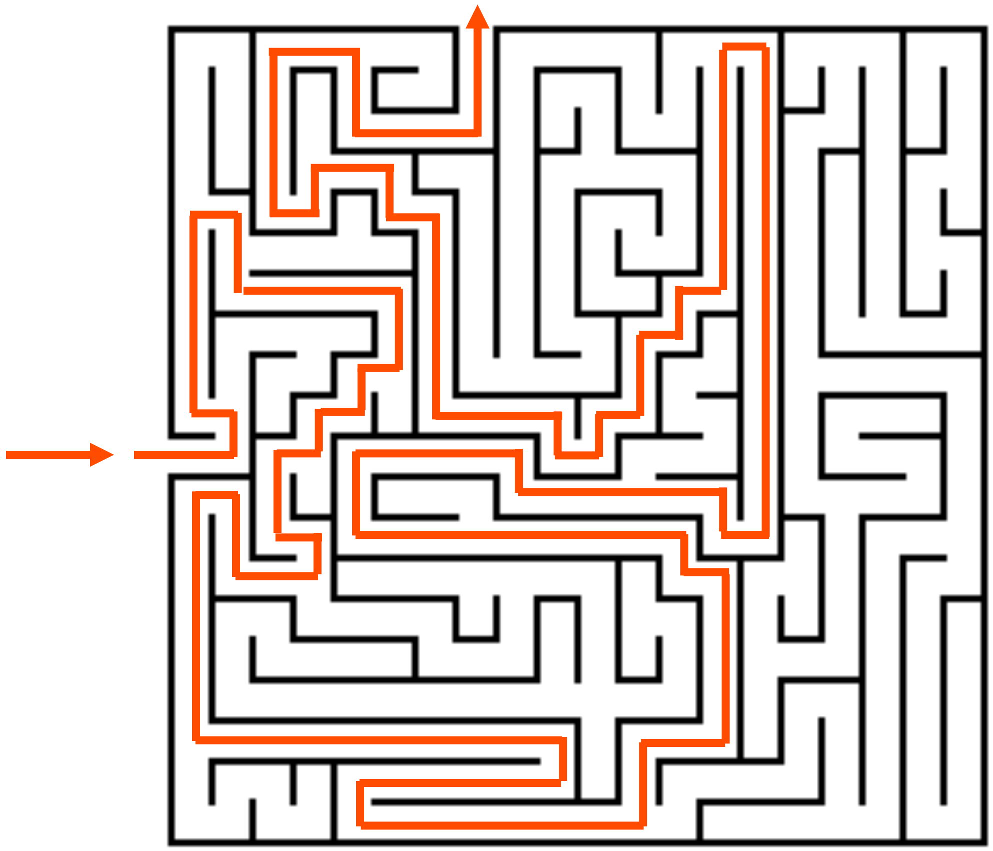
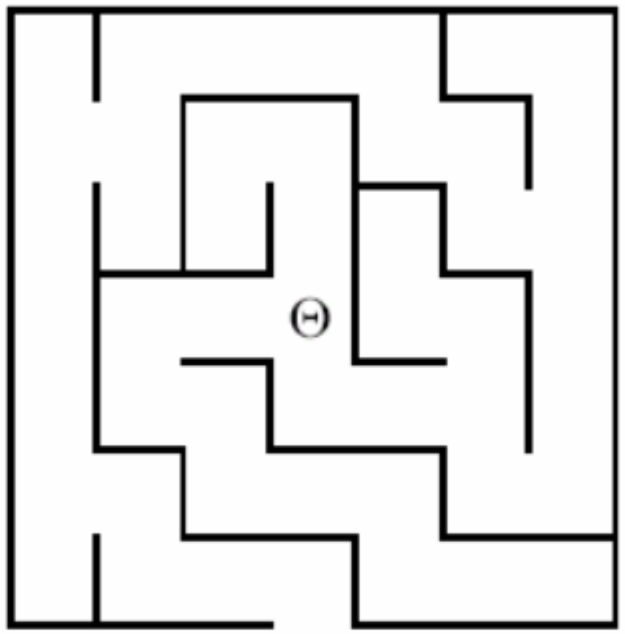
Backtracking

Labirinth



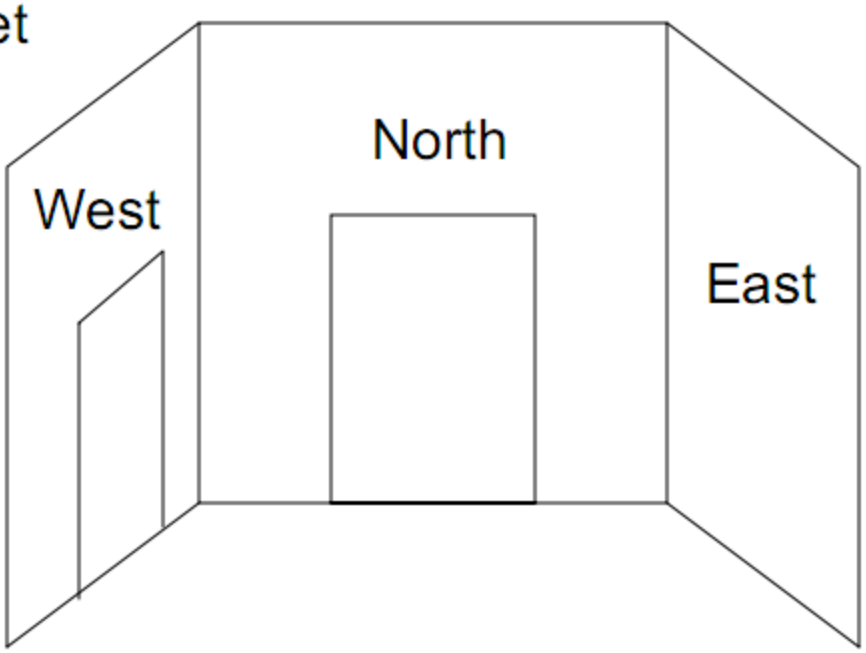
A Simple Maze



Search maze until way out is found. If no way out possible report that.

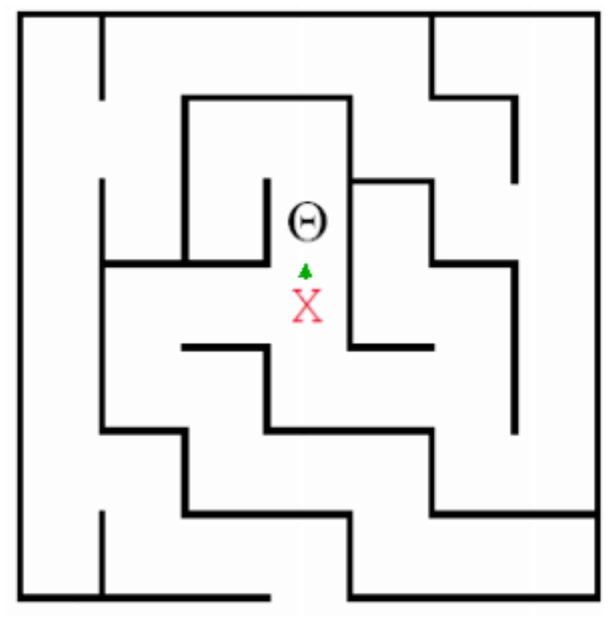
The Local View

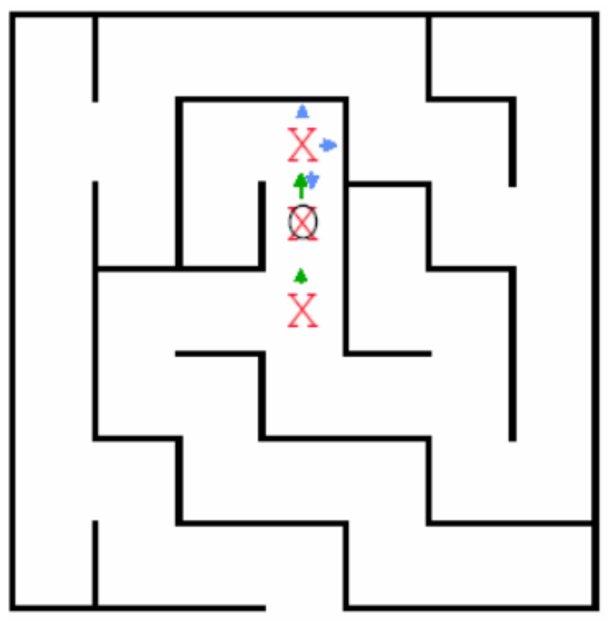
Which way do
I go to get
out?

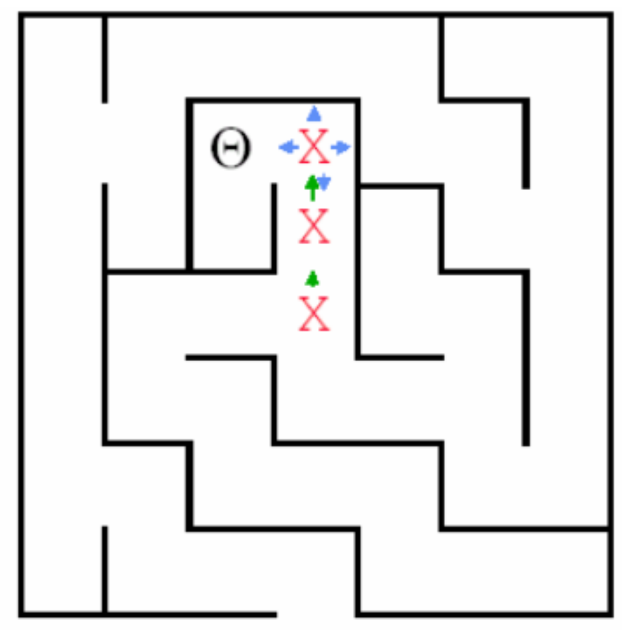


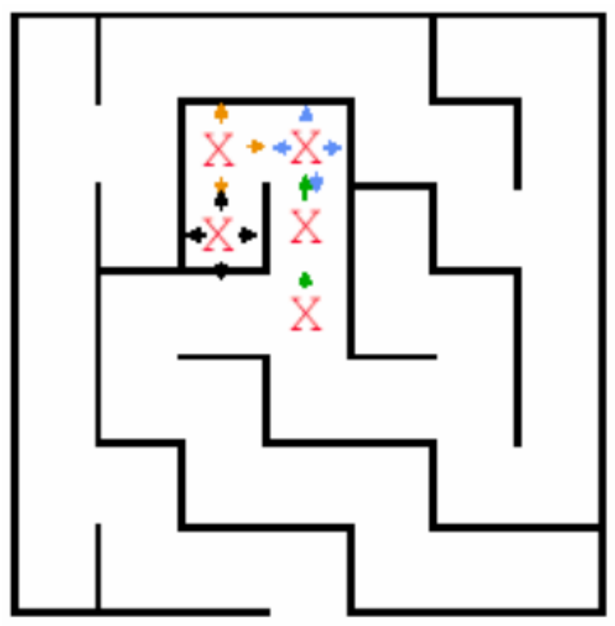
Algorithm for Maze

```
If the current square is outside, return TRUE to indicate that a solution has been
        found.
If the current square is marked, return FALSE to indicate that this path has been
        tried.
Mark the current square.
for (each of the four compass directions)
{       if ( this direction is not blocked by a wall )
        {           Move one step in the indicated direction from the current square.
                    Try to solve the maze from there by making a recursive call.
                    If this call shows the maze to be solvable, return TRUE to indicate that
                            fact.
        }
}
Unmark the current square.
Return FALSE to indicate that none of the four directions led to a solution.
```





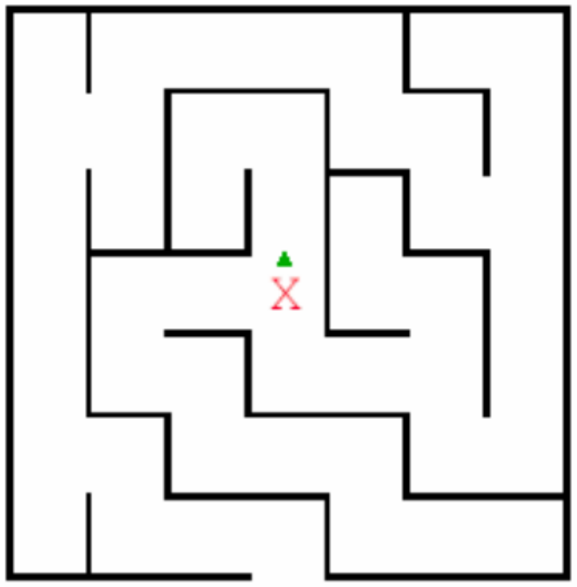




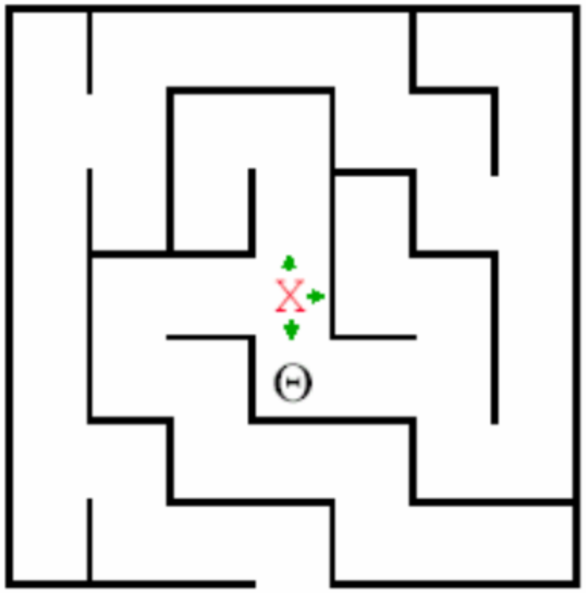
This path reaches a dead end.

Time to backtrack!

Remember the program stack!

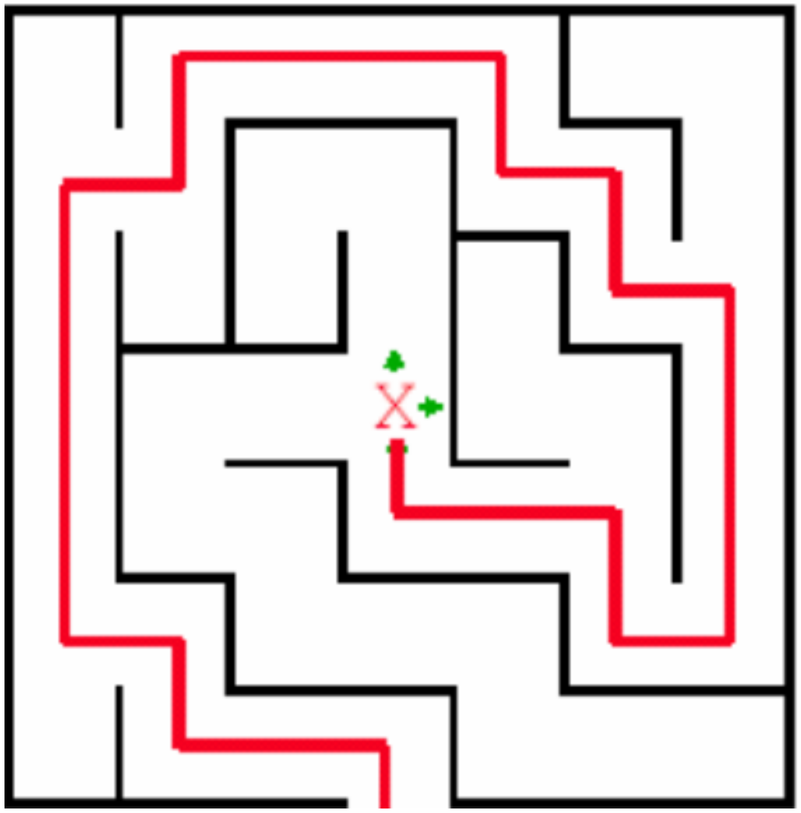


The recursive calls end and return until we find ourselves back here.



And now we try South

Path Eventually Found



Solving Sudoku – Brute Force

- A *brute force* algorithm is a simple but general approach
- Try all combinations until you find one that works
- This approach isn't clever, but computers are fast
- Then try and improve on the brute force resuts

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	1		7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

$\rightarrow$

5	3	1	2	7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

$\rightarrow$

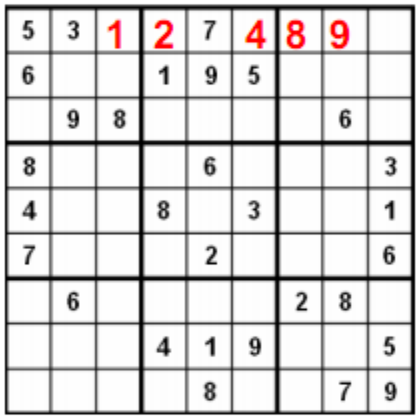
5	3	1	2	7	4			
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	1	2	7	4	8		
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	1	2	7	4	8	9	
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

uh oh!

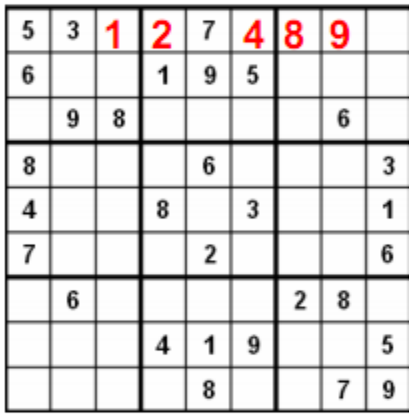
- We have reached a dead end in our search



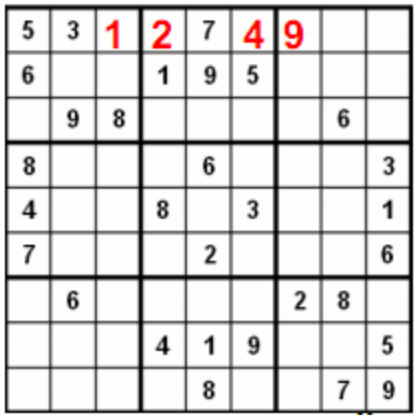
5	3	1	2	7	4	8	9	
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

- With the current set up none of the nine digits work in the top right corner

- When the search reaches a dead end in ***backs up*** to the previous cell it was trying to fill and goes onto to the next digit
- We would back up to the cell with a 9 and that turns out to be a dead end as well so we back up again
  - so the algorithm needs to remember what digit to try next
- Now in the cell with the 8. We try and 9 and move forward again.



5	3	1	2	7	4	8	9	
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9



5	3	1	2	7	4	9		
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

- Brute force algorithms are slow
- The don't employ a lot of logic
  - For example we know a 6 can't go in the last 3 columns of the first row, but the brute force algorithm will plow ahead any way
- But, brute force algorithms are fairly easy to implement as a first pass solution
  - backtracking is a form of a brute force algorithm

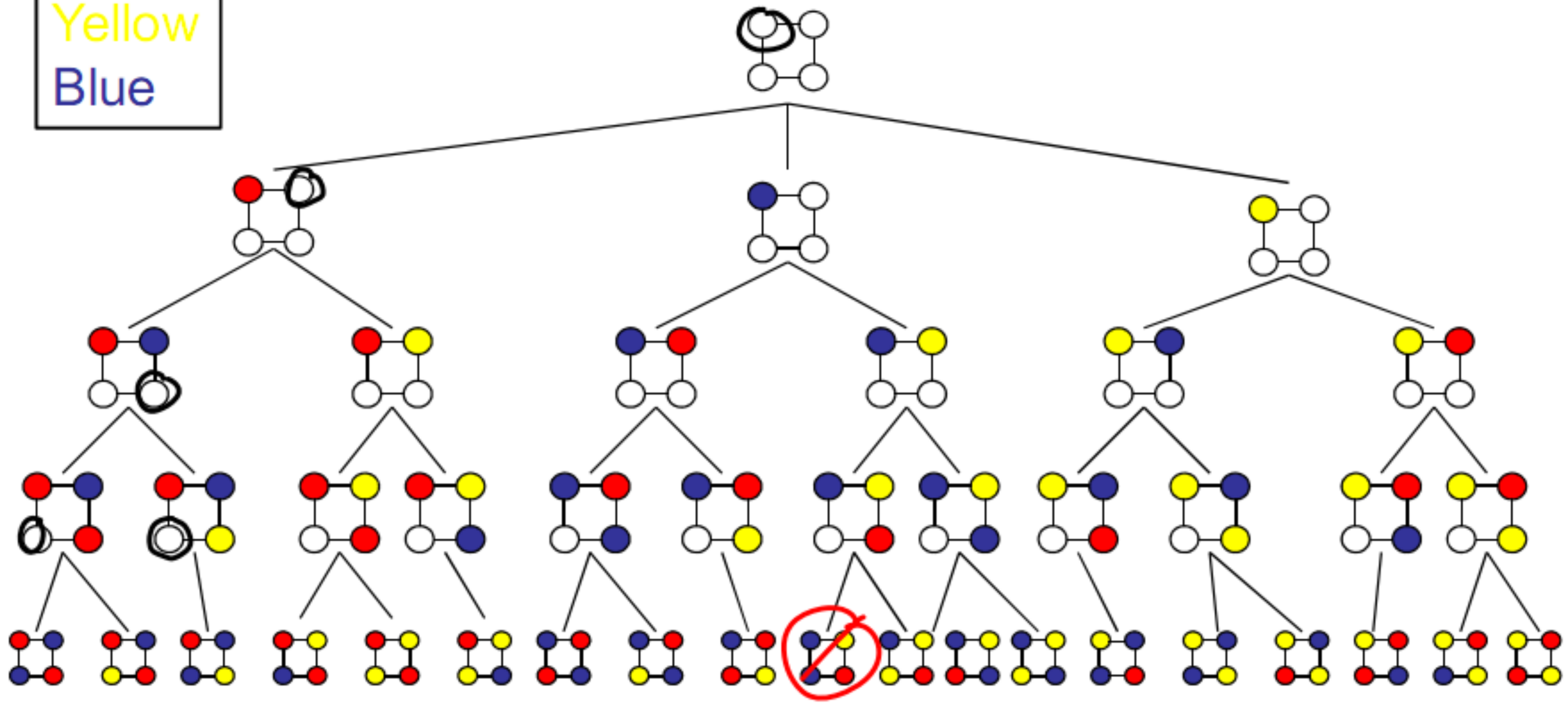
- After trying placing a digit in a cell we want to solve the new sudoku board
  - Isn't that a smaller (or simpler version) of the same problem we started with?!?!?!?
- After placing a number in a cell the we need to remember the next number to try in case things don't work out.
- We need to know if things worked out (found a solution) or they didn't, and if they didn't try the next number
- If we try all numbers and none of them work in our cell we need to *report back* that things didn't work

Recursive Backtracking

- Problems such as Suduko can be solved using recursive backtracking
- recursive because later versions of the problem are just slightly simpler versions of the original
- backtracking because we may have to try different alternatives

m=3
Red
Yellow
Blue

Example: Graph Coloring



- Enumerate all possible states; check for solutions at the leaves.
- Naively, non-solutions will be generated. More on that later.
- Note, most non-solutions **not** shown in this cartoon.

Aside: Coloring a Map

- Assign colors to countries so that no two bordering regions are the same color.
- Map coloring is *planar* graph coloring.

$$\begin{tikzcd}
 & \text{Idaho} \arrow[ddl, dash] \arrow[dd, dash] \arrow[dr, dash] & \\
 & & \text{Wyoming} \arrow[dl, dash] \arrow[d, dash] \\
\text{Nevada} \arrow[r, dash] \arrow[dr, dash] & \text{Utah} \arrow[r, dash] \arrow[d, dash] & \text{Colorado} \arrow[d, dash] \\
 & \text{Arizona} \arrow[r, dash] & \text{New Mexico}
\end{tikzcd}$$

Four Color Theorem

- How many colors do you need for a planar map?
  - Four.
- Haken and Appel using a computer program and 1,200 hours of run time in 1976. Checked 1,476 graphs (i.e., cases in the proof).
- First proposed in 1852.

Simple Recursive Backtracking Algorithm

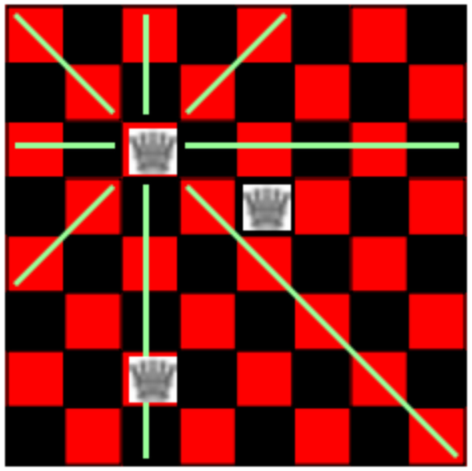
```
procedure explore(int k, state s)
Input: Depth in the search tree k; current state s
Output:  none; prints solutions

   for each s' ∈ extensions_of(s)   ← how to implement?
         if k==n and criterion(s') then
                  output "Solution:" s'
         // if possible solution not yet fully specified
         if (k < n) then
                  explore(k+1, s')
```

- Let’s take a step back from back-tracking for a moment.
- Consider *how large* the search space could be if we don’t approach the formulation of a problem carefully.

n-Queens Problem

- Given: n-queens and an nxn chess board
- Find: A way to place all n queens on the board s.t. no queens are attacking another queen.

- How could you formulate this problem?
- How would you represent a solution?



First Solution Idea

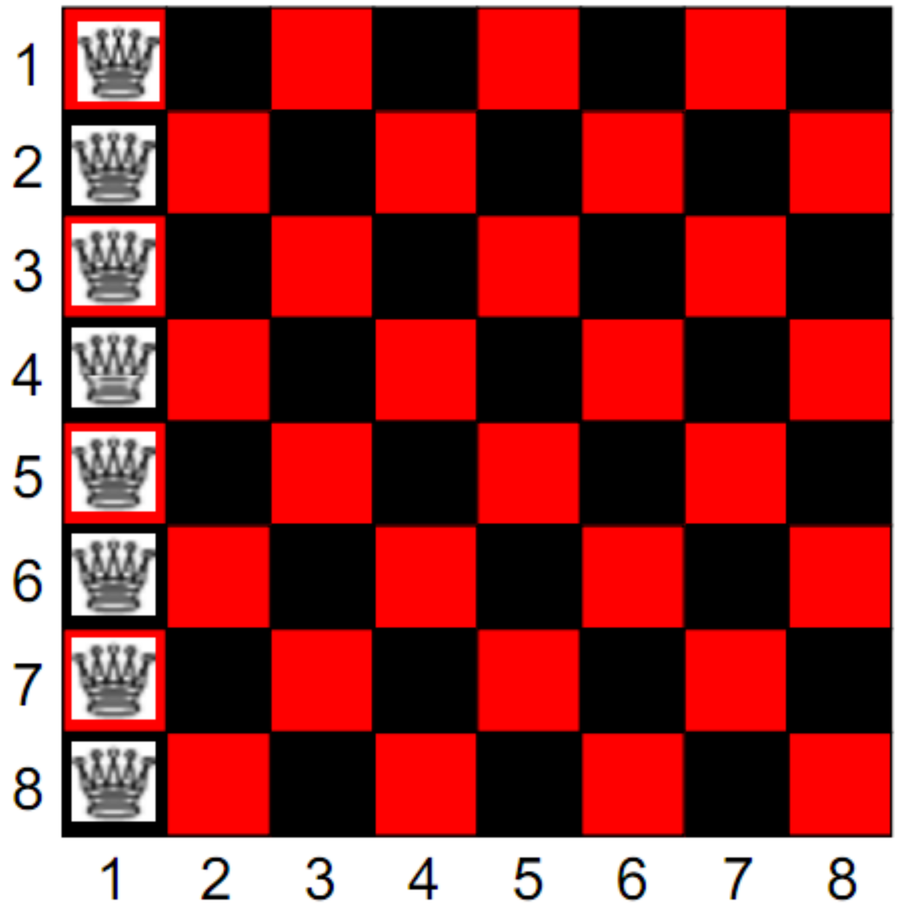
- Consider every placement of each queen 1 at a time.
  - How many placements are there?

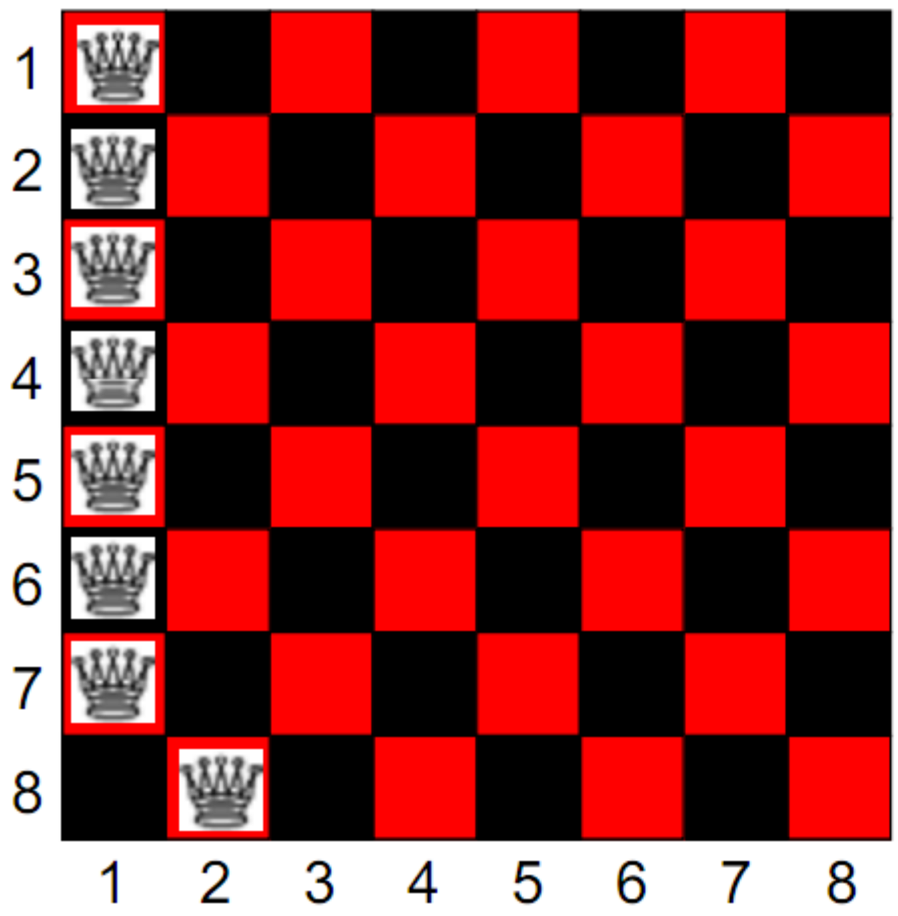
- Consider every placement of each queen 1 at a time.
  - How many placements are there?

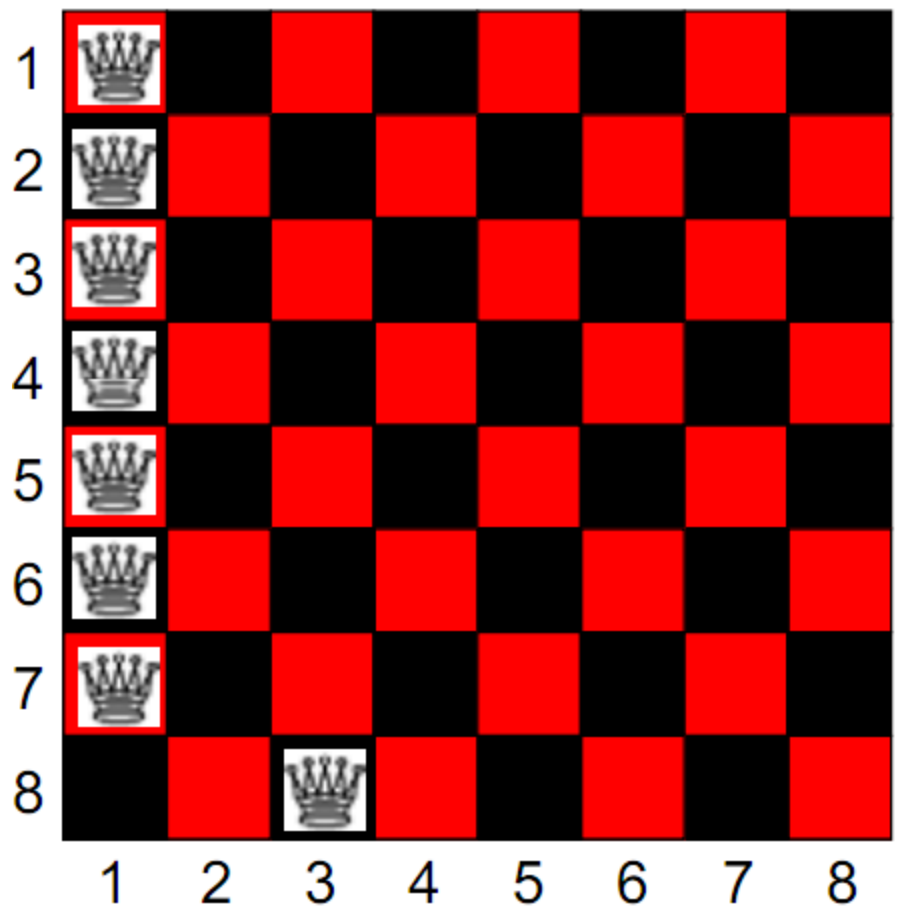
$$\binom{n^2}{n} \Rightarrow \binom{64}{8} = 4{,}426{,}165{,}368$$

Second Solution Idea

- Don’t place 2 queens in the same row.
  - Now how many positions must be checked?







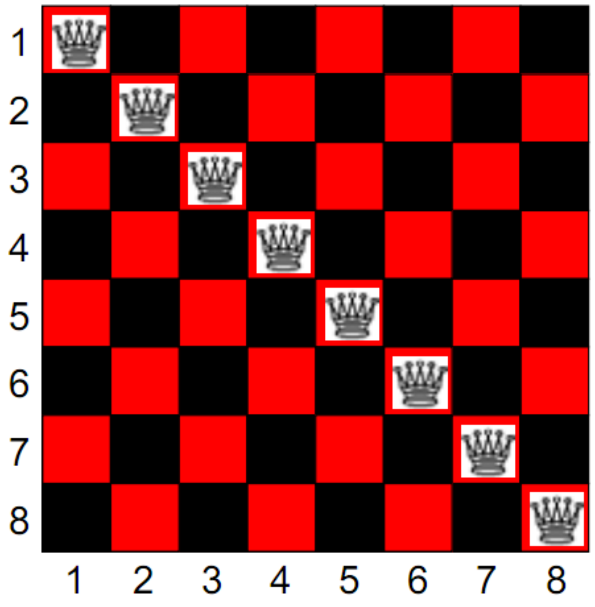
- Don’t place 2 queens in the same row.
  - Now how many positions must be checked?

Represent a positioning as a vector [x_1, …, x_8]
Where each element is an integer 1, …, 8.

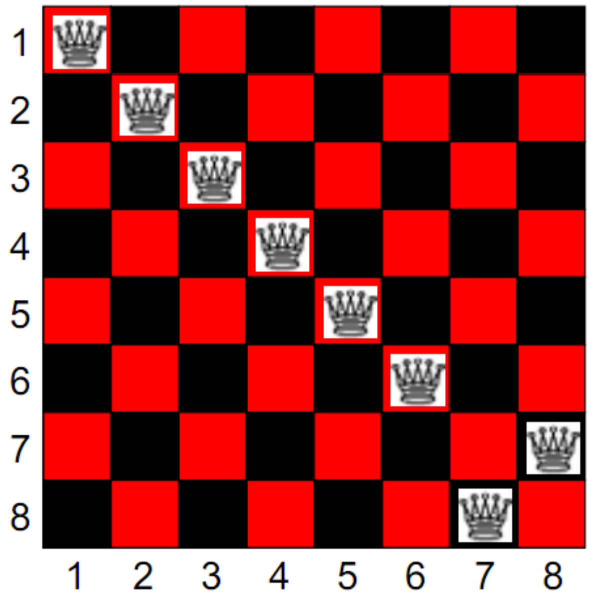
$$n^n = 8^8 = 16,777,216$$

Third Solution Idea

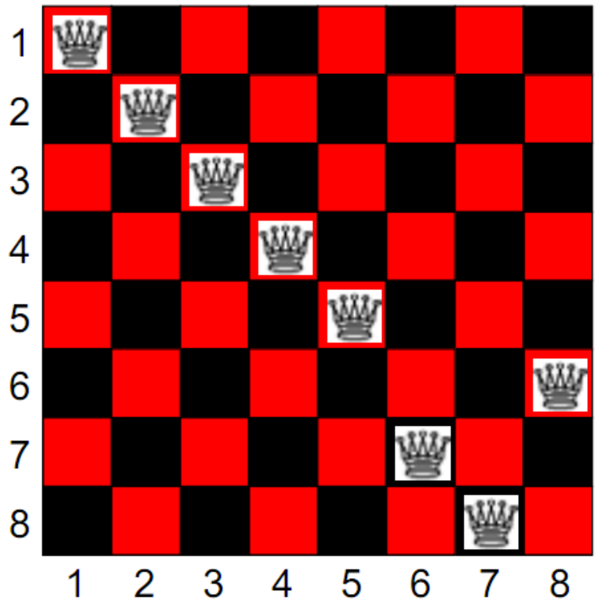
- Don't place 2 queens in the same row or in the same column.
- Generate all permutations of (1,2…8)
  - Now how many positions must be checked?



(1,2,3,4,5,6,7,8)



(1,2,3,4,5,6,8,7)



(1,2,3,4,5,8,6,7)

- Don’t place 2 queens in the same row or in the same column.
- Generate all permutations of (1,2…8)
  - Now how many positions must be checked?

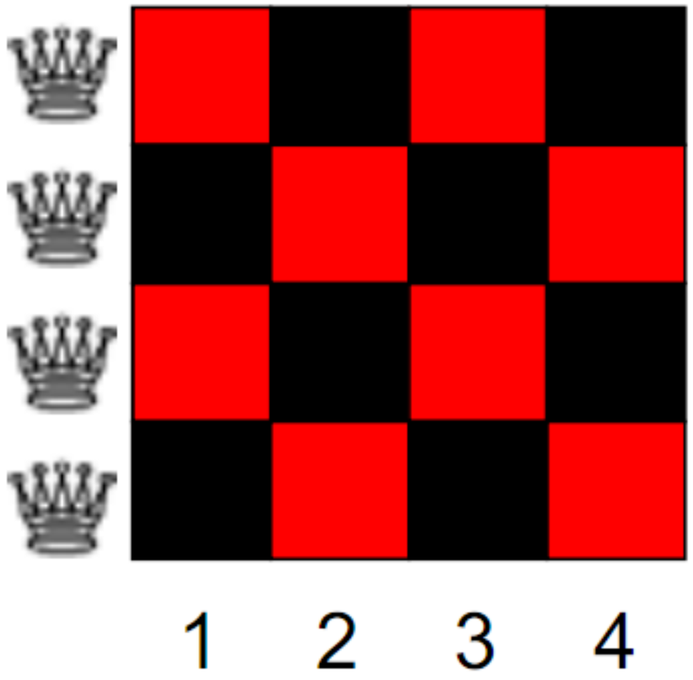
$$n! = 8! = 40,320$$

- We went from $C(n^2, n)$ to n^n to $n!$
  - And we’re happy about it!

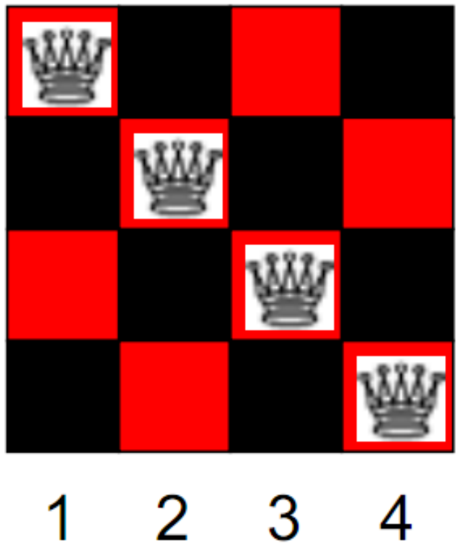
- We applied *explicit constraints* in the enumeration of successor states to shrink our search space.

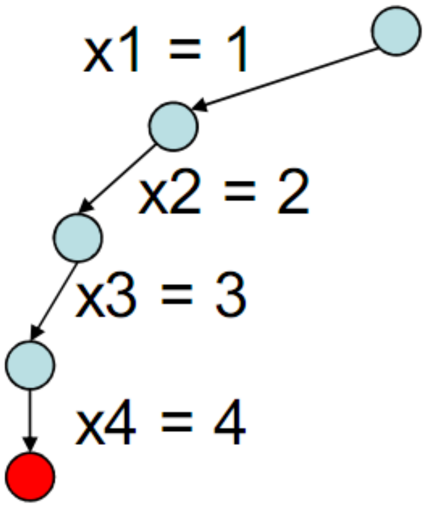
Toward More Lessons

- We've only been checking a solution *after* placing all 8 queens.
    - A **criterion function** (or solution test)
        - A binary test
- A better idea is to check after *each* placement of a queen.
    - A **partial-criterion** function (or feasibility test)
        - A binary test
        - Later: replace with "bounding function", a numerical goodness measure
    - Idea: Determine we are on a dead-end road without exploring the entire road

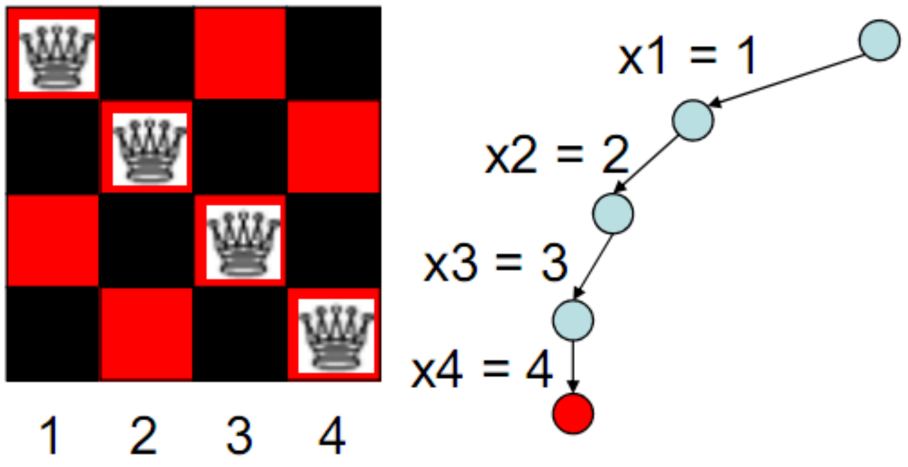


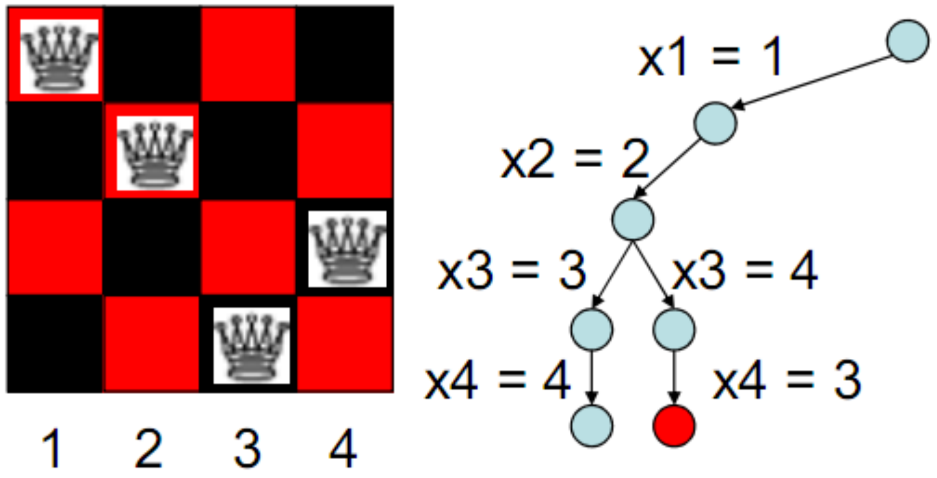
Easier to manage in a lecture.





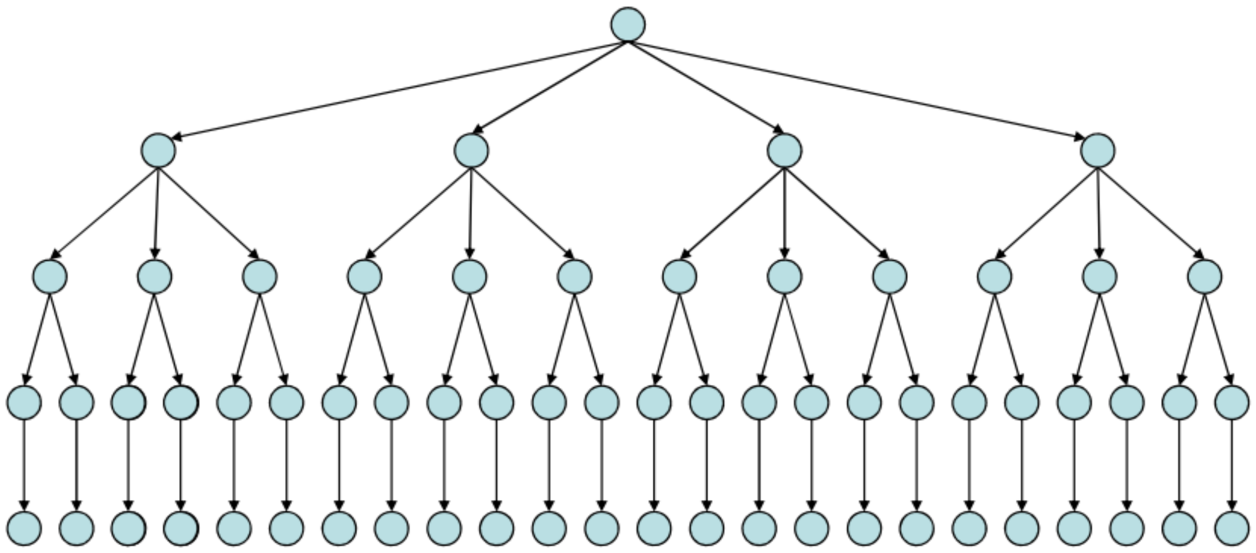
Each branch of the tree represents a decision to place a queen.
The criterion function can only be applied to leaf nodes.



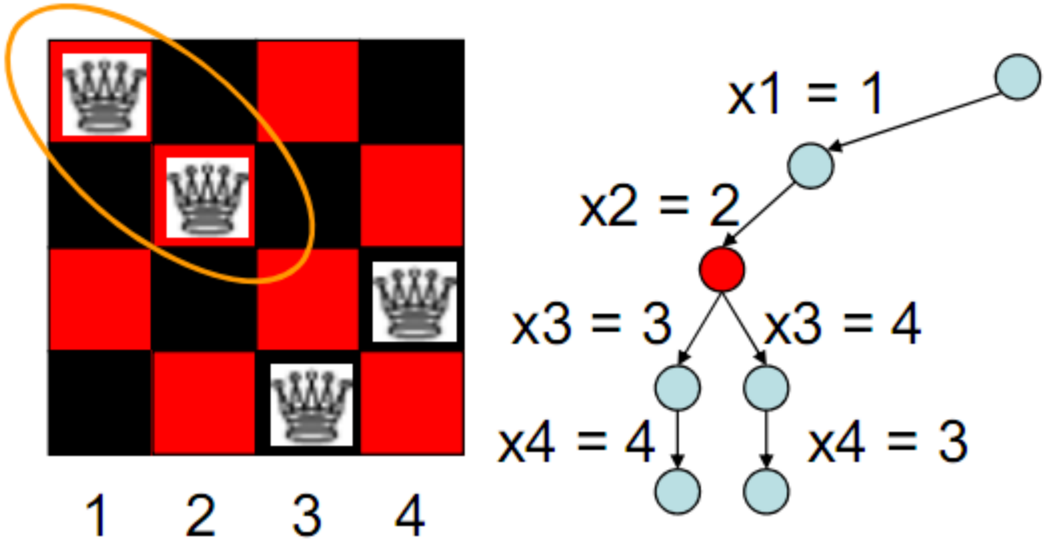


Is this leaf node a solution?

Is this leaf node a solution?

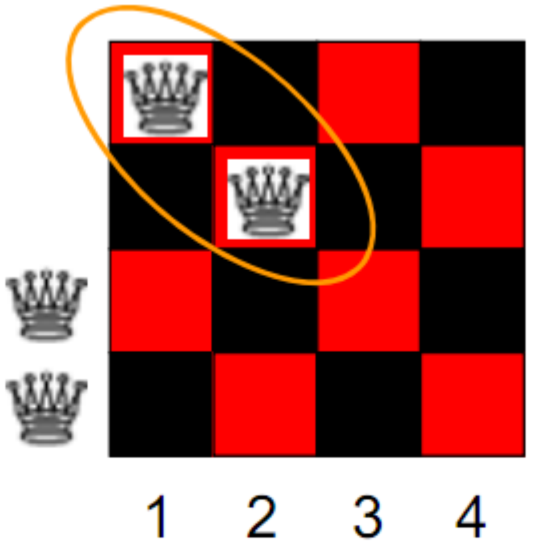


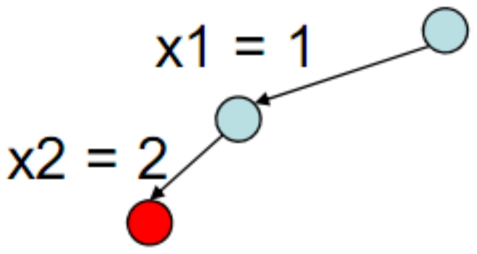
Using the criterion function, this is the search tree.



A better question:
Is any child of this node *ever* going to be a solution?

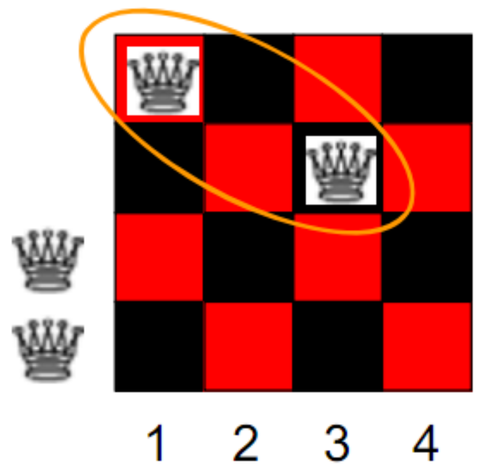
No. 1 is attacking 2. Adding queens won't fix the problem.

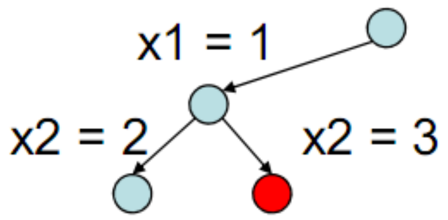




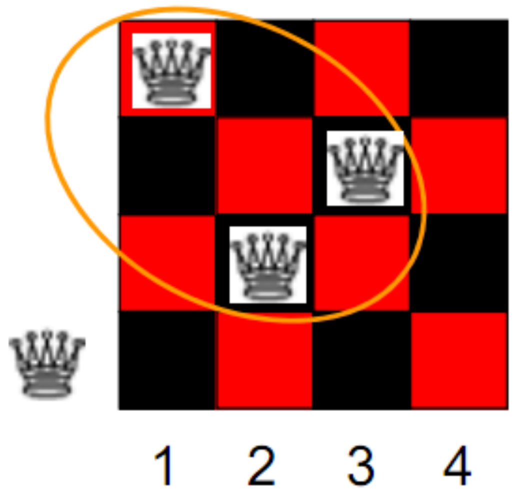
The partial criterion or feasibility function checks that a node may eventually have a solution.

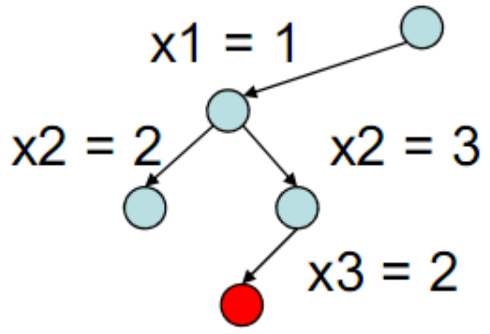
Will this node have a solution? no.



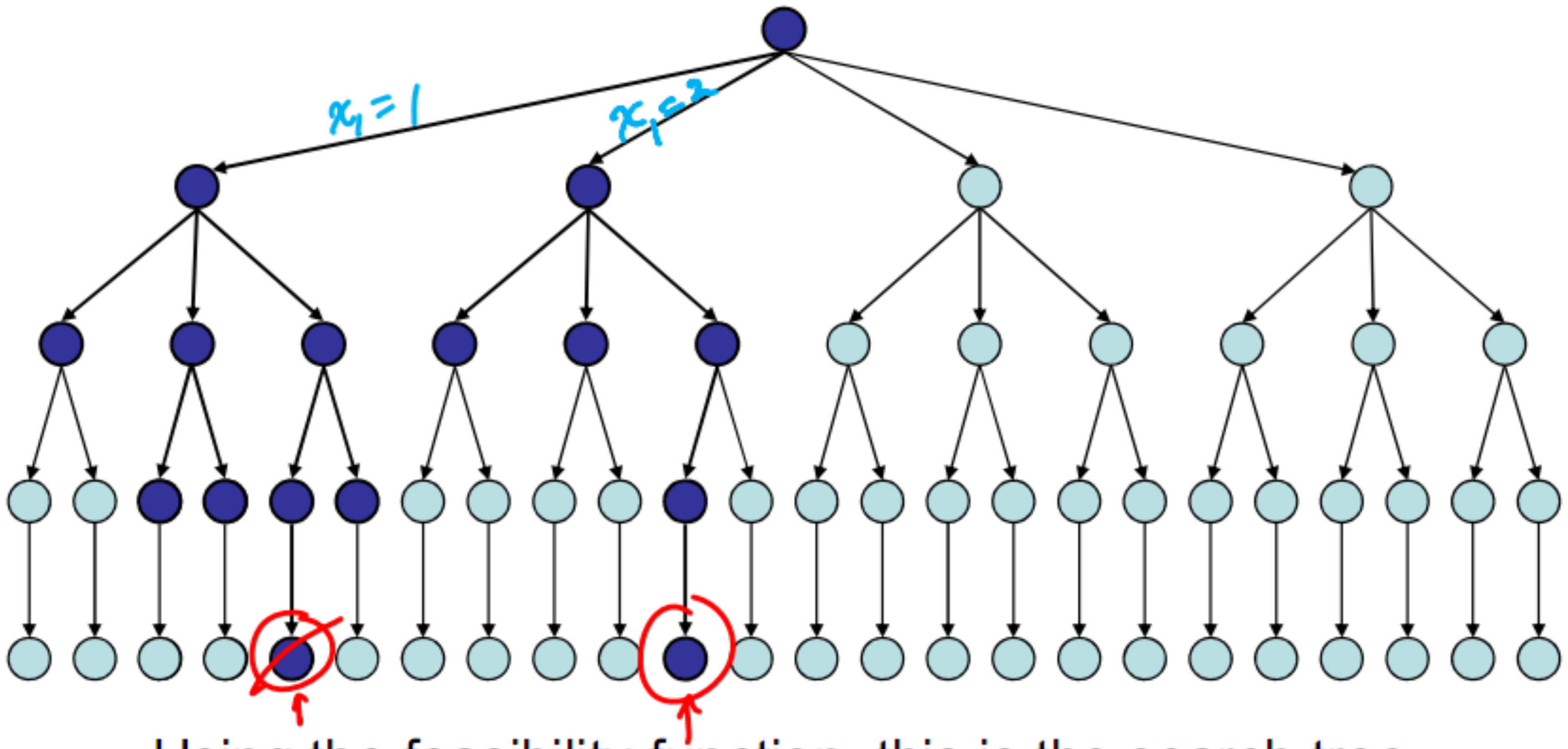


Will this node have a solution? Maybe.

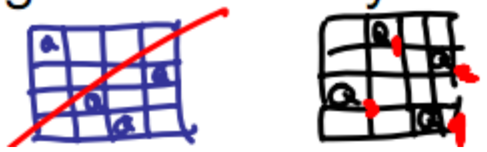




Will this node have a solution? No.
Etc.



Using the feasibility function, this is the search tree.



procedure explore(int k, state s)
Input: Depth in the search tree k; current state s
Output: Solution or {} (failure)

```
for each s' ∈ extensions_of(s)
    if partial_criterion(s') then
        if k==n and criterion(s') then
            output "Solution:" s'
        // if state not yet fully specified
        if (k < n) then
            explore(k+1, s')
```

outer-most call: explore(0, init_state())

Efficiency of Backtracking Algorithms depends on:

1. The time to generate the next possible states in the search tree
2. The number of next possible states satisfying the explicit constraints
3. The time to compute the partial criterion = feasibility
4. The proportion of next possible states satisfying the partial criterion function

- A feasibility function is considered good, if it substantially reduces the number of nodes that are generated.
- But, feasibility functions take more time to evaluate.