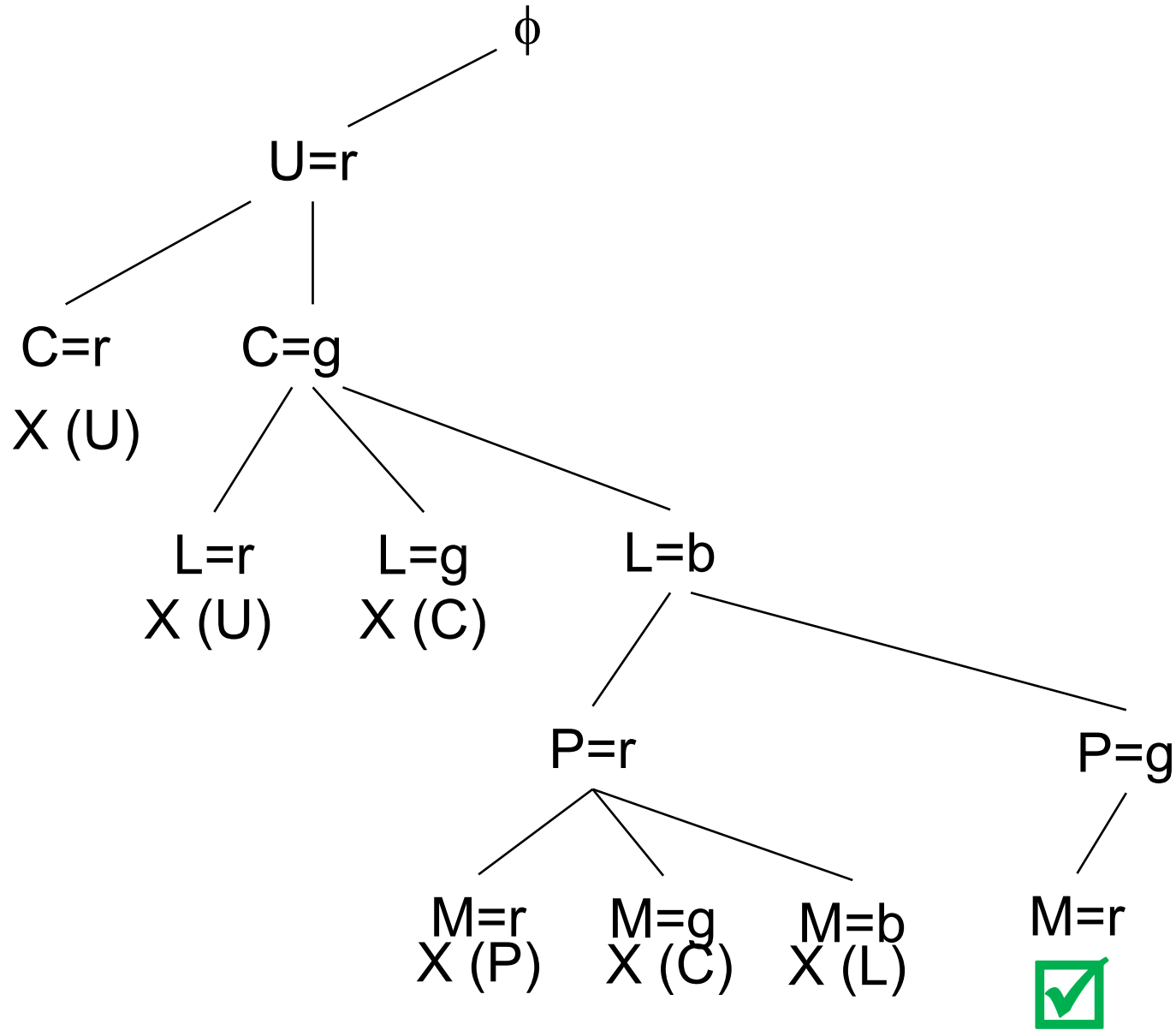
U	C	L	P	M
r	g	b	b	

variable order: U-C-L-P-M; value order: r-g-b



U	C	L	P	M
r	g	b	b	r

variable order: U-C-L-P-M; value order: r-g-b



U	C	L	P	M
r	g	b	b	r

backtrack(V, D, C)

input: V (array of n int) D (2d array of n*d int) C (constraints)

output: (bool, V), where bool = true if a solution is found

```
1. var := 0 //start with the first variable
2. val := 0 //start with the first value
3. done := false //not finished yet
4. while not done
5.     if var == n return (true, values) //if reached end, stop
6.     else if var == -1 return (false, null)//if back to top, stop
7.     else if val == -1 //if no values left
8.         var := var-1 //step back to previous variable
9.         val := nextval(var,values[var]) //continue to next value
10.    else if check(var,val,values,C) == false
11.        val := nextval(var,val) //get next value, or -1 if none
12.    else
13.        values[var] := val //assign current value
14.        var := var + 1 //move to next var
15.        val := 0 //start with first value
```

check(*var*, *val*, *values*, *C*)

```
check(var, val, values, C)
```

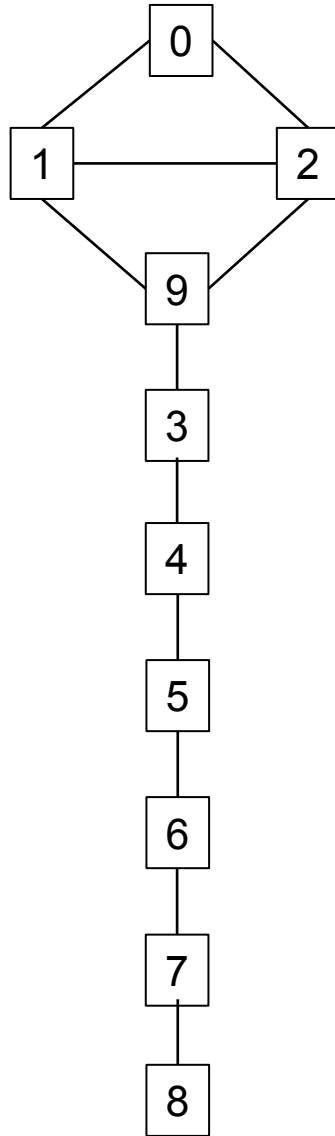
```
output: true if no conflict for var<-val when checking back
```

```
// For each constraint in C whose scope is {var, X}, where X is  
// before var, check var<-val against X's assignment in values
```

```
//i.e. while searching the tree, each time we try a value for  
//a variable, check back against previous assignments to ensure  
//no constraint is violated
```

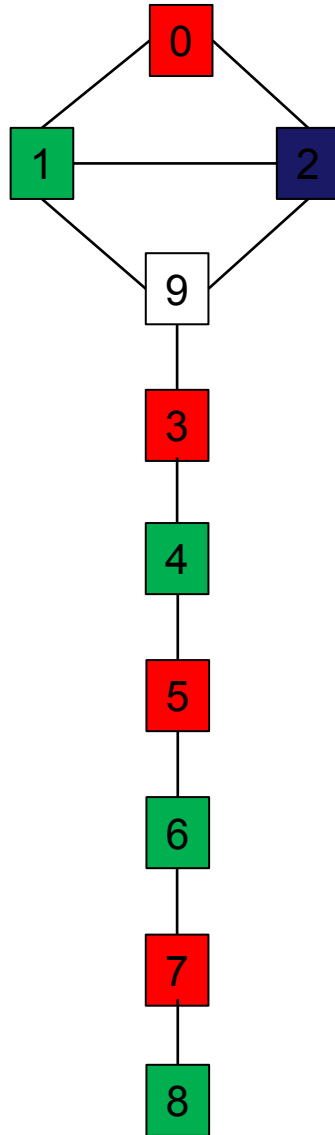
Backtracking is correct

- Backtracking will never output a bad solution.
- If there is a solution, backtracking will return it.
- Backtracking can be extended to return all solutions.



Graph colouring, all domains $\{R,G,B\}$
Assume the numbers give the variable
order.

What happens during backtracking?



If the order of the variables is bad, then backtracking can waste a lot of time ***thrashing***.

0	1	2	3	4	5	6	7	8	9
r	g	b	r	g	r	g	r	g	

must backtrack, trying all combinations for these variables

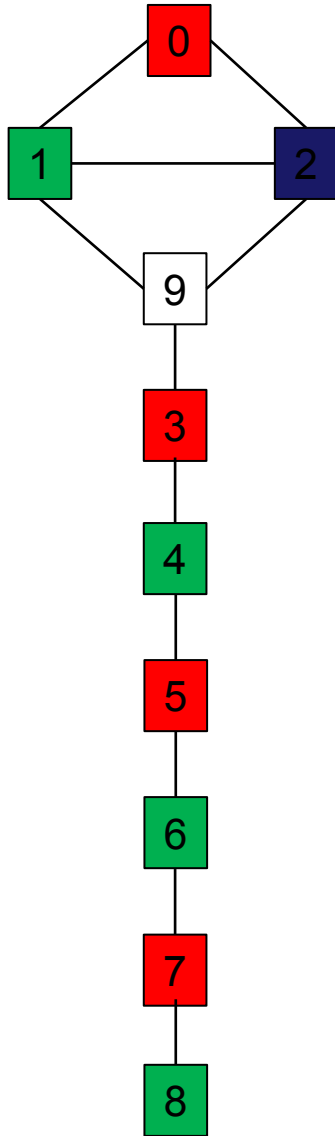
no possible value here

before we realise the conflict is caused here

Backjumping

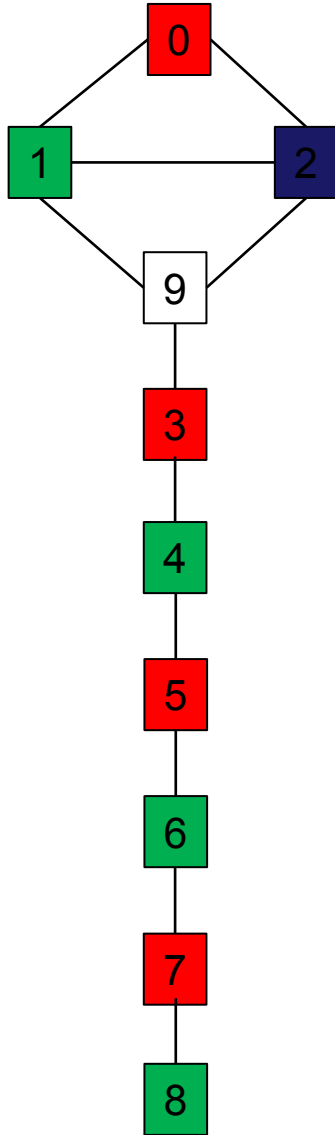
Why not remember, for each variable in the tree, the deepest variable above it that caused a conflict?

Then, when we must backtrack above our current variable, jump back to that variable.



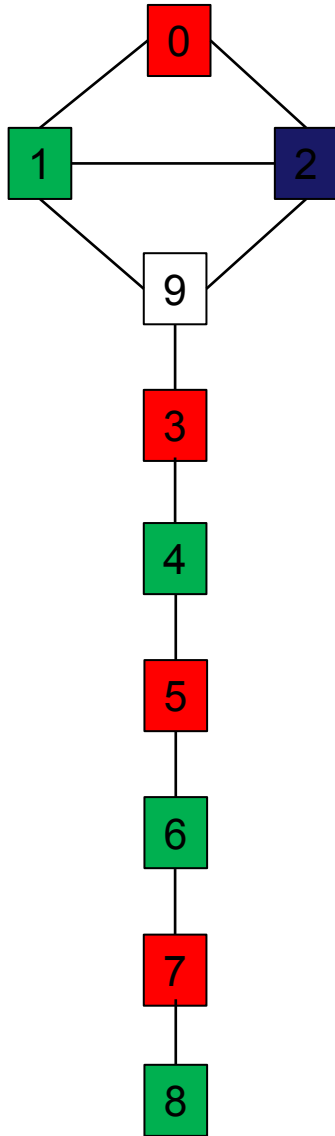
0	1	2	3	4	5	6	7	8	9
r	g	b	r	g	r	g	r	g	
-	0	1	-	3	-	5	-	7	



variables
 assigned values
 deepest conflict

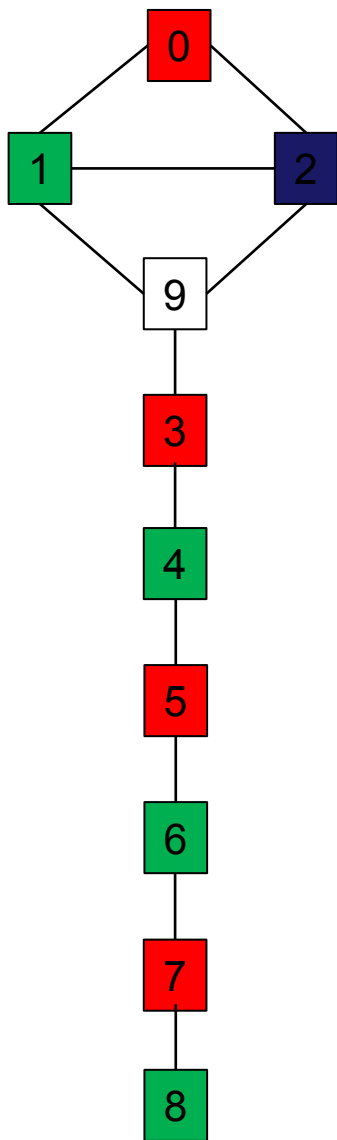


0	1	2	3	4	5	6	7	8	9	variables
r	g	b	r	g	r	g	r	g	r	assigned values
-	0	1	-	3	-	5	-	7	3	deepest conflict

A red arrow points down to the value '3' in the third row, column 4. Another red arrow points up to the value '3' in the third row, column 10.

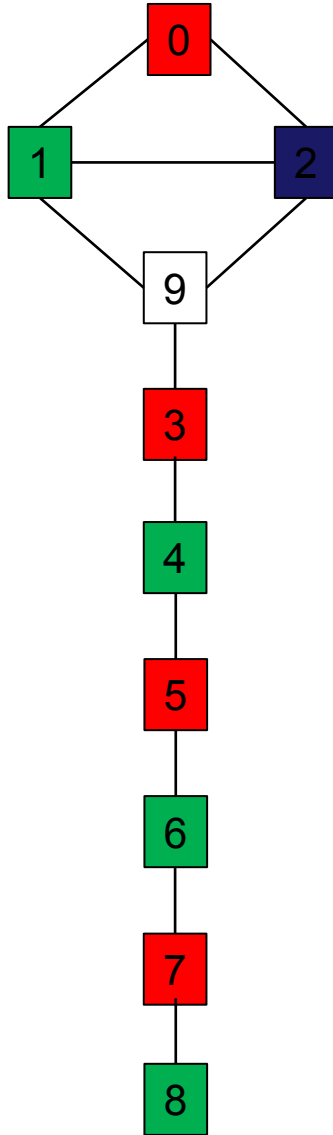


											
0	1	2	3	4	5	6	7	8	9	variables	
r	g	b	r	g	r	g	r	g	g	assigned values	
-	0	1	-	3	-	5	-	7	3	deepest conflict	
											



0	1	2	3	4	5	6	7	8	9	variables
r	g	b	r	g	r	g	r	g	b	assigned values
-	0	1	-	3	-	5	-	7	3	deepest conflict

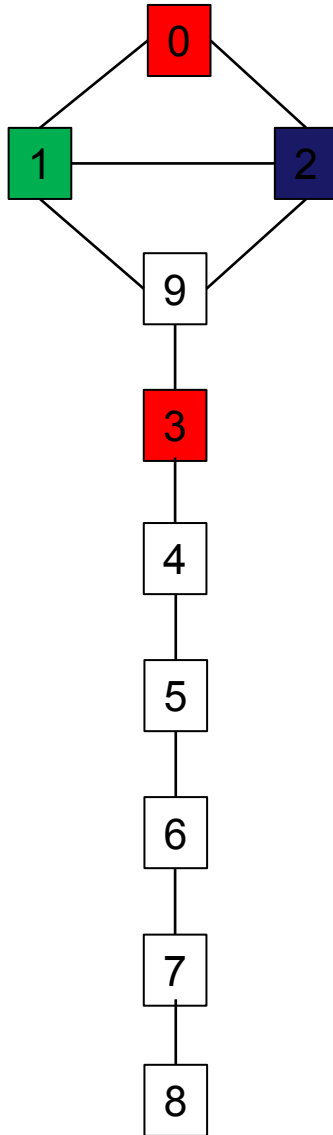
A red arrow points down to the value '2' in the first row. A red arrow points up to the value '3' in the third row.



No values left for 9 ...
so jump back to 3.

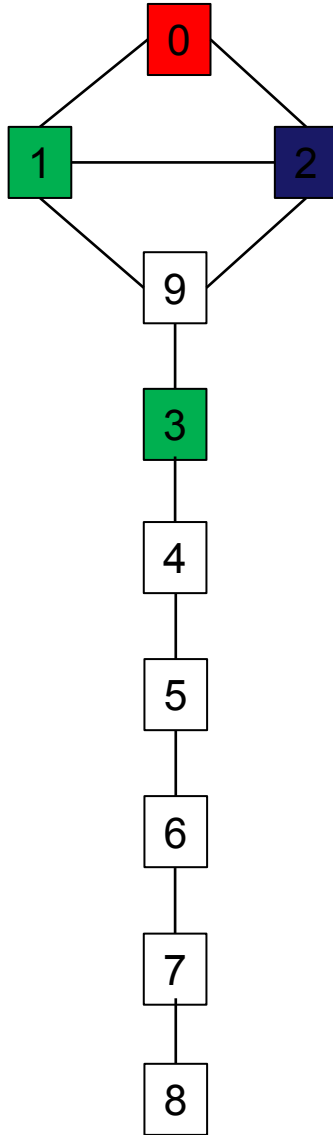
0	1	2	3	4	5	6	7	8	9	variables
r	g	b	r	g	r	g	r	g	b	assigned values
-	0	1	-	3	-	5	-	7	3	deepest conflict

↑



0	1	2	3	4	5	6	7	8	9
r	g	b	r						
-	0	1	-						

variables
 assigned values
 deepest conflict



and we can finish from here...

0	1	2	3	4	5	6	7	8	9
r	g	b	g						
-	0	1	-						

variables
assigned values
deepest conflict

$$V_0 \neq V_3$$
$$V_3 = V_9$$
$$V_6 = V_{12}$$
$$V_9 = V_{12}$$

BUT: consider this simple problem

all domains: $\{a,b\}$

0 1 2 3 4 5 6 7 8 9 10 11 12

variables

assigned values

deepest conflict

$V_0 \neq V_3$

$V_3 = V_9$

$V_6 = V_{12}$


$V_9 = V_{12}$ 

all domains: {a,b}

0	1	2	3	4	5	6	7	8	9	10	11	12	variables
a	a	a	b	a	a	a	a	a	b	a	a	a	assigned values
-	-	-	0	-	-	-	-	-	3	-	-	9	deepest conflict

$V0 \neq V3$

$V3 = V9$

$V6 = V12$ 

$V9 = V12$

all domains: {a,b}

V12 tries value b

That has a conflict with V6

But V6 is not deeper than V9,
so we don't update.

Now jump back to V9

0	1	2	3	4	5	6	7	8	9	10	11	12	variables
a	a	a	b	a	a	a	a	a	b	a	a	b	assigned values
-	-	-	0	-	-	-	-	-	3	-	-	9	deepest conflict

$V_0 \neq V_3$

$V_3 = V_9$

$V_6 = V_{12}$

$V_9 = V_{12}$

all domains: {a,b}

0	1	2	3	4	5	6	7	8	9	10	11	12	variables
a	a	a	b	a	a	a	a	a	b				assigned values
-	-	-	0	-	-	-	-	-	3				deepest conflict

$V0 \neq V3$

$V3 = V9$

$V6 = V12$

$V9 = V12$

all domains: {a,b}

V9 has no more values.

It has a deepest conflict at V3

So should we jump back to V3?

0	1	2	3	4	5	6	7	8	9	10	11	12	variables
a	a	a	b	a	a	a	a	a	b				assigned values
-	-	-	0	-	-	-	-	-	3				deepest conflict

$V_0 \neq V_3$

$V_3 = V_9$

$V_6 = V_{12}$

$V_9 = V_{12}$

all domains: {a,b}

No!

V_6 was also a conflict for V_{12}

If we set $V_6 \leftarrow b$, we can finish.

If we jump back to V_3 , we will miss a solution.

0	1	2	3	4	5	6	7	8	9	10	11	12	variables
a	a	a	b	a	a	a	a	a	b				assigned values
-	-	-	0	-	-	-	-	-	3				deepest conflict

$V0 \neq V3$

$V3 = V9$

$V6 = V12$

$V9 = V12$

After one jump back, we need to backtrack one step at a time, to make sure we don't miss a solution.

all domains: {a,b}

0	1	2	3	4	5	6	7	8	9	10	11	12	variables
a	a	a	b	a	a	a	a	a	b				assigned values
-	-	-	0	-	-	-	-	-	3				deepest conflict

Can we do better?

Conflict-Directed Backjumping (CBJ)

For each variable, as we move down through the tree, we record in a set the variable that caused the conflict for each value we rejected.

When we jump back up the tree, we pass back the records of all variable conflicts, and add to the conflict set of the variable we are jumping back to.

We delete the conflict sets of any variable we are backtracking from or jumping over.

$V_0 \neq V_3$

$V_1 = V_6$

$V_3 = V_9$

$V_6 = V_{12}$

$V_9 = V_{12}$

Consider this simple problem
with one new constraint added



all domains: $\{a, b\}$

0	1	2	3	4	5	6	7	8	9	10	11	12	variables
													assigned values

conflict sets

$V_0 \neq V_3$ ←

$$V1=V6$$
$$V_3 = V_9$$
$$V_6 = V_{12}$$
$$V_9 = V_{12}$$

all domains: $\{a,b\}$

[illegible]

- - - 0 conflict sets


$$V1=V6$$
$$V_3 = V_9$$
$$V_6 = V_{12}$$
$$V_9 = V_{12}$$

all domains: $\{a,b\}$

[illegible]

- - - 0 conflict sets

$V0 \neq V3$

$V1 = V6$ 

$V3 = V9$

$V6 = V12$

$V9 = V12$

all domains: {a,b}

0	1	2	3	4	5	6	7	8	9	10	11	12	variables
a	a	a	b	a	a	a							assigned values

-	-	-	0	-	-	-							conflict sets
---	---	---	---	---	---	---	--	--	--	--	--	--	---------------

$V_0 \neq V_3$

$V_1 = V_6$

$V_3 = V_9$ 

$V_6 = V_{12}$

$V_9 = V_{12}$

all domains: $\{a, b\}$

0	1	2	3	4	5	6	7	8	9	10	11	12	variables
a	a	a	b	a	a	a	a	a	a				assigned values

-	-	-	0	-	-	-	-	-	-	3			conflict sets
---	---	---	---	---	---	---	---	---	---	---	--	--	---------------

$V_0 \neq V_3$

$V_1 = V_6$

$V_3 = V_9$ 

$V_6 = V_{12}$

$V_9 = V_{12}$

all domains: $\{a, b\}$


0	1	2	3	4	5	6	7	8	9	10	11	12	variables
a	a	a	b	a	a	a	a	a	b				assigned values

-	-	-	0	-	-	-	-	-	-	3			conflict sets
---	---	---	---	---	---	---	---	---	---	---	--	--	---------------

$V_0 \neq V_3$

$V_1 = V_6$

$V_3 = V_9$

$V_6 = V_{12}$ 

$V_9 = V_{12}$ 

all domains: $\{a, b\}$

0	1	2	3	4	5	6	7	8	9	10	11	12	variables
a	a	a	b	a	a	a	a	a	b	a	a	a	assigned values


-	-	-	0	-	-	-	-	-	3	-	-	9	conflict sets
---	---	---	---	---	---	---	---	---	---	---	---	---	---------------

$V_0 \neq V_3$

$V_1 = V_6$

$V_3 = V_9$

$V_6 = V_{12}$

$V_9 = V_{12}$ 

all domains: $\{a, b\}$

0	1	2	3	4	5	6	7	8	9	10	11	12	variables
a	a	a	b	a	a	a	a	a	b	a	a	b	assigned values

-	-	-	0	-	-	-	-	-	3	-	-	9	conflict sets
---	---	---	---	---	---	---	---	---	---	---	---	---	---------------

$$V_0 \neq V_3$$
$$V_1 = V_6$$
$$V_3 = V_9$$
$$V_6 = V_{12}$$
$$V_9 = V_{12}$$


all domains: $\{a,b\}$

At this point, 12 has no values left, and the conflicts were caused by V6 and V9.

V10 & V11 are irrelevant.

0	1	2	3	4	5	6	7	8	9	10	11	12	variables
a	a	a	b	a	a	a	a	a	b	a	a	b	assigned values

- - - 0 - - - - 3 - - 9 **conflict sets**
 6

$V_0 \neq V_3$

$V_1 = V_6$

$V_3 = V_9$

$V_6 = V_{12}$


$V_9 = V_{12}$

So jump back to V_9 ...

but remember V_6 as the
earlier conflict.

all domains: $\{a, b\}$

0	1	2	3	4	5	6	7	8	9	10	11	12	variables
a	a	a	b	a	a	a	a	a	b				assigned values



-	-	-	0	-	-	-	-	-	3				conflict sets
									6				

$V0 \neq V3$

$V1 = V6$

$V3 = V9$


$V6 = V12$

$V9 = V12$

all domains: {a,b}

V9 has no values left, and
V3 was its deepest conflict.

But the reason we are
backtracking is because of
V12, so we should try V6



0	1	2	3	4	5	6	7	8	9	10	11	12	variables
a	a	a	b	a	a	a	a	a	b				assigned values
-	-	-	0	-	-	-	-	-	3				conflict sets
									6				

$V0 \neq V3$

$V1 = V6$

$V3 = V9$

$V6 = V12$

$V9 = V12$

So jump back to V6 ...

but remember V3 (the
conflict for V9)

all domains: {a,b}

0	1	2	3	4	5	6	7	8	9	10	11	12	variables
a	a	a	b	a	a	a							assigned values
-	-	-	0	-	-	3							conflict sets

$V_0 \neq V_3$

$V_1 = V_6$



$V_3 = V_9$

$V_6 = V_{12}$

$V_9 = V_{12}$

all domains: $\{a, b\}$

0	1	2	3	4	5	6	7	8	9	10	11	12	variables
a	a	a	b	a	a	b							assigned values

-	-	-	0	-	-	3							conflict sets
---	---	---	---	---	---	---	--	--	--	--	--	--	---------------

$V_0 \neq V_3$

$V_1 = V_6$


$V_3 = V_9$

$V_6 = V_{12}$

$V_9 = V_{12}$

all domains: {a,b}

0	1	2	3	4	5	6	7	8	9	10	11	12	variables
a	a	a	b	a	a	b							assigned values



-	-	-	0	-	-	3							conflict sets
						1							

$V_0 \neq V_3$

$V_1 = V_6$


$V_3 = V_9$

$V_6 = V_{12}$

$V_9 = V_{12}$

all domains: {a,b}

0	1	2	3	4	5	6	7	8	9	10	11	12	variables
a	a	a	b										assigned values



-	-	-	0	conflict sets
			1	

$V_0 \neq V_3$

$V_1 = V_6$


$V_3 = V_9$

$V_6 = V_{12}$

$V_9 = V_{12}$

all domains: $\{a, b\}$

0	1	2	3	4	5	6	7	8	9	10	11	12	variables
a	a												assigned values



– 0 conflict sets

$V_0 \neq V_3$

$V_1 = V_6$

$V_3 = V_9$

$V_6 = V_{12}$

$V_9 = V_{12}$

all domains: {a,b}

0	1	2	3	4	5	6	7	8	9	10	11	12	variables
a	b												assigned values

– 0 conflict sets

$$V_0 \neq V_3$$

$$V1=V6$$
$$V_3 = V_9$$
$$V_6 = V_{12}$$
$$V_9 = V_{12}$$

all domains: $\{a,b\}$

[illegible]

$- \quad 0 \quad - \quad 0$ conflict sets


$$V1=V6$$
$$V_3 = V_9$$
$$V_6 = V_{12}$$
$$V_9 = V_{12}$$

all domains: $\{a,b\}$

[illegible]

$- \quad 0 \quad - \quad 0$ conflict sets

$V_0 \neq V_3$

$V_1 = V_6$



$V_3 = V_9$

$V_6 = V_{12}$

$V_9 = V_{12}$

all domains: $\{a, b\}$

0	1	2	3	4	5	6	7	8	9	10	11	12	variables
a	b	a	b	a	a	a							assigned values

-	0	-	0	-	-	1							conflict sets
---	---	---	---	---	---	---	--	--	--	--	--	--	---------------

$V_0 \neq V_3$

$V_1 = V_6$ 

$V_3 = V_9$

$V_6 = V_{12}$

$V_9 = V_{12}$

all domains: $\{a, b\}$

0	1	2	3	4	5	6	7	8	9	10	11	12	variables
a	b	a	b	a	a	b							assigned values

-	0	-	0	-	-	1							conflict sets
---	---	---	---	---	---	---	--	--	--	--	--	--	---------------

$V_0 \neq V_3$

$V_1 = V_6$

$V_3 = V_9$



$V_6 = V_{12}$

$V_9 = V_{12}$

all domains: $\{a, b\}$

0	1	2	3	4	5	6	7	8	9	10	11	12	variables
a	b	a	b	a	a	b	a	a	a				assigned values

-	0	-	0	-	-	1	-	-	3				conflict sets
---	---	---	---	---	---	---	---	---	---	--	--	--	---------------

$V_0 \neq V_3$

$V_1 = V_6$

$V_3 = V_9$ 

$V_6 = V_{12}$

$V_9 = V_{12}$

all domains: $\{a, b\}$

0	1	2	3	4	5	6	7	8	9	10	11	12	variables
a	b	a	b	a	a	b	a	a	b				assigned values

-	0	-	0	-	-	1	-	-	3				conflict sets
---	---	---	---	---	---	---	---	---	---	--	--	--	---------------

$V_0 \neq V_3$

$V_1 = V_6$

$V_3 = V_9$

$V_6 = V_{12}$ 

$V_9 = V_{12}$

all domains: $\{a, b\}$

0	1	2	3	4	5	6	7	8	9	10	11	12	variables
a	b	a	b	a	a	b	a	a	b	a	a	a	assigned values


-	0	-	0	-	-	1	-	-	3	-	-	6	conflict sets
---	---	---	---	---	---	---	---	---	---	---	---	---	---------------

$V_0 \neq V_3$

$V_1 = V_6$

$V_3 = V_9$

$V_6 = V_{12}$ 

$V_9 = V_{12}$ 

all domains: $\{a, b\}$

0	1	2	3	4	5	6	7	8	9	10	11	12	variables
a	b	a	b	a	a	b	a	a	b	a	a	b	assigned values

-	0	-	0	-	-	1	-	-	3	-	-	6	conflict sets
---	---	---	---	---	---	---	---	---	---	---	---	---	---------------

cbj (V, D, C)

input: V (array of n int) D (2d array of n*d int) C (constraints)
output: (bool, V), where bool = true if a solution is found

```
0. conflict[] = {{}, {}, ..., {}} //array of n empty sets
1. var := 0 //start with the first variable
2. val := 0 //start with the first value
3. done := false //not finished yet
4. while not done
5.     if var == n return (true, values) //if reached end, stop
6.     else if var == -1 return (false, null) //if back to top, stop
7.     else if val == -1 //if no values left
8.         temp := max(conflict[var]) //find jump back point
9.         conflict[temp] := conflict[temp] ∪ conflict[var] - {temp}
10.        var := temp //merge conflict sets
11.        val := values[var] + 1 //continue to next value
12.    else if check(var, val, values, C) == (false, i)
13.        conflict[var] := conflict[var] ∪ {i}
14.        val := nextval(var, val) //get next value, or -1 if none
15.    else
16.        values[var] := val //assign current value
17.        var := var + 1 //move to next var
18.        conflict[var] := {} //start new conflict set
20.        val := 0 //start with first value
```

Things to note

- conflict-directed backjumping (cbj) is correct and complete
- cbj offers significant speed-ups over standard backtracking and backjumping
- the variable ordering appears to be significant
- cbj is believed to be used in most SAT solvers
- we will see another use for conflict sets later on, for automated explanations of why a given CSP allows some solutions but denies others.

Notes

- Backtracking was first (?) formalised by Golomb & Baumert (1965)
- Patrick Prosser first described CBJ (1993)

S. W. Golomb & L. D. Baumert , "Backtrack Programming", *Journal of the ACM*, Volume 12 Issue 4, pp516-524, 1965.

P. Prosser, "Hybrid Algorithms for the Constraint Satisfaction Problem", *Computational Intelligence*, Volume 9, Issue 3, pp268–299, 1993.

Next lecture ...

Combining Search and Inference

The map colouring example and the CSPs for illustrating CBJ were taken from examples by Patrick Prosser