LogicProblem.mws

Solving Constraint Satisfaction Problems Using Equivalence Relations

Author: Carl Devore <devore@math.udel.edu>

31 March 2001

If you use this program, please send me email at <devore@math.udel.edu>, either to discuss the program or just to say "hi". I like to keep track of how the program is being distributed. I am eager to see any suggestions about making this program better.

Introduction

This is a program for solving or partially solving a class of **Constraint Satisfaction Problems** (CSPs). Specifically, this program works on CSPs that can be stated in terms of finding an equivalence relation on a finite set, given that the user can specify a partition of the set into Systems of Distinct Representatives (SDRs) of the equivalence classes. In the examples that follow, we will see that many CSPs that do not at first appear to be in this form can be readily put into this form. In particular, many problems that appear in puzzle magazines as "Logic Problems" can be stated in this form.

```
> restart;
```

This application uses a Maple package called **LP** , contained in the file "LogicProblem.mpl", which is read by the next line. Make sure that file is in the same directory as this worksheet before executing.

```
> read "LogicProblem.mpl";
```

A simple classic logic problem - "Who Owns the Zebra?"

This problem is a classic example of a CSP. I believe that this problem is from an issue of *Reader's Digest* from the 1950's. I have seen it copied many places.

1. There five houses, each of a different color and inhabited by people of different nationalities, with different pets (one is a zebra), drinks (one drinks water), and cigarettes.

2. The English person lives in the red house.

3. The Spaniard owns the dog.

4. Coffee is drunk in the green house.

5. The Ukrainian drinks tea.

6. The green house is immediately to the right (your right) of the ivory house.

7. The Old Gold smoker owns snails.

8. Kools are smoked in the yellow house.

9. Milk is drunk in the middle house.

10. The Norwegian lives in the first house on the left.

11. The person who smokes Chesterfields lives next to the house with the fox.

12. The person who smokes Kools lives next to the house with the horse.

13. The Lucky Strike smoker drinks orange juice.

14. The Japanese smokes Parliaments.

15. The Norwegian lives next to the blue house.

Questions: Who owns the zebra? Who drinks water?

Solution:

The "variables" are House Number, Color, Pet, Nationality, Drink, and Cigarette.

> **Vars:= ['HN','Col','Pet','Nat','Drink','Cig'];**

$$Vars := [HN, Col, Pet, Nat, Drink, Cig]$$

For each variable, we must specify the values that it can assume.

> **HN:= [$1..5];**

$$HN := [1, 2, 3, 4, 5]$$

> **Col:= [red,green,yellow,blue,ivory];**

$$Col := [red, green, yellow, blue, ivory]$$

> **Pet:= [dog,snail,fox,horse,zebra];**

$$Pet := [dog, snail, fox, horse, zebra]$$

> **Nat:= [Eng,Spa,Ukr,Nor,Jap];**

$$Nat := [Eng, Spa, Ukr, Nor, Jap]$$

> **Drink:= [coffee,tea,water,oj,milk];**

$$Drink := [coffee, tea, water, oj, milk]$$

> **Cig:= [OG,Kool,Ches,Lucky,Parl];**

$$Cig := [OG, Kool, Ches, Lucky, Parl]$$

In the next line, set the number to the highest number of columns that will print neatly on your screen when you do the &? commands that print charts. I suppose that this will depend on monitor size. When I am using Windows '98, I press Control-1 to get the small font. When I am using Unix, the small font is just slightly too small to be comfortable, so I use the Control-2 font. In that case, only 24 columns (for this problem) will fit neatly across the screen.

> **interface(rtablesize=30);**

Note that the module (class) structure means that is possible to be working on multiple logic puzzles at the same time without any conflicts. Some of the subsequent worksheets explore this idea in detail. Each instantiation has its own copies of the "global" variables.

> **LP:= LogicProblem(Vars):**

> **with(LP);**

$$[\&!!, \&-, \&<, \&>, \&?, \&G, \&Soln, AutoGuess, CPV, CollectStats, ConstNum, Consts, ConstsInV, DifferentBlock, Equation, FreeGuess, GoBack, Guess, InternalRep, IsComplete, IsUnique, NC, NV, PrintConst, Quiet, Reinitialize, SameBlock, Satisfy, Separated, UniquenessProof, VarNum, VarNumC, X\_O]$$

There is a wide degree of control over how much internal information the program will print out. Higher infolevel numbers on the next line means you see more info; lower numbers, less info. Values higher than 3 give a great deal of information about the recursion. (TC stands for Transitive Closure).

> **infolevel['Constraints']:= 2: infolevel['TC']:= 1: CollectStats:= true:**

Record starting time to show efficiency.

> **st:= time():**

There is no need to Reinitialize on the first time through the problem. I just put the command here to make it easier to come back up to this point and change things.

```
> Reinitialize();
```

Okay

The constraints in the next line are entered in the order that they appear in the problem, using the notation that I developed for this. The program automatically sorts the constraints and processes them in an order that it believes is most efficient. Hopefully, the notation is obvious after reading through the above clues. "Succ" stands for "Successor". For every constraint type that relies on ordinal information (Succ, NextTo, Less, Rel), it is necessary to specify the variable with respect to which the order is measured. In this case HN (House Number) is the only ordinal variable. The order is not determined by the numerical values in HN; rather it is determined by the order that the values were assigned to the list HN.

```
> Satisfy([Eng=red, Spa=dog, coffee=green, Ukr=tea, Succ(green,ivory,HN), OG=snail, Kool=yellow, milk=3, Nor=1
,NextTo(fox,Ches,HN), NextTo(Kool,horse,HN), Lucky=oj, Jap=Parl
,NextTo(blue,Nor,HN)
]);
```

```
Know/NextTo:   Constraint   NextTo(blue,Nor,1)   completely satisfied.
```

$$Unsatisfied = [[-5, \mathrm{Succ}(green, ivory, 1)], [-8, \mathrm{NextTo}(fox, Ches, 1)], [-8, \mathrm{NextTo}(Kool, horse, 1)]]$$

```
Know/NextTo:   Constraint   NextTo(Kool,horse,1)   completely satisfied.
```

$$Unsatisfied = [[-5, \mathrm{Succ}(green, ivory, 1)], [-8, \mathrm{NextTo}(fox, Ches, 1)]]$$

```
Satisfy:
```

Attempting to fulfill unsatisfied constraints by guessing.

```
Backup:   Guessing   tea = horse   Depth=   1
```

$$Unsatisfied = [[-5, \mathrm{Succ}(green, ivory, 1)], [-8, \mathrm{NextTo}(fox, Ches, 1)]]$$

```
Know/Succ:   Constraint   Succ(green,ivory,1)   completely satisfied.

Know/NextTo:   Constraint   NextTo(fox,Ches,1)   completely satisfied.
```

$$Unsatisfied = [\ ]$$

```
Satisfy:
```

Complete, consistent solution found. Attempting to prove uniqueness.

```
GoBack:   Restoring saved state from before guess   tea = horse   Depth=   0

&<!=>:   Contradicts   horse = oj
```

Unique solution:

$$\begin{bmatrix} 1 & yellow & fox & Nor & water & Kool \\ 2 & blue & horse & Ukr & tea & Ches \\ 3 & red & snail & Eng & milk & OG \\ 4 & ivory & dog & Spa & oj & Lucky \\ 5 & green & zebra & Jap & coffee & Parl \end{bmatrix}$$

Note that the program decided on its own to make a guess at one point. That guess was proven necessary.

> **time()-st;**

.451

House number is obviously the "key" SDR. Let's look at everything we know about the house number.

> **&? HN;**

[, red , green , yellow , blue , ivory , . , dog , snail , fox , horse , zebra , . , Eng , Spa , Ukr , Nor , Jap , . , coffee , tea , water , oj , milk , . , OG , Kool , Ches , Lucky , Parl]
[1 , X , X , O , X , X , . , X , X , O , X , X , . , X , X , X , O , X , . , X , X , O , X , X , . , X , O , X , X , X]
[2 , X , X , X , O , X , . , X , X , X , O , X , . , X , X , O , X , X , . , X , O , X , X , X , . , X , X , O , X , X]
[3 , O , X , X , X , X , . , X , O , X , X , X , . , O , X , X , X , X , . , X , X , X , X , O , . , O , X , X , X , X]
[4 , X , X , X , X , O , . , O , X , X , X , X , . , X , O , X , X , X , . , X , X , X , O , X , . , X , X , X , O , X]
[5 , X , O , X , X , X , . , X , X , X , X , O , . , X , X , X , X , O , . , O , X , X , X , X , . , X , X , X , X , O]

CollectStats must be set to true in order to collect the information shown below.

> **&? ShowStats;**

$$Sets = 42,\ Unsets = 92,\ RuleOuts = 184,\ Elims = 398,\ Transfers = 154,\ Pivots = 182,\ MaxLevel = 2331,\ Guesses = 1,\ MaxDepth = 1$$

Let's see if it got to a solution first and then tried to prove uniqueness (this is obvious from the "infolevel" info printed above, but I am just illustrating the commands.)

> **&? PreviousSolution;**

$$\begin{bmatrix} 1 & yellow & fox & Nor & water & Kool \\ 2 & blue & horse & Ukr & tea & Ches \\ 3 & red & snail & Eng & milk & OG \\ 4 & ivory & dog & Spa & oj & Lucky \\ 5 & green & zebra & Jap & coffee & Parl \end{bmatrix}$$

A few more examples of the &? command.

One argument, a constant:

> **&? zebra;**

$$zebra = [5, green, Jap, coffee, Parl]$$

> **&? water;**

$$water = [1, yellow, fox, Nor, Kool]$$

Two arguments, both constants:

> **water &? fox;**

They are in the same equivalence class.

> **water &? zebra;**

They are in different equivalence classes.

> **water &? oj;**

They are in the same variable.

Two arguments, both variables:

> **Drink &? Pet;**

	coffee	tea	water	oj	milk
dog	X	X	X	O	X
snail	X	X	X	X	O
fox	X	X	O	X	X
horse	X	O	X	X	X
zebra	O	X	X	X	X

No arguments:

> **&?();**

A complete consistent solution has been found.

It has been proven unique.

1	yellow	fox	Nor	water	Kool
2	blue	horse	Ukr	tea	Ches
3	red	snail	Eng	milk	OG
4	ivory	dog	Spa	oj	Lucky
5	green	zebra	Jap	coffee	Parl

```
There are no undisproved guesses.

There are no unsatisfied constraints.
```

The solution chart can be displayed with respect to any leading variable:

> **Nat &? ``;**

$$\begin{bmatrix} Eng & 3 & red & snail & milk & OG \\ Spa & 4 & ivory & dog & oj & Lucky \\ Ukr & 2 & blue & horse & tea & Ches \\ Nor & 1 & yellow & fox & water & Kool \\ Jap & 5 & green & zebra & coffee & Parl \end{bmatrix}$$

The whole grid can be returned as a matrix: (It's probably not worth looking at, but it might be useful programmatically).

> **G:= &? Grid:**

The matrix is declared symmetric. The entries above the diagonal don't really exist as separate entities in memory. Also, each entry is tored as a one-byte integer.

The whole grid can be returned in printable form:

> **&? AllGrids;**

[, red , green , yellow , blue , ivory , . , dog , snail , fox , horse , zebra , . , Eng , Spa , Ukr , Nor , Jap , . , coffee , tea , water , oj , milk , . , OG , Kool , Ches , Lucky , Parl]
[1 , X , X , O , X , X , . , X , X , O , X , X , . , X , X , X , O , X , . , X , X , O , X , X , . , X , O , X , X , X]
[2 , X , X , X , O , X , . , X , X , X , O , X , . , X , X , O , X , X , . , X , O , X , X , X , . , X , X , O , X , X]
[3 , O , X , X , X , X , . , X , O , X , X , X , . , O , X , X , X , X , . , X , X , X , X , O , . , O , X , X , X , X]
[4 , X , X , X , X , O , . , O , X , X , X , X , . , X , O , X , X , X , . , X , X , X , O , X , . , X , X , X , O , X]
[5 , X , O , X , X , X , . , X , X , X , X , O , . , X , X , X , X , O , . , O , X , X , X , X , . , X , X , X , X , O]

[, 1 , 2 , 3 , 4 , 5 , . , dog , snail , fox , horse , zebra , . , Eng , Spa , Ukr , Nor , Jap , . , coffee , tea , water , oj , milk , . , OG , Kool , Ches , Lucky , Parl]
[red , X , X , O , X , X , . , X , O , X , X , X , . , O , X , X , X , X , . , X , X , X , X , O , . , O , X , X , X , X]
[green , X , X , X , X , O , . , X , X , X , X , O , . , X , X , X , X , O , . , O , X , X , X , X , . , X , X , X , X , O]
[yellow , O , X , X , X , X , . , X , X , O , X , X , . , X , X , X , O , X , . , X , X , O , X , X , . , X , O , X , X , X]
[blue , X , O , X , X , X , . , X , X , X , O , X , . , X , X , O , X , X , . , X , O , X , X , X , . , X , X , O , X , X]
[ivory , X , X , X , O , X , . , O , X , X , X , X , . , X , O , X , X , X , . , X , X , X , O , X , . , X , X , X , O , X]

[, 1 , 2 , 3 , 4 , 5 , . , *red* , *green* , *yellow* , *blue* , *ivory* , . , *Eng* , *Spa* , *Ukr* , *Nor* , *Jap* , . , *coffee* , *tea* , *water* , *oj* , *milk* , . , *OG* , *Kool* , *Ches* , *Lucky* , *Parl*]
[*dog* , *X* , *X* , *X* , O , *X* , . , *X* , *X* , *X* , *X* , O , . , *X* , O , *X* , *X* , *X* , . , *X* , *X* , *X* , O , *X* , . , *X* , *X* , *X* , O , *X*]
[*snail* , *X* , *X* , O , *X* , *X* , . , O , *X* , *X* , *X* , *X* , . , O , *X* , *X* , *X* , *X* , . , *X* , *X* , *X* , *X* , O , . , O , *X* , *X* , *X* , *X*]
[*fox* , O , *X* , *X* , *X* , *X* , . , *X* , *X* , O , *X* , *X* , . , *X* , *X* , *X* , O , *X* , . , *X* , *X* , O , *X* , *X* , . , *X* , O , *X* , *X* , *X*]
[*horse* , *X* , O , *X* , *X* , *X* , . , *X* , *X* , *X* , O , *X* , . , *X* , *X* , O , *X* , *X* , . , *X* , O , *X* , *X* , *X* , . , *X* , *X* , O , *X* , *X*]
[*zebra* , *X* , *X* , *X* , *X* , O , . , *X* , O , *X* , *X* , *X* , . , *X* , *X* , *X* , *X* , O , . , O , *X* , *X* , *X* , *X* , . , *X* , *X* , *X* , *X* , O]

[, 1 , 2 , 3 , 4 , 5 , . , *red* , *green* , *yellow* , *blue* , *ivory* , . , *dog* , *snail* , *fox* , *horse* , *zebra* , . , *coffee* , *tea* , *water* , *oj* , *milk* , . , *OG* , *Kool* , *Ches* , *Lucky* , *Parl*]
[*Eng* , *X* , *X* , O , *X* , *X* , . , O , *X* , *X* , *X* , *X* , . , *X* , O , *X* , *X* , *X* , . , *X* , *X* , *X* , *X* , O , . , O , *X* , *X* , *X* , *X*]
[*Spa* , *X* , *X* , *X* , O , *X* , . , *X* , *X* , *X* , *X* , O , . , O , *X* , *X* , *X* , *X* , . , *X* , *X* , *X* , O , *X* , . , *X* , *X* , *X* , O , *X*]
[*Ukr* , *X* , O , *X* , *X* , *X* , . , *X* , *X* , *X* , O , *X* , . , *X* , *X* , *X* , O , *X* , . , *X* , O , *X* , *X* , *X* , . , *X* , *X* , O , *X* , *X*]
[*Nor* , O , *X* , *X* , *X* , *X* , . , *X* , *X* , O , *X* , *X* , . , *X* , *X* , O , *X* , *X* , . , *X* , *X* , O , *X* , *X* , . , *X* , O , *X* , *X* , *X*]
[*Jap* , *X* , *X* , *X* , *X* , O , . , *X* , O , *X* , *X* , *X* , . , *X* , *X* , *X* , *X* , O , . , O , *X* , *X* , *X* , *X* , . , *X* , *X* , *X* , *X* , O]

[, 1 , 2 , 3 , 4 , 5 , . , *red* , *green* , *yellow* , *blue* , *ivory* , . , *dog* , *snail* , *fox* , *horse* , *zebra* , . , *Eng* , *Spa* , *Ukr* , *Nor* , *Jap* , . , *OG* , *Kool* , *Ches* , *Lucky* , *Parl*]
[*coffee* , *X* , *X* , *X* , *X* , O , . , *X* , O , *X* , *X* , *X* , . , *X* , *X* , *X* , *X* , O , . , *X* , *X* , *X* , *X* , O , . , *X* , *X* , *X* , *X* , O]
[*tea* , *X* , O , *X* , *X* , *X* , . , *X* , *X* , *X* , O , *X* , . , *X* , *X* , *X* , O , *X* , . , *X* , *X* , O , *X* , *X* , . , *X* , *X* , O , *X* , *X*]
[*water* , O , *X* , *X* , *X* , *X* , . , *X* , *X* , O , *X* , *X* , . , *X* , *X* , O , *X* , *X* , . , *X* , *X* , *X* , O , *X* , . , *X* , O , *X* , *X* , *X*]
[*oj* , *X* , *X* , *X* , O , *X* , . , *X* , *X* , *X* , *X* , O , . , O , *X* , *X* , *X* , *X* , . , *X* , O , *X* , *X* , *X* , . , *X* , *X* , *X* , O , *X*]
[*milk* , *X* , *X* , O , *X* , *X* , . , O , *X* , *X* , *X* , *X* , . , *X* , O , *X* , *X* , *X* , . , O , *X* , *X* , *X* , *X* , . , O , *X* , *X* , *X* , *X*]

[, 1 , 2 , 3 , 4 , 5 , . , *red* , *green* , *yellow* , *blue* , *ivory* , . , *dog* , *snail* , *fox* , *horse* , *zebra* , . , *Eng* , *Spa* , *Ukr* , *Nor* , *Jap* , . , *coffee* , *tea* , *water* , *oj* , *milk*]
[*OG* , *X* , *X* , O , *X* , *X* , . , O , *X* , *X* , *X* , *X* , . , *X* , O , *X* , *X* , *X* , . , O , *X* , *X* , *X* , *X* , . , *X* , *X* , *X* , *X* , O]
[*Kool* , O , *X* , *X* , *X* , *X* , . , *X* , *X* , O , *X* , *X* , . , *X* , *X* , O , *X* , *X* , . , *X* , *X* , *X* , O , *X* , . , *X* , *X* , O , *X* , *X*]
[*Ches* , *X* , O , *X* , *X* , *X* , . , *X* , *X* , *X* , O , *X* , . , *X* , *X* , *X* , O , *X* , . , *X* , *X* , O , *X* , *X* , . , *X* , O , *X* , *X* , *X*]
[*Lucky* , *X* , *X* , *X* , O , *X* , . , *X* , *X* , *X* , *X* , O , . , O , *X* , *X* , *X* , *X* , . , *X* , O , *X* , *X* , *X* , . , *X* , *X* , *X* , O , *X*]
[*Parl* , *X* , *X* , *X* , *X* , O , . , *X* , O , *X* , *X* , *X* , . , *X* , *X* , *X* , *X* , O , . , *X* , *X* , *X* , *X* , O , . , O , *X* , *X* , *X* , *X*]

The whole Stack of solution states can be returned as a list. (Since we have a complete, consistent, unique solution, the stack is necessarily empty at this point). In general, the

stack would be too much information to look at on the screen, but it could be assigned to a variable.

> **&? Stack;**

[ ]

The current list of Unsatisfied clues can be returned (necessarily empty if the solution is complete).

> **&? Unsat;**

[ ]

The current list of undisproved guesses (necessarily empty if the solution is unique):

> **&? ActiveGuesses;**

[ ]

Often, you only want to know the number of active guesses:

> **&? CountGuesses;**

0

I will redo the problem and alter the format of some of the constraints. Note that a NextTo constraint is logically equivalent to the disjunction of two Succ constraints.

> **Reinitialize();**

Okay

> **st:= time():**

> **Satisfy([Eng=red, Spa=dog, coffee=green, Ukr=tea, Succ(green,ivory,HN), OG=snail, Kool=yellow, milk=3, Nor=1**
,OR([Succ(fox,Ches,HN), Succ(Ches, fox, HN)]), OR([Succ(Kool,horse,HN), Succ(horse,Kool,HN)]), Lucky=oj, Jap=Parl
,OR([Succ(blue,Nor,HN), Succ(Nor,blue,HN)])
]);

```
Backup:   Guessing   OR([1, Succ(fox,Ches,[1, 2, 3, 4, 5]), Succ(Ches,fox,[1, 2, 3, 4, 5])])   Depth=   1

Backup:   Guessing   OR([1, Succ(blue,Nor,[1, 2, 3, 4, 5]), Succ(Nor,blue,[1, 2, 3, 4, 5])])   Depth=   2
```

Know/Succ: Constraint Succ(blue,Nor,1) completely satisfied.

Backup: Guessing OR([1, Succ(Kool,horse,[1, 2, 3, 4, 5]), Succ(horse,Kool,[1, 2, 3, 4, 5])]) Depth= 3

$$Unsatisfied = [[-5, \mathrm{Succ}(green, ivory, 1)], [-5, \mathrm{Succ}(fox, Ches, 1)], [-5, \mathrm{Succ}(Kool, horse, 1)]]$$

&<!=>: Contradicts ivory = 1

NegateLastGuess: Backing up to last guess. Level = 95 Depth = 3

GoBack: Restoring saved state from before guess OR([1, Succ(Kool,horse,[1, 2, 3, 4, 5]), Succ(horse,Kool,[1, 2, 3, 4, 5])]) Depth= 2

$$Unsatisfied = [[-5, \mathrm{Succ}(green, ivory, 1)], [-5, \mathrm{Succ}(fox, Ches, 1)], [-5, \mathrm{Succ}(horse, Kool, 1)], [-10, \mathrm{Rel}(NotSucc, Kool, horse, 1, [\,])]]$$

Know/Succ: Constraint Succ(horse,Kool,1) completely satisfied.

Rel/Check: Constraint Rel(NotSucc,Kool,horse,1,[]) satisfied.

$$Unsatisfied = [[-5, \mathrm{Succ}(green, ivory, 1)], [-5, \mathrm{Succ}(fox, Ches, 1)]]$$

Satisfy:

Attempting to fulfill unsatisfied constraints by guessing.

Backup: Guessing Ukr = 2 Depth= 3

$$Unsatisfied = [[-5, \mathrm{Succ}(green, ivory, 1)], [-5, \mathrm{Succ}(fox, Ches, 1)]]$$

Know/Succ: Constraint Succ(green,ivory,1) completely satisfied.

&<!=>: Contradicts Ches = 2

NegateLastGuess: Backing up to last guess. Level = 95 Depth = 3

GoBack: Restoring saved state from before guess Ukr = 2 Depth= 2

&<!=>: Contradicts 2 = Jap

NegateLastGuess: Backing up to last guess. Level = 90 Depth = 2

GoBack: Restoring saved state from before guess OR([1, Succ(blue,Nor,[1, 2, 3, 4, 5]), Succ(Nor,blue,[1, 2, 3, 4, 5])])

```
Depth=   1

&<!=>:   Contradicts   Nor = 1

NegateLastGuess:   Backing up to last guess.   Level = 90   Depth = 1

GoBack:   Restoring saved state from before guess   OR([1, Succ(fox,Ches,[1, 2, 3, 4, 5]), Succ(Ches,fox,[1, 2, 3, 4, 5])])
Depth=   0

Backup:   Guessing   OR([1, Succ(blue,Nor,[1, 2, 3, 4, 5]), Succ(Nor,blue,[1, 2, 3, 4, 5])])   Depth=   1

Know/Succ:   Constraint   Succ(blue,Nor,1)   completely satisfied.

Backup:   Guessing   OR([1, Succ(Kool,horse,[1, 2, 3, 4, 5]), Succ(horse,Kool,[1, 2, 3, 4, 5])])   Depth=   2
```

$$Unsatisfied = [[-5, \mathrm{Succ}(green, ivory, 1)], [-5, \mathrm{Succ}(Ches, fox, 1)], [-10, \mathrm{Rel}(NotSucc, fox, Ches, 1, [\,])], [-5, \mathrm{Succ}(Kool, horse, 1)]]$$

```
&<!=>:   Contradicts   ivory = 1

NegateLastGuess:   Backing up to last guess.   Level = 95   Depth = 2

GoBack:   Restoring saved state from before guess   OR([1, Succ(Kool,horse,[1, 2, 3, 4, 5]), Succ(horse,Kool,[1, 2, 3, 4,
5])])   Depth=   1
```

$$Unsatisfied = [[-5, \mathrm{Succ}(green, ivory, 1)], [-5, \mathrm{Succ}(Ches, fox, 1)], [-10, \mathrm{Rel}(NotSucc, fox, Ches, 1, [\,])], [-5, \mathrm{Succ}(horse, Kool, 1)], [-10, \mathrm{Rel}(NotSucc, Kool, horse, 1, [\,])]]$$

```
Know/Succ:   Constraint   Succ(horse,Kool,1)   completely satisfied.

Rel/Check:   Constraint   Rel(NotSucc,Kool,horse,1,[])   satisfied.
```

$$Unsatisfied = [[-5, \mathrm{Succ}(green, ivory, 1)], [-5, \mathrm{Succ}(Ches, fox, 1)], [-10, \mathrm{Rel}(NotSucc, fox, Ches, 1, [\,])]]$$

$$Unsatisfied = [[-5, \mathrm{Succ}(green, ivory, 1)], [-5, \mathrm{Succ}(Ches, fox, 1)], [-10, \mathrm{Rel}(NotSucc, fox, Ches, 1, [\,])]]$$

```
Satisfy:
```

Attempting to fulfill unsatisfied constraints by guessing.

```
Backup:   Guessing   tea = horse   Depth=   2
```

$$Unsatisfied = [[-5, Succ(green, ivory, 1)], [-5, Succ(Ches, fox, 1)], [-10, Rel(NotSucc, fox, Ches, 1, [\ ])]]$$

```
Know/Succ:   Constraint   Succ(green,ivory,1)   completely satisfied.

Know/Succ:   Constraint   Succ(Ches,fox,1)   completely satisfied.

Rel/Check:   Constraint   Rel(NotSucc,fox,Ches,1,[])   satisfied.

Satisfy:
```

Complete, consistent solution found. Attempting to prove uniqueness.

```
GoBack:   Restoring saved state from before guess   tea = horse   Depth=   1

&<!=>:   Contradicts   horse = oj

NegateLastGuess:   Backing up to last guess.   Level = 90   Depth = 1

GoBack:   Restoring saved state from before guess   OR([1, Succ(blue,Nor,[1, 2, 3, 4, 5]), Succ(Nor,blue,[1, 2, 3, 4, 5])])
Depth=   0

&<!=>:   Contradicts   Nor = 1
```

Unique solution:

$$\begin{bmatrix} 1 & yellow & fox & Nor & water & Kool \\ 2 & blue & horse & Ukr & tea & Ches \\ 3 & red & snail & Eng & milk & OG \\ 4 & ivory & dog & Spa & oj & Lucky \\ 5 & green & zebra & Jap & coffee & Parl \end{bmatrix}$$

```
> time()-st;
```

.441

```
> &? ShowStats;
```

$$Sets = 77, Unsets = 196, RuleOuts = 356, Elims = 734, Transfers = 318, Pivots = 352, MaxLevel = 2418, Guesses = 7, MaxDepth = 3$$

Note that if takes far more effort for the program to solve the problem this way. The OR constraint is the most inefficient constraint, especially when a uniqueness proof is attempted. Avoid using the OR whenever possible.

See the worksheet "MoreLogicProblems" for more complex examples.