ColorASquareShort.mws

A combinatorial problem: Coloring a Square

Carl Devore <devore@math.udel.edu>

31 March 2001

```
> 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 more abstract problem with several programmatically generated constraints and programmatically analyzed output.

Since this problem is much more abstract and combinatorial than the previous problems, it is easier to see how it can generalize to real-world problems.

Problem:

Adapted from "60. Pigeonhole" in *Official's Logic Problems* (vol. 6, no. 6, December 2000, ISSN 1088-3096)

Beverly has a pigeonhole shelf with 36 square holes in a 6x6 grid. She keeps a minature fork, knife, or spoon in each hole such that in each row and column there are exactly two of each.

We label the rows A through F, and label the columns 1 through 6.

1. In row A, each fork is horizontally adjacent only to knives.

2. In row B, the two forks are either two or three holes apart.

3. In row C, the two edge holes contain, in some order, a fork and a spoon.

4. In both row D and row F, the two spoons are at least three holes apart.

5. In column 1, the two center holes contain, in some order, a fork and a knife.

6. In both column 2 and column 6, the two knives are two holes apart.

7. In column 4, the two knives are adjacent and the two forks are at least three holes apart.

8. In column 5, the two knives are either adjacent or four holes apart.

Solution:

The problem can be stated in a more abstract and mathematical way thus: We seek to color a 6x6 grid with 3 colors in such a way that there are two of each color in every row and column and such that the other constraints are satisfied. I will take this interpretation later when I animate the solution process,

It is obvious that there are 12 of each item. To encode the problem, we must think of each fork, knife, and spoon as a distinct entity, although in the final solution, it does matter *which* fork, knife, or spoon goes in a particular hole. Once we think of them as distinct entities, we need to specify the constraints in such a manner that the item in each position in uniquely specified. There are two approaches to this, which I'll call "ByRows" and "ByCols".

In approach "ByRows", we consider the left fork in row A to be fork1, the right fork in row A to be fork2, the left fork in row B to be fork3, etc., and similarly for the spoons and knives. In approach "ByCols", we consider the upper fork in column 1 to be fork1, the lower fork in column 1 to be fork2, etc. "ByRows" makes constraints 2 and 4 easy to encode, and 6, 7, and 8 difficult. Vice versa for "ByCols". Constraint 1 is slightly easier in "ByRows". Constraints 3 and 5 are easy in either case. For the present solution, I will use "ByRows". In a subsequent solution, I will use both.

```
> V:= ['holes_BR', 'items']:
```

The "d" is lowercase below to avoid conflict with the differentiation operator.

```
> rows:= [A,B,C,d,E,F]:
```

"BR" stands for By Rows.

```
> holes_BR:= [seq(seq(cat(j,i), i= 1..6), j= rows)];
```

```
holes_BR :=
[A1, A2, A3, A4, A5, A6, B1, B2, B3, B4, B5, B6, C1, C2, C3, C4, C5, C6, d1, d2, d3, d4, d5, d6, E1, E2, E3, E4, E5, E6, F1, F2, F3, F4, F5, F6]
```

```
> forks:= {f||(1..12)}: spoons:= {s||(1..12)}: knives:= {k||(1..12)}:
```

```
> items:= [op(spoons), op(knives), op(forks)];
```

```
items := [s7, s8, s10, s11, s1, s2, s6, s9, s3, s4, s5, s12, k4, k5, k12, k6, k9, k3, k10, k11, k7, k8, k2, k1, f3, f4, f12, f5, f9, f10, f2, f11, f6, f7, f8, f1]
```

```
> Types:= [fork, knife, spoon]:

> TypeToItem:= table([knife= knives, fork= forks, spoon= spoons]):

> NotItem:= table([seq(t = {op(items)} minus TypeToItem[t], t= Types)]):

> for t in Types do for i in TypeToItem[t] do setattribute(i, t) od od;

> BR:= LogicProblem(V):
```

This procedural constraint forces the forks in row A to be only next to knives.

```
> Clue1:= proc(M)
local pos_f1, pos_f2, pos_k1, pos_k2, Vh, C, L, k;
use VarNum= M:-VarNum, ConstNum= M:-ConstNum, `&Soln`= M:-`&Soln` in
Vh:= VarNum(holes_BR);
pos_f1:= ConstNum(f1) &Soln Vh;
pos_f2:= ConstNum(f2) &Soln Vh;
pos_k1:= ConstNum(k1) &Soln Vh;
pos_k2:= ConstNum(k2) &Soln Vh;
C:= [];
if pos_f1 > ConstNum(A1) then
if pos_k2 > 0 and pos_k2<>pos_f1+1 then return true fi;
if pos_k1 > 0 and pos_k1<>pos_f1-1 then return true fi;
C:= [k1 = holes_BR[pos_f1-1], k2 = holes_BR[pos_f1+1]]
elif pos_f1 = ConstNum(A1) then
if pos_k1 > 0 and pos_k1<>pos_f1+1 then return true fi;
C:= [k1 = A2]
fi;
if pos_f2 > 0 and pos_f2 < ConstNum(A6) then
if pos_k2 > 0 and pos_k2<>pos_f2+1 then return true fi;
if pos_k1 > 0 and pos_k1<>pos_f2-1 then return true fi;
C:= [op(C), k1 = holes_BR[pos_f2-1], k2 = holes_BR[pos_f2+1]]
elif pos_f2 = ConstNum(A6) then
if pos_k2 > 0 and pos_k2<>pos_f2-1 then return true fi;
C:= [op(C), k2 = A5]
fi;
for k in [k1,k2] do
L:= map(rhs, select(e -> lhs(e) = k, C));
if nops(L)=2 and L[1]<>L[2] then return true fi
od;
end use;
false, evalb(pos_f1>0 and pos_f2>0 and pos_k1>0 and pos_k2>0), C
```

```
end proc:
```

This procedural constraint enforces the 2-each-per-column constraint and constraints 6, 7, and 8.

This procedure uses the selectremove builtin function which is new to Maple6.

```
> Cols:= proc(M, item_col)
local Type, Vi, col, item_count, blanks, row, blank_count, pos, result, ExcludeRows;
Type:= item_col[1];
col:= item_col[2];
use `&Soln`= M:-`&Soln`, ConstNum= M:-ConstNum, VarNum= M:-VarNum, Consts= M:-Consts, `&?`= M:-`&?` in
userinfo(4, 'Constraints', print(PrintSoln(holes_BR &? ``)));
Vi:= VarNum(items);
pos:= [];
blanks:=
select
(proc(row)
local item;
item:= ConstNum(cat(rows[row], col)) &Soln Vi;
if item<>0 then if attributes(Consts[item]) = Type then pos:= [op(pos), row] fi; false else true fi;
end
,[$1..6]
);
end use;
blank_count:= nops(blanks);
item_count:= nops(pos);
if item_count > 2 or item_count+blank_count < 2 then return true
elif item_count>0 then
ExcludeRows:= proc(Cond)
local C, row, bad_rows, good_rows;
# Note that it is possible at this point to make a sophisticated analysis for the case item_count=0.
# However, I have not attempted that here. To do so would significantly improve the runtime by reducing the number of guesses.
if item_count=1 then
bad_rows, good_rows:= selectremove(r -> Cond(r, pos[1]), blanks);
userinfo(4,'Constraints', good_rows, bad_rows);
C:= [seq(Distinct([cat(rows[row], col), TypeToItem[Type]]), row= bad_rows)];
if nops(good_rows)=1 then
return false, false, [op(C), Distinct([cat(rows[good_rows[1]], col), NotItem[Type]])]
elif nops(good_rows)=0 then
return true
fi;
false, false, C
elif Cond(pos[2], pos[1]) then
```

```
true
else
NULL
fi
end proc;

result:=
`if`((col=2 or col=6) and Type=knife
,ExcludeRows(proc(p,q) option inline; abs(p-q)<>2 end) #Clue 6
,`if`(col=4 and Type=knife
,ExcludeRows(proc(p,q) option inline; abs(p-q)<>1 end) #Clue 7a
,`if`(col=4 and Type=fork
,ExcludeRows(proc(p,q) option inline; abs(p-q)<3 end) #Clue 7b
,`if`(col=5 and Type=knife
,ExcludeRows(proc(p,q) option inline; abs(p-q)<>1 and abs(p-q)<>4 end) #Clue 8
,NULL
)
)
)
);
if result<>NULL then userinfo(4,'Constraints',result); return result fi
fi;
if blank_count=0 or item_count=2 then false, true
elif blank_count+item_count=2 then
false, false, [seq(Distinct([cat(rows[row], col), NotItem[Type]]), row= blanks)]
else false, false
fi
end proc:
```

This procedure interprets the solution chart in a way that is meaningful to this problem.

This procedure uses an array-constructor procedure, a feature new to Maple6.

```
> PrintSoln:= proc(Sol)
rtable
(1..6, 1..6
,proc(i,j)
local item, F, K, S, _;
item:= attributes(Sol[6*(i-1)+j, 2]);
`if`(item=fork, F, `if`(item=spoon, S, `if`(item=knife, K, _)))
end
)
end:
```

This procedure will be used in a dummy constraint so that we can watch the solution progress. It is useful to watch the solution progress so that you can improve the efficiency of your procedural constraints. Also, I will save each iteration for an animation later. Note that I used &? Solved to stop collecting this information once a consistent solution is reached. The second return value, true, causes the dummy constraint to be removed from the unsatisfied list.

Note how I refer to the global variables with :-. This feature is new to Maple6.

```
> ShowIter:= proc(M)
local S;
use `&?`= M:-`&?` in
if &? Solved then return false, true fi;
S:= PrintSoln(holes_BR &? ``);
print(`\n`, S, ` Depth`= &? CountGuesses)
end use;
:-frame:= :-frame+1;
:-Frames[:-frame]:= eval(S);
false, false
end proc:
```

This procedure illustrates a technique that is very useful in debugging large problems -- we change the infolevel settings after a certain number of guesses have been made. Note how I retrieve the number of guesses as the eigth member of the stats list.

Note how I specify that infolevel is a global. This feature is new to Maple6.

```
> IncreaseInfo:= proc(M, opts)
use `&?`= M:-`&?` in
if [&? Stats][8] = opts[1] then :-infolevel[all]:= opts[2]; return false, true fi
end use;
false, false
end proc:
```

Note in the following how large sequences of constraints can be generated. Note that it is allowed to refer to the variables (but not the constants) by their internal reference numbers. This usually makes the output of a long list of programmatically generated constraints easier to read for debugging purposes.

```
> Vholes:= BR:-VarNum(holes_BR); Vitems:= BR:-VarNum(items):
```

$$Vholes := 1$$

```
> Constraints:=
[#This guarantees that there are two of each item in each row and that they are ordered as specified above.
seq(op([Less([cat(it,1), cat(it,2), B1], Vholes)
,seq(Less([holes_BR[6*(i-1)], cat(it,2*i-1), cat(it,2*i), holes_BR[6*i+1]], Vholes), i= 2..5)
,Less([E6, cat(it,11), cat(it,12)], Vholes)
])
,it= [f,s,k]
)

#Use the procedure Cols for each item-column pair.
,seq(seq(Proc(Cols, [item, col]), item= Types), col= 1..6)

#Clue 1
,NextTo(f1, k1, Vholes), NextTo(f2, k2, Vholes), Less(k1, f2, Vholes), Less(f1, k2, Vholes)
,Distinct([ {A1, A6}, knives])
,Proc(Clue1, [])

#Clue 2
# Note that the various procedures that I have included for use with "Rel" are exports, and thus the module prefix
# is needed if the module is not invoked with "with".
,Rel(BR:-Equation, f3, f4, Vholes, [{_A+2, _A+3}, {_B-2, _B-3}])

#Clue 3
,Distinct([{C1,C6}, knives]), Rel(BR:-DifferentBlock, C1, C6, Vitems, [forks, spoons])

#Clue 4
# Here I introduce the Separated constraint type. The number in square brackets indicates the MINimum number
# of other items that must be between the two specified items.
,Rel(BR:-Separated, s7, s8, Vholes, [2])
,Rel(BR:-Separated, s11, s12, Vholes, [2])

#Clue 5
,Distinct([{C1,d1}, spoons]), Rel(BR:-DifferentBlock, C1, d1, Vitems, [forks, knives])

,Dummy(proc() :-frame:= 0; false, true end, [])
,Dummy(ShowIter, [])
,Dummy(IncreaseInfo, [89, 1])
]:

> BR:-Reinitialize();

> infolevel['Constraints']:= 0: infolevel['TC']:= 0: infolevel[all]:= 0: BR:-CollectStats:= true:

> BR:-UniquenessProof:= false:

> interface(rtablesize=37);

> frame:= 0:
```

```
> st:= time():
```

Note: In order to load the HTML display of this worksheet efficiently, the output for the first half of the worksheet has been greatly abbreviated. Please download the actual .mws file to view the complete output.

```
> PrintSoln(BR:-Satisfy(Constraints));
```

$$, \left[\begin{array}{cccccc} \_ & \_ & \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ & \_ & \_ \end{array}\right], \mathit{Depth} = 0$$

```
Warning, computation interrupted
```

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

```
> use BR in &? ShowStats end;
```

Animate the progression of the solution. We will consider a fork to be a pink square, a spoon to be a yellow square, and a knife to be a turquoise square. Currently unknown positions will be shown as black.

```
> plotgrid:= proc(G)
local i,j, colors, F,S,K;
colors:=
rtable
(1..6, 1..6
,(i,j) -> `if`(cat(G[i,j])=cat(F), [1,.5,.7]
,`if`(cat(G[i,j])=cat(S), [1,1,0]
,`if`(cat(G[i,j])=cat(K), [0,1,.8], [0,0,0])
)
)
);
PLOT
(POLYGONS
(seq(seq([[j,-i], [j,-i-1], [j+1,-i-1], [j+1,-i]], i= 1..6), j= 1..6)
,COLOUR(RGB, seq(seq(op(colors[i,j]), i= 1..6), j= 1..6))
)
)
end proc:
```

```
> AnimateSoln:=
Frames ->
plots[display]
([seq(plotgrid(Frames[i]), i= 1..frame)]
,insequence= true, axes= none, scaling= constrained
):
```

```
> AnimateSoln(Frames);
```

Remember to click on plot and then click on the animation controls at the top of the screen to play it. I would also suggest that you slow down the frame rate some.

It is interesting to note how many times the program changes its mind about the first row. This suggests that it may be possible to improve the solution time bysharpening procedural constraint forthe first row.

Now I will redo the problem and try for a uniqueness proof.

```
> infolevel[all]:= 0:
```

```
> BR:-Reinitialize();
```

```
> BR:-UniquenessProof:= true:
```

```
> st:= time():
```

I will remove the 2 dummy constraints to speed up the solution a bit.

Warning: the next command takes about 3 minutes on my AMD K6 600 MHz processor.

```
> PrintSoln(BR:-Satisfy(subsop(-1= NULL, -2= NULL, Constraints)));
```

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

```
> use BR in &? ShowStats end;
```

The technique of multiple communicating problems

A far more sophicated approach to solving this problem is as follows: We set up two problems. The first problem is encoded in the "ByRows" manner; the second in the "ByCols" manner. Each problem only has the constraints that are easy for that problem (as long as every constraint is specified in at least one of the two problems!). Then a procedural constraint is used to pass information between the problems. This technique is extremely powerful.

This approach shows the awesome power of Maple6's module construct.

Construct the list of holes by columns:

```
> holes_BC:= [seq(seq(cat(j,i), j= rows), i= 1..6)];
```

```
holes_BC :=
[A1, B1, C1, d1, E1, F1, A2, B2, C2, d2, E2, F2, A3, B3, C3, d3, E3, F3, A4, B4, C4, d4, E4, F4, A5, B5, C5, d5, E5, F5, A6, B6, C6, d6, E6, F6]
```

```
> BC:= LogicProblem([holes_BC, items]):
```

```
> BR:-UniquenessProof:= true:
```

```
> BC:-CollectStats:= true:
```

We consider ByRows to be the master problem because it has the procedural constraint for clue 1. This decision is somewhat arbitrary, but it makes some sense to me. All guessing will be done in ByRows.

```
> BC:-AutoGuess:= false:
```

It doesn't make any sense to try a uniqueness proof in the slave problem.

```
> BC:-UniquenessProof:= false:
```

```
> BC:-Quiet:= true:
```

We need a procedure that can interpret the grids in a way that is meaningful to both problems so that information can be passed between the two. Note that the position letters and numbers mean the same thing in both problems; whereas the item numbers mean different things. For example, suppose that in ByRows we know that position C3 is matched with fork5. Then we can tell ByRows that position C3 is a fork, but we can't say which particular fork it is. Also, if we merely know in ByRows that position C3 is not any spoon, we can pass that information directly to ByCols.

We also need this utility procedure for subset checking.

```
> `&subset`:= proc(A,B)
local i;
for i in A do if not member(i,B) then return false fi od;
```

```
true
end proc:

> Interpret:= proc(M)
local h, Type, Q, C, Nots,i;
C:= [];
# Note that 'items' is identical, both internally and externally, in both problems.
# Also that holes_BR and holes_BC are equal as sets.
use `&?`= M:-`&?` in
for h in holes_BR do
Q:= &? h;
if Q::`=` then C:= [op(C), Distinct([h, NotItem[attributes(op([2,1], Q))]])]
else
Nots:= op(2, Q);
for Type in [knives, forks, spoons] do
if Type &subset Nots then C:= [op(C), Distinct([h, Type])] fi
od
fi
od
end use;
C
end proc:
```

Now we need a procedural constraint for ByRows that coordinates the exchange of information with ByCols.

We preserve the initial state of ByCols (that is, the state reached by its initial constraints) by making dummy guesses. Before every call to ByCols, we go back to this initial state. While it may at first seem wasteful to always go back to this initial state, note that a complete solution to the puzzle consists of 36^2 = 1296 bits of information. The initial state already has 1138 bits determined (as I will show you shortly).

If ByCols reaches a contradiction, we consider that a contradiction in ByRows also. Otherwise, we run Interpret on ByCols to pass constraints back to ByRows. This procedural constraint stays unsatisfied until a complete solution that is consistent in both problems is reached.

```
> Coordinate:= proc(This, That)
local changes;
# The "Interpretation" process is quite expensive. We only want to do it if there is nothing else that can be
# concluded in the Master under the current guess. Thus, we check "Changes".
use `&?`= This:-`&?` in changes:= &? Changes end use;
if changes = :-sav_changes then return false, false else :-sav_changes:= changes fi;
use `&?`= That[1]:-`&?`, Satisfy= That[1]:-Satisfy, GoBack= That[1]:-GoBack, Guess= That[1]:-Guess in
if &? CountGuesses = 1 then GoBack() fi;
Guess(Dummy(proc() false, true end, []));
Satisfy(Interpret(This));
evalb(&? CountGuesses = 0), This:-IsComplete(), `if`(&? Changes > 0, Interpret(That[1]), NULL)
```

```
end use;
end:
```

We just take the previous Constraint set, remove the Procedural constraints specified by the procedure "Cols", and add the above.

```
> RowConstraints:=
[#This guarantees that there are two of each item in each row and that they are ordered as specified above.
seq(op([Less([cat(it,1), cat(it,2), B1], Vholes)
,seq(Less([holes_BR[6*(i-1)], cat(it,2*i-1), cat(it,2*i), holes_BR[6*i+1]], Vholes), i= 2..5)
,Less([E6, cat(it,11), cat(it,12)], Vholes)
])
,it= [f,s,k]
)

#Clue 1
,NextTo(f1, k1, Vholes), NextTo(f2, k2, Vholes), Less(k1, f2, Vholes), Less(f1, k2, Vholes)
,Distinct([ {A1, A6}, knives])
,Proc(Clue1, [])

#Clue 2
,Rel(BR:-Equation, f3, f4, Vholes, [{_A+2, _A+3}, {_B-2, _B-3}])

#Clue 3
,Distinct([{C1,C6}, knives])
,Rel(BR:-DifferentBlock, C1, C6, Vitems, [forks, spoons])

#Clue 4
,Rel(BR:-Separated, s7, s8, Vholes, [2])
,Rel(BR:-Separated, s11, s12, Vholes, [2])

#Clue 5
,Distinct([{C1,d1}, spoons]), Rel(BR:-DifferentBlock, C1, d1, Vitems, [forks, knives])

,Proc(Coordinate, [BC])

,Dummy(proc() :-sav_changes:= 0; false, true end, [])
,Dummy(proc() :-frame:= 0; false, true end, [])
,Dummy(ShowIter, [])
,Dummy(proc() :-infolevel['sendmail']:= 0; false, true end, [])
]:
```

Here, we put the constraints that are easy to specify by columns. Note that constraints that are easy to specify either way are included in both lists. Those would be the constraints that refer merely to an item's type rather than its number.

```
> ColConstraints:=
[#This guarantees that there are two of each item in each column and that they are ordered as specified above.
seq(op([Less([cat(it,1), cat(it,2), A2], Vholes)
,seq(Less([holes_BC[6*(i-1)], cat(it,2*i-1), cat(it,2*i), holes_BC[6*i+1]], Vholes), i= 2..5)
,Less([F5, cat(it,11), cat(it,12)], Vholes)
])
,it= [f,s,k]
)

#Clue 1
,Distinct([{A1, A6}, knives])

#Clue 3
,Distinct([{C1,C6}, knives]), Rel(BC:-DifferentBlock, C1, C6, Vitems, [forks, spoons])

#Clue 5
,Distinct([{C1,d1}, spoons]), Rel(BC:-DifferentBlock, C1, d1, Vitems, [forks, knives])

#Clue 6
,Rel(BC:-Equation, k3, k4, Vholes, [{_A+2}, {_B-2}])
,Rel(BC:-Equation, k11, k12, Vholes, [{_A+2}, {_B-2}])

#Clue 7a
,Succ(k8, k7, Vholes)

#Clue 7b
,Rel(BC:-Separated, f7, f8, Vholes, [2])

#Clue 8
,Rel(BC:-Equation, k9, k10, Vholes, [{_A+1, _A+4}, {_B-1, _B-4}])
]:
```

Note how much easier it has been to specify the problem in this form.

```
> infolevel[all]:= 0:
```

```
> BR:-Reinitialize(): BC:-Reinitialize():
```

```
> st:= time():
```

Generate the "initial state" for ByCols.

```
> BC:-Satisfy(ColConstraints):
```

That initial state contains a significant amount of information:

```
> use BC in &? Changes end;
```

Interpret the initial state and pass it to the main problem.

```
> PrintSoln(BR:-Satisfy([op(RowConstraints), op(Interpret(BC))]));
```

$$, \left[\begin{array}{cccccc} \_ & \_ & \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ & \_ & \_ \end{array}\right], \ Depth = 0$$

$$, \left[\begin{array}{cccccc} \_ & \_ & \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ & \_ & \_ \\ F & \_ & \_ & \_ & \_ & S \\ K & \_ & \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ & \_ & \_ \end{array}\right], \ Depth = 0$$

$$, \left[\begin{array}{cccccc} \_ & \_ & \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ & \_ & \_ \\ F & \_ & \_ & \_ & \_ & S \\ K & S & \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ & \_ & \_ \end{array}\right], \; Depth = 1$$

$$, \left[\begin{array}{cccccc} \_ & \_ & \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ & \_ & \_ \\ F & \_ & \_ & \_ & \_ & S \\ K & S & \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ & \_ & \_ \end{array}\right], \; Depth = 1$$

$$, \left[\begin{array}{cccccc} \_ & \_ & \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ & \_ & \_ \\ F & \_ & \_ & \_ & \_ & S \\ K & S & \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ & \_ & \_ \end{array}\right], \; Depth = 1$$

$$, \left[\begin{array}{cccccc} \_ & \_ & \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ & \_ & \_ \\ F & \_ & \_ & \_ & \_ & S \\ K & S & \_ & \_ & \_ & S \\ \_ & \_ & \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ & \_ & \_ \end{array}\right], \; Depth = 2$$

$$, \left[\begin{array}{cccccc} \_ & \_ & \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ & \_ & \_ \\ F & \_ & \_ & \_ & \_ & S \\ K & S & \_ & \_ & S & \_ \\ \_ & \_ & \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ & \_ & \_ \end{array}\right], \; Depth = 1$$

$$, \left[\begin{array}{cccccc} \_ & \_ & \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ & \_ & \_ \\ F & \_ & \_ & \_ & \_ & S \\ K & S & F & \_ & S & \_ \\ \_ & \_ & \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ & \_ & \_ \end{array}\right], \; Depth = 2$$

$$,\left[\begin{array}{cccccc} \_ & \_ & \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ & \_ & \_ \\ F & \_ & \_ & \_ & \_ & S \\ K & S & F & K & S & F \\ \_ & \_ & \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ & \_ & \_ \end{array}\right], Depth = 3$$

$$,\left[\begin{array}{cccccc} \_ & \_ & \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ & \_ & \_ \\ F & \_ & \_ & \_ & \_ & S \\ K & S & F & F & S & K \\ \_ & \_ & \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ & \_ & \_ \end{array}\right], Depth = 2$$

$$,\left[\begin{array}{cccccc} \_ & \_ & \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ & \_ & \_ \\ F & \_ & \_ & \_ & \_ & S \\ K & S & F & F & S & K \\ \_ & \_ & \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ & \_ & \_ \end{array}\right], Depth = 2$$

$$, \left[\begin{array}{cccccc} \_ & \_ & \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ & \_ & \_ \\ F & \_ & \_ & \_ & \_ & S \\ K & S & F & F & S & K \\ \_ & \_ & \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ & \_ & \_ \end{array}\right], \mathit{Depth} = 2$$

$$, \left[\begin{array}{cccccc} \_ & \_ & \_ & F & \_ & S \\ \_ & \_ & \_ & \_ & \_ & \_ \\ F & \_ & \_ & \_ & \_ & S \\ K & S & F & F & S & K \\ \_ & \_ & \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ & \_ & \_ \end{array}\right], \mathit{Depth} = 3$$

$$, \left[\begin{array}{cccccc} \_ & \_ & K & F & K & S \\ \_ & \_ & \_ & \_ & \_ & \_ \\ F & \_ & \_ & \_ & \_ & S \\ K & S & F & F & S & K \\ \_ & \_ & \_ & \_ & \_ & F \\ \_ & \_ & \_ & \_ & \_ & \_ \end{array}\right], \mathit{Depth} = 3$$

$$, \begin{bmatrix} S & \_ & \_ & F & \_ & F \\ \_ & \_ & \_ & \_ & \_ & \_ \\ F & \_ & \_ & \_ & \_ & S \\ K & S & F & F & S & K \\ \_ & \_ & \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ & \_ & \_ \end{bmatrix}, \; Depth = 2$$

$$, \begin{bmatrix} S & S & K & F & K & F \\ \_ & \_ & \_ & \_ & \_ & \_ \\ F & \_ & \_ & \_ & \_ & S \\ K & S & F & F & S & K \\ \_ & \_ & \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ & \_ & \_ \end{bmatrix}, \; Depth = 2$$

$$, \begin{bmatrix} S & S & K & F & K & F \\ \_ & F & \_ & \_ & \_ & \_ \\ F & K & \_ & \_ & \_ & S \\ K & S & F & F & S & K \\ \_ & \_ & \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ & \_ & \_ \end{bmatrix}, \; Depth = 2$$

$$, \begin{bmatrix} S & S & K & F & K & F \\ \_ & F & \_ & \_ & F & \_ \\ F & K & \_ & \_ & \_ & S \\ K & S & F & F & S & K \\ \_ & \_ & \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ & \_ & \_ \end{bmatrix}, Depth = 2$$

$$, \begin{bmatrix} S & S & K & F & K & F \\ \_ & F & \_ & \_ & F & \_ \\ F & K & \_ & \_ & \_ & S \\ K & S & F & F & S & K \\ \_ & K & \_ & \_ & K & \_ \\ \_ & \_ & \_ & \_ & \_ & \_ \end{bmatrix}, Depth = 2$$

$$, \begin{bmatrix} S & S & K & F & K & F \\ \_ & F & \_ & \_ & F & \_ \\ F & K & \_ & K & \_ & S \\ K & S & F & F & S & K \\ \_ & K & \_ & \_ & K & \_ \\ \_ & \_ & \_ & S & \_ & \_ \end{bmatrix}, Depth = 2$$

$$, \begin{bmatrix} S & S & K & F & K & F \\ \_ & F & \_ & \_ & F & \_ \\ F & K & \_ & K & \_ & S \\ K & S & F & F & S & K \\ \_ & K & \_ & \_ & K & \_ \\ S & \_ & K & S & \_ & K \end{bmatrix}, \mathit{Depth} = 2$$

$$, \begin{bmatrix} S & S & K & F & K & F \\ \_ & F & \_ & \_ & F & \_ \\ F & K & \_ & K & \_ & S \\ K & S & F & F & S & K \\ \_ & K & \_ & \_ & K & \_ \\ S & F & K & S & F & K \end{bmatrix}, \mathit{Depth} = 2$$

$$, \begin{bmatrix} S & S & K & F & K & F \\ K & F & S & K & F & S \\ F & K & F & K & S & S \\ K & S & F & F & S & K \\ F & K & \_ & \_ & K & F \\ S & F & K & S & F & K \end{bmatrix}, \mathit{Depth} = 2$$

$$, \begin{bmatrix} S & S & K & F & K & F \\ K & F & S & K & F & S \\ F & K & F & K & S & S \\ K & S & F & F & S & K \\ F & K & S & S & K & F \\ S & F & K & S & F & K \end{bmatrix}, \mathit{Depth} = 2$$

$$\mathit{Unique\ solution:}$$

$$\begin{bmatrix} S & S & K & F & K & F \\ K & F & S & K & F & S \\ F & K & F & K & S & S \\ K & S & F & F & S & K \\ F & K & S & S & K & F \\ S & F & K & S & F & K \end{bmatrix}$$

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

$$7.852$$

```
> use BC in &? ShowStats end;
```

$$\mathit{Sets} = 0, \mathit{Unsets} = 1818, \mathit{RuleOuts} = 414, \mathit{Elims} = 3952, \mathit{Transfers} = 0, \mathit{Pivots} = 0, \mathit{MaxLevel} = 319, \mathit{Guesses} = 19, \mathit{MaxDepth} = 1$$

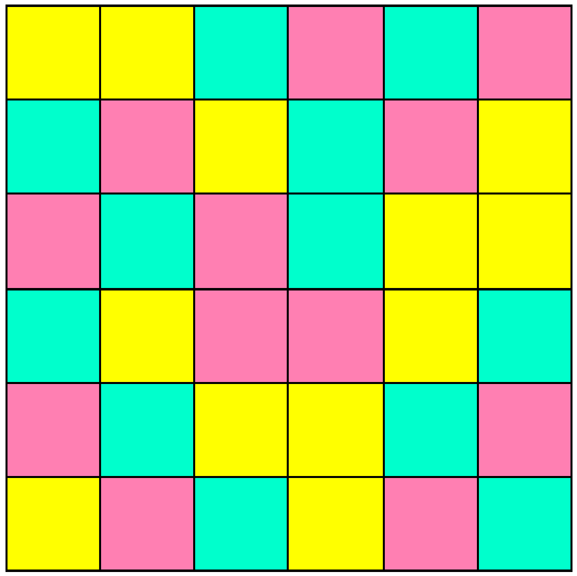
```
> use BR in &? ShowStats end;
```

$$\mathit{Sets} = 7, \mathit{Unsets} = 1212, \mathit{RuleOuts} = 120, \mathit{Elims} = 2512, \mathit{Transfers} = 0, \mathit{Pivots} = 0, \mathit{MaxLevel} = 200, \mathit{Guesses} = 5, \mathit{MaxDepth} = 3$$

Note how few guesses are made in the master problem, and how shallow the search space is!!! Note how much faster this technique is!!!

There are never any guesses made in ByCols. The "guesses" that are reported above are merely the dummy guesses discussed earlier that are made to preserve its initial state.

```
> AnimateSoln(Frames);
```



Play the animation at a very slow frame rate, and notice how much more direct this solution is. There is very little backtracking. The program is for the most part taking the same steps that an expert human puzzle solver would.

The above technique of two interacting copies of the problem can be generalized to whole classes of puzzles. Please proceed to the worksheet PaintByNumbers for an example.

```
>
```