BACKTRACKING ALGORITHM

SOLVING SUDOKU PUZZLE

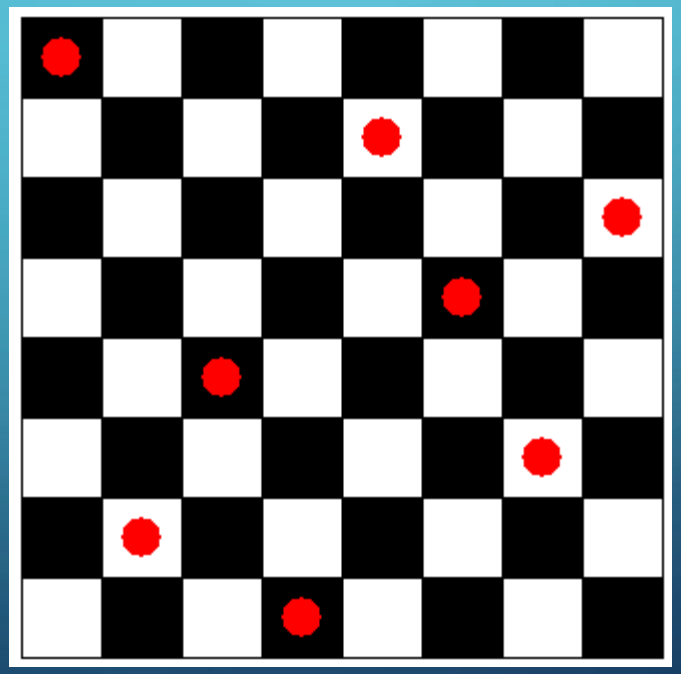
ANDREI

CONSTRAINT SATISFACTION PROBLEMS

- These are logic problems with constraint, e.g.
  1. Eight queen problem
  2. Sudoku
  3. Fox, goose and bag of beans puzzle (River Crossing Problem)
  4. Latin Square

EIGHT QUEEN PROBLEM

- Placing 8 queens on a chest board without any of the queens attacking any others, there are 92 solutions, here is a solution:



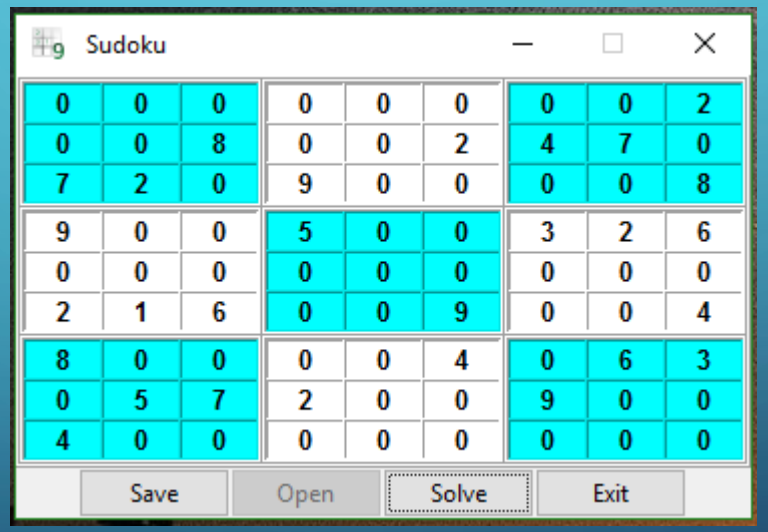
RIVER CROSSING PROBLEM

- A farmer had a fox, a goose, and a bag of beans. He wanted to cross a river with a boat. The farmer could carry only himself and a single animal or the bag of beans on each crossing. Unattended, the fox would eat the goose, the goose would eat the beans.
- Look up the solution for the puzzle in Wikipedia

https://en.wikipedia.org/wiki/Fox,_goose_and_bag_of_beans_puzzle

SUDOKU

- Here an example of a Sudoku puzzle



0	0	0	0	0	0	0	0	2
0	0	8	0	0	2	4	7	0
7	2	0	9	0	0	0	0	8
9	0	0	5	0	0	3	2	6
0	0	0	0	0	0	0	0	0
2	1	6	0	0	9	0	0	4
8	0	0	0	0	4	0	6	3
0	5	7	2	0	0	9	0	0
4	0	0	0	0	0	0	0	0

BACKTRACKING ALGORITHM

```
   input: n current position
 1 Backtracking (n)
 2 if n < end then
 3 |   pv ← possibleValues(n)
 4 |   foreach p in pv do
 5 |   |   commit(p)
 6 |   |   Backtracking (n + 1)
 7 |   |   if n+1 == end and iSsolved() then
 8 |   |   |   printSolution()
 9 |   |   end
10 |   |   uncommit(p)
11 |   end
12 end
```

Algorithm 1: Backtracking Algorithm

LATIN SQUARE PROBLEM

- A square matrix in which the sum of each row, column and the two main diagonals are all equal
- This small problem and we can use what tantamount to a brute force solution

2	7	6
6	5	1
4	3	8

LATIN SQUARE ALGORITHM

```
   input:  n current position
 1 latinSquare (n)
 2 if n < 9 then
 3 |   pv ← possibleValues(n)
 4 |   foreach p in pv do
 5 |   |   vector[n] = p
 6 |   |   latinSquare (n + 1)
 7 |   |   if n+1 == 9 and iSsolved() then
 8 |   |   |   printVector()
 9 |   |   end
10 |   |   vector[n] = 0
11 |   end
12 end
```

Algorithm 2: Latin Square

POSSIBLE VALUES FUNCTION

input : n current position
output: r = list of all possible values for position n

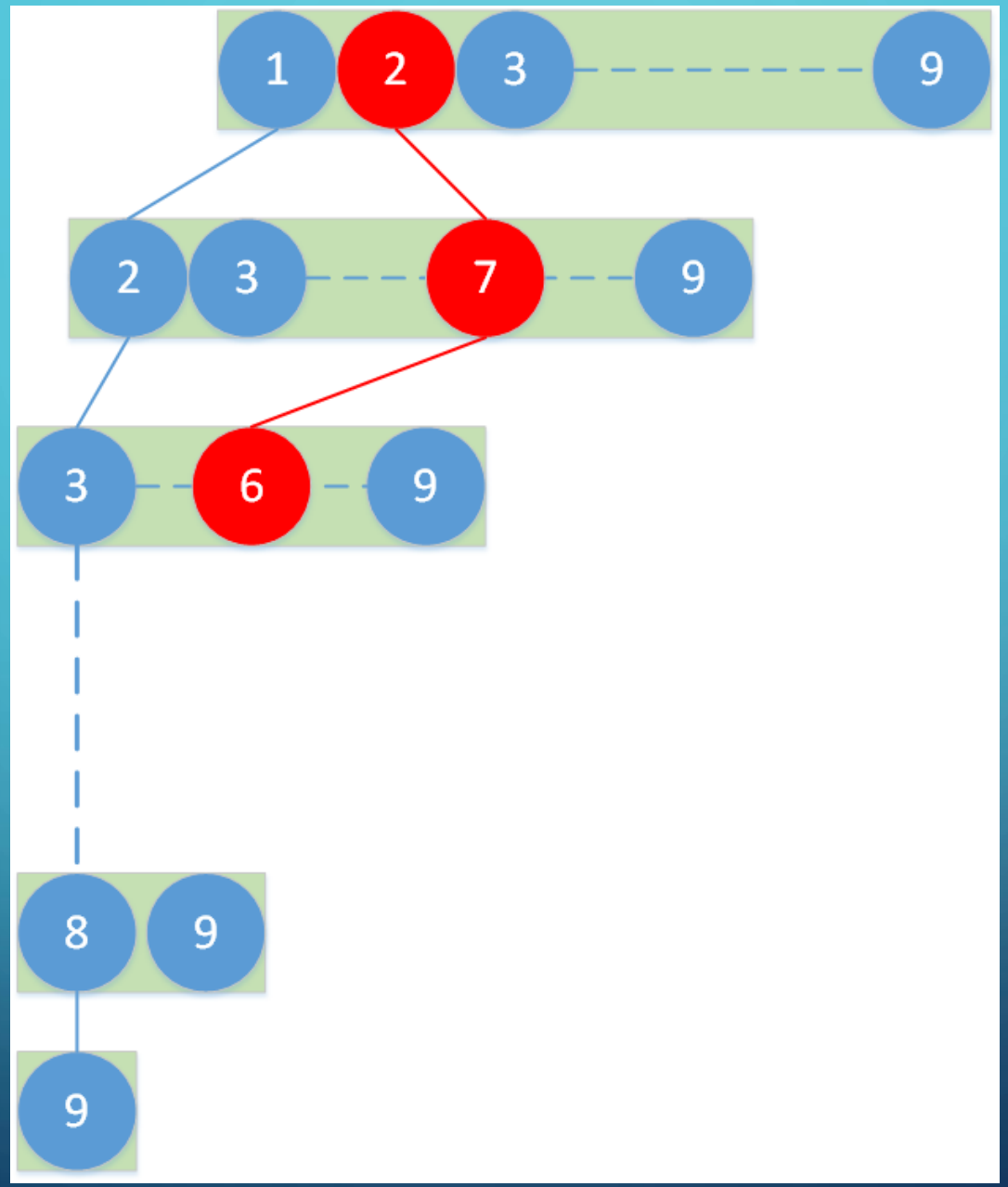
```
1 possibleValues (n)
2 r ← [x | 1 ≤ x ≤ 9 and x ∉ vector]
3 return r
```

Algorithm 3: possibleValues

FUNCTION IS_SOLVED FOR LATIN SQUARE

```
output: r = returns a Boolean value
1  isSolved ()
2  sum ← vector[0] + vector[1] + vector[2]
3  if if vector[3] + vector[4] + vector[5] ≠ sum then
4  |  return False
5  else if vector[6] + vector[7] + vector[8] ≠ sum then
6  |  return False
7  else if vector[0] + vector[3] + vector[6] ≠ sum then
8  |  return False
9  else if vector[1] + vector[4] + vector[7] ≠ sum then
10 |  return False
11 else if vector[2] + vector[5] + vector[8] ≠ sum then
12 |  return False
13 else if vector[0] + vector[4] + vector[8] ≠ sum then
14 |  return False
15 else if vector[2] + vector[4] + vector[6] ≠ sum then
16 |  return False
17 else
18 |  return True
19 end
```

Algorithm 4: isSolved


1
2
3
9
2
3
7
9
3
6
9
8
9
9

IF WE REMOVE THE CONSTRAINTS WE GET 9! = 362880 PERMUTATION OF NUMBERS 1,2,..,9. THIS IS THE FIRST 10 PERMUTATIONS. (A BIT OF EXAGGERATION – WHAT ABOUT THE CONSTRAINT THAT ARE GIVEN)

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

LATIN SQUARE 3X3 SOLUTIONS

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

$$\mu = \frac{1}{9}\sum_{i=1}^{9} i = 5$$

SOLVING SUDOKU

- Brute force means checking $(9!)^9 = 1.1 \text{ X } 10^{50}$ permutations, this would be astronomical
- We need to prune the search space
- Pruning comes when we calculate the possible values for each position
- Also, how we check if a solution is found.

```
input: n current position
1  Sudoku (n)
2  if n < 81 then
3      r, c ← index(n)
4      if puzzle[r][c] == 0 then
5          pv ← possibleValues(n)
6          foreach p in pv do
7              puzzle[r][c] ← p
8              Sudoku (n + 1)
9              if n+1 == 81 and iSsolved() then
10                 printSolution()
11             end
12             puzzle[r][c] ← 0
13         end
14     end
15     else
16         if n+1 == end and iSsolved() then
17             printSolution()
18         end
19         Sudoku (n + 1)
20     end
21 end
```

INDEX FUNCTION

- Notice index function below, retunes a tuple

```
input  : n current position
output: r, c = row, column for position n
1 index (n)
2 r ← n Div 9
3 c ← n Mod 9
4 return r, c
```

FUNCTION FOR COMPUTING POSSIBLE_VALUES

```
input  : i index of row
input  : j index of column
output: pv = possible values for puzzle[i][j]
1 possibleValues (i, j)
2 pv ← ∅
3 if puzzle[i][j] == 0 then
4 |   pv ← pv ∪ row(i) ∪ column(j) ∪ box(i, j)
5 end
6 return pv
```

FUNCTION BOX

```
input  : i index of row
input  : j index of column
output: s = for cell, puzzle[i][j], returns set of all elements that are not
          in the same box

box (i, j)
si ← find_box_start(i)
sj ← find_box_start(j)
s ← ∅
for p in range(si, si+3) do
    for q in range(sj, sj+3) do
        if not (puzzle[p][q] == 0) then
            s ← s ∪ {puzzle[p][q]}
        end
    end
end
return s
```

FUNCTION FIND_BOX_START

input : i index of row or column
output: v = for any given row or column it returns the starting value of respective row or column of the box they belong to

```
1 find_box_start(i)
2 v ← 3 * (n Div 3)
3 return v
```

BOOLEAN FUNCTION SOLVED

output: *boolean* - checks if the content of each cell in the puzzle is unique in its row, column and box it occupies

```
1 iSsolved()
2 for i in range(9) do
3 |   for j in range(9) do
4 |   |   if not is_unique(i, j) then
5 |   |   |   return False
6 |   |   end
7 |   end
8 end
9 return True
```

```
input: n current position
1  Sudoku (n)
2  if n < 81 then
3  |  r, c ← index(n)
4  |  if puzzle[r][c] == 0 then
5  |  |  pv ← possibleValues(n)
6  |  |  foreach p in pv do
7  |  |  |  puzzle[r][c] ← p
8  |  |  |  Sudoku (n + 1)
9  |  |  |  if n+1 == 81 then
10 |  |  |  |  printSolution()
11 |  |  |  end
12 |  |  |  puzzle[r][c] ← 0
13 |  |  end
14 |  end
15 |  else
16 |  |  if n+1 == end then
17 |  |  |  printSolution()
18 |  |  end
19 |  |  Sudoku (n + 1)
20 |  end
21 end
```

For this problem (the way we wrote possible_values) we don’t need is_solved, hence the algorithm as presented in slide 21