SUDOKU SOLVER

Leonardo Iurada - Computational Intelligence

A scheduling problem

➔ A company, every day we need a technician

A scheduling problem

- A company, every day we need a technician
- Freelance available days:

	Mon	Tue	Wed	Thu	Fri	Sat	Sun
Alice	x			x			x
Bob	x				x		
Chuck	x		x		x		x
Dan		x		x			
Eve			x			x	
Frank		x		x		x	

A scheduling problem

➔ Constraints:

	Mon	Tue	Wed	Thu	Fri	Sat	Sun
Alice	x			x			x
Bob	x				x		
Chuck	x		x		x		x
Dan		x		x			
Eve			x			x	
Frank		x		x		x	

A scheduling problem

- Constraints:
  - Only 1 person per day

	Mon	Tue	Wed	Thu	Fri	Sat	Sun
Alice	x			x			x
Bob	x				x		
Chuck	x		x		x		x
Dan		x		x			
Eve			x			x	
Frank		x		x		x	

A scheduling problem

- Constraints:
  - Only 1 person per day
  - Rows picked as they are (no slicing)

	Mon	Tue	Wed	Thu	Fri	Sat	Sun
Alice	x			x			x
Bob	x				x		
Chuck	x		x		x		x
Dan		x		x			
Eve			x			x	
Frank		x		x		x	

A scheduling problem

- Constraints:
  - Only 1 person per day
  - Rows picked as they are (no slicing)
- Solution:
  - A set of rows that cover all days

	Mon	Tue	Wed	Thu	Fri	Sat	Sun
Alice	x			x			x
Bob	x				x		
Chuck	x		x		x		x
Dan		x		x			
Eve			x			x	
Frank		x		x		x	

A scheduling problem

$$\mathcal{O}\left(\sum_{k=1}^{6}\binom{6}{k}\right) \rightarrow 63$$

- ➔ Constraints:
  - ◆ Only 1 person per day
  - ◆ Rows picked as they are (no slicing)
- ➔ Solution:
  - ◆ A set of rows that cover all days

	Mon	Tue	Wed	Thu	Fri	Sat	Sun
Alice	x			x			x
Bob	x				x		
Chuck	x		x		x		x
Dan		x		x			
Eve			x			x	
Frank		x		x		x	

A scheduling problem

- Constraints:
  - Only 1 person per day
  - Rows picked as they are (no slicing)
- Solution:
  - A set of rows that cover all days

	Mon	Tue	Wed	Thu	Fri	Sat	Sun
A	1	0	0	1	0	0	1
B	1	0	0	0	1	0	0
C	1	0	1	0	1	0	1
D	0	1	0	1	0	0	0
E	0	0	1	0	0	1	0
F	0	1	0	1	0	1	0

Solution: []

A scheduling problem

- Constraints:
  - Only 1 person per day
  - Rows picked as they are (no slicing)
- Solution:
  - A set of rows that cover all days

	Mon	Tue	Wed	Thu	Fri	Sat	Sun
A	1	0	0	1	0	0	1
B	1	0	0	0	1	0	0
C	1	0	1	0	1	0	1
D	0	1	0	1	0	0	0
E	0	0	1	0	0	1	0
F	0	1	0	1	0	1	0

Solution: []

A scheduling problem

- Constraints:
  - Only 1 person per day
  - Rows picked as they are (no slicing)
- Solution:
  - A set of rows that cover all days

	Mon	Tue	Wed	Thu	Fri	Sat	Sun
A	1	0	0	1	0	0	1
B	1	0	0	0	1	0	0
C	1	0	1	0	1	0	1
D	0	1	0	1	0	0	0
E	0	0	1	0	0	1	0
F	0	1	0	1	0	1	0

Solution: []

A scheduling problem

- Constraints:
  - Only 1 person per day
  - Rows picked as they are (no slicing)
- Solution:
  - A set of rows that cover all days

	Mon	Tue	Wed	Thu	Fri	Sat	Sun
A	1	0	0	1	0	0	1
B	1	0	0	0	1	0	0
C	1	0	1	0	1	0	1
D	0	1	0	1	0	0	0
E	0	0	1	0	0	1	0
F	0	1	0	1	0	1	0

Solution: [D]

A scheduling problem

- Constraints:
  - Only 1 person per day
  - Rows picked as they are (no slicing)
- Solution:
  - A set of rows that cover all days

	Mon	Tue	Wed	Thu	Fri	Sat	Sun
A	1	0	0	1	0	0	1
B	1	0	0	0	1	0	0
C	1	0	1	0	1	0	1
D	0	1	0	1	0	0	0
E	0	0	1	0	0	1	0
F	0	1	0	1	0	1	0

Solution: [D]

A scheduling problem

- Constraints:
  - Only 1 person per day
  - Rows picked as they are (no slicing)
- Solution:
  - A set of rows that cover all days

	Mon	Tue	Wed	Thu	Fri	Sat	Sun
A	1	0	0	1	0	0	1
B	1	0	0	0	1	0	0
C	1	0	1	0	1	0	1
D	0	1	0	1	0	0	0
E	0	0	1	0	0	1	0
F	0	1	0	1	0	1	0

Solution: [D]

A scheduling problem

- Constraints:
  - Only 1 person per day
  - Rows picked as they are (no slicing)
- Solution:
  - A set of rows that cover all days

	Mon	Tue	Wed	Thu	Fri	Sat	Sun
A	1	0	0	1	0	0	1
B	1	0	0	0	1	0	0
C	1	0	1	0	1	0	1
D	0	1	0	1	0	0	0
E	0	0	1	0	0	1	0
F	0	1	0	1	0	1	0

Solution: [D]

A scheduling problem

- ➔ Constraints:
  - ◆ Only 1 person per day
  - ◆ Rows picked as they are (no slicing)
- ➔ Solution:
  - ◆ A set of rows that cover all days

	Mon	Wed	Fri	Sat	Sun
B	1	0	1	0	0
C	1	1	1	0	1
E	0	1	0	1	0

Solution: [D]

A scheduling problem

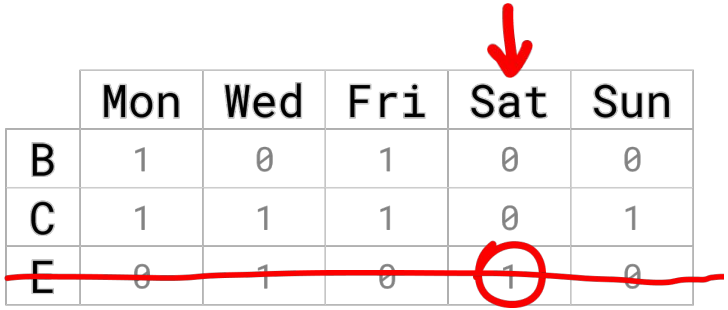
- Constraints:
  - Only 1 person per day
  - Rows picked as they are (no slicing)
- Solution:
  - A set of rows that cover all days

	Mon	Wed	Fri	Sat	Sun
B	1	0	1	0	0
C	1	1	1	0	1
E	0	1	0	1	0

Solution: [D]

A scheduling problem

- Constraints:
  - Only 1 person per day
  - Rows picked as they are (no slicing)
- Solution:
  - A set of rows that cover all days

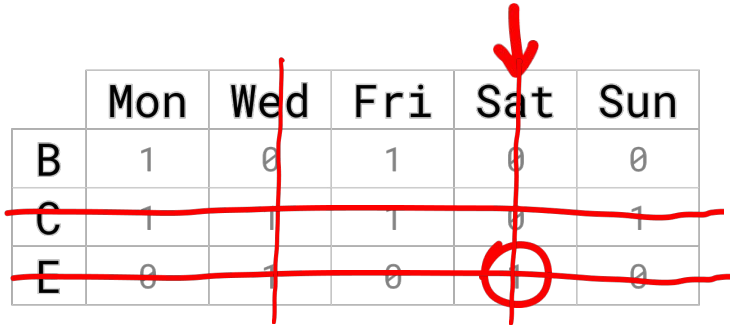


	Mon	Wed	Fri	Sat	Sun
B	1	0	1	0	0
C	1	1	1	0	1
E	0	1	0	1	0

Solution: [D,E]

A scheduling problem

- Constraints:
  - Only 1 person per day
  - Rows picked as they are (no slicing)
- Solution:
  - A set of rows that cover all days



	Mon	Wed	Fri	Sat	Sun
B	1	0	1	0	0
C	1	1	1	0	1
E	0	1	0	1	0

Solution: [D,E]

A scheduling problem

- ➔ Constraints:
  - ◆ Only 1 person per day
  - ◆ Rows picked as they are (no slicing)
- ➔ Solution:
  - ◆ A set of rows that cover all days

	Mon	Fri	Sun
B	1	1	0

Solution: [D,E,B]

A scheduling problem

- ➔ Constraints:
  - ◆ Only 1 person per day
  - ◆ Rows picked as they are (no slicing)
- ➔ Solution:
  - ◆ A set of rows that cover all days

	Mon	Fri	Sun
B	1	1	0

Solution: [D,E,B] is not a solution!

A scheduling problem

- Constraints:
  - Only 1 person per day
  - Rows picked as they are (no slicing)
- Solution:
  - A set of rows that cover all days

Backtrack!

	Mon	Fri	Sun
B	1	1	0

Solution: [D,E,~~B~~] is not a solution!

A scheduling problem

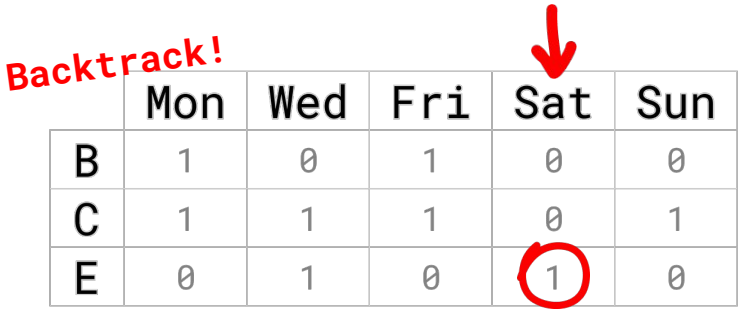
- Constraints:
  - Only 1 person per day
  - Rows picked as they are (no slicing)
- Solution:
  - A set of rows that cover all days

	Mon	Wed	Fri	Sat	Sun
B	1	0	1	0	0
C	1	1	1	0	1
E	0	1	0	1	0

Solution: [D,E]

A scheduling problem

- Constraints:
  - Only 1 person per day
  - Rows picked as they are (no slicing)
- Solution:
  - A set of rows that cover all days



	Mon	Wed	Fri	Sat	Sun
B	1	0	1	0	0
C	1	1	1	0	1
E	0	1	0	1	0

Solution: [D,E]

A scheduling problem

- Constraints:
  - Only 1 person per day
  - Rows picked as they are (no slicing)
- Solution:
  - A set of rows that cover all days

	Mon	Tue	Wed	Thu	Fri	Sat	Sun
A	1	0	0	1	0	0	1
B	1	0	0	0	1	0	0
C	1	0	1	0	1	0	1
D	0	1	0	1	0	0	0
E	0	0	1	0	0	1	0
F	0	1	0	1	0	1	0

Solution: [D]

Expanded nodes: 1

A scheduling problem

- Constraints:
  - Only 1 person per day
  - Rows picked as they are (no slicing)
- Solution:
  - A set of rows that cover all days

	Mon	Tue	Wed	Thu	Fri	Sat	Sun
A	1	0	0	1	0	0	1
B	1	0	0	0	1	0	0
C	1	0	1	0	1	0	1
D	0	1	0	1	0	0	0
E	0	0	1	0	0	1	0
F	0	1	0	1	0	1	0

Solution: [D]

Expanded nodes: 2

A scheduling problem

- Constraints:
  - Only 1 person per day
  - Rows picked as they are (no slicing)
- Solution:
  - A set of rows that cover all days

	Mon	Tue	Wed	Thu	Fri	Sat	Sun
A	1	0	0	1	0	0	1
B	1	0	0	0	1	0	0
C	1	0	1	0	1	0	1
D	0	1	0	1	0	0	0
E	0	0	1	0	0	1	0
F	0	1	0	1	0	1	0

Solution: [F]

Expanded nodes: 2

A scheduling problem

- Constraints:
  - Only 1 person per day
  - Rows picked as they are (no slicing)
- Solution:
  - A set of rows that cover all days

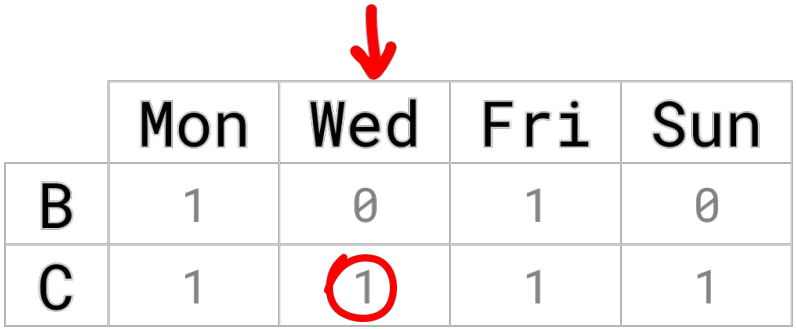
	Mon	Tue	Wed	Thu	Fri	Sat	Sun
A	1	0	0	1	0	0	1
B	1	0	0	0	1	0	0
C	1	0	1	0	1	0	1
D	0	1	0	0	0	0	0
E	0	0	1	0	0	1	0
F	0	1	0	1	0	1	0

Solution: [F]

Expanded nodes: 2

A scheduling problem

- Constraints:
  - Only 1 person per day
  - Rows picked as they are (no slicing)
- Solution:
  - A set of rows that cover all days



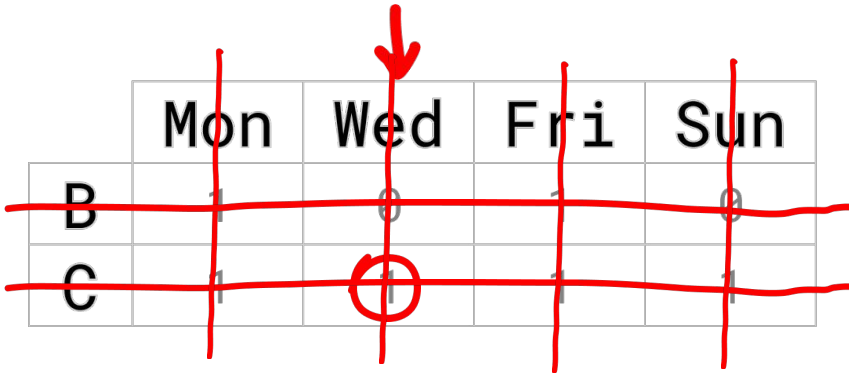
	Mon	Wed	Fri	Sun
B	1	0	1	0
C	1	1	1	1

Solution: [F]

Expanded nodes: 2

A scheduling problem

- Constraints:
  - Only 1 person per day
  - Rows picked as they are (no slicing)
- Solution:
  - A set of rows that cover all days



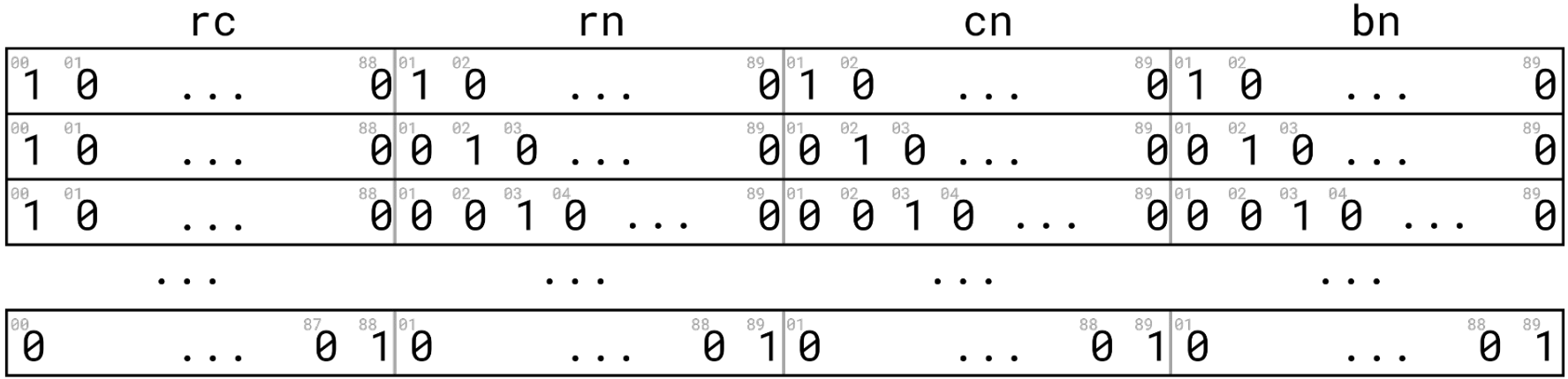
Solution: [F,C] is a solution!

Expanded nodes: 2

Sudoku, finally...

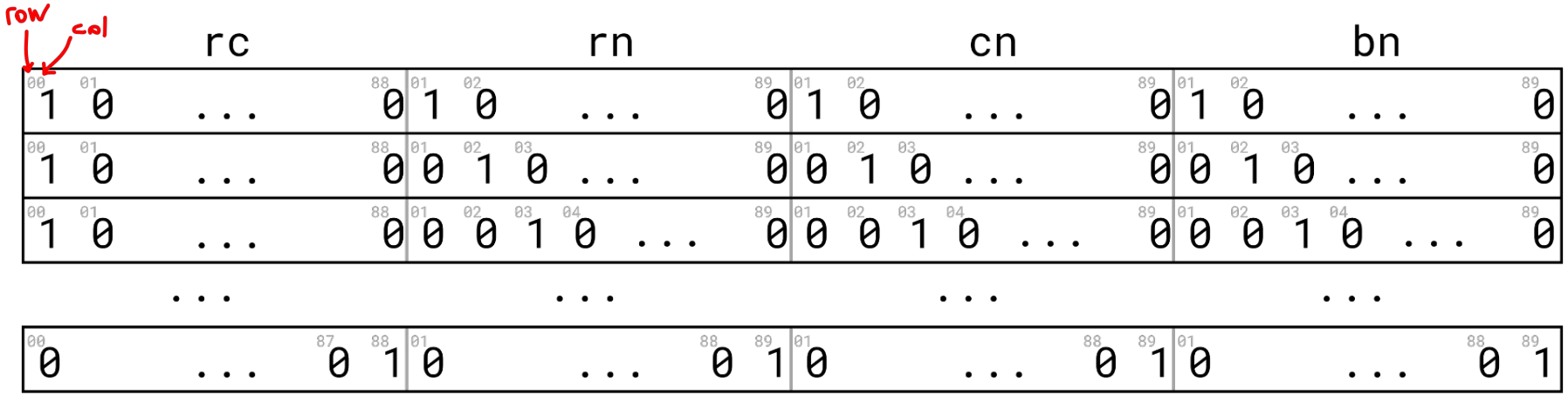
Sudoku representation

- ➔ Sudoku as a graph, adjacency matrix:
  - ◆ Easy representation of constraints



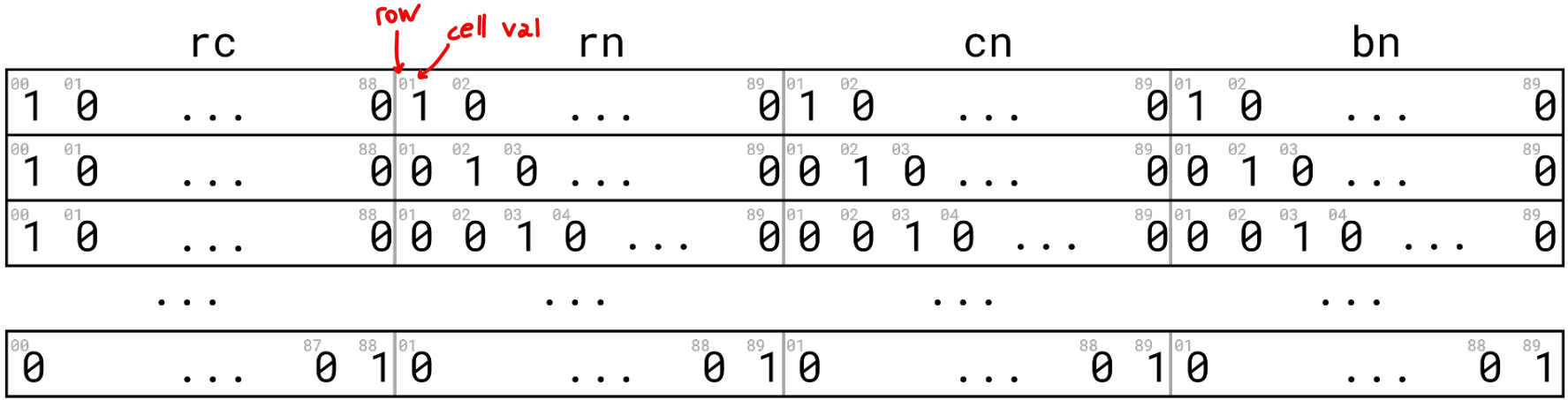
Sudoku representation

- ➔ Sudoku as a graph, adjacency matrix:
  - ◆ Easy representation of constraints



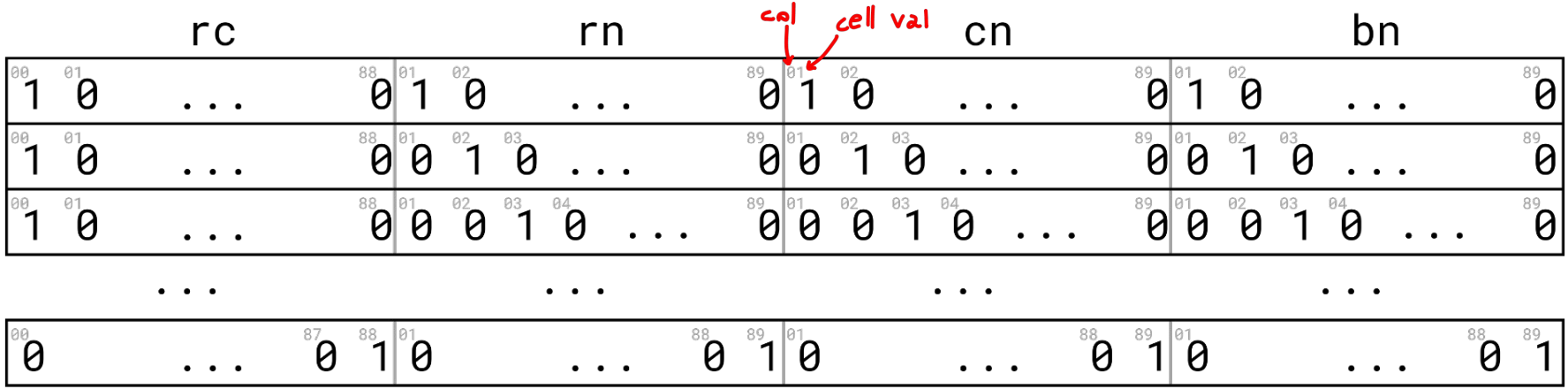
Sudoku representation

- ➔ Sudoku as a graph, adjacency matrix:
  - ◆ Easy representation of constraints



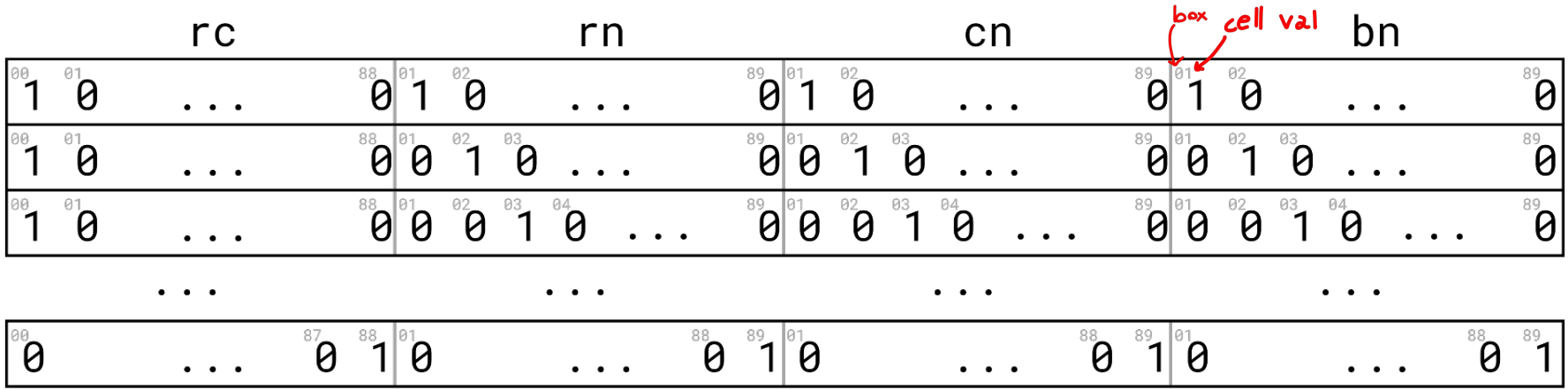
Sudoku representation

- ➔ Sudoku as a graph, adjacency matrix:
  - ◆ Easy representation of constraints



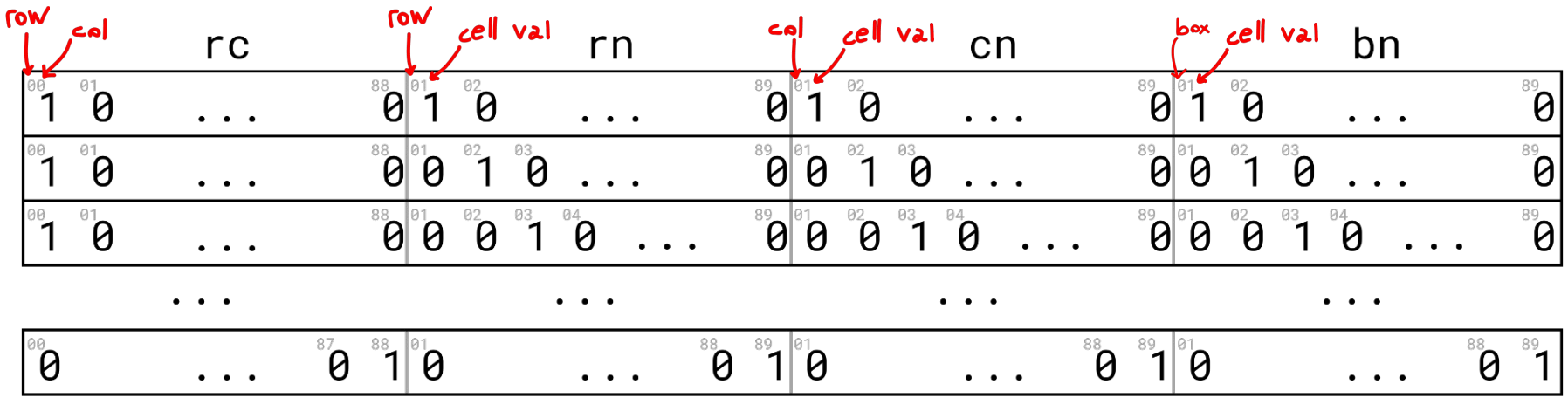
Sudoku representation

- ➔ Sudoku as a graph, adjacency matrix:
  - ◆ Easy representation of constraints



Sudoku representation

- ➔ Sudoku as a graph, adjacency matrix:
  - ◆ Easy representation of constraints



Sudoku representation

➔ Better to use an Adjacency List

```
# The frontier
possible_values = {
    # rc                  # each tuple in the set is (row, col, val)
    ('rc', (0,0)) : { (0,0,1), (0,0,2), (0,0,3), ... , (0,0,8), (0,0,9) },
    ('rc', (0,1)) : { (0,1,1), (0,1,2), (0,1,3), ... , (0,1,8), (0,1,9) },
        ...
    ('rc', (8,8)) : { (8,8,1), (8,8,2), (8,8,3), ... , (8,8,8), (8,8,9) },

    # rn
    ('rn', (0,1)) : { (0,0,1), (0,1,1), (0,2,1), ... , (0,7,1), (0,8,1) },
        ...
    ('rn', (0,9)) : { (0,0,9), (0,1,9), (0,2,9), ... , (0,7,9), (0,8,9) },
    ('rn', (1,1)) : { (1,0,1), (1,1,1), (1,2,1), ... , (1,7,1), (1,8,1) },
        ...
    ('rn', (1,9)) : { (1,0,9), (1,1,9), (1,2,9), ... , (1,7,9), (1,8,9) },
        ...
    ('rn', (8,9)) : { (8,0,9), (8,1,9), (8,2,9), ... , (8,7,9), (8,8,9) },

    # cn
    ('cn', (0,1)) : { (0,0,1), (1,0,1), (2,0,1), ... , (7,0,1), (8,0,1) },
        ...
    ('cn', (1,1)) : { (0,1,1), (1,1,1), (2,1,1), ... , (7,1,1), (8,1,1) },
        ...
    ('cn', (8,9)) : { (0,8,9), (1,8,9), (2,8,9), ... , (7,8,9), (8,8,9) },

    # bn
    ('bn', (0,1)) : { (0,0,1), (0,1,1), (0,2,1), ... , (2,1,1), (2,2,1) },
        ...
    ('bn', (0,9)) : { (0,0,9), (0,1,9), (0,2,9), ... , (2,1,9), (2,2,9) },
    ('bn', (1,1)) : { (0,3,1), (0,4,1), (0,5,1), ... , (2,4,1), (2,5,1) },
        ...
    ('bn', (1,9)) : { (0,3,9), (0,4,9), (0,5,9), ... , (2,4,9), (2,5,9) },
        ...
    ('bn', (8,9)) : { (6,6,9), (6,7,9), (6,8,9), ... , (8,7,9), (8,8,9) },
}
```

Sudoku representation

➔ Better to use an Adjacency List

```
# Links used to navigate the graph when we insert/delete values in cells
links = {
    (0,0,1) : [ ('rc', (0,0)), ('rn', (0,1)), ('cn', (0,1)), ('bn', (0,1)) ],
    (0,0,2) : [ ('rc', (0,0)), ('rn', (0,2)), ('cn', (0,2)), ('bn', (0,2)) ],
        ...
    (8,8,8) : [ ('rc', (8,8)), ('rn', (8,8)), ('cn', (8,8)), ('bn', (8,8)) ],
    (8,8,9) : [ ('rc', (8,8)), ('rn', (8,9)), ('cn', (8,9)), ('bn', (8,9)) ]
}
```

```
# The frontier
possible_values = {
    # rc                  # each tuple in the set is (row, col, val)
    ('rc', (0,0)) : { (0,0,1), (0,0,2), (0,0,3), ... , (0,0,8), (0,0,9) },
    ('rc', (0,1)) : { (0,1,1), (0,1,2), (0,1,3), ... , (0,1,8), (0,1,9) },
        ...
    ('rc', (8,8)) : { (8,8,1), (8,8,2), (8,8,3), ... , (8,8,8), (8,8,9) },

    # rn
    ('rn', (0,1)) : { (0,0,1), (0,1,1), (0,2,1), ... , (0,7,1), (0,8,1) },
        ...
    ('rn', (0,9)) : { (0,0,9), (0,1,9), (0,2,9), ... , (0,7,9), (0,8,9) },
    ('rn', (1,1)) : { (1,0,1), (1,1,1), (1,2,1), ... , (1,7,1), (1,8,1) },
        ...
    ('rn', (1,9)) : { (1,0,9), (1,1,9), (1,2,9), ... , (1,7,9), (1,8,9) },
        ...
    ('rn', (8,9)) : { (8,0,9), (8,1,9), (8,2,9), ... , (8,7,9), (8,8,9) },

    # cn
    ('cn', (0,1)) : { (0,0,1), (1,0,1), (2,0,1), ... , (7,0,1), (8,0,1) },
        ...
    ('cn', (1,1)) : { (0,1,1), (1,1,1), (2,1,1), ... , (7,1,1), (8,1,1) },
        ...
    ('cn', (8,9)) : { (0,8,9), (1,8,9), (2,8,9), ... , (7,8,9), (8,8,9) },

    # bn
    ('bn', (0,1)) : { (0,0,1), (0,1,1), (0,2,1), ... , (2,1,1), (2,2,1) },
        ...
    ('bn', (0,9)) : { (0,0,9), (0,1,9), (0,2,9), ... , (2,1,9), (2,2,9) },
    ('bn', (1,1)) : { (0,3,1), (0,4,1), (0,5,1), ... , (2,4,1), (2,5,1) },
        ...
    ('bn', (1,9)) : { (0,3,9), (0,4,9), (0,5,9), ... , (2,4,9), (2,5,9) },
        ...
    ('bn', (8,9)) : { (6,6,9), (6,7,9), (6,8,9), ... , (8,7,9), (8,8,9) },
}
```

Sudoku algorithm

```
    # Initialization (transform the 2D array into a graph via adjacency lists)
    links = {}
    for row, col, n in product(range(9), range(9), range(1, 9+1)):
        box = (row//3)*3 + (col//3)
        links[(row, col, n)] = [('rc', (row, col)),
                                ('rn', (row, n)),
                                ('cn', (col, n)),
                                ('bn', (box, n))]

    possible_values = ([('rc', x) for x in product(range(9), range(9))] +
                       [('rn', x) for x in product(range(9), range(1, 9+1))] +
                       [('cn', x) for x in product(range(9), range(1, 9+1))] +
                       [('bn', x) for x in product(range(9), range(1, 9+1))])

    possible_values = { k : set() for k in possible_values }
    for i, row in links.items(): # i is the key, row is the corresp value of the dict
        for j in row:
            possible_values[j].add(i)

    # Trim illegal moves based on initial cells values
    for r, row in enumerate(sudoku):
        for c, n in enumerate(row):
            if n:
                trim(possible_values, links, (r,c,n))
```

Sudoku algorithm

```
# Initialization (transform the 2D array into a graph via adjacency lists)
links = {}
for row, col, n in product(range(9), range(9), range(1, 9+1)):
```

```
# Start searching for a solution
num_nodes = [1]

solution = search(possible_values, links, [], num_nodes)
```

```
# Trim illegal moves based on initial cells values
for r, row in enumerate(sudoku):
    for c, n in enumerate(row):
        if n:
            trim(possible_values, links, (r,c,n))
```

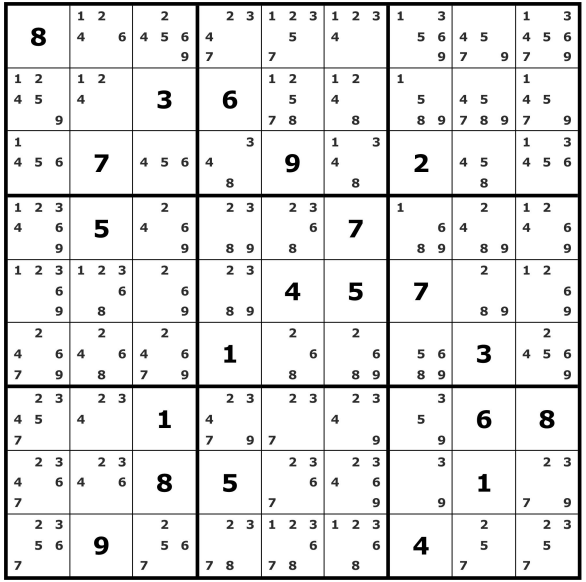
Sudoku algorithm

```python
def search(possible_values, links, solution, num_nodes):

    # If we have run out of possibilites we could have found a solution
    if not possible_values:
        return solution
    else:
        # Pick the cell with least possibilities to prevent massive branching
        col = min(possible_values, key=lambda c: len(possible_values[c]))

        for r in list(possible_values[col]):
            solution.append(r)

            # When we select a column we need to eliminate all rows intersecting
            trimmed_columns = trim(possible_values, links, r)

            s = search(possible_values, links, solution, num_nodes)
            if s:
                return s

            if expand(possible_values, links, r, trimmed_columns):
                num_nodes[0] += 1

            solution.pop()
```

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

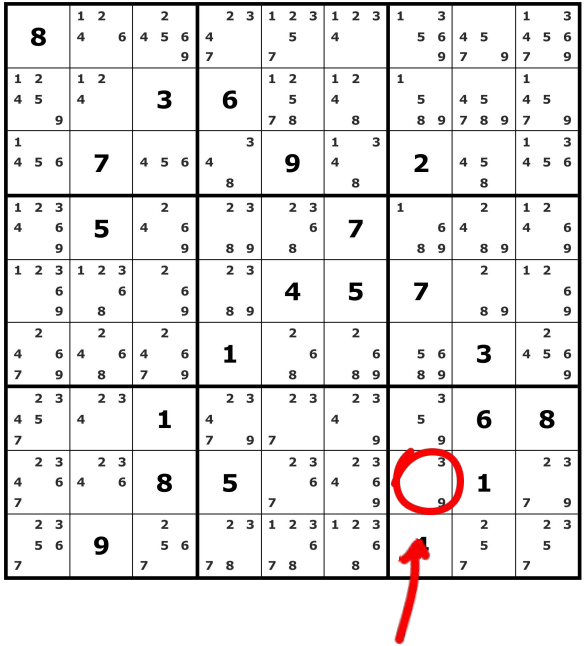
Sudoku algorithm

```python
def search(possible_values, links, solution, num_nodes):

    # If we have run out of possibilites we could have found a solution
    if not possible_values:
        return solution
    else:
        # Pick the cell with least possibilities to prevent massive branching
        col = min(possible_values, key=lambda c: len(possible_values[c]))

        for r in list(possible_values[col]):
            solution.append(r)

            # When we select a column we need to eliminate all rows intersecting
            trimmed_columns = trim(possible_values, links, r)

            s = search(possible_values, links, solution, num_nodes)
            if s:
                return s

            if expand(possible_values, links, r, trimmed_columns):
                num_nodes[0] += 1

            solution.pop()
```



Sudoku algorithm

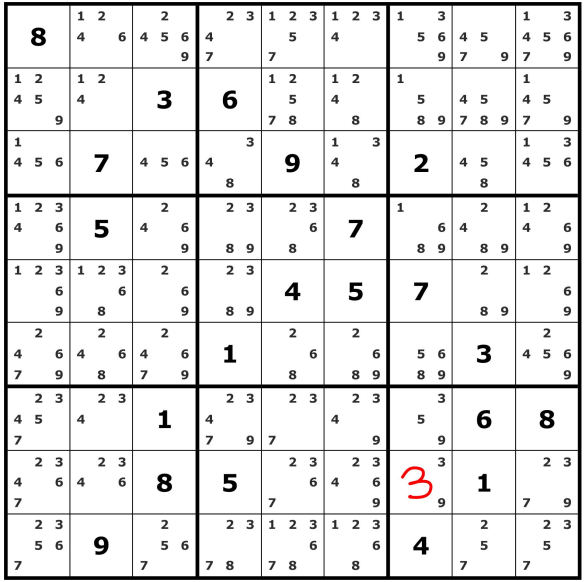
```python
def search(possible_values, links, solution, num_nodes):

    # If we have run out of possibilites we could have found a solution
    if not possible_values:
        return solution
    else:
        # Pick the cell with least possibilities to prevent massive branching
        col = min(possible_values, key=lambda c: len(possible_values[c]))

        for r in list(possible_values[col]):
            solution.append(r)

            # When we select a column we need to eliminate all rows intersecting
            trimmed_columns = trim(possible_values, links, r)

            s = search(possible_values, links, solution, num_nodes)
            if s:
                return s

            if expand(possible_values, links, r, trimmed_columns):
                num_nodes[0] += 1

            solution.pop()
```



→ Either 3 or 9
→ Both could lead to a (different) solution

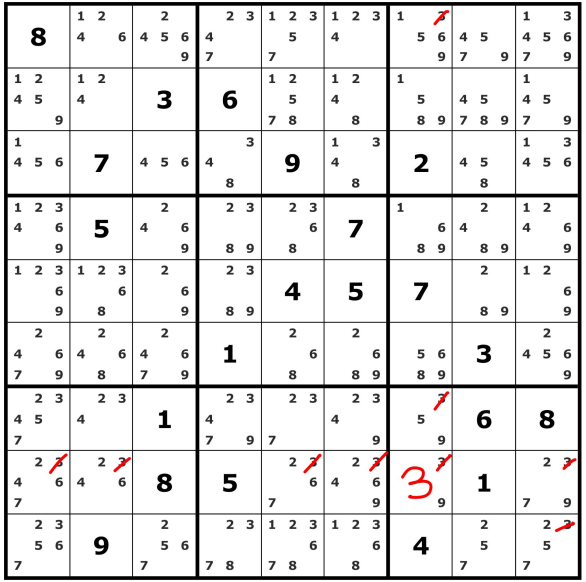
Sudoku algorithm

```python
def search(possible_values, links, solution, num_nodes):

    # If we have run out of possibilites we could have found a solution
    if not possible_values:
        return solution
    else:
        # Pick the cell with least possibilities to prevent massive branching
        col = min(possible_values, key=lambda c: len(possible_values[c]))

        for r in list(possible_values[col]):
            solution.append(r)

            # When we select a column we need to eliminate all rows intersecting
            trimmed_columns = trim(possible_values, links, r)

            s = search(possible_values, links, solution, num_nodes)
            if s:
                return s

            if expand(possible_values, links, r, trimmed_columns):
                num_nodes[0] += 1

            solution.pop()
```



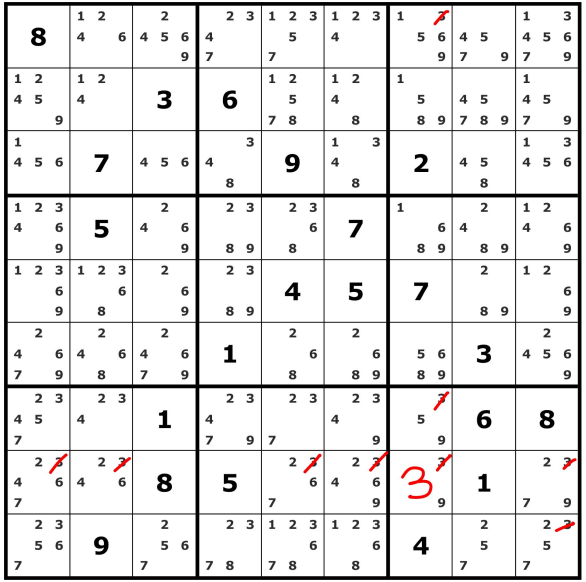
Sudoku algorithm

```python
def trim(possible_values, links, r):

    trimmed_columns = []

    for j in links[r]:
        for i in possible_values[j]:
            for k in links[i]:

                if k != j:
                    possible_values[k].remove(i)

        trimmed_columns.append(possible_values.pop(j))

    return trimmed_columns
```



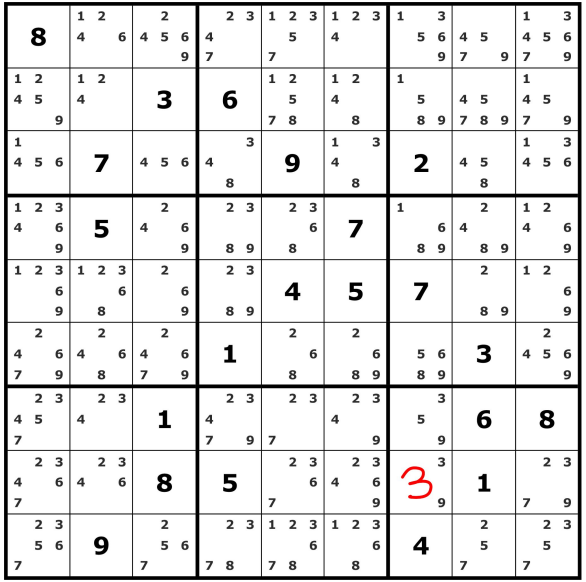
Sudoku algorithm

```python
def search(possible_values, links, solution, num_nodes):

    # If we have run out of possibilites we could have found a solution
    if not possible_values:
        return solution
    else:
        # Pick the cell with least possibilities to prevent massive branching
        col = min(possible_values, key=lambda c: len(possible_values[c]))

        for r in list(possible_values[col]):
            solution.append(r)

            # When we select a column we need to eliminate all rows intersecting
            trimmed_columns = trim(possible_values, links, r)

            s = search(possible_values, links, solution, num_nodes)
            if s:
                return s

            if expand(possible_values, links, r, trimmed_columns):
                num_nodes[0] += 1

            solution.pop()
```



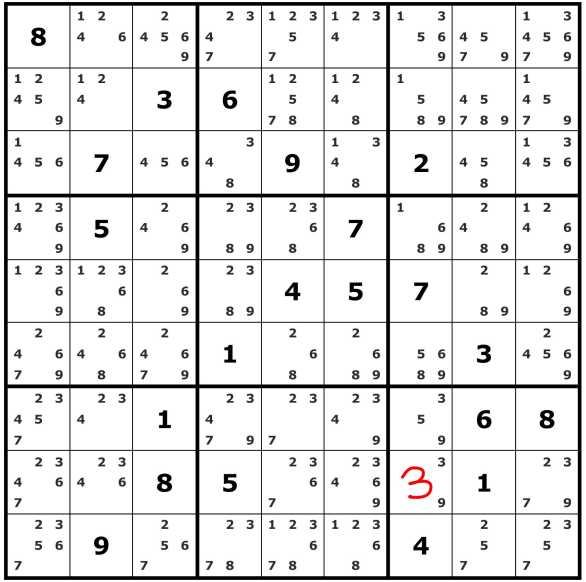
Sudoku algorithm

```python
def expand(possible_values, links, r, trimmed_columns):
    # r is a tuple: (row, col, val)
    expanded = False

    for j in reversed(links[r]):

        possible_values[j] = trimmed_columns.pop()

        for i in possible_values[j]:
            for k in links[i]:

                if k != j:
                    possible_values[k].add(i)
                    expanded = True

    return expanded
```



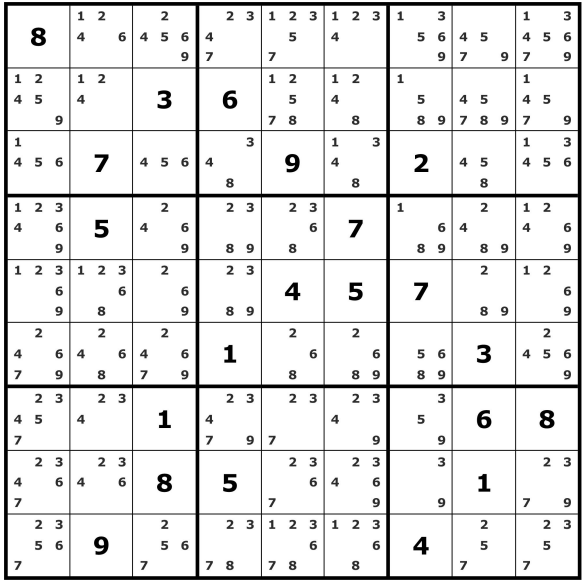
Sudoku algorithm

```python
def search(possible_values, links, solution, num_nodes):

    # If we have run out of possibilites we could have found a solution
    if not possible_values:
        return solution
    else:
        # Pick the cell with least possibilities to prevent massive branching
        col = min(possible_values, key=lambda c: len(possible_values[c]))

        for r in list(possible_values[col]):
            solution.append(r)

            # When we select a column we need to eliminate all rows intersecting
            trimmed_columns = trim(possible_values, links, r)

            s = search(possible_values, links, solution, num_nodes)
            if s:
                return s

            if expand(possible_values, links, r, trimmed_columns):
                num_nodes[0] += 1

            solution.pop()
```



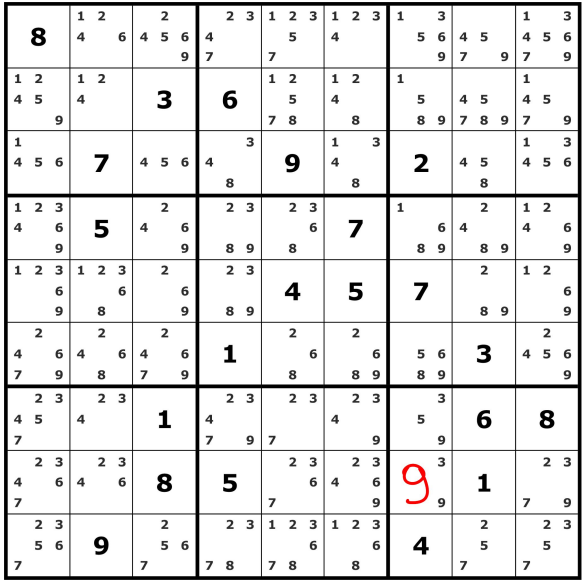
Sudoku algorithm

```python
def search(possible_values, links, solution, num_nodes):

    # If we have run out of possibilites we could have found a solution
    if not possible_values:
        return solution
    else:
        # Pick the cell with least possibilities to prevent massive branching
        col = min(possible_values, key=lambda c: len(possible_values[c]))

        for r in list(possible_values[col]):
            solution.append(r)

            # When we select a column we need to eliminate all rows intersecting
            trimmed_columns = trim(possible_values, links, r)

            s = search(possible_values, links, solution, num_nodes)
            if s:
                return s

            if expand(possible_values, links, r, trimmed_columns):
                num_nodes[0] += 1

            solution.pop()
```

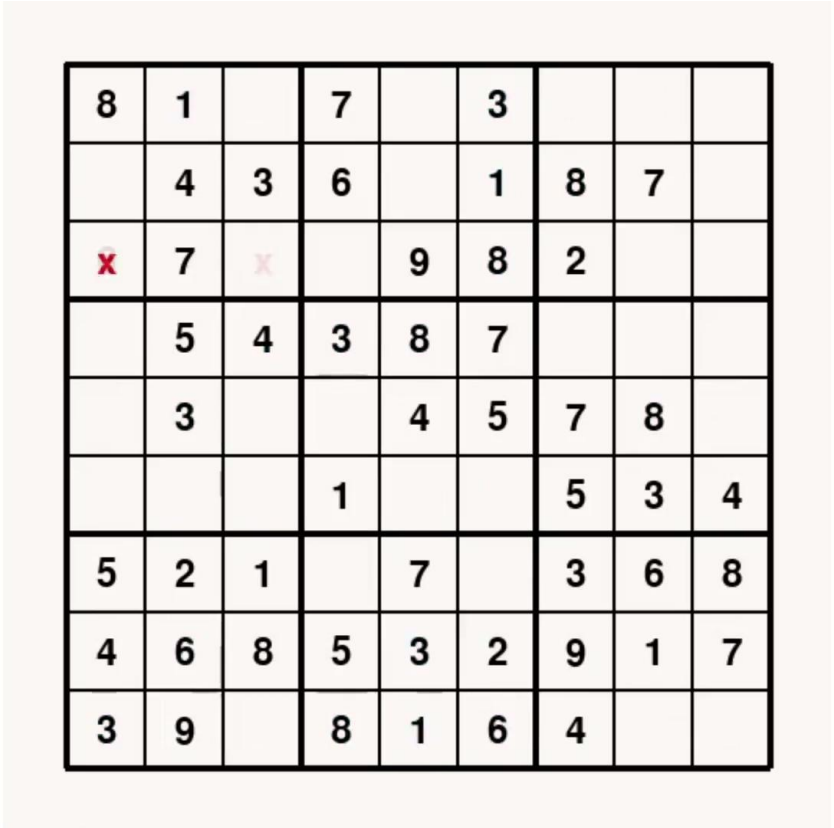


Sudoku algorithm

```python
def search(possible_values, links, solution, num_nodes):

    # If we have run out of possibilites we could have found a solution
    if not possible_values:
        return solution
    else:
        # Pick the cell with least possibilities to prevent massive branching
        col = min(possible_values, key=lambda c: len(possible_values[c]))

        for r in list(possible_values[col]):
            solution.append(r)

            # When we select a column we need to eliminate all rows intersecting
            trimmed_columns = trim(possible_values, links, r)

            s = search(possible_values, links, solution, num_nodes)
            if s:
                return s

            if expand(possible_values, links, r, trimmed_columns):
                num_nodes[0] += 1

            solution.pop()
```



Sudoku Demo


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

Thanks for your attention!

Leonardo Iurada - Computational Intelligence