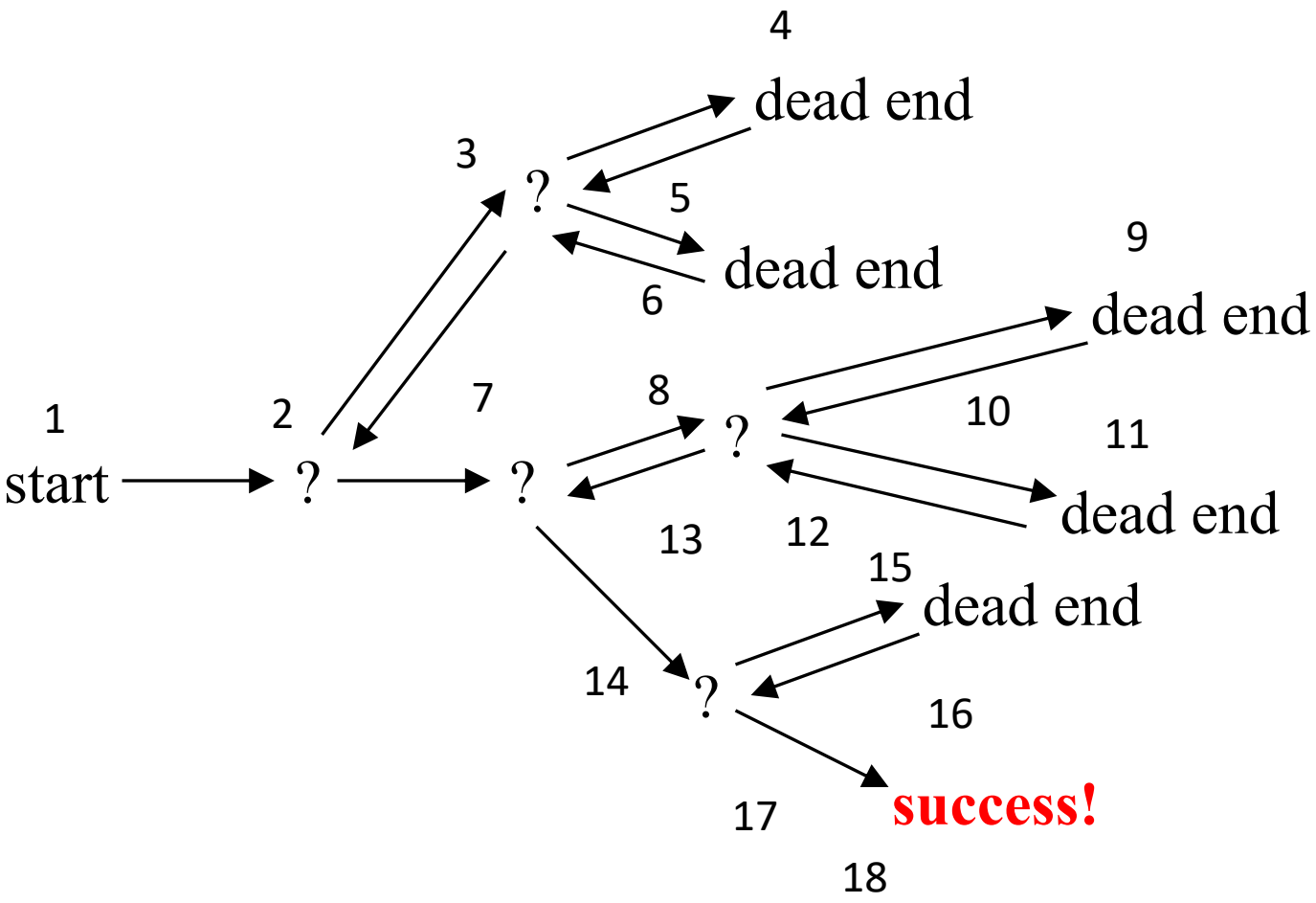
Backtracking Algorithms



Algorithm Design Techniques

- Divide and Conquer
  - Divide the problem into smaller subproblems, solve them, and combine into the overall solution
- Greedy
  - Choice is made based on locally optimal solution
- Dynamic Programming
  - Brute force through all possible solutions, storing solutions to subproblems to avoid repeat computation
- Backtracking
  - A smart form of exhaustive search
- …

Backtracking

- Suppose you have to make a series of *decisions,* among various *choices,* where
  - You don't have enough information to know what to choose
  - Each decision leads to a new set of choices
  - Some sequence of choices (possibly more than one) may be a solution to your problem

- Backtracking is a methodical way of trying out various sequences of decisions, until you find one that "works"

Backtracking

- **Backtracking** is basically a form of **recursion**. It is a general approach for **finding all solutions** to some computational problems called **constraint-satisfaction problems** without trying all possibilities.

- These problems are interesting because there are so many **candidate solutions**, the vast majority of which do not satisfy the given constraints.

- Why can't we just try them all, one-by-one, until finding the solution? Not really, because of so many possibilities!

- In the **N-Queens Problem**, there are $\binom{n^2}{n}$ ways to place n queens on an $n \times n$ board. For an 8×8 board there are 4,426,165,368 possible arrangements.
- For **Sudoku**, there are 9^n ways to fill the n blank squares. Suppose we were given 20 cells filled in. There are 16,173,092,699,229,880,893,718,618,465,586,445,357,583,280,647,840,659,957,609 possible ways to fill in the remaining 61 cells.
- For the **No Equal Adjacent Substrings Problem**, there are n^k possible strings. That's a huge number. For a 100-character long string of trinary digits, we have 515,377,520,732,011,331,036,461,129,765,621,272,702,107,522,001 possibilities.
- When **k-coloring maps of n regions**, there are k^n possible colorings. For example, there are 1,267,650,600,228,229,401,496,703,205,376 ways to color a 50-region map with 4 colors.
- There are $(15!)^7$ ways to arrange the walkers in a 7-day **Kirkman Schoolgirls Problem**. That is an 85-digit number.
- In the **Traveling Salesperson Problem**, there are $n!$ possible paths. 200! is a 375-digit number. 500! has 1,135 digits.

Backtracking vs Brute Force

- Backtracking algorithms are often much faster than brute force enumeration (when we consider all the possible solutions one by one) of all complete candidates because it can eliminate a large number of candidates with a single test.

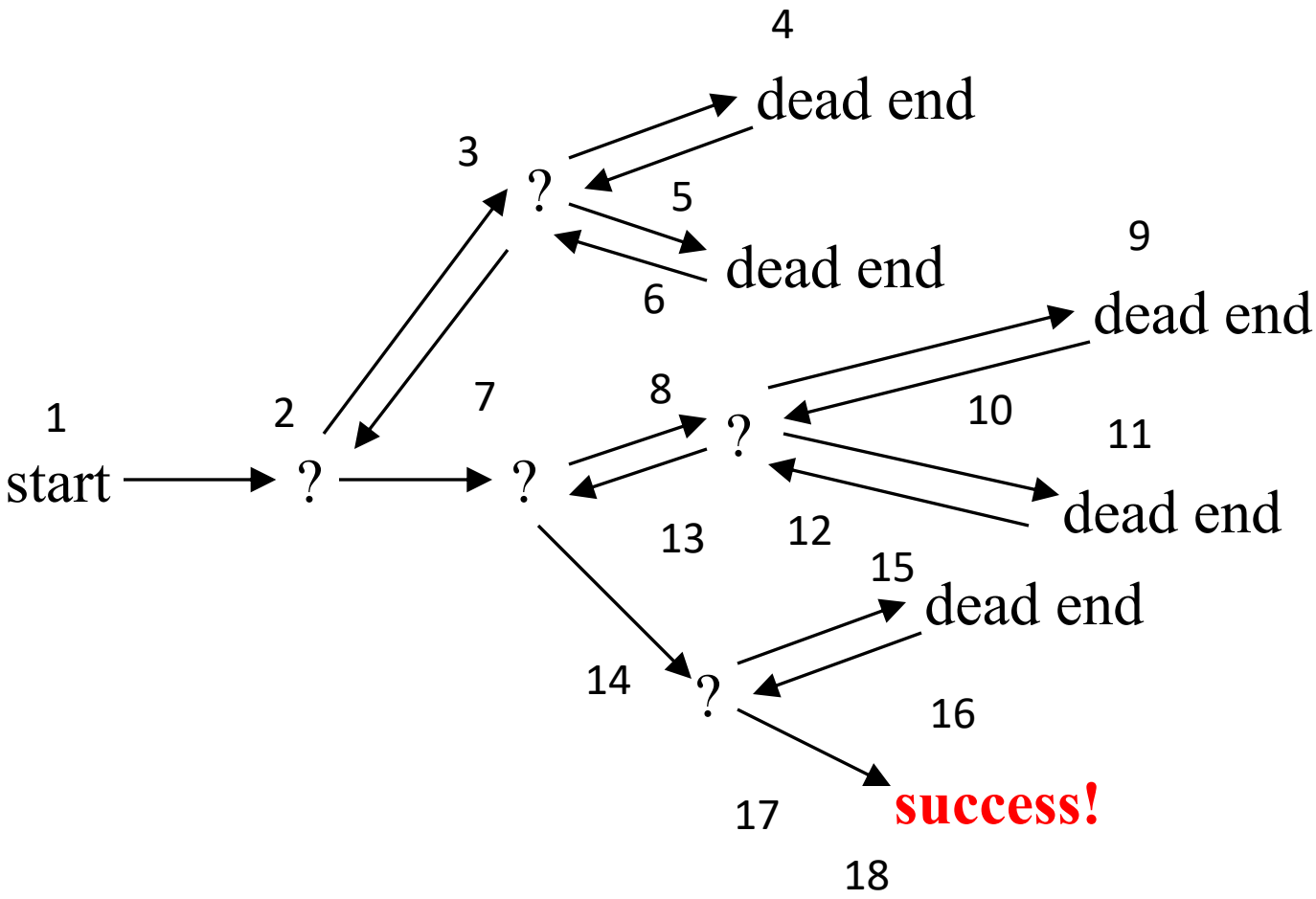
- Backtracking algorithms help in finding a solution to the first sub-problem and then recursively attempting to resolve other sub-problems based on the solution of the first issue. If the current issue cannot be resolved, the step is backtracked, and the next possible solution is applied to previous steps and then proceeds further.

How it works?

- We can construct a search tree based on the constraint satisfaction problem. Backtracking is basically a simple depth-first search (DFS) on this search tree.

- For every node (state) the algorithm checks whether the given node can be completed to a valid solution (good state). If it can not then the whole subtree is skipped. *And this is exactly why it is faster than exhaustive (brute-force) search*. It recursively enumerates all subtree (neighbors) of the actual node.

Backtracking



The backtracking algorithm

- Backtracking is really quite simple--we "explore" each node, as follows:
- To "explore" node N:
    1. If N is a goal node, return "success"
    2. If N is a leaf node, return "failure"
    3. For each child C of N,
        3.1. Explore C
            3.1.1. If C was successful, return "success"
    4. Return "failure"

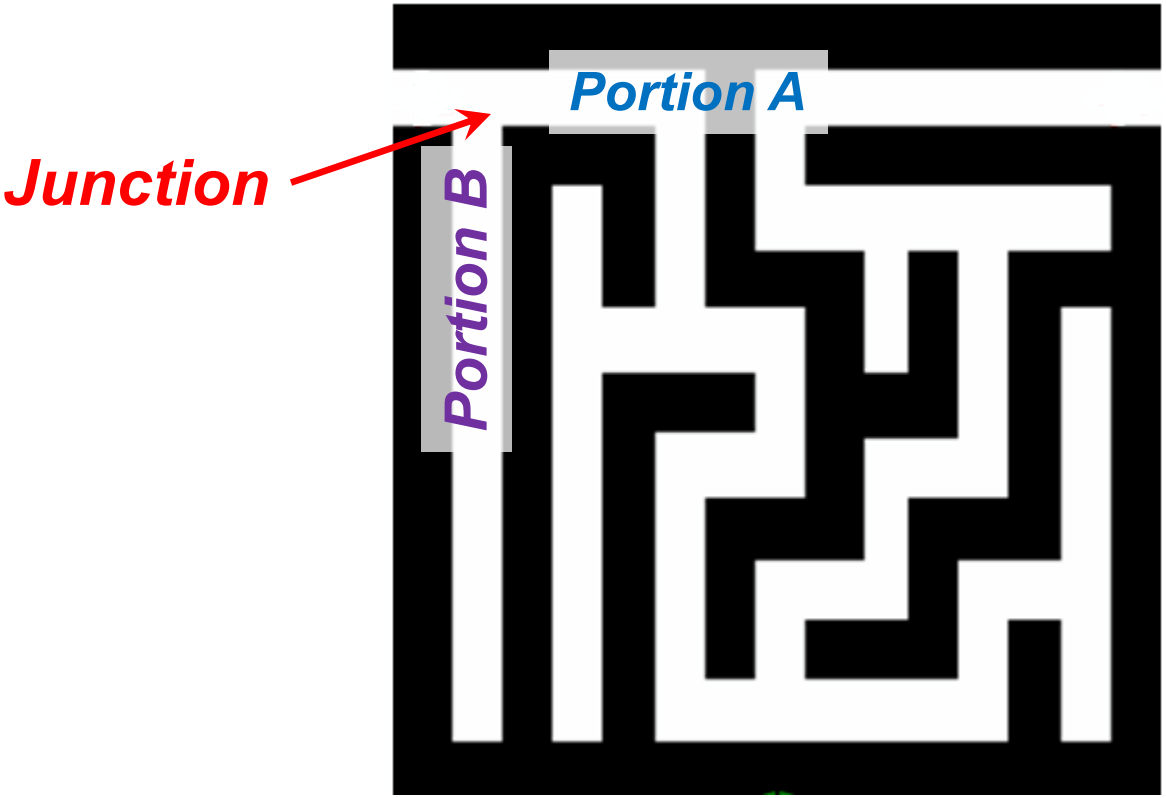
Classical applications

Classic examples include:

- Maze solving
- Map coloring
- Eight queens puzzle
- Knight's tour
- Logic programming languages
- Crossword puzzles

Example: Solving a Maze

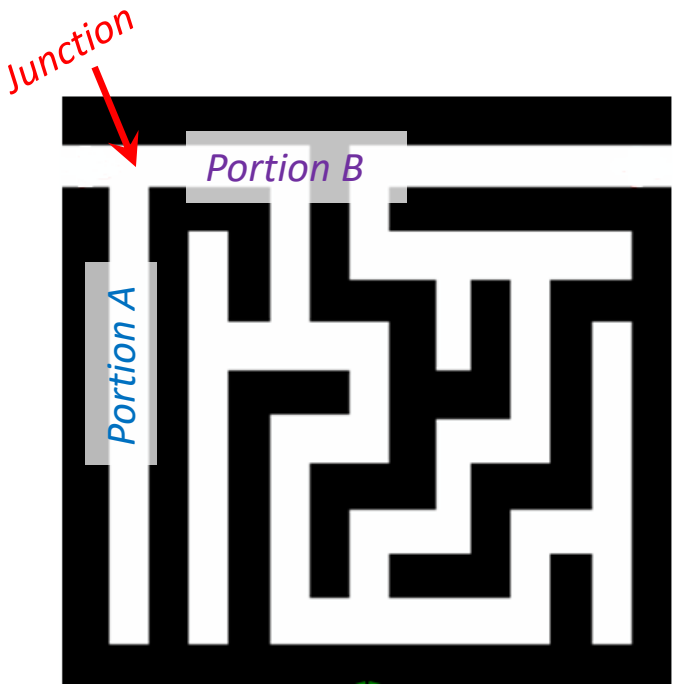
- Backtracking is a technique used to solve problems with a large search space, by systematically trying and eliminating possibilities.
- Example: Going through a maze. At some point, you might have two options of which direction to go:



Backtracking

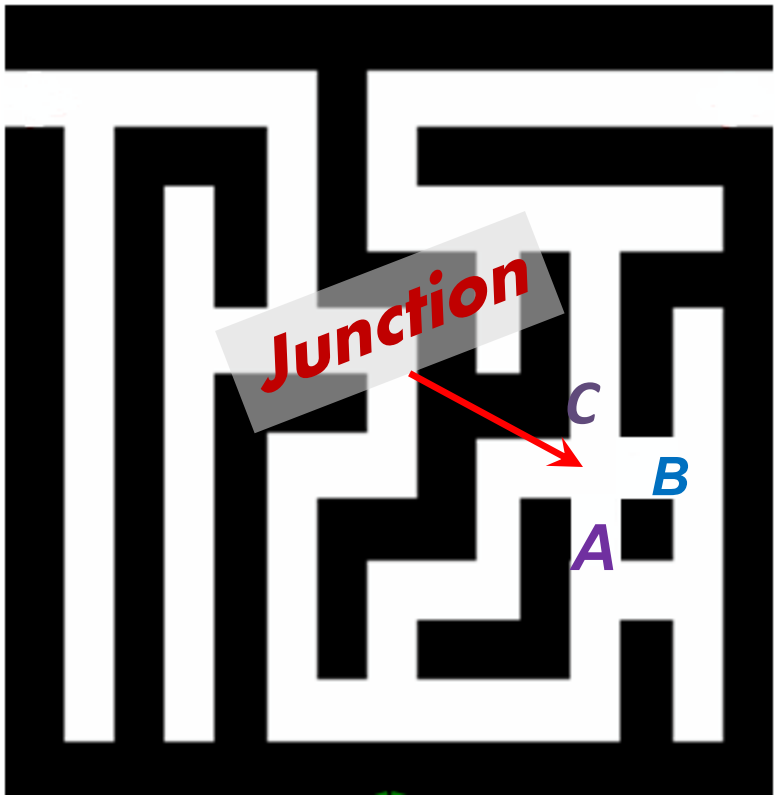
One strategy would be to try going through Portion A of the maze. If you get stuck before you find your way out, then you *"backtrack"* to the junction.

At this point in time you know that Portion A will *NOT* lead you out of the maze, so you then start searching in Portion B



Backtracking

- Clearly, at a single junction you could have even more than 2 choices.
- The backtracking strategy says to try each choice, one after the other. If you ever get stuck, *"backtrack"* to the junction and try the next choice.
- If you try all choices and never found a way out, then there IS no solution to the maze.



Backtracking

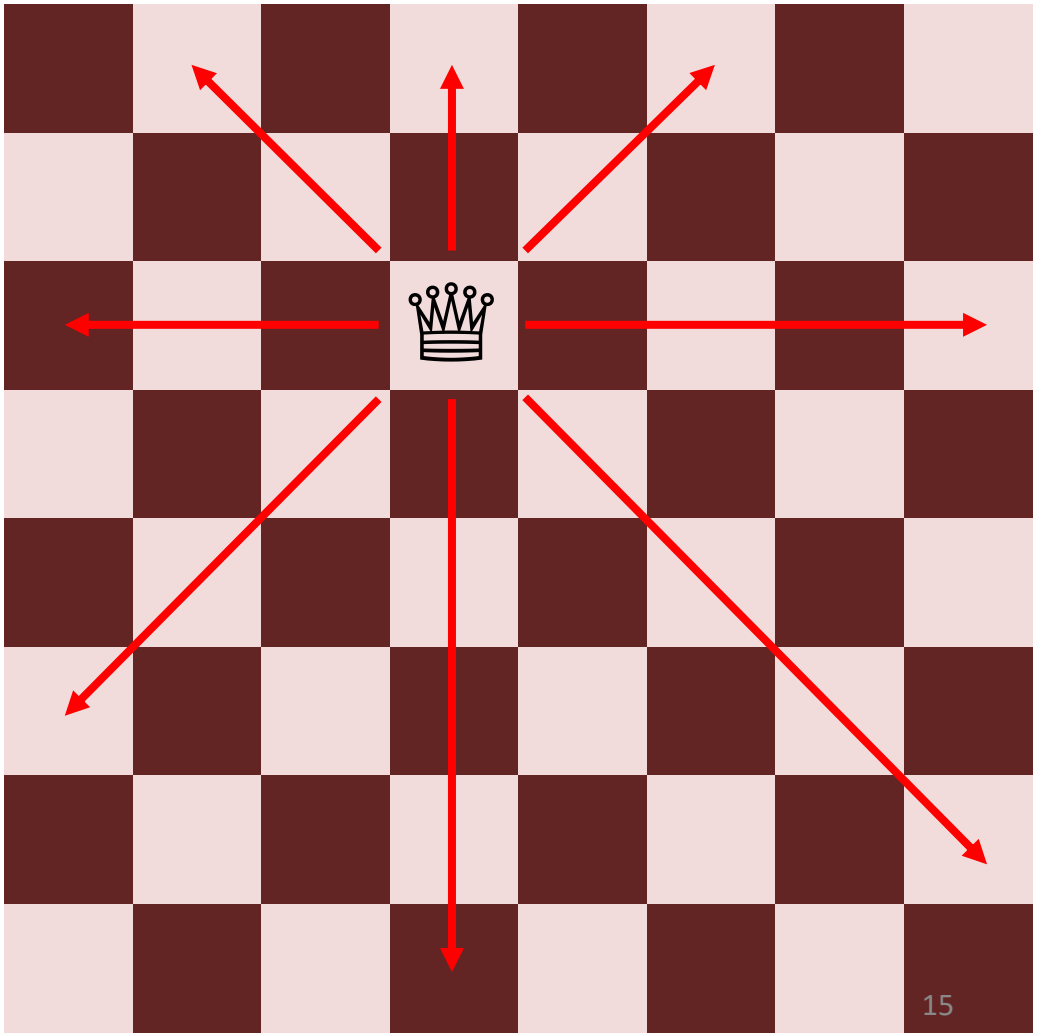
The neat thing about coding up backtracking is that it can be done recursively, without having to do all the bookkeeping at once.

- Instead, the stack of recursive calls does most of the bookkeeping
- (i.e., keeps track of which locations we've tried so far.)

Eight queens puzzle

Arrange eight queens on a chess board so that no two queens are attacking one another.

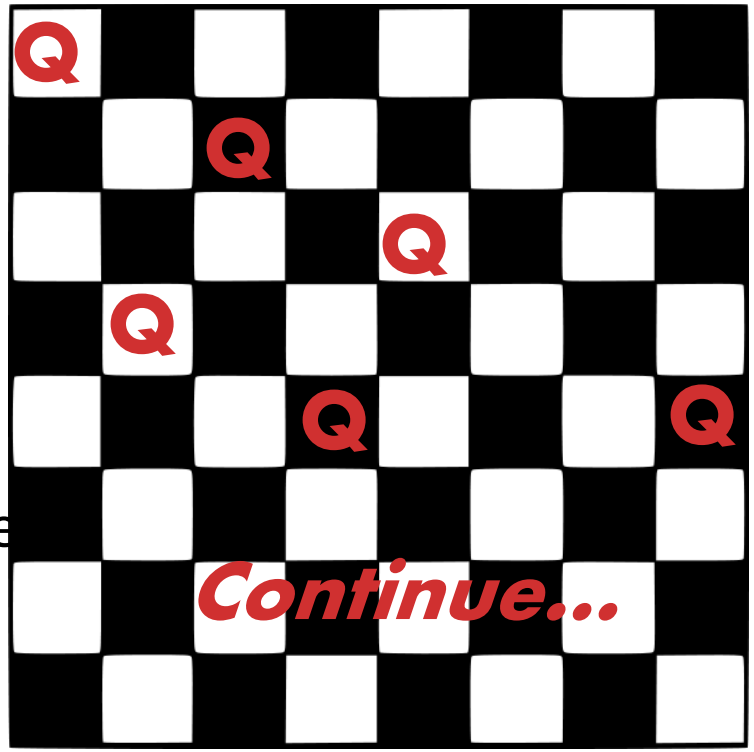
Queens can move all the way down any row, column or diagonal. So it's clear that each row and column of the board will have exactly one queen.



The backtracking strategy is as follows:

1) Place a queen on the first available square in row 1.
2) Move onto the next row, placing a queen on the first available square there (that doesn't conflict with the previously placed queens).
3) Continue in this fashion until either:
   a) You have solved the problem, or
   b) You get stuck.

   When you get stuck, remove the queens that got you there, until you get to a row where there is another valid square to try.



Animated Example:
http://www.hbmeyer.de/backtrack/achtdamen/eight.htm#up

Backtracking – 8 queens Analysis

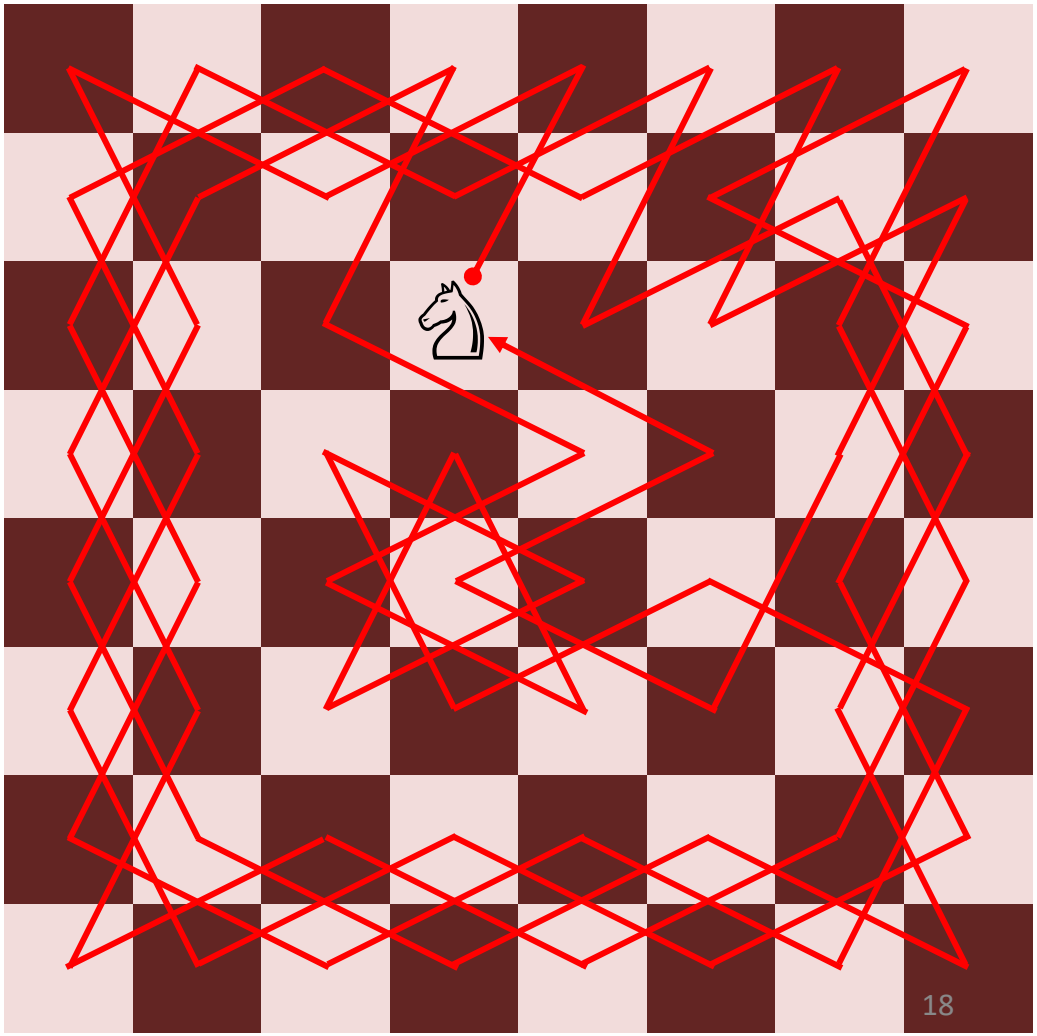
- Another possible brute-force algorithm is generate all possible permutations of the numbers 1 through 8 (there are 8! = 40,320),
  - Use the elements of each permutation as possible positions in which to place a queen on each row.
  - Reject those boards with diagonal attacking positions.

- The backtracking algorithm does a bit better
  - constructs the search tree by considering one row of the board at a time, eliminating most non-solution board positions at a very early stage in their construction.
  - because it rejects row and diagonal attacks even on incomplete boards, it examines only 15,720 possible queen placements.

- 15,720 is still a lot of possibilities to consider
  - Sometimes we have no other choice but to do the best we can ☺

Knight's tour

Have a knight visit all the squares of a chess board either as a path or a cycle



Sudoku

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

A typical Sudoku puzzle ...

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

... and its solution

Sudoku

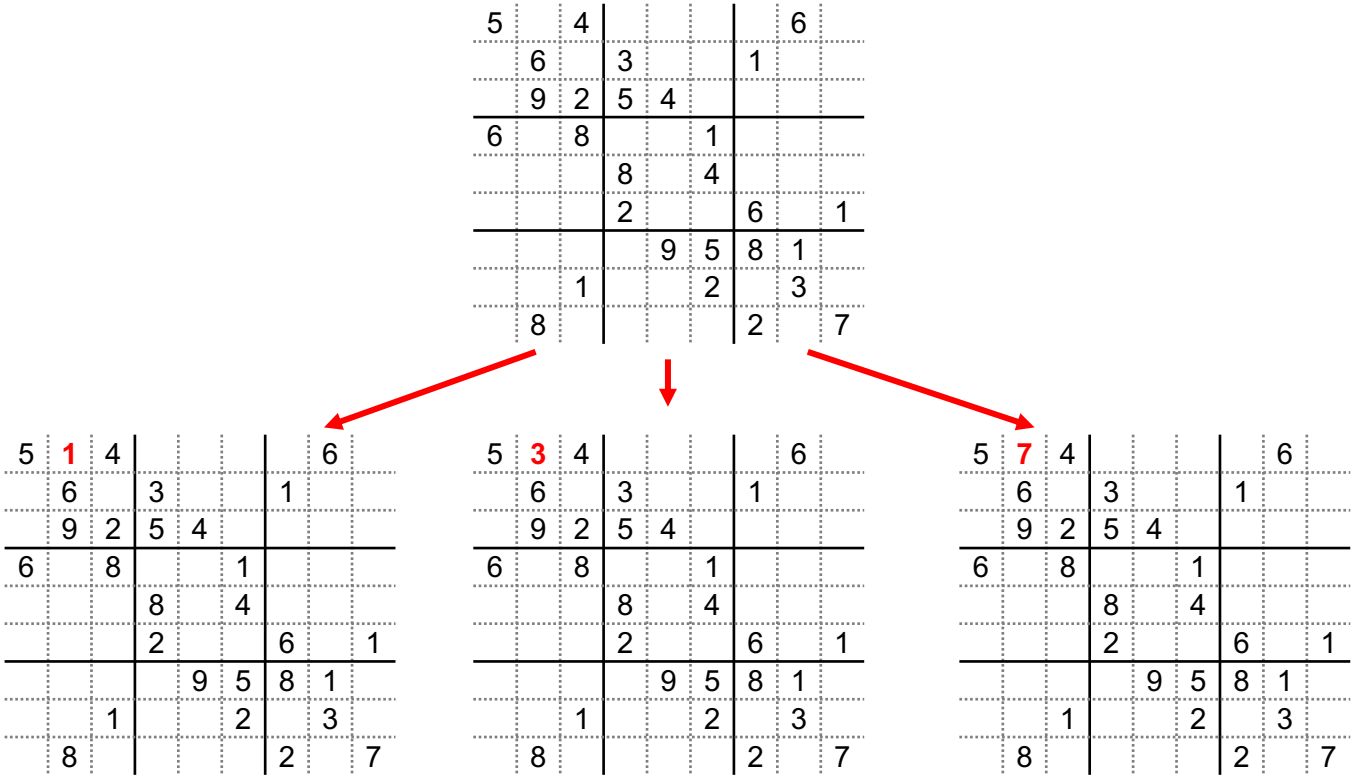
Consider the game Sudoku:

- The search space is 9^{53}

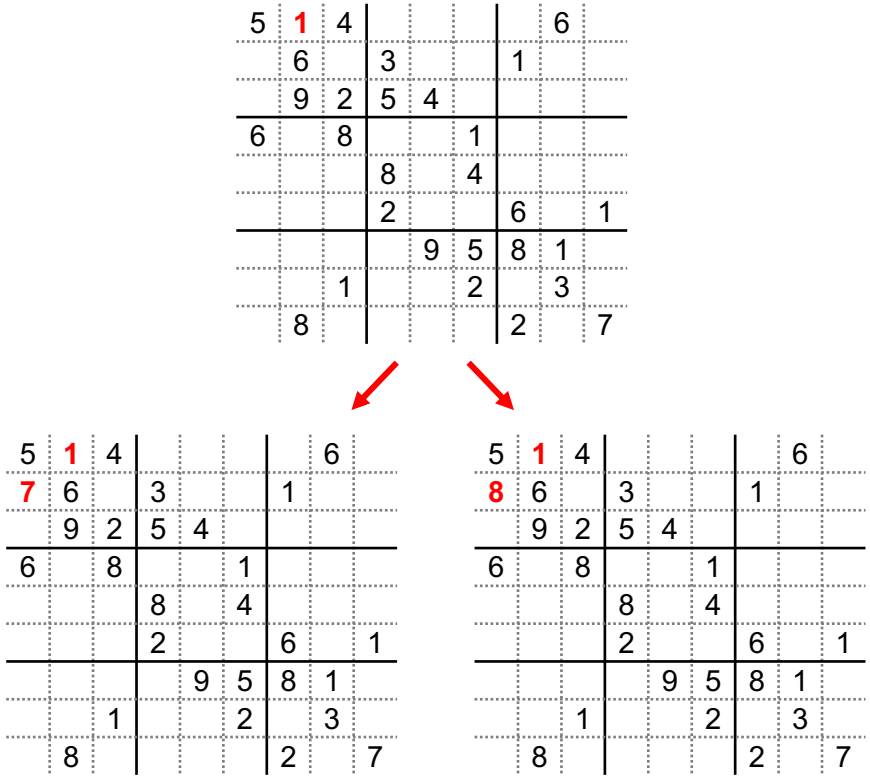
5		4					6	
	6		3			1		
	9	2	5	4				
6		8			1			
			8		4			
			2			6		1
				9	5	8	1	
		1			2		3	
	8					2		7

http://www.sudokuoftheday.com/

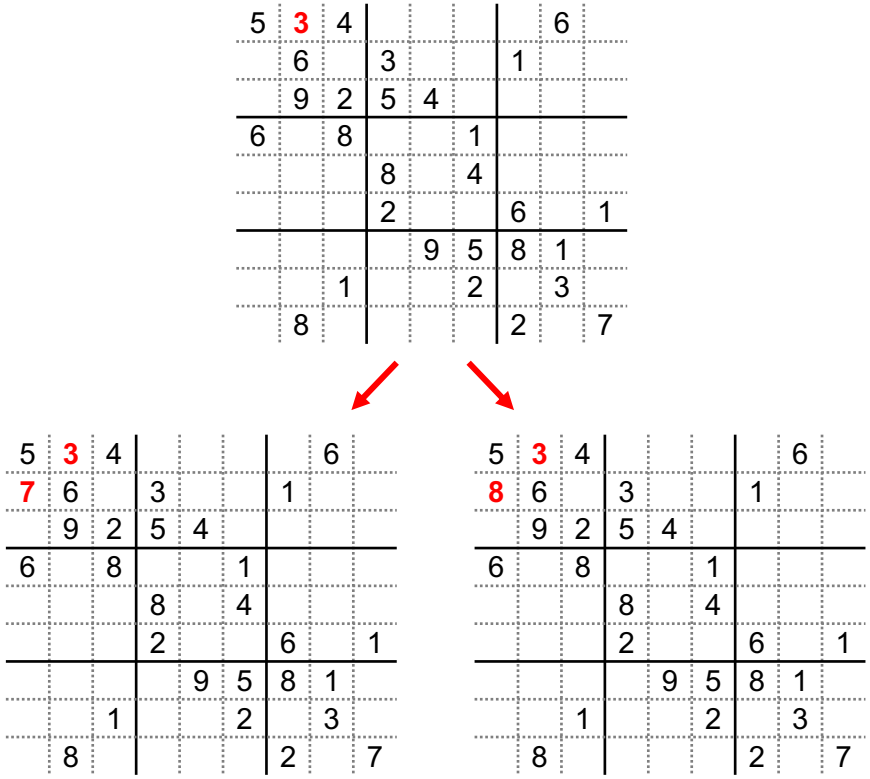
At least for the first entry in the first square, only 1, 3, 7 fit



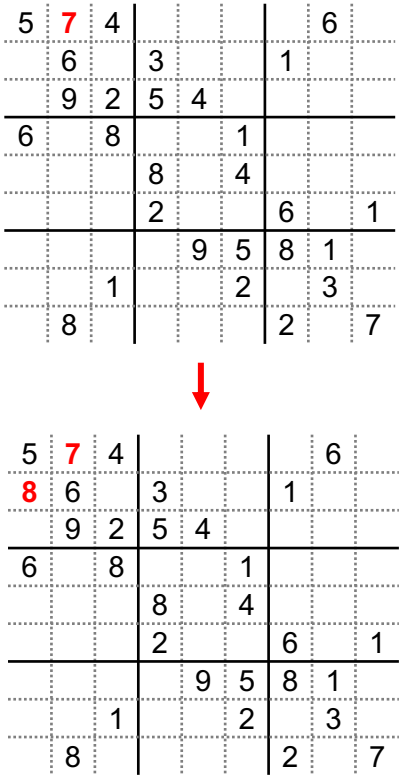
If the first entry has a 1, the 2nd entry in that square could be 7 or 8



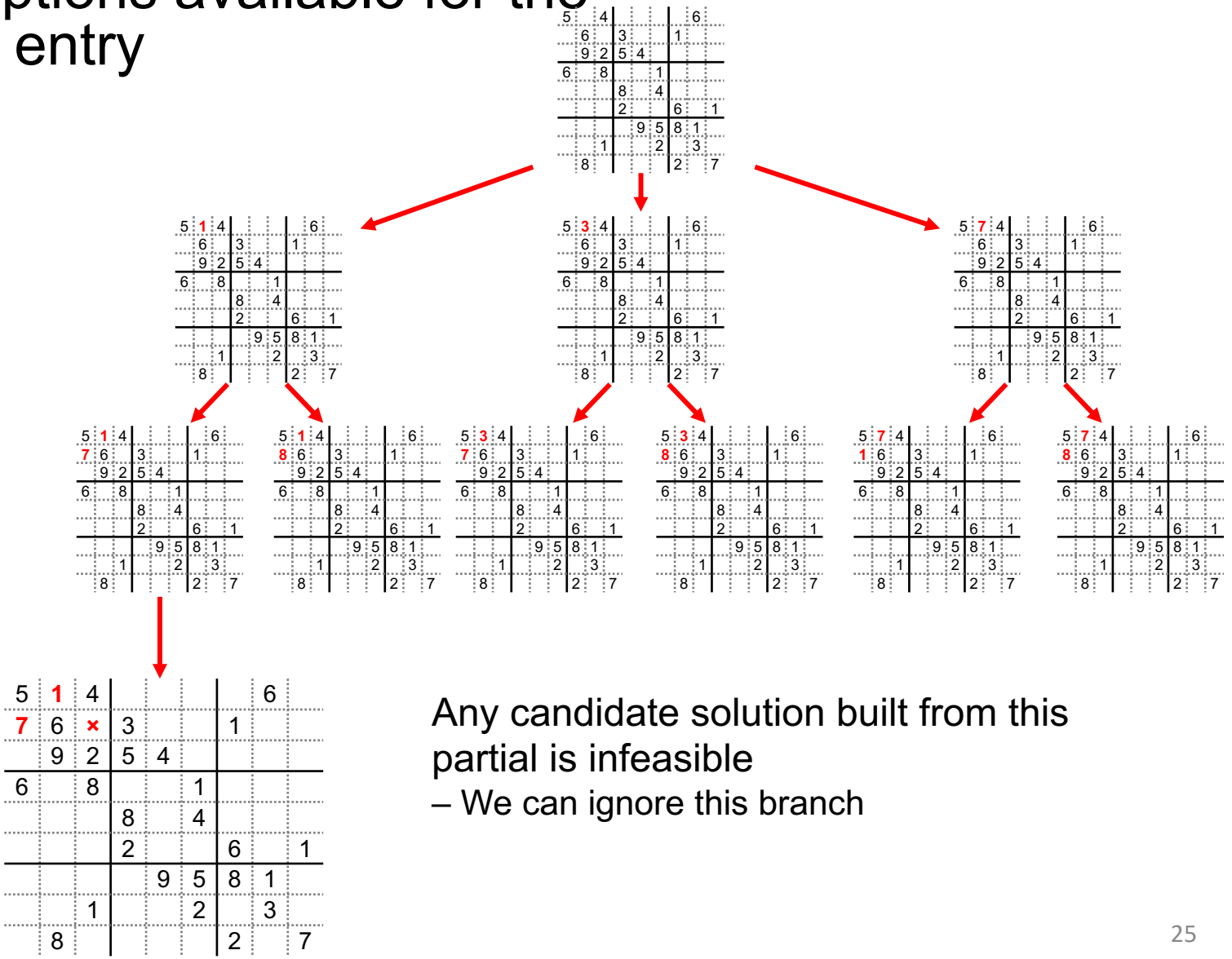
If the first entry has a 3, the 2nd entry in that square could be 7 or 8



If the first entry has a 7, the 2nd entry in that square could be 8



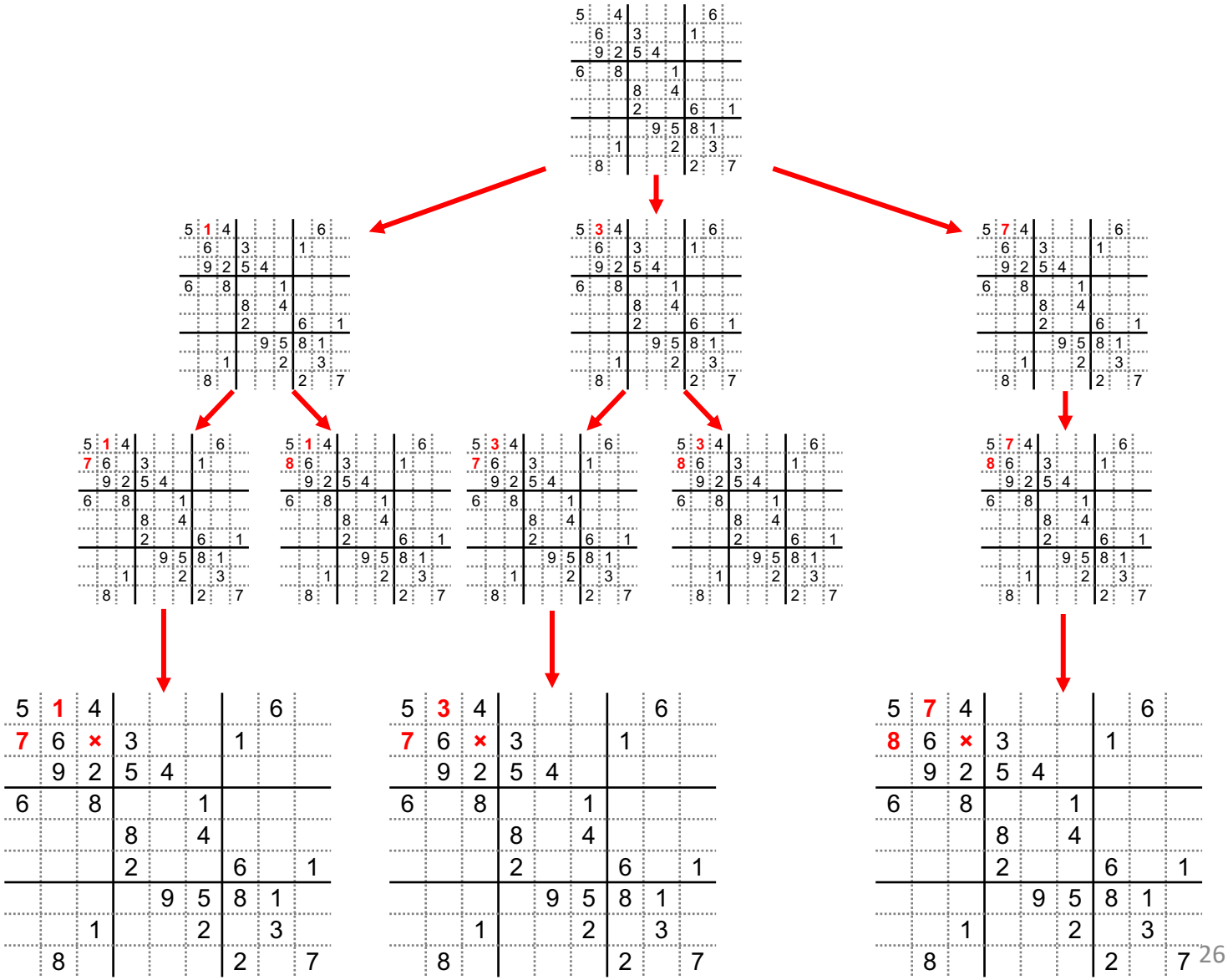
In the next child, there are no options available for the next entry



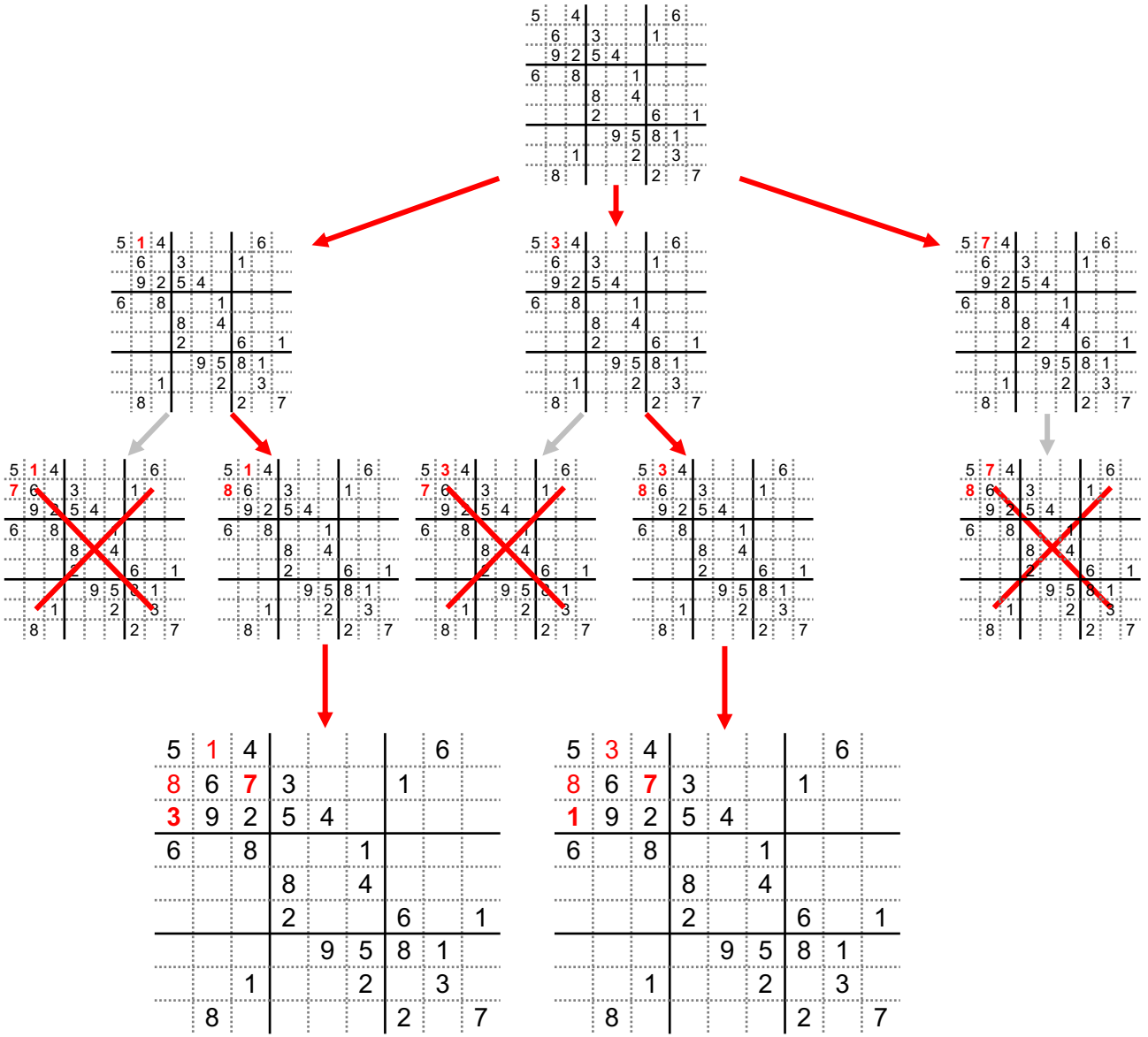
Any candidate solution built from this partial is infeasible

– We can ignore this branch

Three other branches lead to similar dead ends

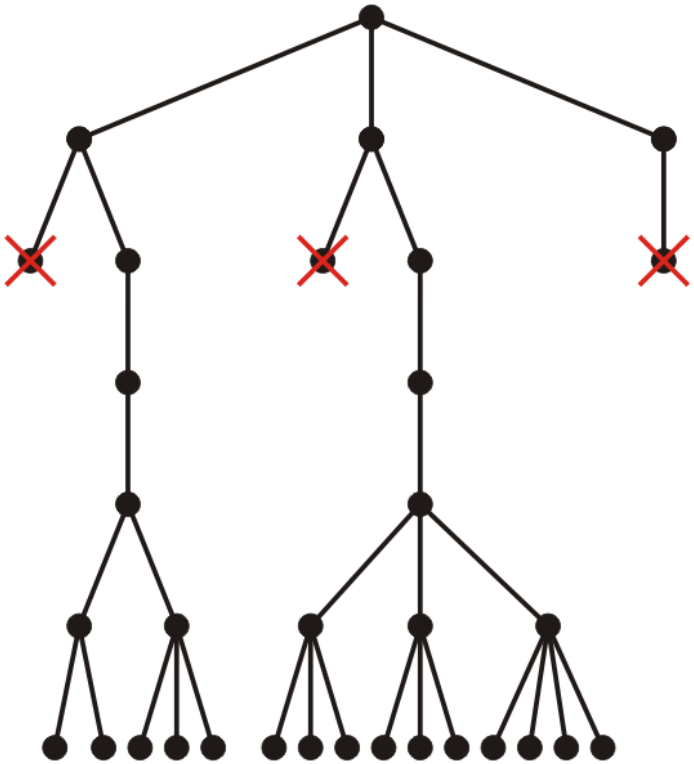


With the other two, there is only one candidate for each of the last two entries



Sudoku

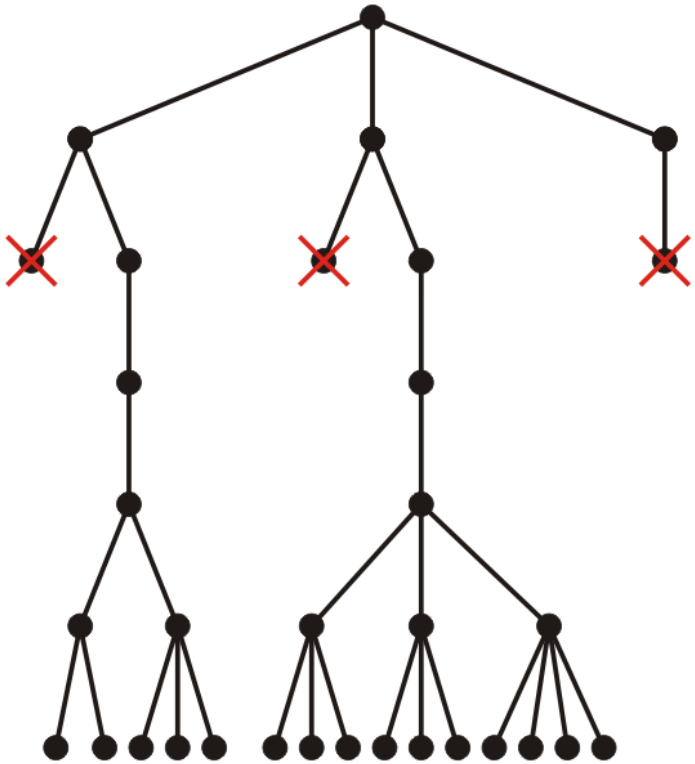
It may seem that this is a reasonably straight-forward method; however, the decision tree continues to branch quick once we start filling the second square



Sudoku

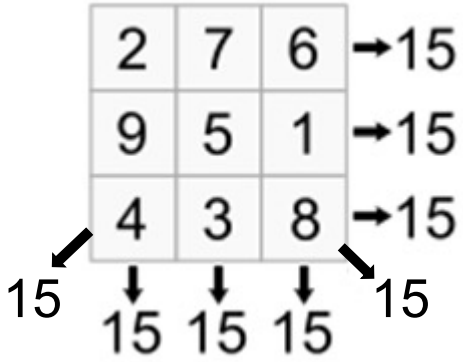
A binary tree of this height would have around $2^{54} - 1$ nodes

- Fortunately, as we get deeper into the tree, more get cut



Magic Number Squares

- A 3x3 magic number square has 9 numbers {1,2,… 9} that must be arranged in such a way that every row, column, and diagonal has the same sum.
  - For a 3x3 magic number square the sum is 3*(3*3+1)/2 = 15.



Magic Number Square

- It is possible to have different sized magic number squares of size nxn,
  - where numbers {1,2,… n*n} must be arranged in such a way that every row and column has the same sum.
  - For a nxn magic number square the sum is n*(n*n+1)/2

Magic Number Square

- Say we want to create a class MagicSquare that takes in an int n and returns all valid Magic Number Squares of size nxn.
  - Example, n = 3 returns 8 magic squares:

2	7	6
9	5	1
4	3	8

4	9	2
3	5	7
8	1	6

8	1	6
3	5	7
4	9	2

2	9	4
7	5	3
6	1	8

6	1	8
7	5	3
2	9	4

8	3	4
1	5	9
6	7	2

4	3	8
9	5	1
2	7	6

6	7	2
1	5	9
8	3	4

Magic Number Square

- How would we start?
  - The plan like we did with Eight Queens is just to try a number in the first position.
    - Then we can recursively solve the board at the next position given that the number has been placed.
    - Then we systematically “throw out” boards that do not meet our constraints.
  - What do we need to do that?
    - First we will want an empty board that we will fill in one square at a time.
    - Then we will want to mark which numbers in the set (1, … , n*n) have already been used.
    - We need to calculate the sum that each row, column, and diagonal must add up to.