BACKTRACKING ALGORITHM

SOLVING SUDOKU PUZZLE

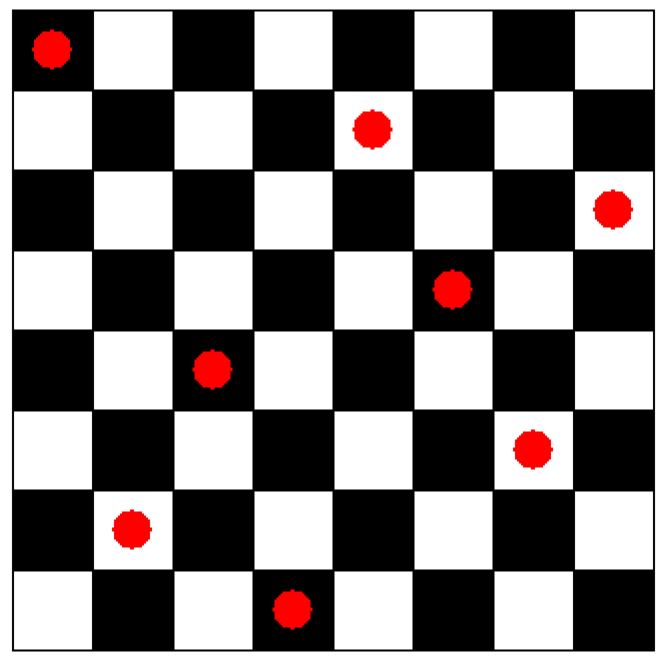
ANDREI

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:



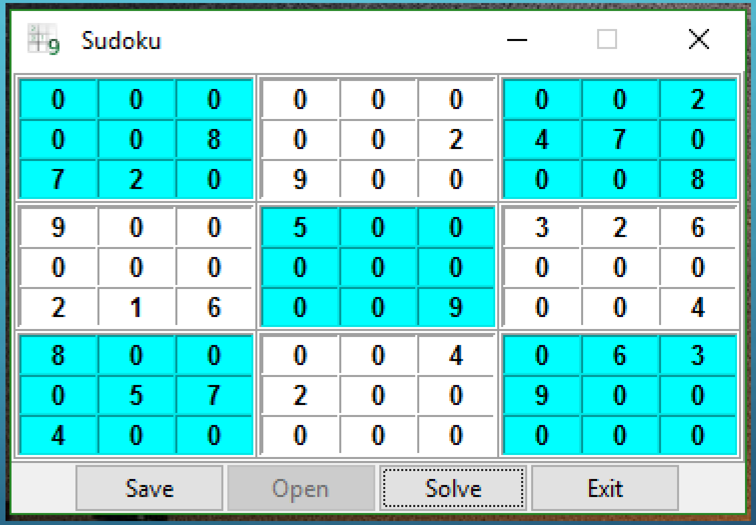
RIVER CROSSING PROBLEM

- 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



BACKTRACKING ALGORITHM

```
input:  n current position
Backtracking (n)
if n < end then
    pv ← possibleValues(n)
    foreach p in pv do
        commit(p)
        Backtracking (n + 1)
        if n+1 == end and iSsolved() then
            printSolution()
        end
        uncommit(p)
    end
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
latinSquare (n)
if n < 9 then
    pv ← possibleValues(n)
    foreach p in pv do
        vector[n] = p
        latinSquare (n + 1)
        if n+1 == 9 and iSsolved() then
            printVector()
        end
        vector[n] = 0
    end
end
```

Algorithm 2: Latin Square

0	0	0	0	0	0	0	0	0

POSSIBLE VALUES FUNCTION

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

possibleValues (n)
r $\leftarrow [x \mid 1 \leq x \leq 9 \; and \; x \notin vector]$
return r

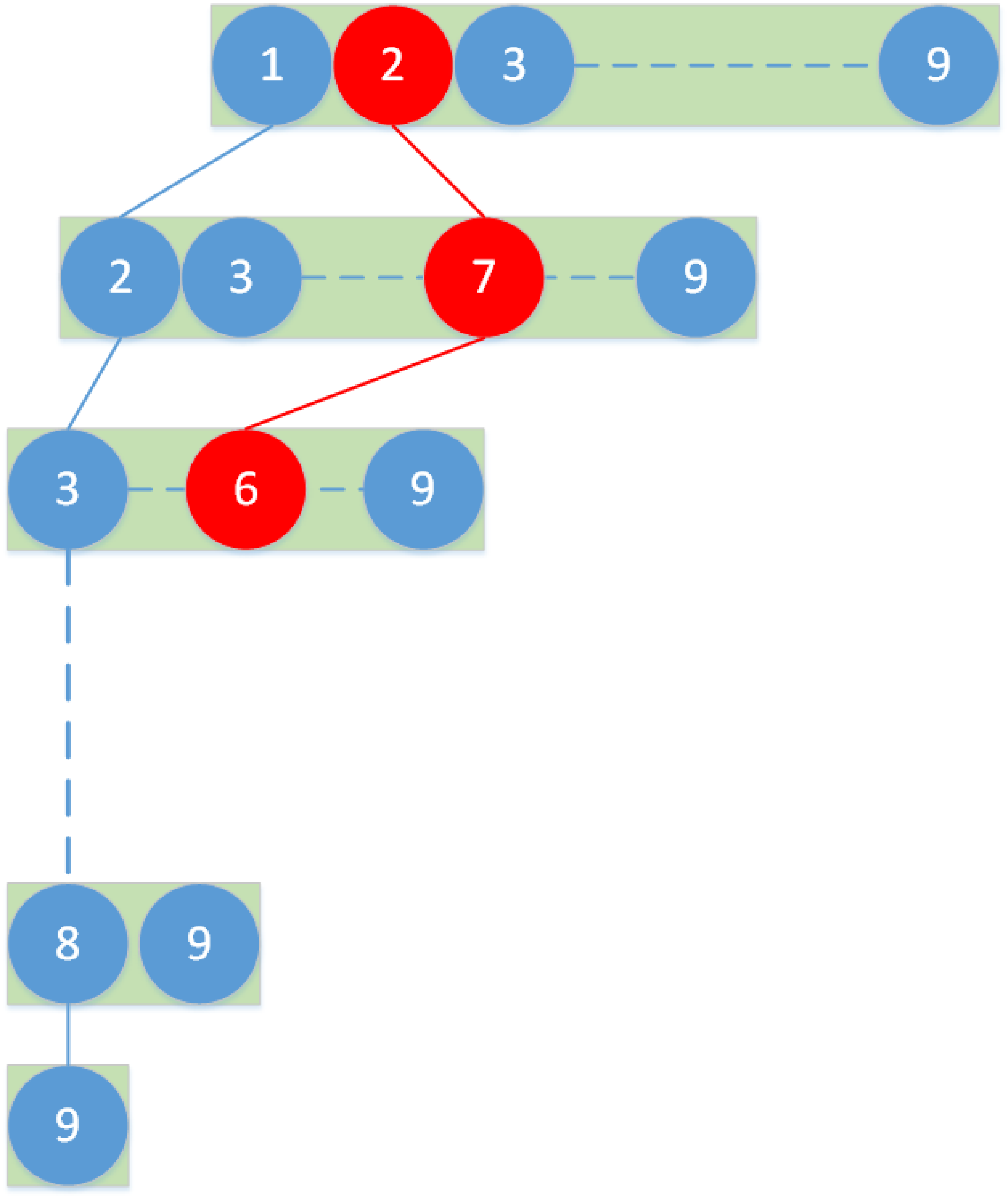
Algorithm 3: possibleValues

FUNCTION IS_SOLVED FOR LATIN SQUARE

output: r = returns a Boolean value

```
isSolved ()
sum ← vector[0] + vector[1] + vector[2]
if if vector[3] + vector[4] + vector[5] ≠ sum then
    return False
else if vector[6] + vector[7] + vector[8] ≠ sum then
    return False
else if vector[0] + vector[3] + vector[6] ≠ sum then
    return False
else if vector[1] + vector[4] + vector[7] ≠ sum then
    return False
else if vector[2] + vector[5] + vector[8] ≠ sum then
    return False
else if vector[0] + vector[4] + vector[8] ≠ sum then
    return False
else if vector[2] + vector[4] + vector[6] ≠ sum then
    return False
else
    return True
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.

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. (a bit of exaggeration – what about the constraint that are given)
- 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

Sudoku (n)
if n < 81 then
    r, c ← index(n)
    if puzzle[r][c] == 0 then
        pv ← possibleValues(n)
        foreach p in pv do
            puzzle[r][c] ← p
            Sudoku (n + 1)
            if n+1 == 81 and iSsolved() then
                printSolution()
            end
            puzzle[r][c] ← 0
        end
    end
    else
        if n+1 == end and iSsolved() then
            printSolution()
        end
        Sudoku (n + 1)
    end
end
```

INDEX FUNCTION

- Notice index function below, retunes a tuple

input : n current position
output: r, c = row, column for position n

index (n)
r $\leftarrow$ n **Div** 9
c $\leftarrow$ n **Mod** 9
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]

possibleValues (i, j)
$pv \leftarrow \emptyset$
if $puzzle[i][j] == 0$ **then**
    $pv \leftarrow pv \cup row(i) \cup column(j) \cup box(i, j)$
end
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

```
find_box_start(i)
v ← 3 * (n Div 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

```
iSsolved()
for i in range(9) do
    for j in range(9) do
        if not is_unique(i, j) then
            return False
        end
    end
end
return True
```

```
input:  n current position

Sudoku (n)
if n < 81 then
    r, c ← index(n)
    if puzzle[r][c] == 0 then
        pv ← possibleValues(n)
        foreach p in pv do
            puzzle[r][c] ← p
            Sudoku (n + 1)
            if n+1 == 81 then
                printSolution()
            end
            puzzle[r][c] ← 0
        end
    end
    else
        if n+1 == end then
            printSolution()
        end
        Sudoku (n + 1)
    end
end
```

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