Artificial Intelligence II

Lesson 3 - Constraint Satisfaction

First Code Academy

Today’s Plan

Teach Back	00 - 5 min
Constraint Satisfaction Problems	10 - 20 min
Backtracking	20 - 25 min
Quiz	25-30 min
Break	30 - 35 min
Project - N-Queens	35 - 90 min

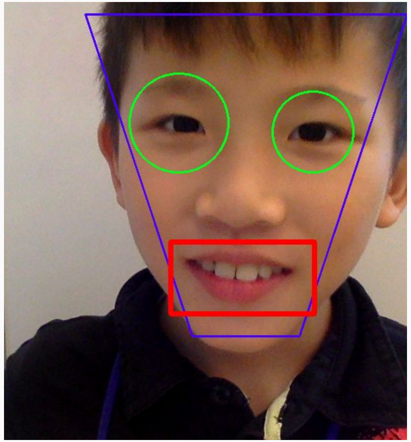
Teach Back

- ___ refers to how much resources our algorithm needs to run
- ___ are just groups of **vectors**
- ___ contain vertices which are connected to each other with edges
- ___ graphs have a weight with every edge, whereas edges in ___ graphs all have the same weight

What did we do last time?

Matrices

Learned about graphs and matrices



Original image

```
[[[12  11  15]
  [ 9  11  12]
  [ 7  12  11]
  ...
  [22  31  21]
  [21  29  18]
  [24  32  21]]

 [[12  13  17]
  [11  12  16]
  [10  14  15]
  ...
```

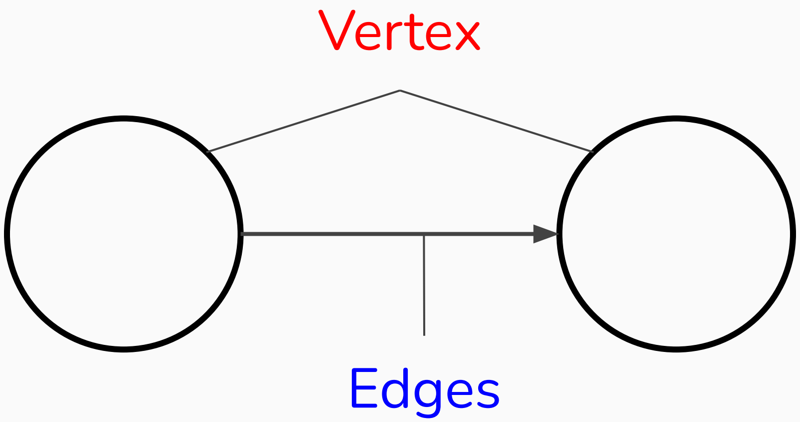
Computer’s understanding

```
[[r g b]
...
[r g b]]
```

Your computer sees images as a giant matrix of red, green and blue values, one for each pixel

Graphs

We also learned about graphs in math!



Key Terms

Constraint Satisfaction

Backtracking

Constraint Satisfaction

Constraint Satisfaction

- **Constraints** refer to conditions that limit our possible choices
- In these problems, we have a group of conditions, and we need to make a choice that satisfies all of them

Terms

We have **variables**, which are the objects that take on values

The set of values the variables can take on is called its **domain**

Example

X	X = [1, 2, 3, ...]	"x is even"
variable	domain	constraint

How to use?

1. Check which values in the variable's domain can make the constraint true
2. Update the values of the remaining variables

Example

Variables: X, Y

Domain: X = {1, 2, 3, 4, 5}, Y = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

Constraint 1: “X is even”

Constraint 2: “X + Y = 4”

Example

$X = \{0, \cancel{1}, 2, \cancel{3}, 4, \cancel{5}\}$

$Y = \{0, \cancel{1}, 2, \cancel{3}, 4, \cancel{5}, \cancel{6}, \cancel{7}, \cancel{8}, \cancel{9}\}$

“X is even”

“X + Y = 4”

Backtracking

Backtracking

In backtracking, we try and make a decision in our problem, and abandon a decision if we know it does not lead to a solution

Backtracking

Backtracking resembles DFS in that it will follow a path until we can be certain it fails.

Backtracking

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

Example

```
   S E N D
+ M O R E
-----------------
 M O N E Y
```

What values could we assign to each letter to make a valid expression?

Backtracking

Our algorithm will look something like:

```
if this node is not valid:
    Return False
If we solved the problem:
    Return True
for all the possible choices:
    Try this choice
```

Quiz: http://bit.ly/FCA_Quiz_AI

House Rules

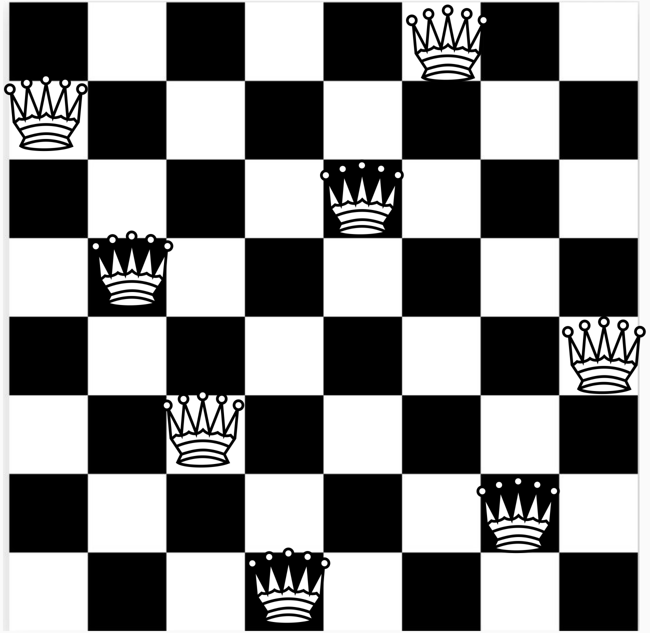

DONE IS
BETTER
THAN
PERFECT


MOVE
FAST AND
BREAK
THINGS

Project: N-Queens

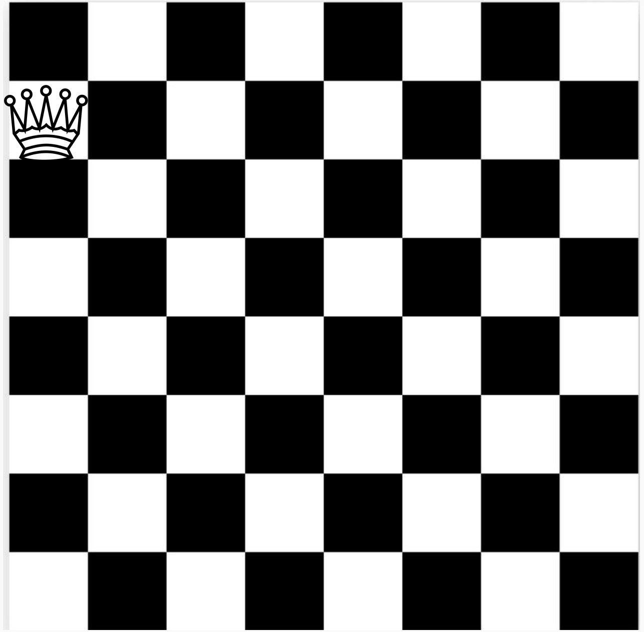
N-Queens

How can we find an arrangement of queens on a chessboard such that none of them attack each other?



N-Queens

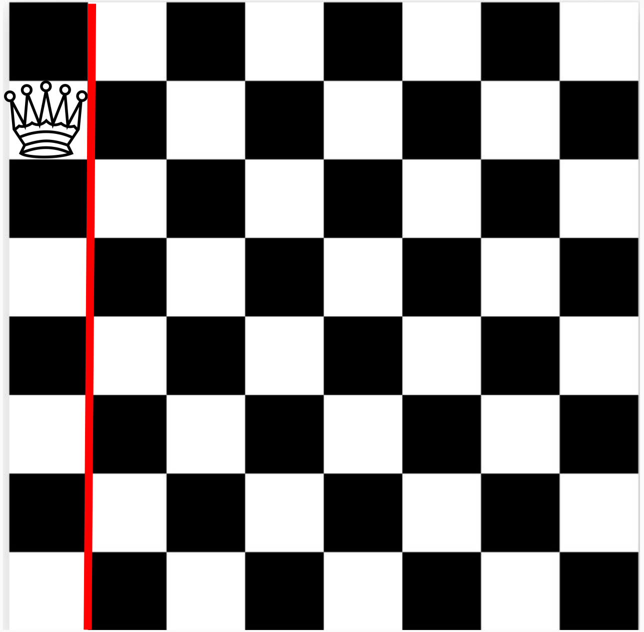
Pick a spot to place the first queen, and check if it is a valid spot



N-Queens

Then, try and solve the same problem on the smaller board, i.e. recurse!

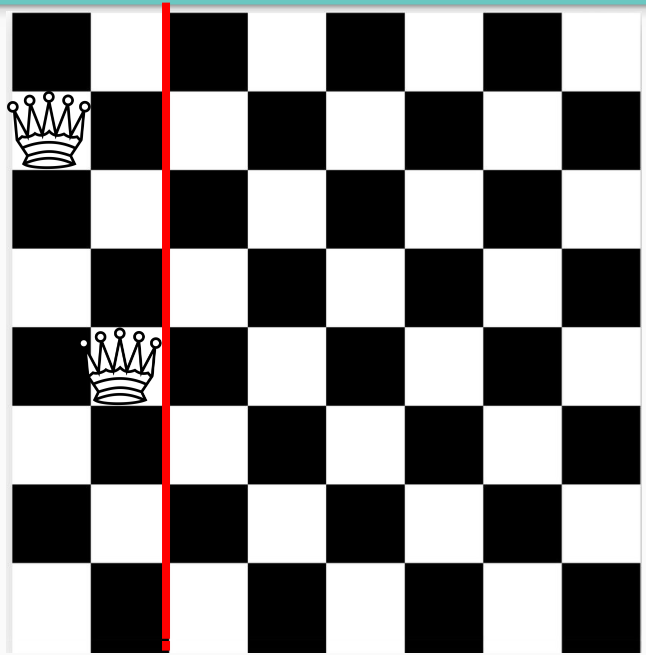
We know the queen can't be on the first column or the second row



N-Queens

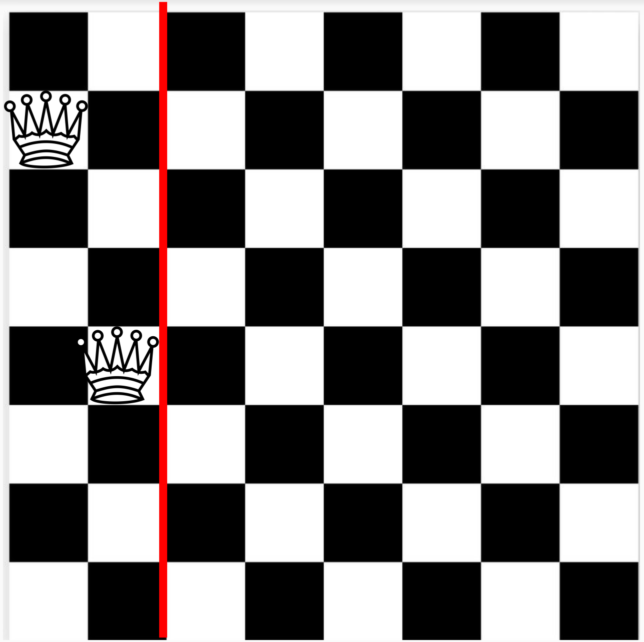
Try and place the next queen on a a valid spot in the second column

Then recurse!



N-Queens

Continue this way until the board is full!



Let’s go to our code!

Get the starter file

http://bit.ly/FCA_AI2_Starter

Create the board

We first create the board and place the first queen in a random spot in first column

```
def new_board():
    board = [['*' for x in range(BOARD_SIZE)] for y in range(BOARD_SIZE)]

    # Set a queen on random tile in first column to vary the solution
    init_queen = random.randint(0, BOARD_SIZE)
    board[init_queen][0] = 'Q'
    return board

```

Solving the board

We loop through all the values in the next column, and see if we can place a queen there

```python
    # Find the spot on the column we can place the queen
    for i in range(0, BOARD_SIZE, 1):
        if is_safe(board, i, col):

            board[i][col] = 'Q'
```

Solving the Board

If it is safe to place a queen, and placing it here leads to a solved board, return True

```
            # If we can place a queen here and solve the board, return True
            if solve_board(board, col + 1):
                return True

```

Solving the Board

If placing a queen here does not work, undo our decision and return False

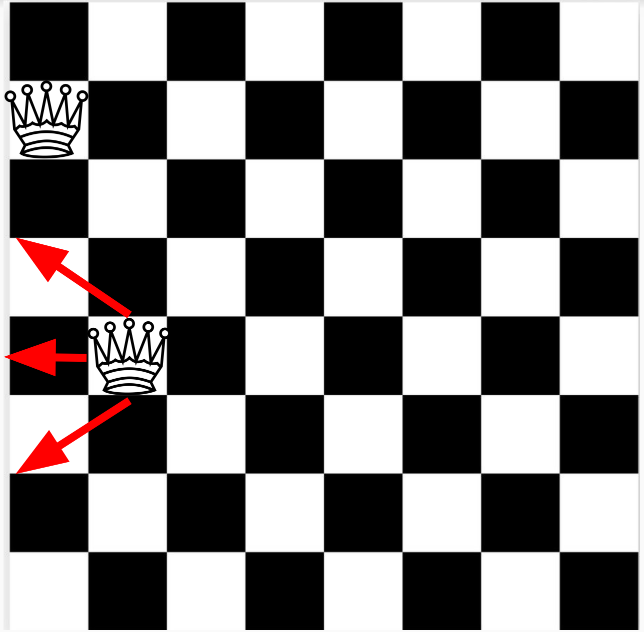
```
        # If we're here, it means we could not solve the board by placing a queen here
        board[i][col] = '*'

    return False
```

Checking if it is safe to place a queen

Checking placement

When trying to place a queen, we need to check the upper and lower diagonals.



Checking placement

Check if there is another queen along the row

```
# Check the same row on left side
for i in range(col):
    if board[row][i] == 'Q':
        return False
```

Checking placement

Next, we check if there is a queen along the upper left hand diagonal

```
    # Move along the upper left hand diagonal
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 'Q':
            return False
```

Checking placement

Lastly, we check if there is a queen along the lower left-hand diagonal

```python
    # Move along the lower left hand diagonal
    for i, j in zip(range(row, BOARD_SIZE, 1), range(col, -1, -1)):
        if board[i][j] == 'Q':
            return False
```

Try it out!

We should get a slightly different board every time

```
C:\Users\Andres Ponce\FCA_AI\FCA_AI2\L3>python L3.py

---------------------------------------
*   |  Q  |  *  |  *  |  *  |  *  |  *  |  *  |
---|-----|-----|-----|-----|-----|-----|-----|
*   |  *  |  *  |  *  |  Q  |  *  |  *  |  *  |
---|-----|-----|-----|-----|-----|-----|-----|
*   |  *  |  *  |  *  |  *  |  *  |  Q  |  *  |
---|-----|-----|-----|-----|-----|-----|-----|
Q   |  *  |  *  |  *  |  *  |  *  |  *  |  *  |
---|-----|-----|-----|-----|-----|-----|-----|
*   |  *  |  Q  |  *  |  *  |  *  |  *  |  *  |
---|-----|-----|-----|-----|-----|-----|-----|
*   |  *  |  *  |  *  |  *  |  *  |  *  |  Q  |
---|-----|-----|-----|-----|-----|-----|-----|
*   |  *  |  *  |  *  |  *  |  Q  |  *  |  *  |
---|-----|-----|-----|-----|-----|-----|-----|
*   |  *  |  *  |  Q  |  *  |  *  |  *  |  *  |
---|-----|-----|-----|-----|-----|-----|-----|
```

Challenges

Currently our program will scan a column from top to bottom to pick a suitable position for our queen.

Could we choose a random position from each column to check?

That’s it for today!

Artificial Intelligence II

Lesson 3 - Constraint Satisfaction

First Code Academy - Creator