Lecture 27 Backtracking

The matrix map below indicates countries that share boundaries: 1 if they share a boundary, 0 if not.

<table>
<thead>
<tr>
<th></th>
<th>(1)</th>
<th>(2)</th>
<th>(3)</th>
<th>(4)</th>
<th>(5)</th>
<th>(6)</th>
<th>(7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

We want to find, if possible, a 3-coloring of this map. In the tree for this problem, each of the the countries 1, 2, ..., 7 has its own level. For each country, there are three colors to try, r, g, b for red, green, blue. Each path from the root through the tree determines a coloring of the map. We have to find a path which colors the map so that countries (provinces in this case) which share a boundary get different colors.
Suppose \( \text{map} \) is a matrix such that for any countries \( i,j \), \( \text{map}(i,j) = 1 \) if countries \( i,j \) share a boundary and 0 if not. Write a recursive subroutine \text{find\_coloring} \) which uses backtracking to find a 3-coloring (red, green blue) of a map or prints “There is no 3-coloring of this map” if there is none. Assume by recursion that the subroutine can solve all problems with uncolored countries.

recursive subroutine \text{extend\_coloring}(\text{coloring, colorable})
!n=number of countries, uncolored = first uncolored country
!colorable is true if all countries can be colored without color clashes.
implicit none
integer::color, uncolored, coloring(n), mloc(1)
logical::colorable
... 3 lines to set colorable if we have a color clash.
... 3 lines to set colorable and return if everything is colored
mloc=minloc(\text{coloring}); uncolored=mloc(1)
!uncolored = first uncolored country
do color=1,3
... 3 lines to color the uncolored country, return if can be extended
enddo
end subroutine

logical function \text{color\_clash}(\text{coloring})
integer::color, coloring(n)
\text{color\_clash}=.false.
do i=1,n; do j=1,n
if(i==j .or. coloring(i)==0 .or. coloring(j)==0) cycle
... 3 lines to check if we have a color clash.
enddo; enddo
end function

end program
**Homework 27.1(7) mapcolor2.f95**

**email:** dale@math.hawaii.edu  **subject line:** 190 h27.1(7)

Modify the mapcolor program of **Classwork 27.1(6)** mapcoloring to search for a 3-coloring of the provinces of Canada (for simplicity the maritime provinces Nova Scotia, Prince Edward Island are omitted). There are 10 provinces whose shared boundaries must be encoded with the matrix **map**.

---

**Sudoku**

A **Sudoku** board is a $9 \times 9$ matrix whose entries are digits 1, 2, 3, ..., 9 or blanks (which we will represent with 0). The matrix is divided into 9 $3 \times 3$ blocks. A board is **fully-filled**, there are no blanks (no 0's). A board is **legally** filled if, no row has duplicates (no repeated entries), no column has duplicates, and none of the $9 \times 3$ block has duplicates. Some entries may be filled. A board is **solved** if it is fully and legally filled. It is **solvable** if it can be extended to a solved board. Given an initial solvable partially-filled board, the goal is to extend it (i.e., fill in the blanks) to a solved board.

We solve this with a backtracking strategy. For each blank position we try all possible legal values (values 1, 2, ..., 9 which don’t produce duplicates). If none works, we backtrack to an earlier position and try an alternate value.

http://www.nytimes.com/crosswords/game/sudoku/easy?page=sudoku&difficulty=easy&_r=0

http://www.sudokukingdom.com/
Write function `is_legal(b)` which determines if the board value `b` is legal (0’s represent unfilled blanks). Write a recursive subroutine `extend_board(b, solvable)` which given a board `b` will set the truth value `solvable` to `.true.` if `b` is solvable iff it can be extended to a solved board. We assume by recursion that the subroutine can solve all problems with fewer blanks.

```fortran
subroutine extend_board(b, solvable)
  integer::b(9,9)
  logical:: solvable
  !c27_2_6sudoku.f95
  subroutine sudoku_printer(b)
    integer::b(9,9)
    character(*),parameter::dashes='----- ----- ----- -----
    print*, dashes
    do j=1,9,3
      do i=j,j+2; print 10,b(i, :); enddo
      print *,dashes
    enddo
  ends subroutine
  program sudoku
    integer::b(9,9)
    logical:: solvable
    b(1,:)=(/0,0,3,0,2,0,6,0,0/)
    b(2,:)=(/9,0,0,3,0,5,0,0,1/)
    b(3,:)=(/0,0,1,8,0,6,4,0,0/)
    b(4,:)=(/0,0,8,1,0,2,9,0,0/)
    b(5,:)=(/7,0,0,0,0,0,0,0,8/)
    b(6,:)=(/0,0,6,7,0,8,2,0,0/)
    b(7,:)=(/0,0,2,6,0,9,5,0,0/)
    b(8,:)=(/8,0,0,2,0,3,0,0,9/)
    b(9,:)=(/0,0,5,0,1,0,3,0,0/)
    call sudoku_printer(b)
    print*
    call extend_board(b, solvable)
    if(.not. solvable) then;
      print*,"No solution possible."
    else
      call sudoku_printer(b)
    endif
  endprogram
```

<table>
<thead>
<tr>
<th>5</th>
<th>3</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1</td>
<td>9 5</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>2 8</td>
<td></td>
</tr>
<tr>
<td>4 1 9 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>7 9</td>
<td></td>
</tr>
</tbody>
</table>

```
<table>
<thead>
<tr>
<th>5 3 4</th>
<th>6 7 8 9 1 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 7 2 1 9 5 3 4 8</td>
<td></td>
</tr>
<tr>
<td>1 9 8 3 4 2 5 6 7</td>
<td></td>
</tr>
<tr>
<td>8 5 9 7 6 1 4 2 3</td>
<td></td>
</tr>
<tr>
<td>4 2 6 8 5 3 7 9 1</td>
<td></td>
</tr>
<tr>
<td>7 1 3 9 2 4 8 5 6</td>
<td></td>
</tr>
<tr>
<td>9 6 1 5 3 7 2 8 4</td>
<td></td>
</tr>
<tr>
<td>2 8 7 4 1 9 6 3 5</td>
<td></td>
</tr>
<tr>
<td>3 4 5 2 8 6 1 7 9</td>
<td></td>
</tr>
</tbody>
</table>
```
recursive subroutine extend_board(b,solvable)
! solvable=true if b can be extended to a full legal solution
integer::b(9,9),mloc(2),i,j,k
logical::is_legal,solvable
... 3 lines to set solvable if board is not legal.
... 3 lines to set solvable and return if everything is solvable
mloc=minloc(b); i=mloc(1); j=mloc(2);
! line above gets first blank location
do k=1,9
... 3 lines to fill b(i,j), return if can be extended
enddo
b(i,j)=0
solvable=.false.
endsubroutine

logical function is_legal(b)
integer::b(9,9)
is_legal=.true.
do i=1,9;do j=1,9;do i2=1,9;do j2=1,9
if(b(i,j)==0 .or. b(i2,j2)==0) cycle
if(b(i,j)/=b(i2,j2)) cycle
if(i==i2 .and. j==j2) cycle
if(i==i2 .or. j==j2) then;
is_legal=.false.; return;
endif
if((i-1)/3==(i2-1)/3 .and. (j-1)/3==(j2-1)/3) then
is_legal=.false.; return
endif
endo;endo;endo;endo
endfunction

Do one of 27.2B or 27.2A, not both Preferably B.

Homework 27.2B(5) sudoku_file.f95
email: dale@math.hawaii.edu subject line: 190 h27.2b(5)
Modify the Sudoku program above to read the matrix b from a file sudoku.txt whose nine lines are
0,0,3,0,2,0,6,0,0
9,0,0,3,0,5,0,0,1
0,0,1,8,0,6,4,0,0
0,0,8,1,0,2,9,0,0
7,0,0,0,0,0,0,0,8
0,0,6,7,0,8,2,0,0
0,0,2,6,0,9,5,0,0
8,0,0,2,0,3,0,0,9
0,0,5,0,1,0,3,0,0

Homework 27.2A(5) sudoku_keyboard.f95
email: dale@math.hawaii.edu subject line: 190 h27.2a(5)
Modify the Sudoku program above to read the matrix b from the keyboard. It asks the reader to “Enter 9 lines with 9 positive digits each”. It then gives the solution.

Probability modeling and simulation.
Recall that rand() gives randomly generated numbers in [0, 1).
To find the probability of an event X (say “heads”) when running a process P (say tossing a coin):
Run the process many times, say nruns= inf=10**6.
Count the number, numx, of times event X occurs.
Approximately, the probability of \( X = \frac{\text{numx}}{\text{nruns}} \).
But, instead, you must write $X = \text{real}(\text{numx})/nruns$
since Fortran divides reals, not integers.

Correct this program. Should get 0.5, not 0.

!real_prob.f95
program real_prob
integer:: numx=5,nruns=10; real:: prob
prob=numx/nruns
!don't divide integers, use real() to make numx a real.
print *, 'The probability = ', prob
endprogram