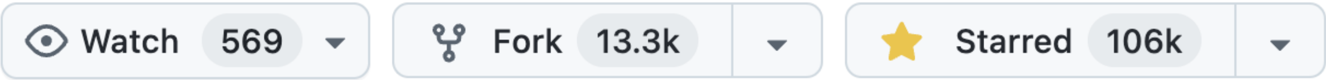
Algorithm Techniques

Constraints Satisfaction Problem and Backtracking

Xian Su

Recommended Reference Material:

- Github Open Source Project: [Hello Algo]
- Github Repo: [krahets/hello-algo]
- Animated Illustrations
- Support More than 10 Programming Languages
- Programming, Data Structure, Algorithm, and Algorithm Techniques



0 - Main Goals in This Topic

1. Constraint Satisfaction Problems

   Understand the definition and characteristics of constraint satisfaction problems (CSPs)

2. Backtracking

   Grasp how backtracking serves as a fundamental approach for CSPs

3. Learn how to enhance backtracking with techniques like constraint propagation

4. Complexity Analysis

   Analyze **time** and **space complexity** through sample code or pseudo-code.

5. Be able to apply these ideas to classic puzzles like Sudoku, N-Queens, Eight Numbers in Cross-shape boards, etc.

What is Constraint Satisfaction Problem?

A CSP is typically defined by:

1. A set of **variables**
2. A set of **domains** (one domain for each variable)
3. A set of **constraints** among the variables

The goal is to assign values to each variable so that **all constraints** are satisfied.

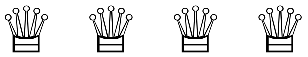
2 - N-Queens Problem

What is Constraint Satisfaction Problem?

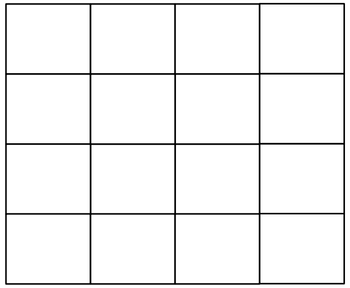
A CSP is typically defined by:

1. A set of **variables**
2. A set of **domains** (one domain for each variable)
3. A set of **constraints** among the variables

The goal is to assign values to each variable so that **all constraints** are satisfied.

Example: Given 4 queens 

How to model this question to CSP?

and one 4*4 board 

Place all Queens on the board, and
no two queens threaten each other.

2 - N-Queens Problem

What is Constraint Satisfaction Problem?

A CSP is typically defined by:

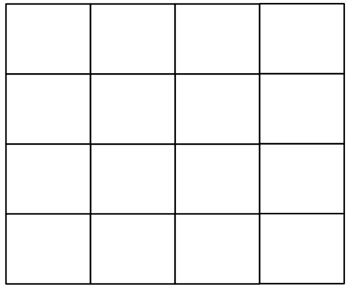
1. A set of **variables**
2. A set of **domains** (one domain for each variable)
3. A set of **constraints** among the variables

The goal is to assign values to each variable so that **all constraints** are satisfied.

Example: Given 4 queens ♕ ♕ ♕ ♕

How to model this question to CSP?

and one 4*4 board



constraints -> "no threaten to each other"

Place all Queens on the board, and
no two queens threaten each other.

What is Constraint Satisfaction Problem?

A CSP is typically defined by:

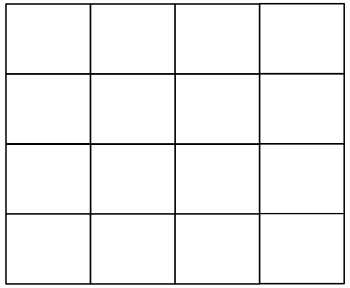
1. A set of **variables**
2. A set of **domains** (one domain for each variable)
3. A set of **constraints** among the variables

The goal is to assign values to each variable so that **all constraints** are satisfied.

Example: Given 4 queens ♕ ♕ ♕ ♕

How to model this question to CSP?

and one 4*4 board



constraints -> "no threaten to each other"

variables -> four queens: q_1, q_2, q_3, q_4

Place all Queens on the board, and no two queens threaten each other.

2 - N-Queens Problem

What is Constraint Satisfaction Problem?

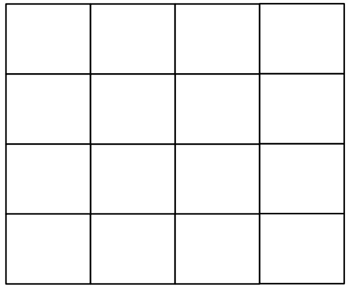
A CSP is typically defined by:

1. A set of **variables**
2. A set of **domains** (one domain for each variable)
3. A set of **constraints** among the variables

The goal is to assign values to each variable so that **all constraints** are satisfied.

Example: Given 4 queens ♕ ♕ ♕ ♕

and one 4*4 board



Place all Queens on the board, and no two queens threaten each other.

How to model this question to CSP?

constraints -> "no threaten to each other"

variables -> four queens: q_1, q_2, q_3, q_4

domains -> ???

2 - N-Queens Problem

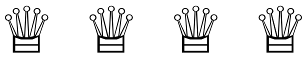
What is Constraint Satisfaction Problem?

A CSP is typically defined by:

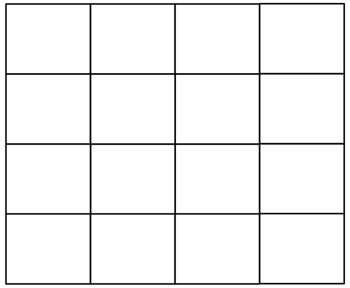
1. A set of **variables**
2. A set of **domains** (one domain for each variable)
3. A set of **constraints** among the variables

The goal is to assign values to each variable so that **all constraints** are satisfied.

Example: Given 4 queens

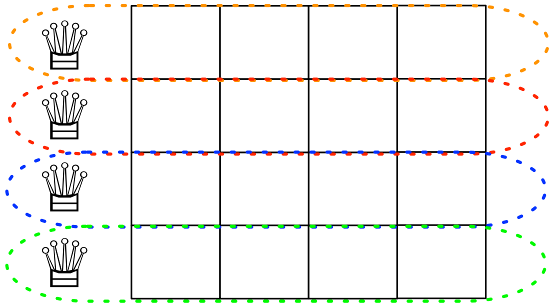


and one 4*4 board



Place all Queens on the board, and no two queens threaten each other.

How to model this question to CSP?



each row could be one container

-> `board[0], board[1],` …

-> board[q_1], board[q_2], …

2 - N-Queens Problem

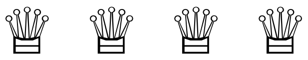
What is Constraint Satisfaction Problem?

A CSP is typically defined by:

1. A set of **variables**
2. A set of **domains** (one domain for each variable)
3. A set of **constraints** among the variables

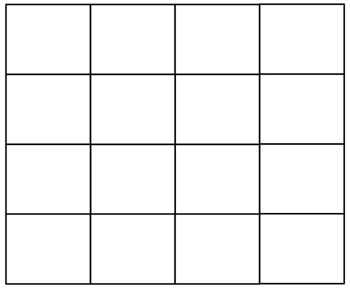
The goal is to assign values to each variable so that **all constraints** are satisfied.

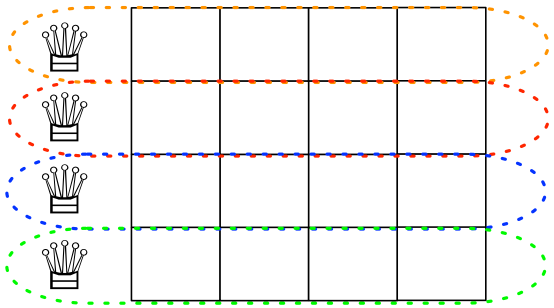
Example: Given 4 queens



How to model this question to CSP?

and one 4*4 board





$\texttt{board}[q_1], \texttt{board}[q_2], \ldots$

what is domain in this setup?

Place all Queens on the board, and no two queens threaten each other.

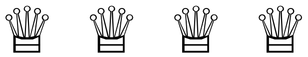
2 - N-Queens Problem

What is Constraint Satisfaction Problem?

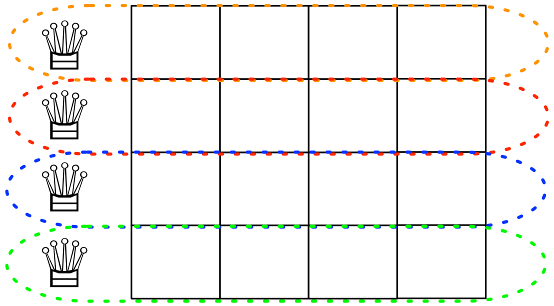
A CSP is typically defined by:

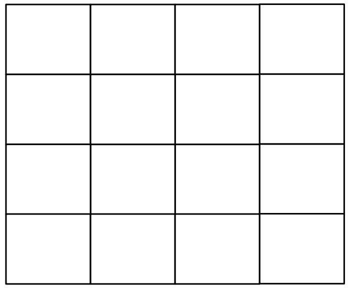
1. A set of **variables**
2. A set of **domains** (one domain for each variable)
3. A set of **constraints** among the variables

The goal is to assign values to each variable so that **all constraints** are satisfied.

Example: Given 4 queens 

How to model this question to CSP?



and one 4*4 board 

board[q_1], board[q_2], ...

Place all Queens on the board, and no two queens threaten each other.

what is domain in this setup?

The positions in each row.

2 - N-Queens Problem

What is Constraint Satisfaction Problem?

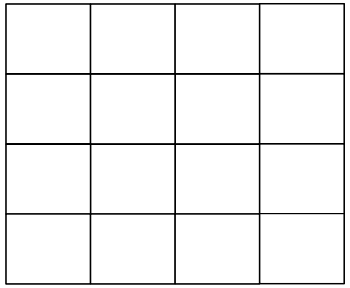
A CSP is typically defined by:

1. A set of **variables**
2. A set of **domains** (one domain for each variable)
3. A set of **constraints** among the variables

The goal is to assign values to each variable so that **all constraints** are satisfied.

Example: Given 4 queens ♕ ♕ ♕ ♕

and one 4*4 board



Place all Queens on the board, and no two queens threaten each other.

How to model this question to CSP?

constraints -> "no threaten to each other"

variables -> four queens: q_1, q_2, q_3, q_4

domains -> the index of columns in each row.

2 - N-Queens Problem

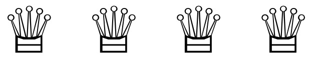
What is Constraint Satisfaction Problem?

A CSP is typically defined by:

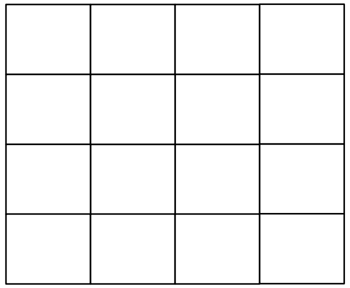
1. A set of **variables**
2. A set of **domains** (one domain for each variable)
3. A set of **constraints** among the variables

The goal is to assign values to each variable so that **all constraints** are satisfied.

Example: Given 4 queens

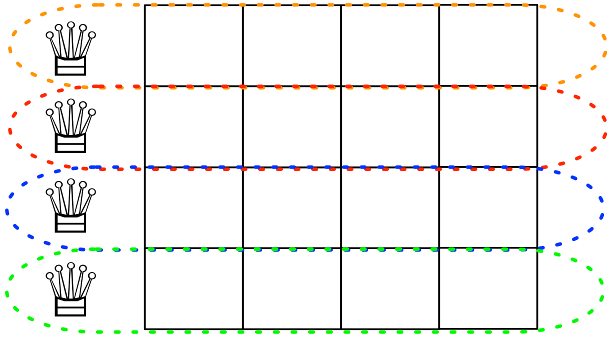


and one 4*4 board



Place all Queens on the board, and no two queens threaten each other.

What's the brute-force/naive solution?



Initialize a `board`

2 - N-Queens Problem

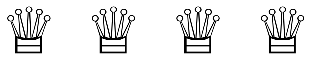
What is Constraint Satisfaction Problem?

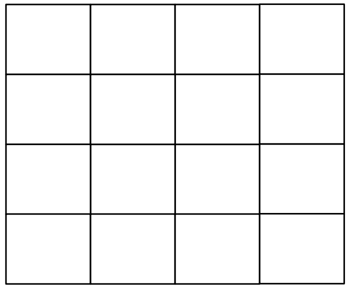
A CSP is typically defined by:

1. A set of **variables**
2. A set of **domains** (one domain for each variable)
3. A set of **constraints** among the variables

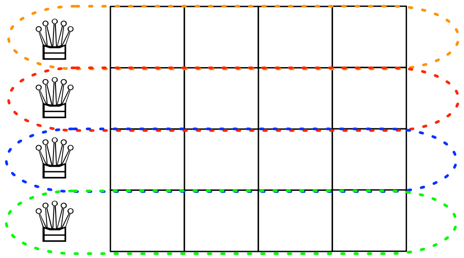
The goal is to assign values to each variable so that **all constraints** are satisfied.

What's the brute-force/naive solution?

Example: Given 4 queens 

and one 4*4 board 

Place all Queens on the board, and no two queens threaten each other.



board = [0, 0, 0, 0]

Any issues?

2 - N-Queens Problem

What is Constraint Satisfaction Problem?

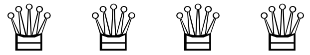
A CSP is typically defined by:

1. A set of **variables**
2. A set of **domains** (one domain for each variable)
3. A set of **constraints** among the variables

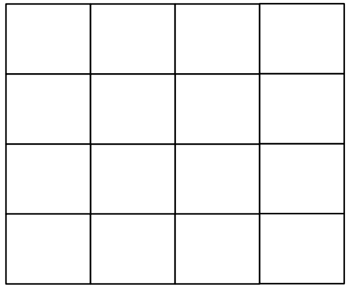
The goal is to assign values to each variable so that **all constraints** are satisfied.

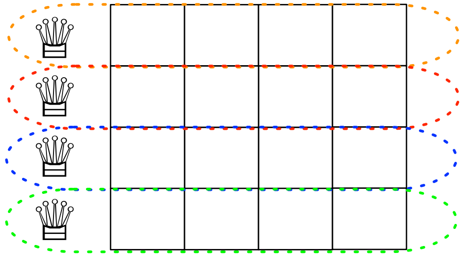
What's the brute-force/naive solution?

Example: Given 4 queens



and one 4*4 board





board = [-1, -1, -1, -1]

Place all Queens on the board, and no two queens threaten each other.

2 - N-Queens Problem

What is Constraint Satisfaction Problem?

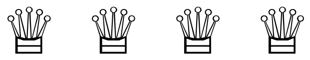
A CSP is typically defined by:

1. A set of **variables**
2. A set of **domains** (one domain for each variable)
3. A set of **constraints** among the variables

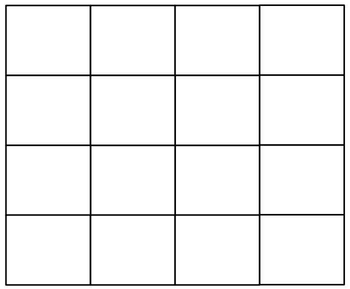
The goal is to assign values to each variable so that **all constraints** are satisfied.

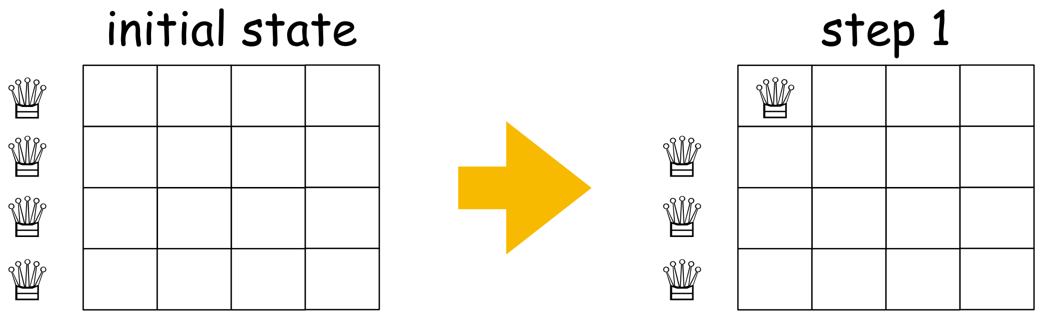
What's the brute-force/naive solution?

Example: Given 4 queens



and one 4*4 board





Place all Queens on the board, and no two queens threaten each other.

2 - N-Queens Problem

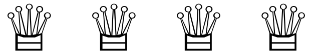
What is Constraint Satisfaction Problem?

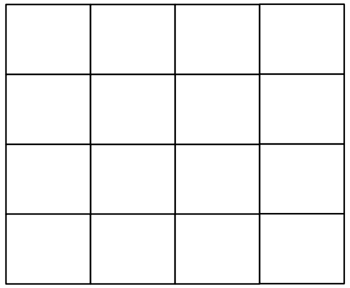
A CSP is typically defined by:

1. A set of **variables**
2. A set of **domains** (one domain for each variable)
3. A set of **constraints** among the variables

The goal is to assign values to each variable so that **all constraints** are satisfied.

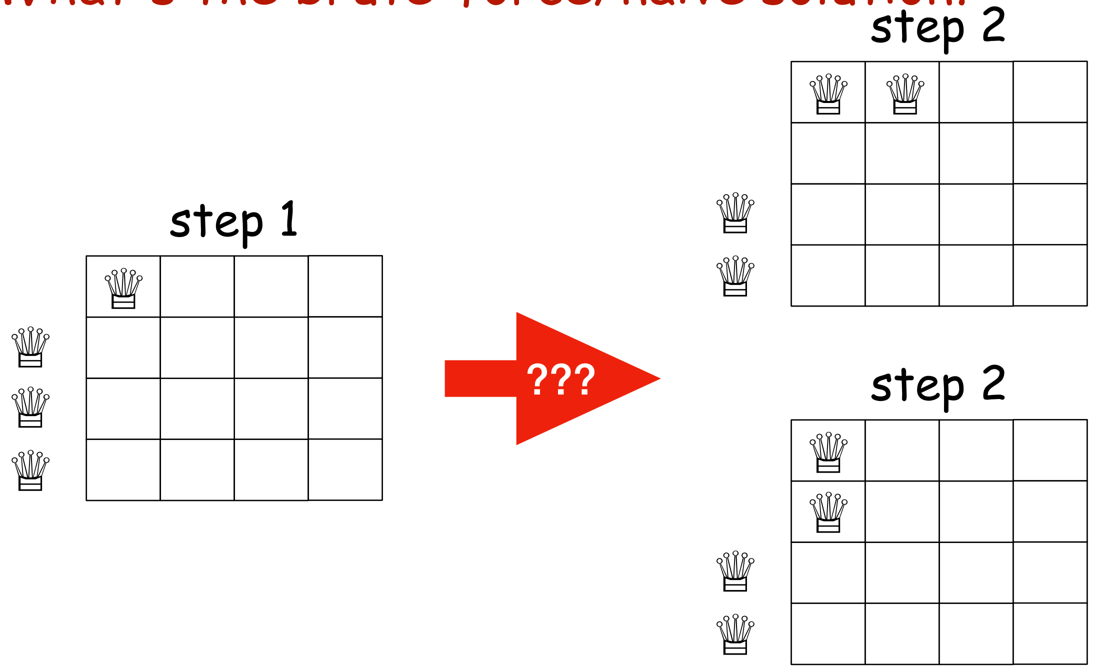
Example:

Given 4 queens 

and one 4*4 board 

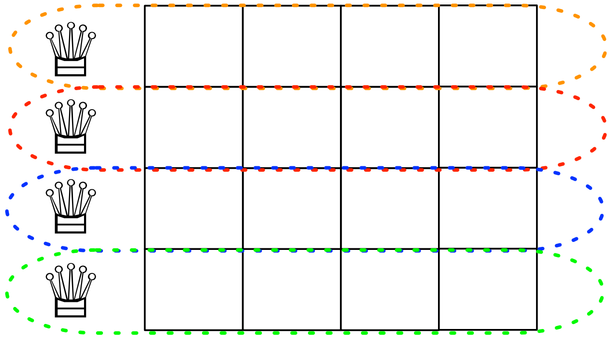
Place all Queens on the board, and no two queens threaten each other.

What's the brute-force/naive solution?



2 - N-Queens Problem

What is Constraint Satisfaction Problem?



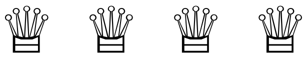
A CSP is typically defined by:

1. A set of **variables**
2. A set of **domains** (one domain for each variable)
3. A set of **constraints** among the variables

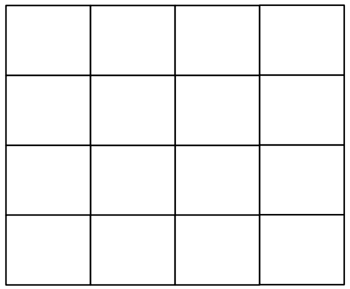
The goal is to assign values to each variable so that **all constraints** are satisfied.

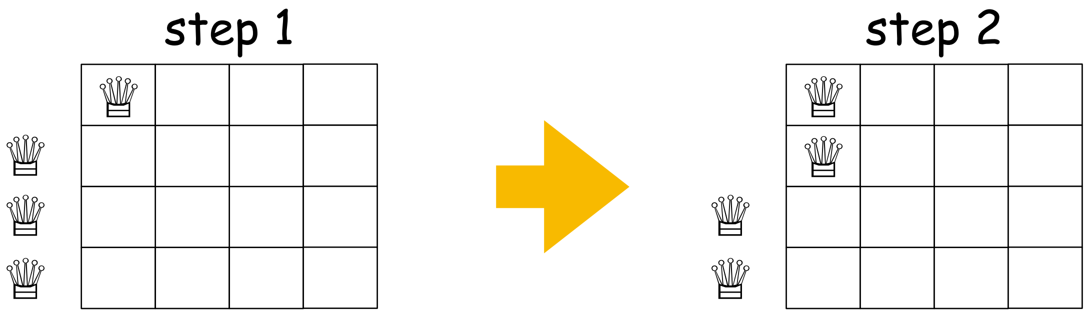
What's the brute-force/naive solution?

Example: Given 4 queens



and one 4*4 board

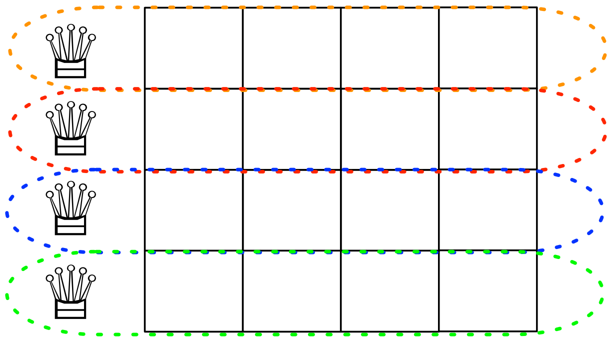




Place all Queens on the board, and no two queens threaten each other.

2 - N-Queens Problem

What is Constraint Satisfaction Problem?

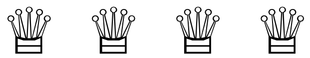


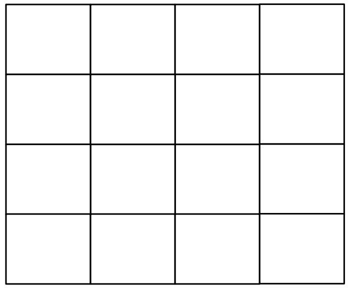
A CSP is typically defined by:

1. A set of **variables**
2. A set of **domains** (one domain for each variable)
3. A set of **constraints** among the variables

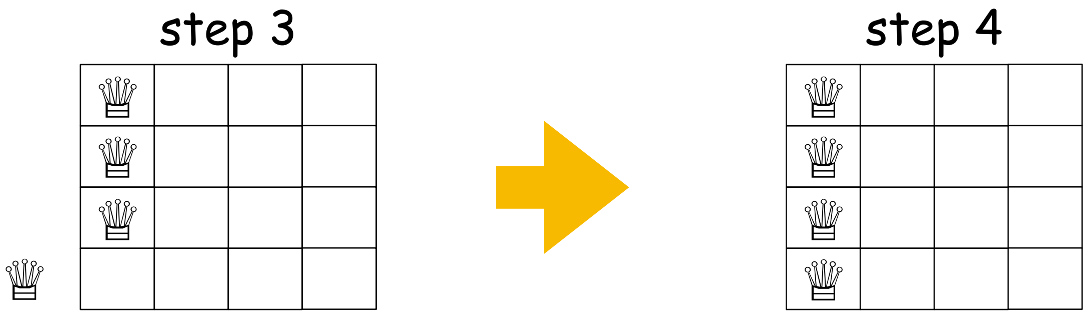
The goal is to assign values to each variable so that **all constraints** are satisfied.

What's the brute-force/naive solution?

Example: Given 4 queens 

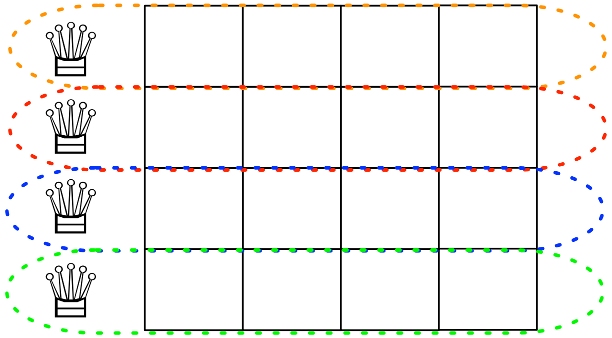
and one 4*4 board 

Place all Queens on the board, and no two queens threaten each other.



2 - N-Queens Problem

What is Constraint Satisfaction Problem?



A CSP is typically defined by:

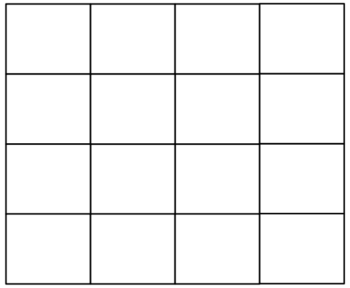
1. A set of **variables**
2. A set of **domains** (one domain for each variable)
3. A set of **constraints** among the variables

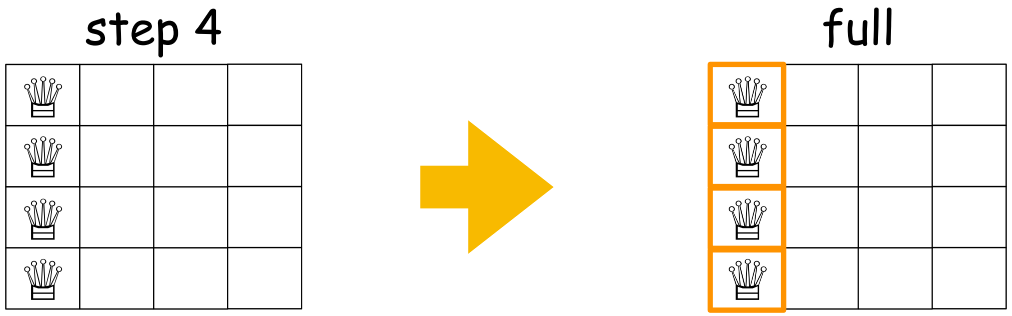
The goal is to assign values to each variable so that **all constraints** are satisfied.

Example: Given 4 queens ♛ ♛ ♛ ♛

What should we do once we placed all ♛ ?

and one 4*4 board

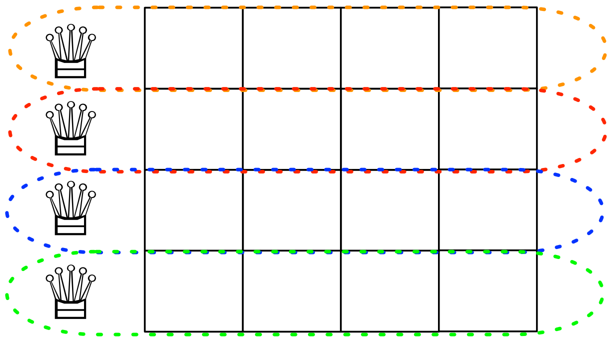




Place all Queens on the board, and no two queens threaten each other.

2 - N-Queens Problem

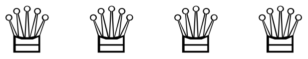
What is Constraint Satisfaction Problem?



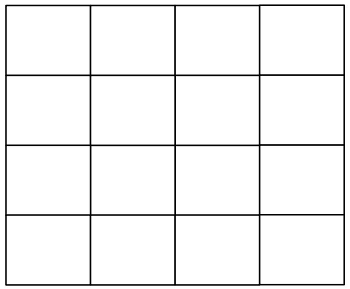
A CSP is typically defined by:

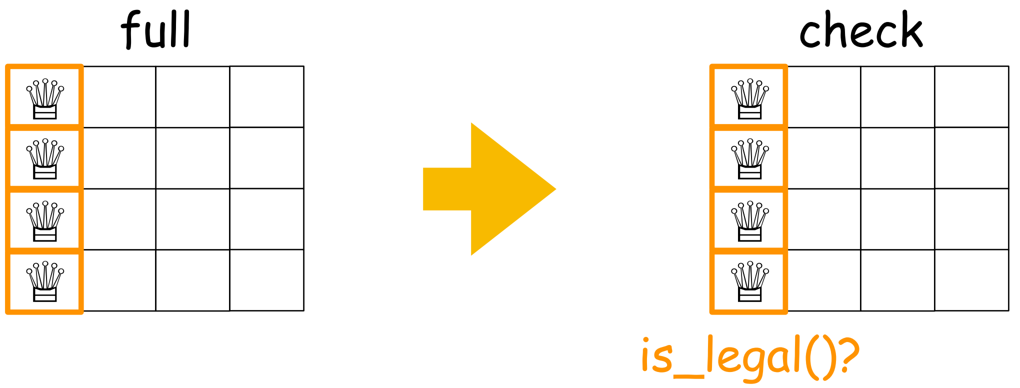
1. A set of **variables**
2. A set of **domains** (one domain for each variable)
3. A set of **constraints** among the variables

The goal is to assign values to each variable so that **all constraints** are satisfied.

Example: Given 4 queens 

What should we do once we placed all ♕ ?

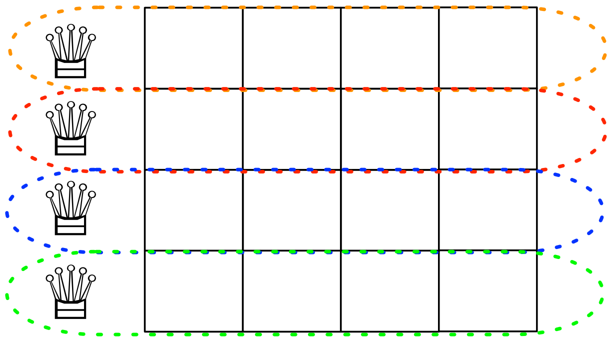
and one 4*4 board 



Place all Queens on the board, and no two queens threaten each other.

2 - N-Queens Problem

What is Constraint Satisfaction Problem?



A CSP is typically defined by:

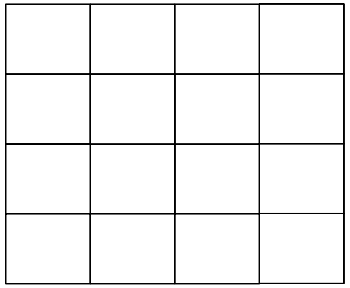
1. A set of **variables**
2. A set of **domains** (one domain for each variable)
3. A set of **constraints** among the variables

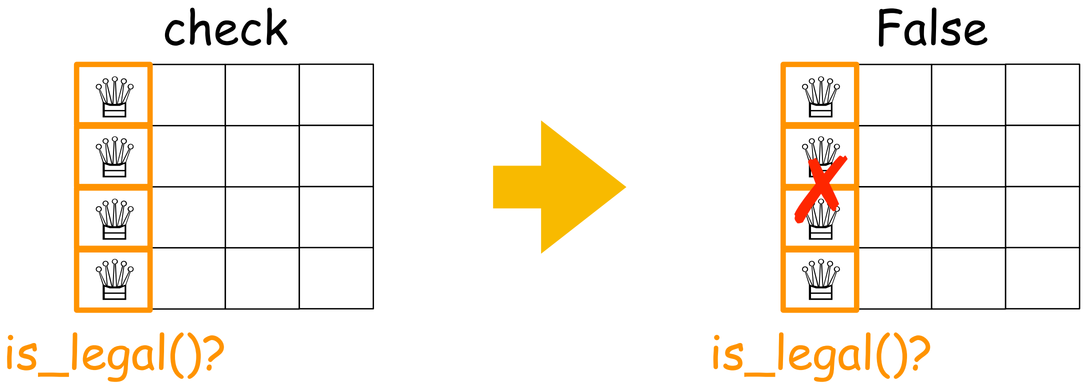
The goal is to assign values to each variable so that **all constraints** are satisfied.

Example: Given 4 queens ♛ ♛ ♛ ♛

What should we do once we placed all ♛ ?

and one 4*4 board

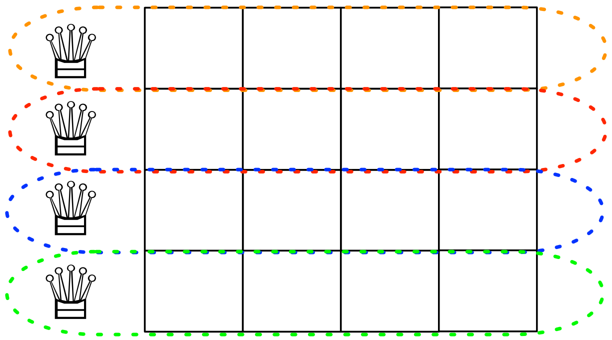




Place all Queens on the board, and no two queens threaten each other.

2 - N-Queens Problem

What is Constraint Satisfaction Problem?

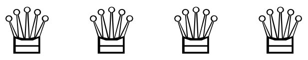


A CSP is typically defined by:

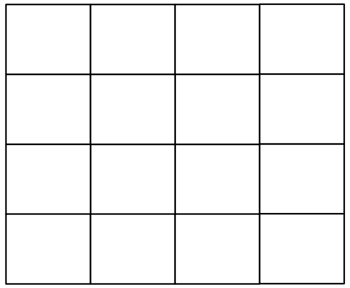
1. A set of **variables**
2. A set of **domains** (one domain for each variable)
3. A set of **constraints** among the variables

The goal is to assign values to each variable so that **all constraints** are satisfied.

Example: Given 4 queens

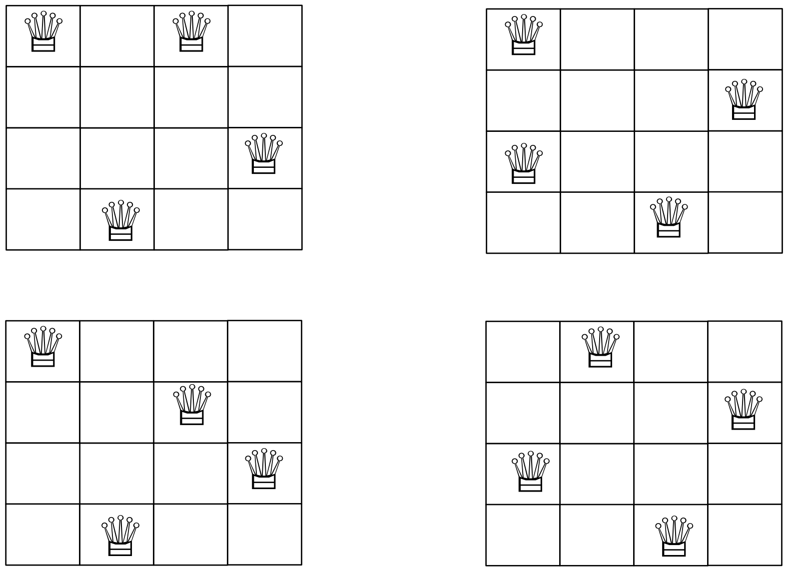


and one 4*4 board



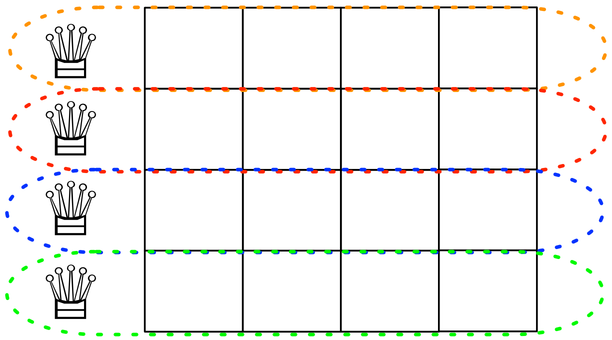
Place all Queens on the board, and no two queens threaten each other.

How to complete `is_legal()`?



2 - N-Queens Problem

What is Constraint Satisfaction Problem?

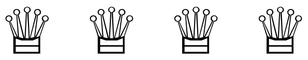


A CSP is typically defined by:

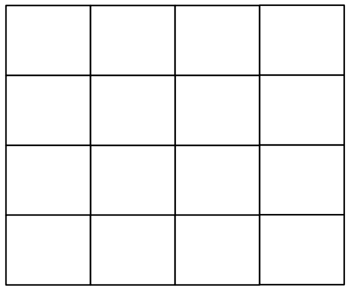
1. A set of **variables**
2. A set of **domains** (one domain for each variable)
3. A set of **constraints** among the variables

The goal is to assign values to each variable so that **all constraints** are satisfied.

Example: Given 4 queens

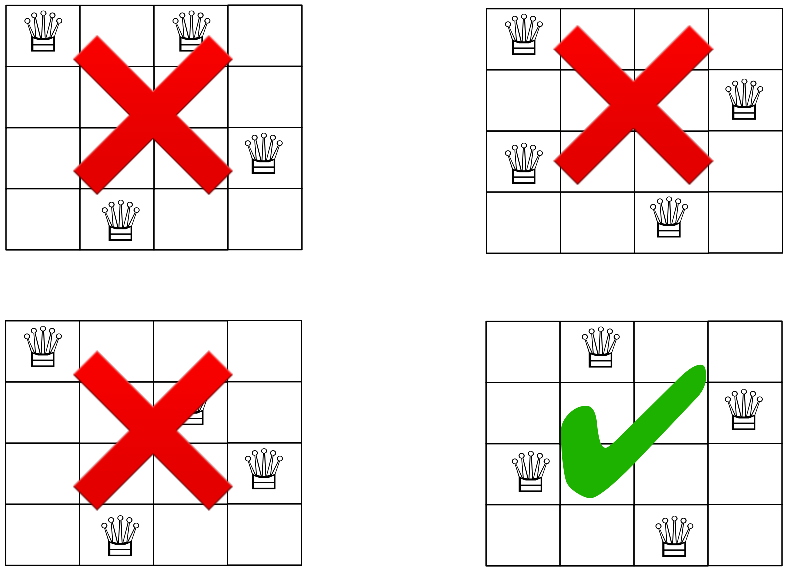


and one 4*4 board



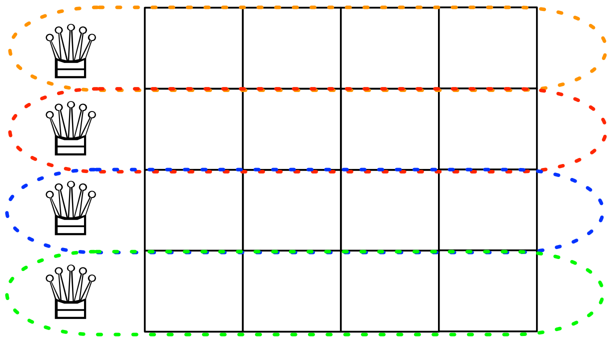
Place all Queens on the board, and no two queens threaten each other.

How to complete `is_legal()`?



2 - N-Queens Problem

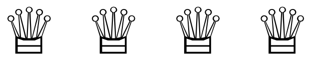
What is Constraint Satisfaction Problem?

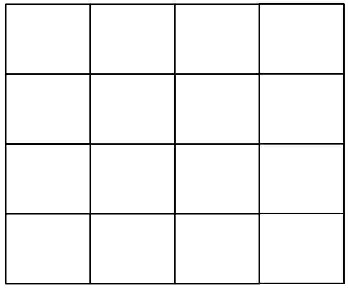


A CSP is typically defined by:

1. A set of **variables**
2. A set of **domains** (one domain for each variable)
3. A set of **constraints** among the variables

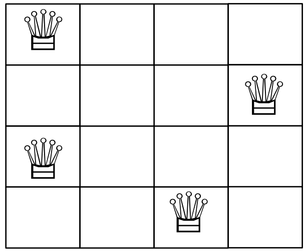
The goal is to assign values to each variable so that **all constraints** are satisfied.

Example: Given 4 queens 

and one 4*4 board 

Place all Queens on the board, and no two queens threaten each other.

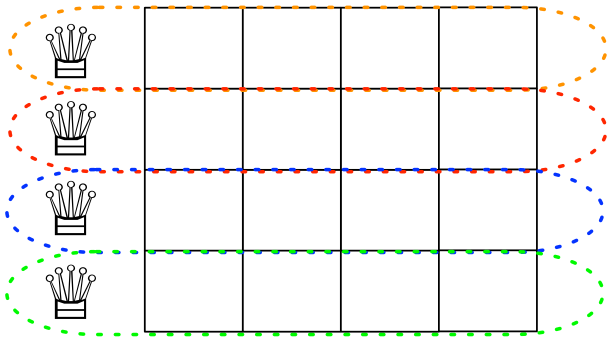
How to complete `is_legal()`?



Why it's in conflict?

2 - N-Queens Problem

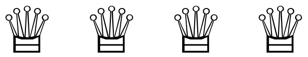
What is Constraint Satisfaction Problem?

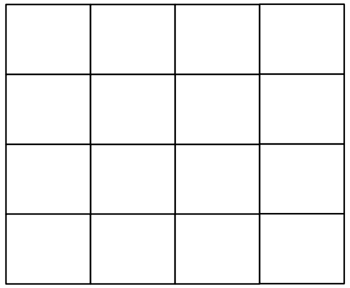


A CSP is typically defined by:

1. A set of **variables**
2. A set of **domains** (one domain for each variable)
3. A set of **constraints** among the variables

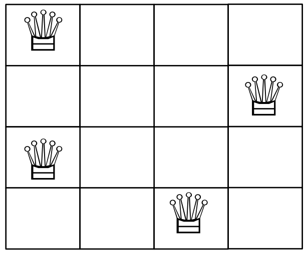
The goal is to assign values to each variable so that **all constraints** are satisfied.

Example: Given 4 queens 

and one 4*4 board 

Place all Queens on the board, and no two queens threaten each other.

How to complete `is_legal()`?

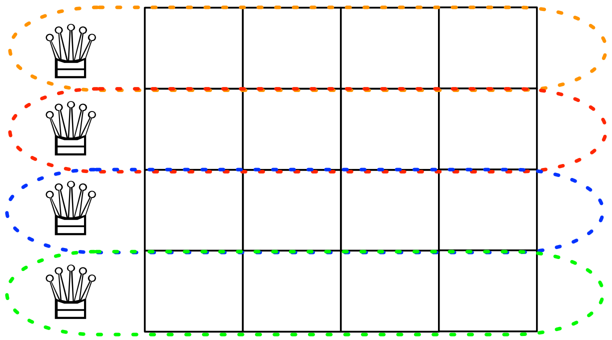


Why it's in conflict?

`board[`q_1`]  ==  board[`q_3`]`

2 - N-Queens Problem

What is Constraint Satisfaction Problem?



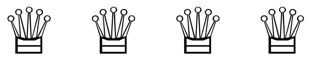
A CSP is typically defined by:

1. A set of **variables**
2. A set of **domains** (one domain for each variable)
3. A set of **constraints** among the variables

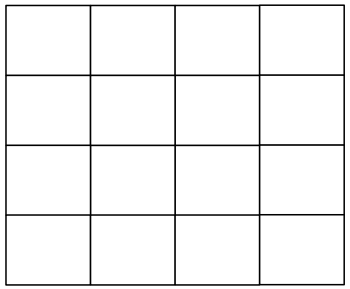
The goal is to assign values to each variable so that **all constraints** are satisfied.

Example:

Given 4 queens

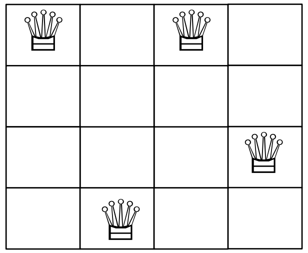


and one 4*4 board



Place all Queens on the board, and no two queens threaten each other.

How to complete `is_legal()`?

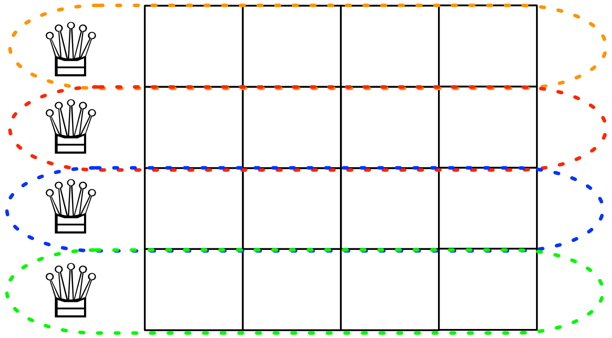


Why it's in conflict?

`No need to check! Impossible.`

2 - N-Queens Problem

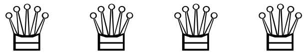
What is Constraint Satisfaction Problem?

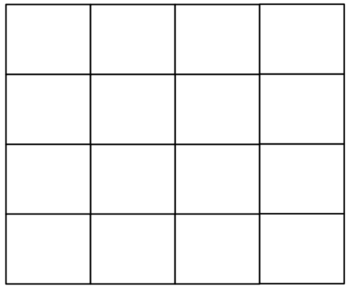


A CSP is typically defined by:

1. A set of **variables**
2. A set of **domains** (one domain for each variable)
3. A set of **constraints** among the variables

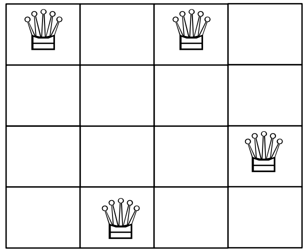
The goal is to assign values to each variable so that **all constraints** are satisfied.

Example: Given 4 queens 

and one 4*4 board 

Place all Queens on the board, and no two queens threaten each other.

How to complete `is_legal()`?

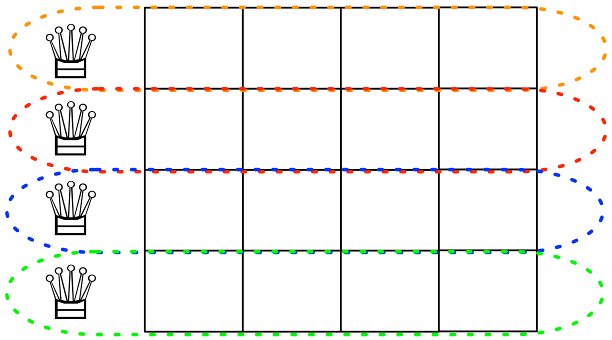


Why it's in conflict?

what will happen if we use 2D array `board`?

2 - N-Queens Problem

What is Constraint Satisfaction Problem?

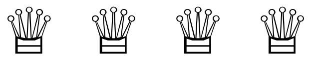


A CSP is typically defined by:

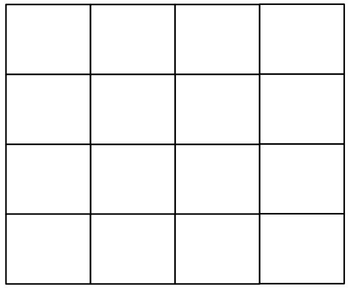
1. A set of **variables**
2. A set of **domains** (one domain for each variable)
3. A set of **constraints** among the variables

The goal is to assign values to each variable so that **all constraints** are satisfied.

Example: Given 4 queens

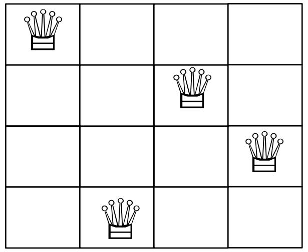


and one 4*4 board



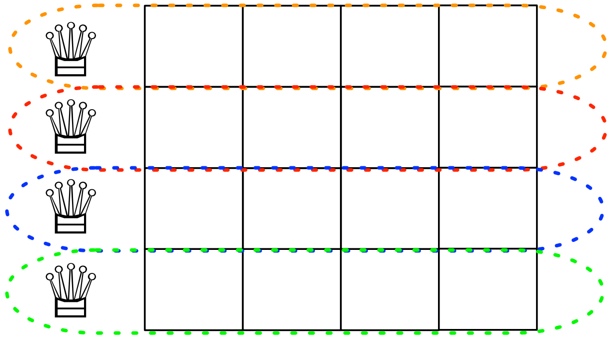
Place all Queens on the board, and no two queens threaten each other.

How to complete `is_legal()`?



Why it's in conflict?

What is Constraint Satisfaction Problem?

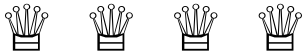


A CSP is typically defined by:

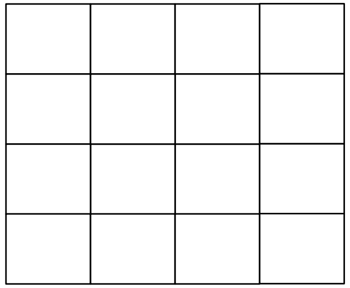
1. A set of **variables**
2. A set of **domains** (one domain for each variable)
3. A set of **constraints** among the variables

The goal is to assign values to each variable so that **all constraints** are satisfied.

Example: Given 4 queens

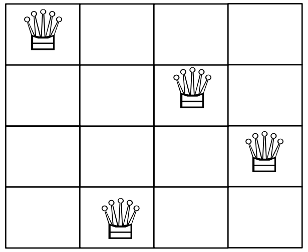


and one 4*4 board



Place all Queens on the board, and no two queens threaten each other.

How to complete `is_legal()`?

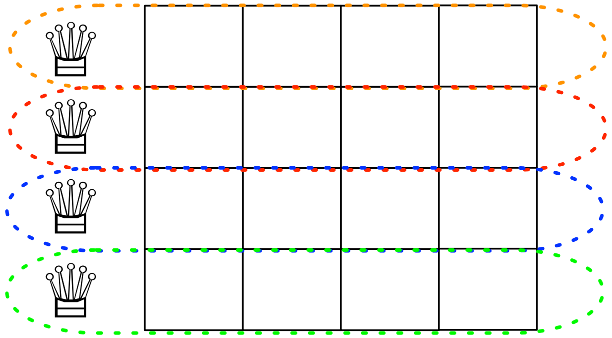


Why it's in conflict?

$q_3 - q_2$ `==` `board[`q_3`]` `-` `board[`q_2`]`

2 - N-Queens Problem

What is Constraint Satisfaction Problem?

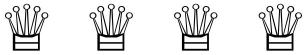


A CSP is typically defined by:

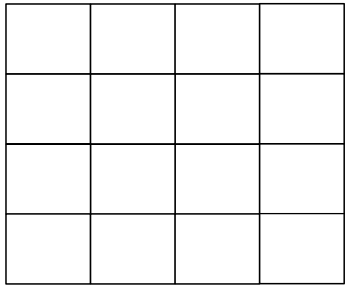
1. A set of **variables**
2. A set of **domains** (one domain for each variable)
3. A set of **constraints** among the variables

The goal is to assign values to each variable so that **all constraints** are satisfied.

Example: Given 4 queens

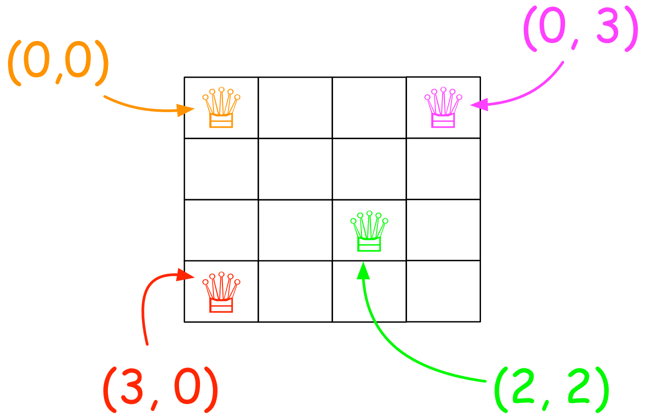


and one 4*4 board



Place all Queens on the board, and no two queens threaten each other.

How to complete `is_legal()`?

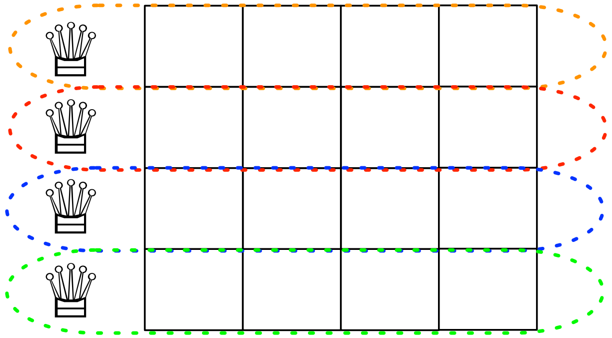


♕ - ♕ = (-2, -2)

♕ - ♕ = (-3, 3)

2 - N-Queens Problem

What is Constraint Satisfaction Problem?

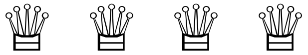


A CSP is typically defined by:

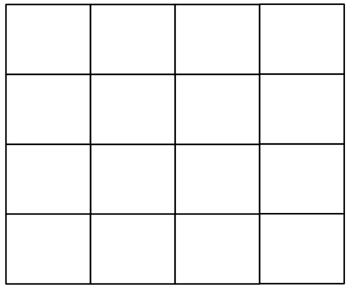
1. A set of **variables**
2. A set of **domains** (one domain for each variable)
3. A set of **constraints** among the variables

The goal is to assign values to each variable so that **all constraints** are satisfied.

Example: Given 4 queens

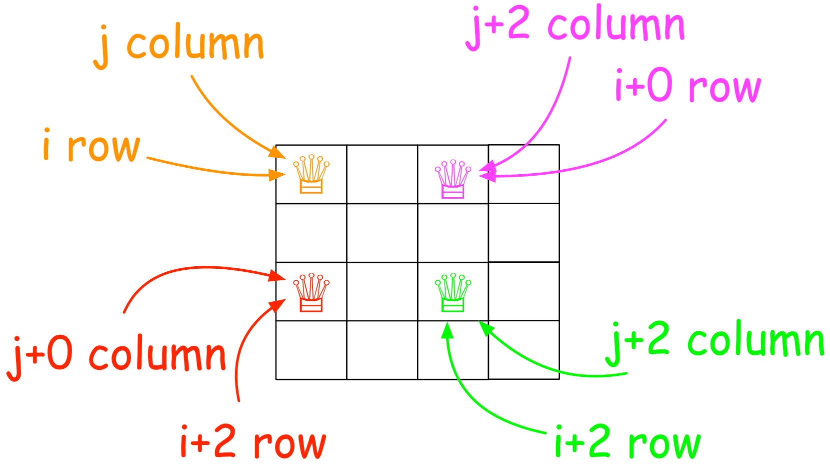


and one 4*4 board



Place all Queens on the board, and no two queens threaten each other.

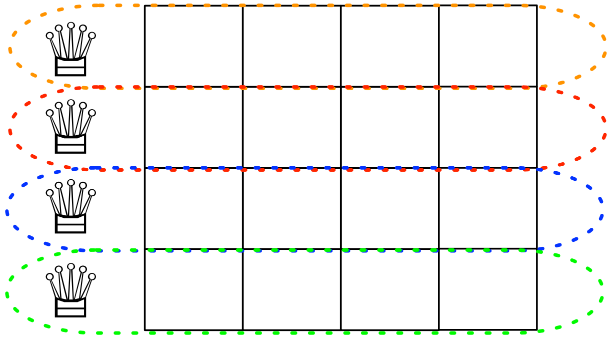
How to complete `is_legal()`?



$\texttt{abs}(x_2 - x_1) == \texttt{abs}(\texttt{board}[x_2] - \texttt{board}[x_1])$

2 - N-Queens Problem

What is Constraint Satisfaction Problem?

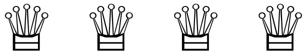


A CSP is typically defined by:

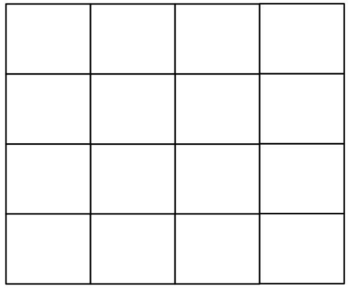
1. A set of **variables**
2. A set of **domains** (one domain for each variable)
3. A set of **constraints** among the variables

The goal is to assign values to each variable so that **all constraints** are satisfied.

Example: Given 4 queens



and one 4*4 board



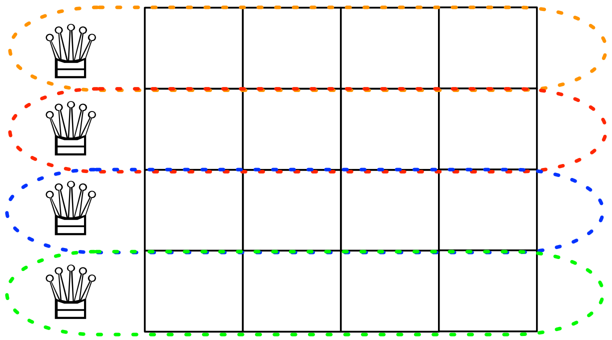
Place all Queens on the board, and no two queens threaten each other.

How to complete `is_legal()`?

```
# Check if the current board configuration is valid.
def is_legal(board):
    # index rows one by one
    for i in range(len(board)):
        # check the remaining rows
        for j in range(i + 1, len(board)):
            # Check any two queens in the same column
            if board[i] == board[j]:
                return False
    return True
```

2 - N-Queens Problem

What is Constraint Satisfaction Problem?



A CSP is typically defined by:

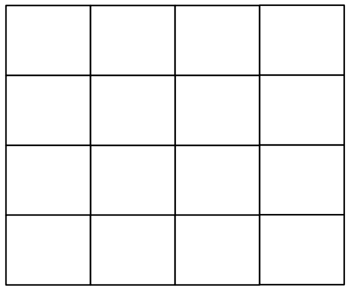
1. A set of **variables**
2. A set of **domains** (one domain for each variable)
3. A set of **constraints** among the variables

The goal is to assign values to each variable so that **all constraints** are satisfied.

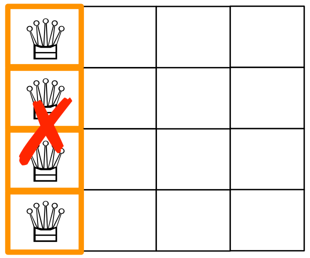
Example: Given 4 queens ♕ ♕ ♕ ♕

Where should we place the next ♕ ?

and one 4*4 board



False

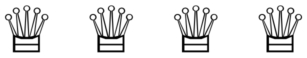


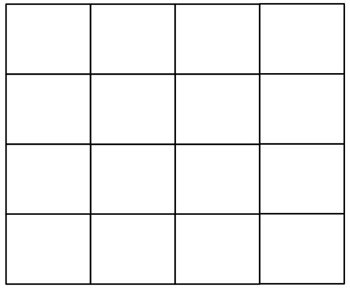
is_legal()?

???

Place all Queens on the board, and no two queens threaten each other.

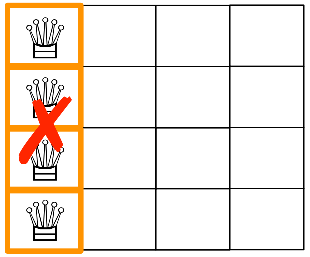
2 - N-Queens Problem

Given 4 queens 

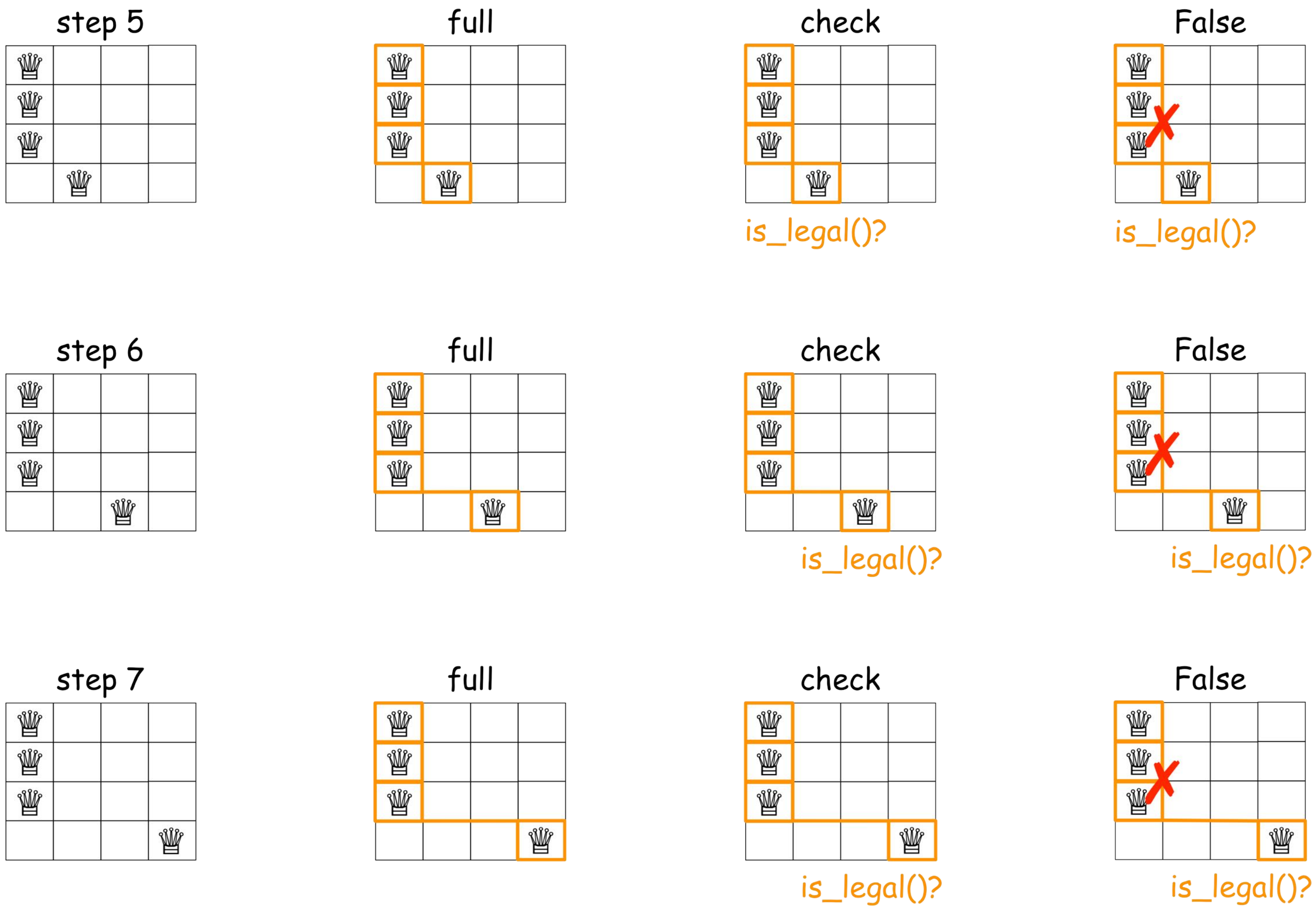
and one 4*4 board 

Place all Queens on the board, and no two queens threaten each other.

False

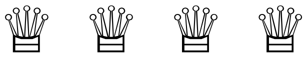


is_legal()?

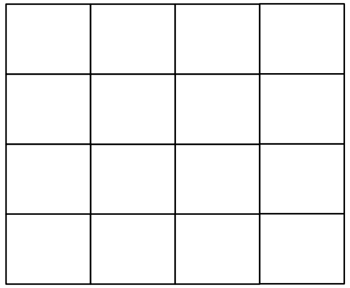


2 - N-Queens Problem

Given 4 queens

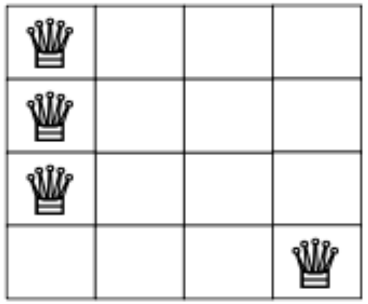


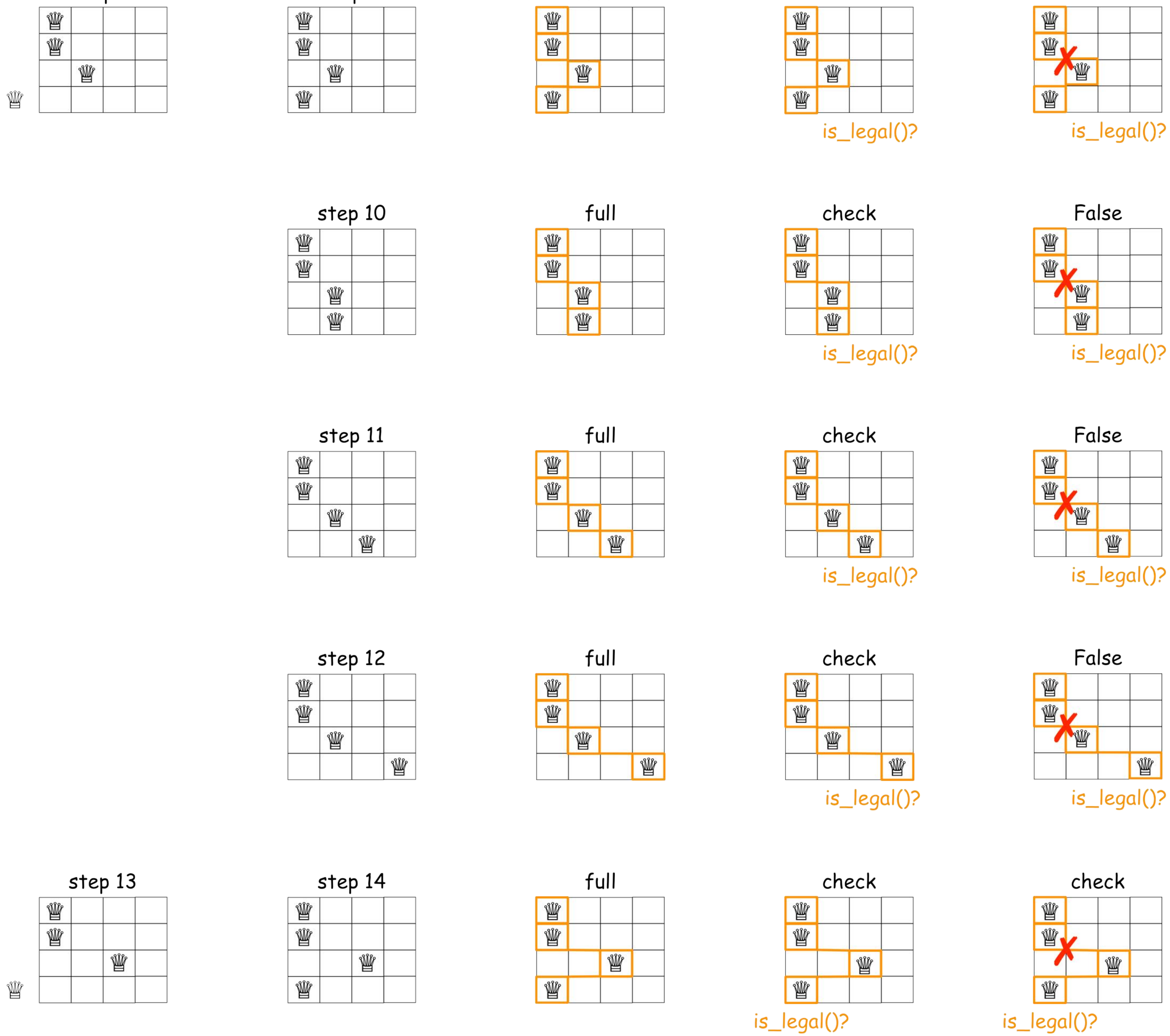
and one 4*4 board



Place all Queens on the board, and no two queens threaten each other.

step 7

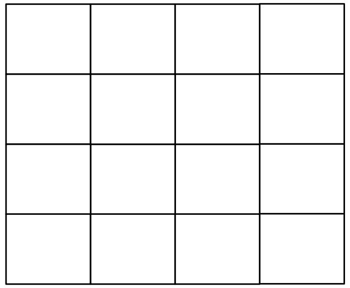




2 - N-Queens Problem

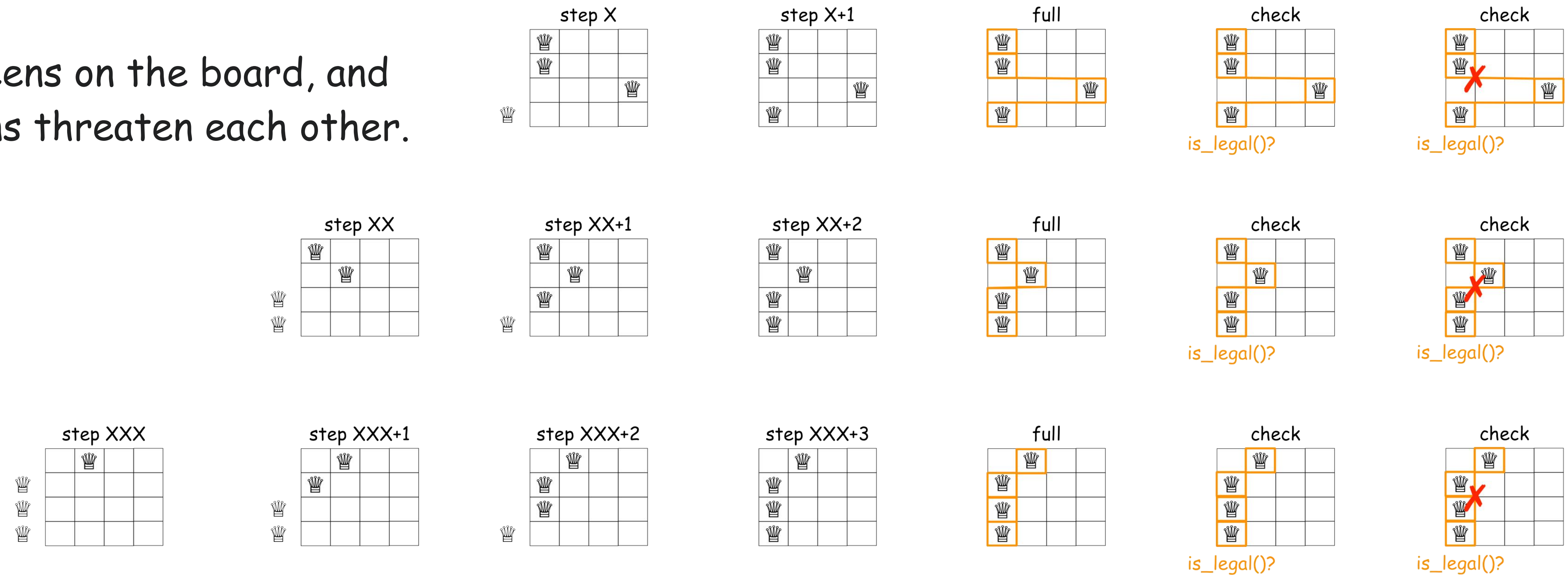
Given 4 queens ♕ ♕ ♕ ♕

and one 4*4 board

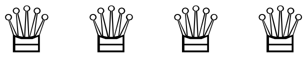


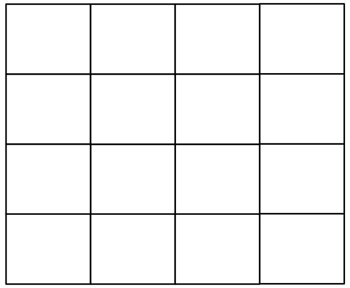
What's the coding structure?

Place all Queens on the board, and no two queens threaten each other.



2 - N-Queens Problem

Given 4 queens 

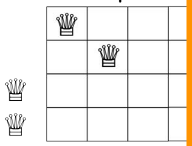
and one 4*4 board 

What's the coding structure?

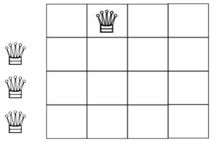
Place all Queens on the board, and no two queens threaten each other

```
for col1 in range(len(board)):
    board[0] = col1
    for col2 in range(len(board)):
        board[1] = col2
        for col3 in range(len(board)):
            board[2] = col3
            for col4 in range(len(board)):
                board[3] = col4

                if is_valid(board):
                    print_board(board, counter)
```

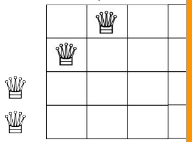
step XX



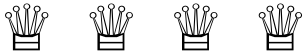
step XXX



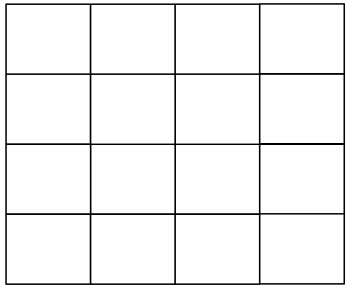
step XXX+



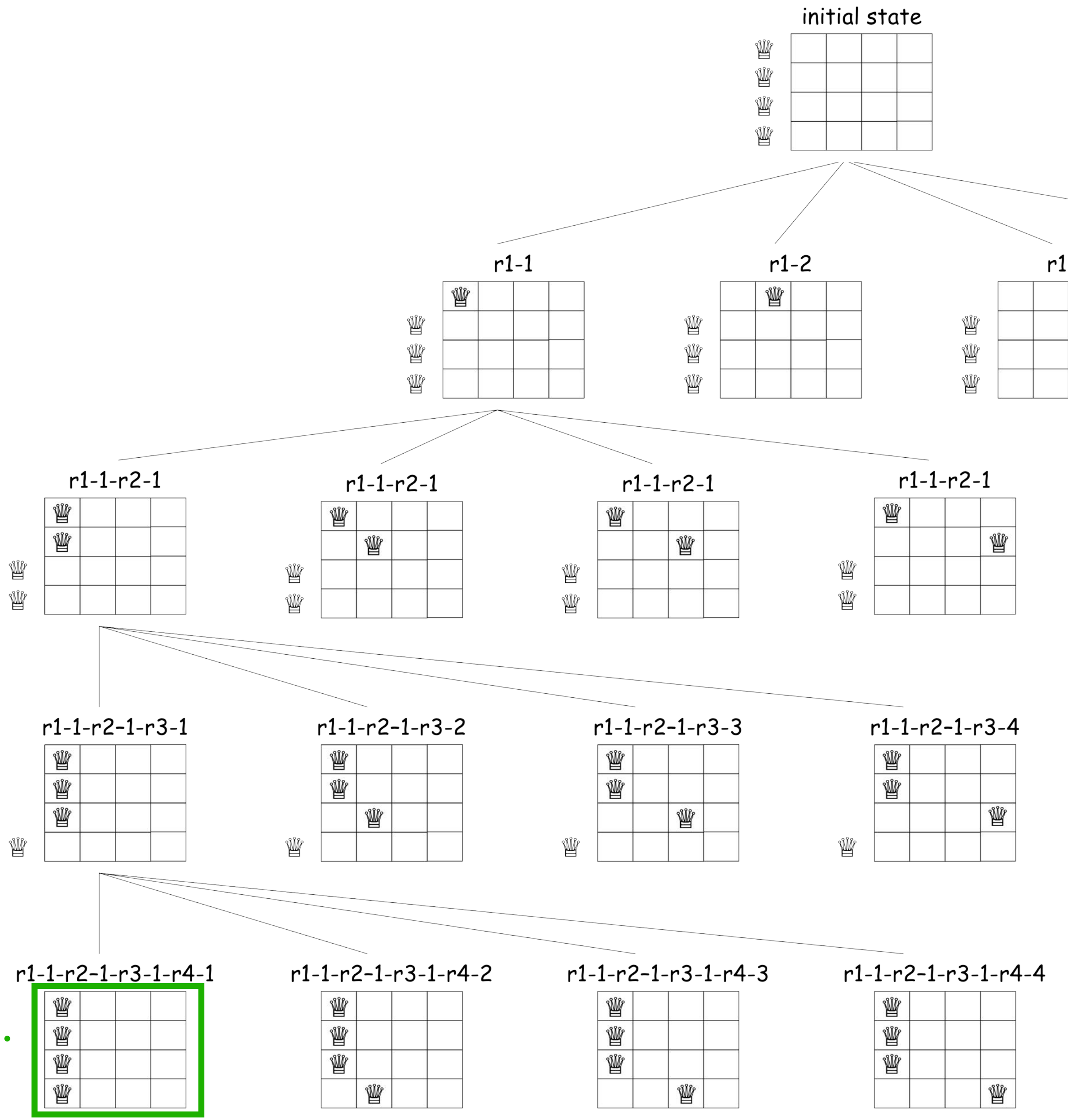
Given 4 queens



and one 4*4 board



Place all Queens on the board, and no two queens threaten each other.

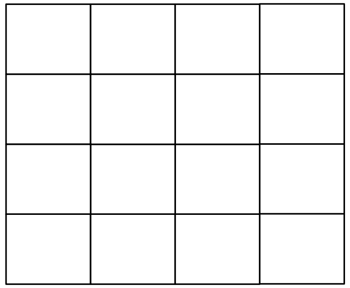


Check on leaf nodes.

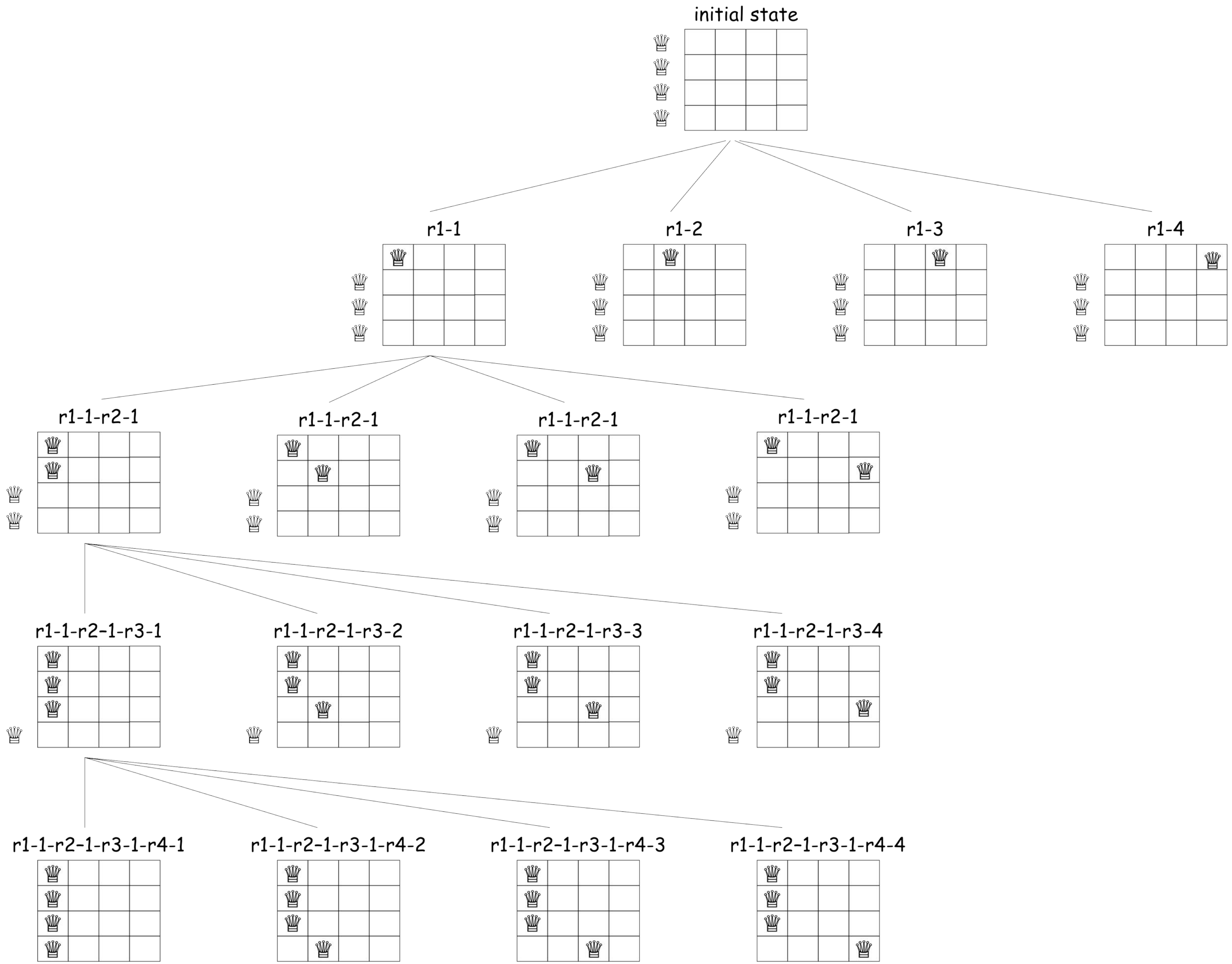
2 - N-Queens Problem

Given 4 queens ♕ ♕ ♕ ♕

and one 4*4 board



Place all Queens on the board, and no two queens threaten each other.



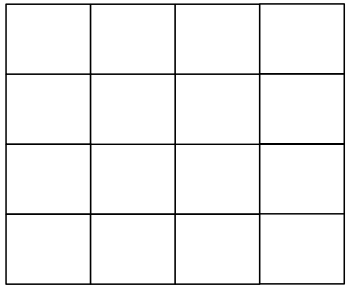
How can we save runtime?

i.e., how can we prune this search tree?

2 - N-Queens Problem

Given 4 queens

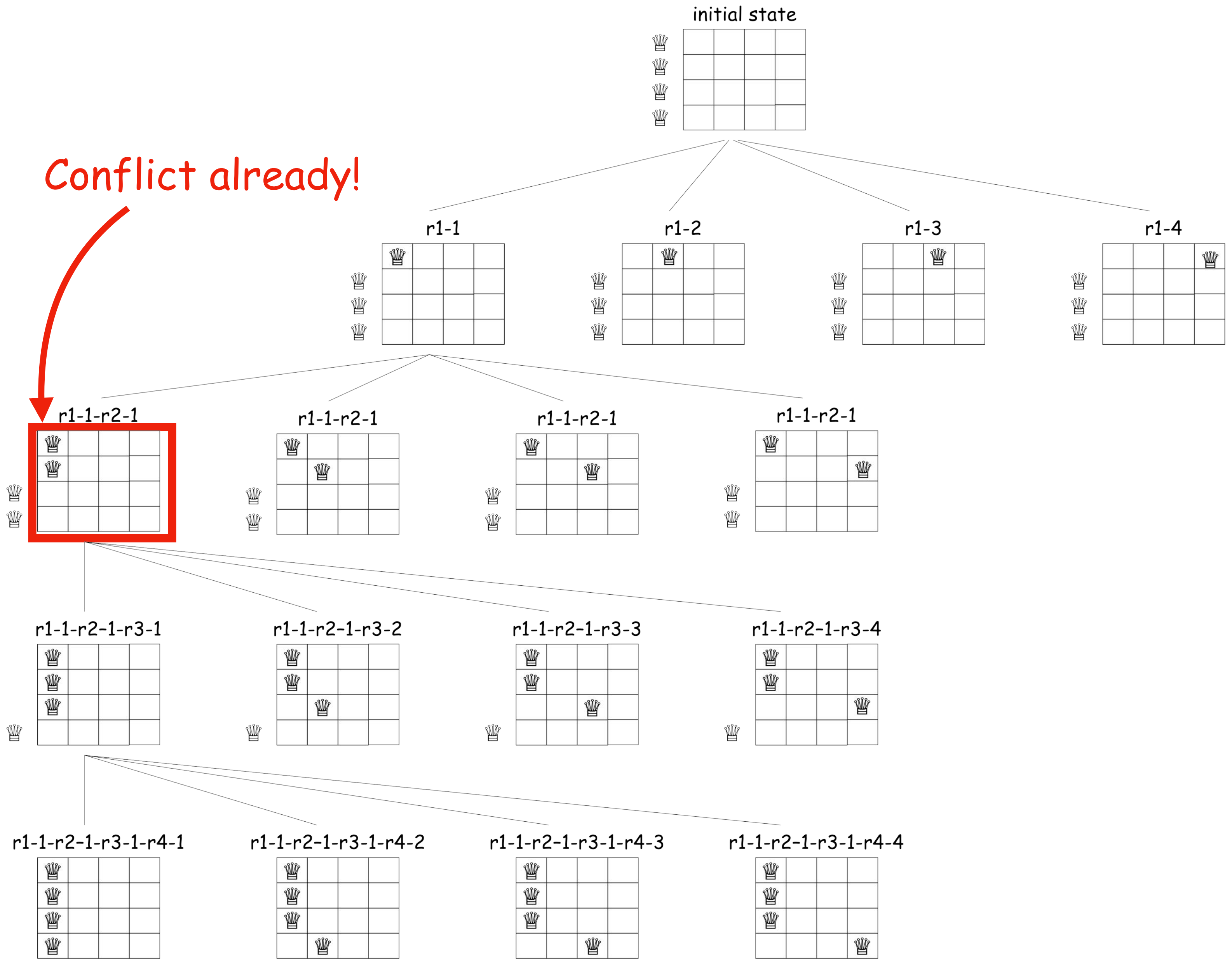
and one 4*4 board



Place all Queens on the board, and no two queens threaten each other.

How can we save runtime?

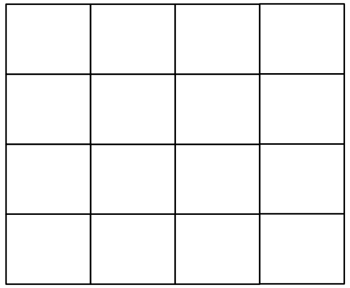
i.e., how can we prune this search tree?



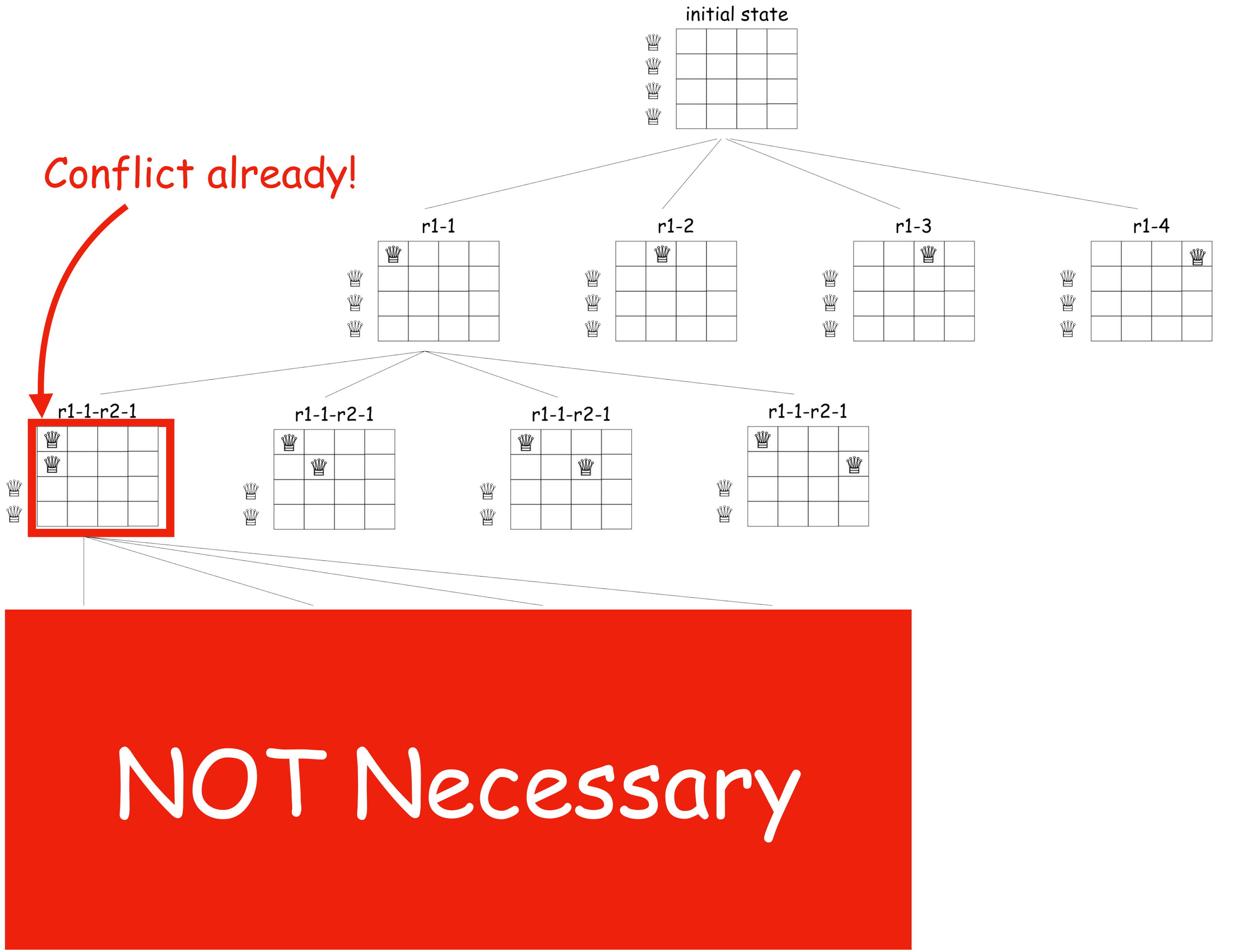
2 - N-Queens Problem

Given 4 queens ♕ ♕ ♕ ♕

and one 4*4 board



Place all Queens on the board, and no two queens threaten each other.



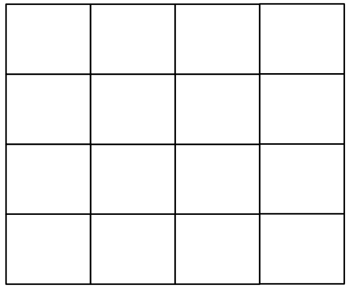
How can we save runtime?

i.e., how can we prune this search tree?

2 - N-Queens Problem

Given 4 queens

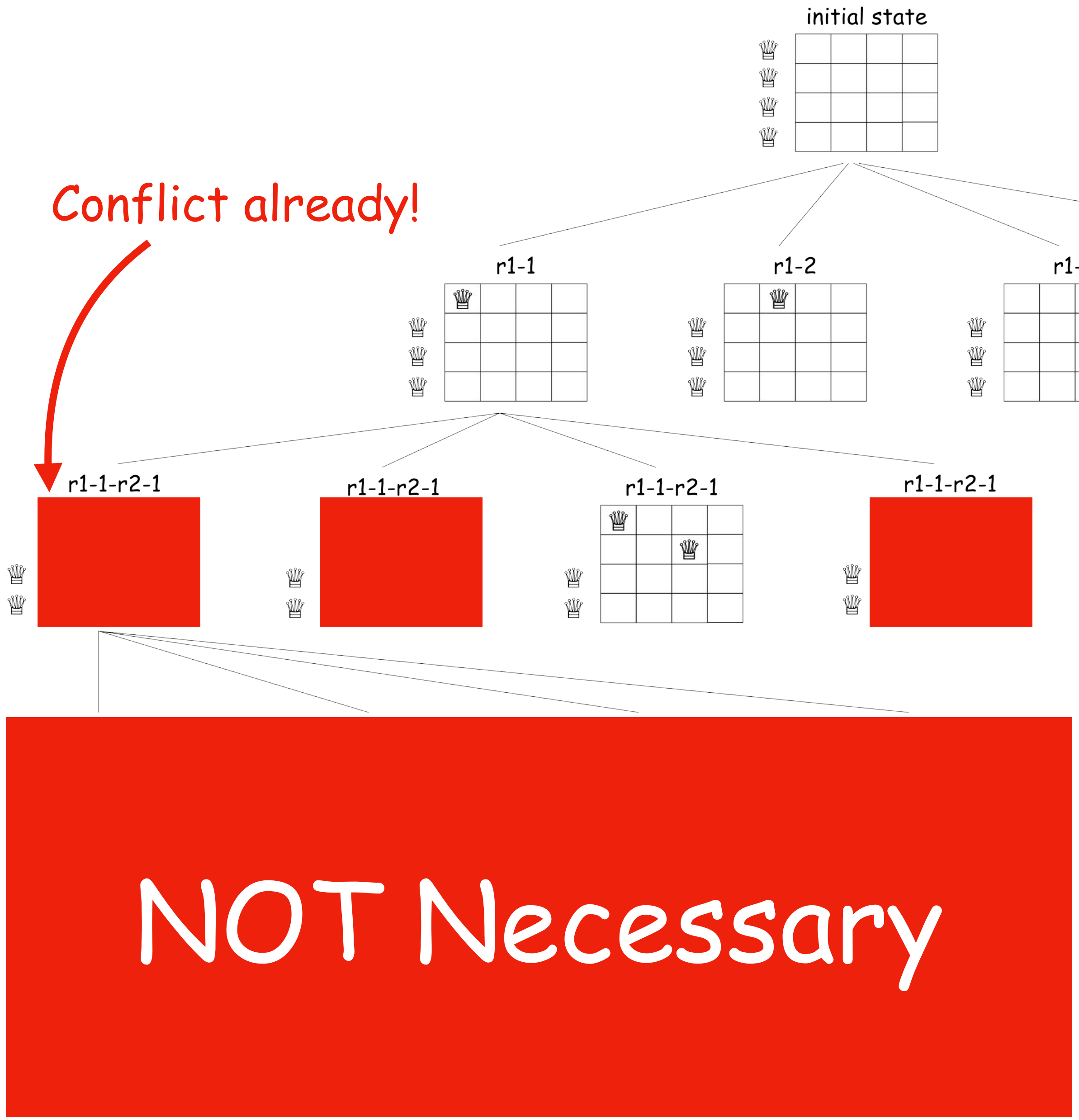
and one 4*4 board



Place all Queens on the board, and no two queens threaten each other.

How can we save runtime?

i.e., how can we prune this search tree?



2 - N-Queens Problem

Given an array `A`, where `A` is defined as follows:

1. How can we systematically enumerate or simulate all possible values of `A`?

```
for candidate_0 in range(len(A)):
    A[0] = candidate_0

    for candidate_1 in range(len(A)):
        A[1] = candidate_1

        for candidate_2 in range(len(A)):
            A[2] = candidate_2

            for candidate_3 in range(len(A)):
                A[3] = candidate_3

                print(A)
```

2 - N-Queens Problem

Given an array `A`, where `A` is defined as follows:

2. What would be the implications if we impose the constraint that each digit can appear only once?

```
for candidate_0 in range(len(A)):
    A[0] = candidate_0

    for candidate_1 in range(len(A)):
        A[1] = candidate_1

        for candidate_2 in range(len(A)):
            A[2] = candidate_2

            for candidate_3 in range(len(A)):
                A[3] = candidate_3

                if is_valid(A):
                    print(A)
```

```
def is_valid(A):
    for i in range(len(A)):
        for j in range(i + 1, len(A)):
            if A[i] == A[j]:
                return False
    return True
```

2 - N-Queens Problem

Given an array `A`, where `A` is defined as follows:

3. How would the behavior change if we introduce the following `is_valid()` function?

```
for candidate_0 in range(len(A)):
    A[0] = candidate_0

    for candidate_1 in range(len(A)):
        A[1] = candidate_1

        for candidate_2 in range(len(A)):
            A[2] = candidate_2

            for candidate_3 in range(len(A)):
                A[3] = candidate_3

                if is_valid(A):
                    print(A)
```

```
def is_valid(A):
    for i in range(len(A)):
        for j in range(i + 1, len(A)):
            if A[i] == A[j]:
                return False
            if abs(i-j) == bas(A[i] - A[j]):
                return False
    return True
```

2 - N-Queens Problem

Given an array `A`, where `A` is defined as follows:

3. How would the behavior change if we introduce the following `is_valid()` function?

```
for candidate_0 in range(len(A)):
    A[0] = candidate_0

    for candidate_1 in range(len(A)):
        A[1] = candidate_1

        for candidate_2 in range(len(A)):
            A[2] = candidate_2

            for candidate_3 in range(len(A)):
                A[3] = candidate_3

                if is_valid(A):
                    print(A)
```

```
def is_valid(A):
    for i in range(len(A)):
        for j in range(i + 1, len(A)):
            if A[i] == A[j]:
                return False
            if abs(i-j) == bas(A[i] - A[j]):
                return False
    return True
```

It's the N-queens Problem.

2 - N-Queens Problem

Global Check!

```
for col1 in range(len(board)):
    board[0] = col1

    for col2 in range(len(board)):
        board[1] = col2

        for col3 in range(len(board)):
            board[2] = col3

            for col4 in range(len(board)):
                board[3] = col4

                   if is_valid(board):
                        print_board(board, counter)
```

Local Check!

```
for tentative_col1 in range(len(board)):
   if is_valid(board, 0, tentative_col1):
       board[0] = tentative_col1

        for tentative_col2 in range(len(board)):
            if is_valid(board, 1, tentative_col2):
               board[1] = tentative_col2

                for tentative_col3 in range(len(board)):
                    if is_valid(board, 2, tentative_col3):
                        board[2] = tentative_col3

                        for tentative_col4 in range(len(board)):
                            if is_valid(board, 3, tentative_col4):
                                board[3] = tentative_col4
                                print_board(board, counter)
```

2 - N-Queens Problem

Global Check!

```
for col1 in range(len(board)):
    board[0] = col1

    for col2 in range(len(board)):
        board[1] = col2

        for col3 in range(len(board)):
            board[2] = col3

            for col4 in range(len(board)):
                board[3] = col4

                if is_valid(board):
                    print_board(board, counter)
```

Local Check!

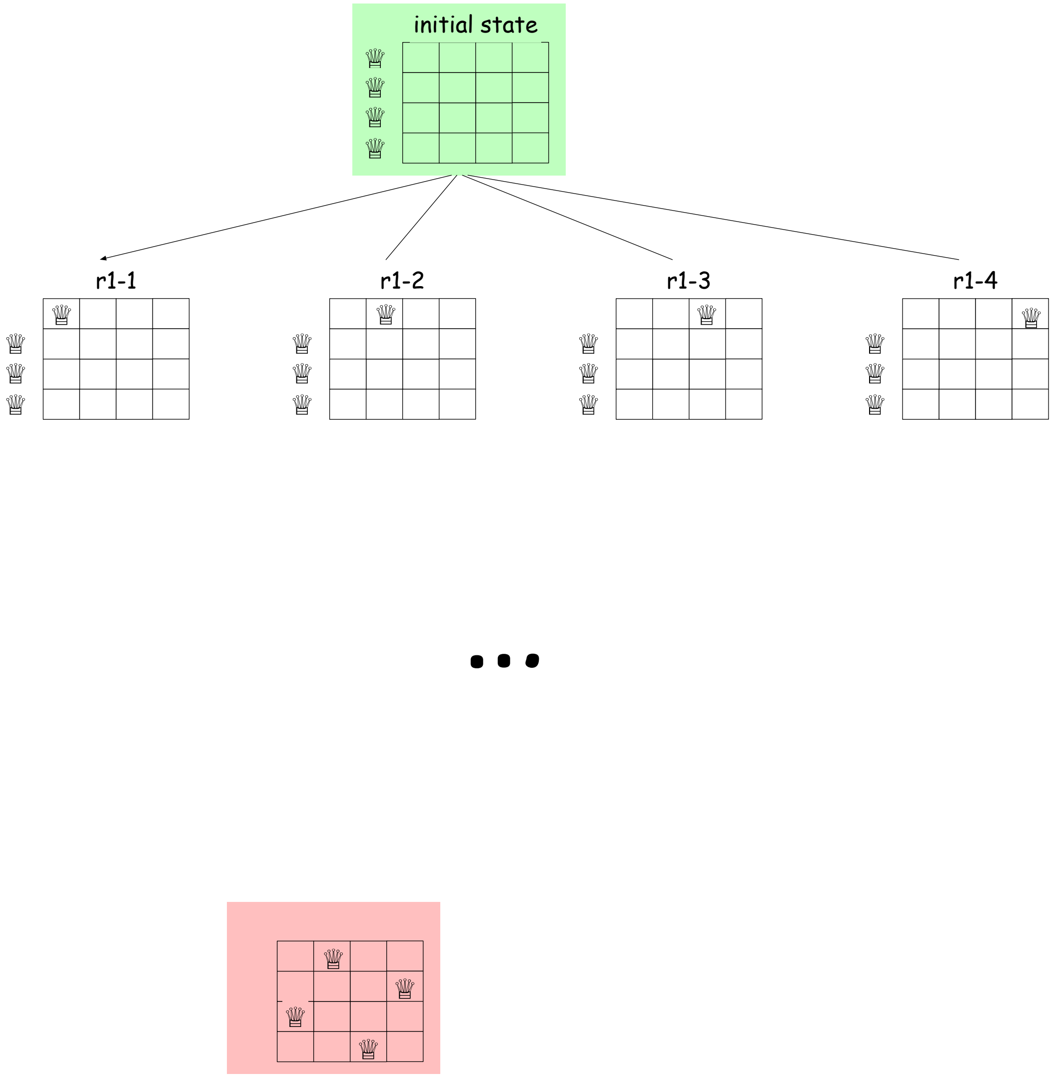
```
for tentative_col1 in range(len(board)):
   if is_valid(board, 0, tentative_col1):
       board[0] = tentative_col1

        for tentative_col2 in range(len(board)):
           if is_valid(board, 1, tentative_col2):
              board[1] = tentative_col2

                for tentative_col3 in range(len(board)):
                   if is_valid(board, 2, tentative_col3):
                       board[2] = tentative_col3

                        for tentative_col4 in range(len(board)):
                           if is_valid(board, 3, tentative_col4):
                               board[3] = tentative_col4
                               print_board(board, counter)
```

```
def solve_n_queens(n):
    solutions = []
     for perm in permutations(range(n)):
         if is_valid(perm):
             solutions.append(perm)
     return solutions
```

1 - Backtracking

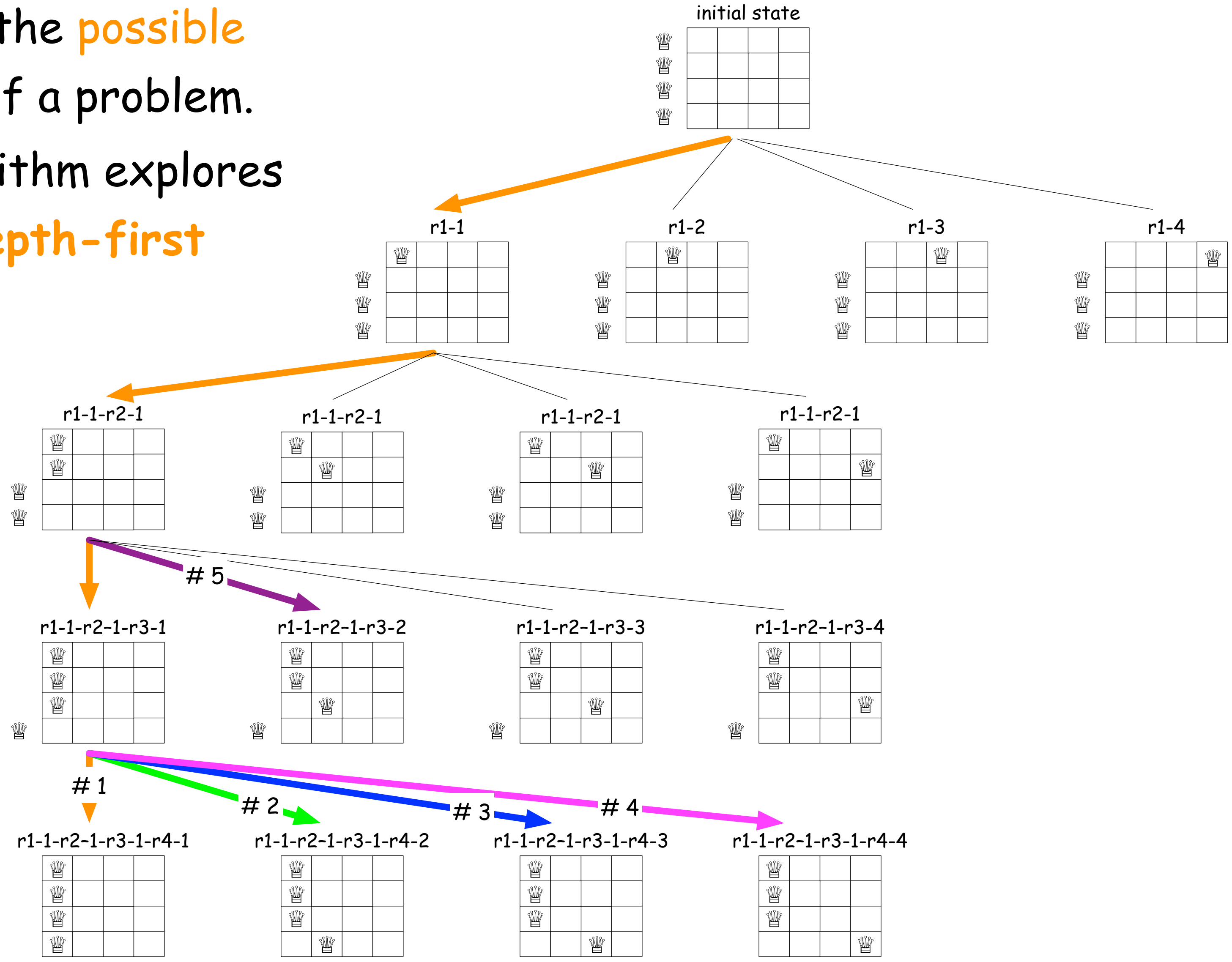
What is Backtracking?

- A methodical way of **searching** through the possible solutions (the "solution/search space") of a problem.



What is Backtracking?

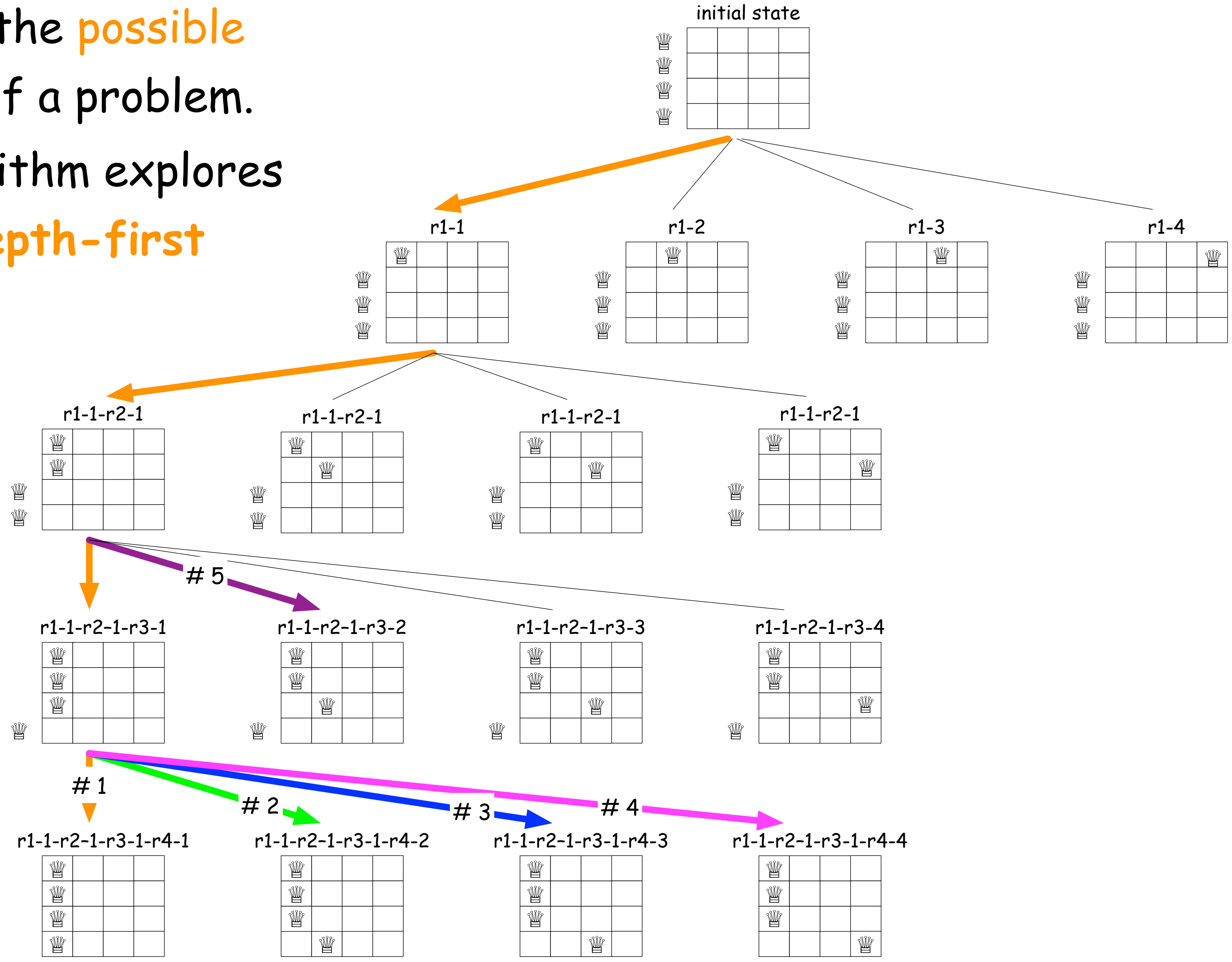
- A methodical way of **searching** through the **possible solutions** (the "**solution/search space**") of a problem.
- Starting from an **initial state**, the algorithm explores possible decisions or assignments in a **depth-first** manner.



What is Backtracking?

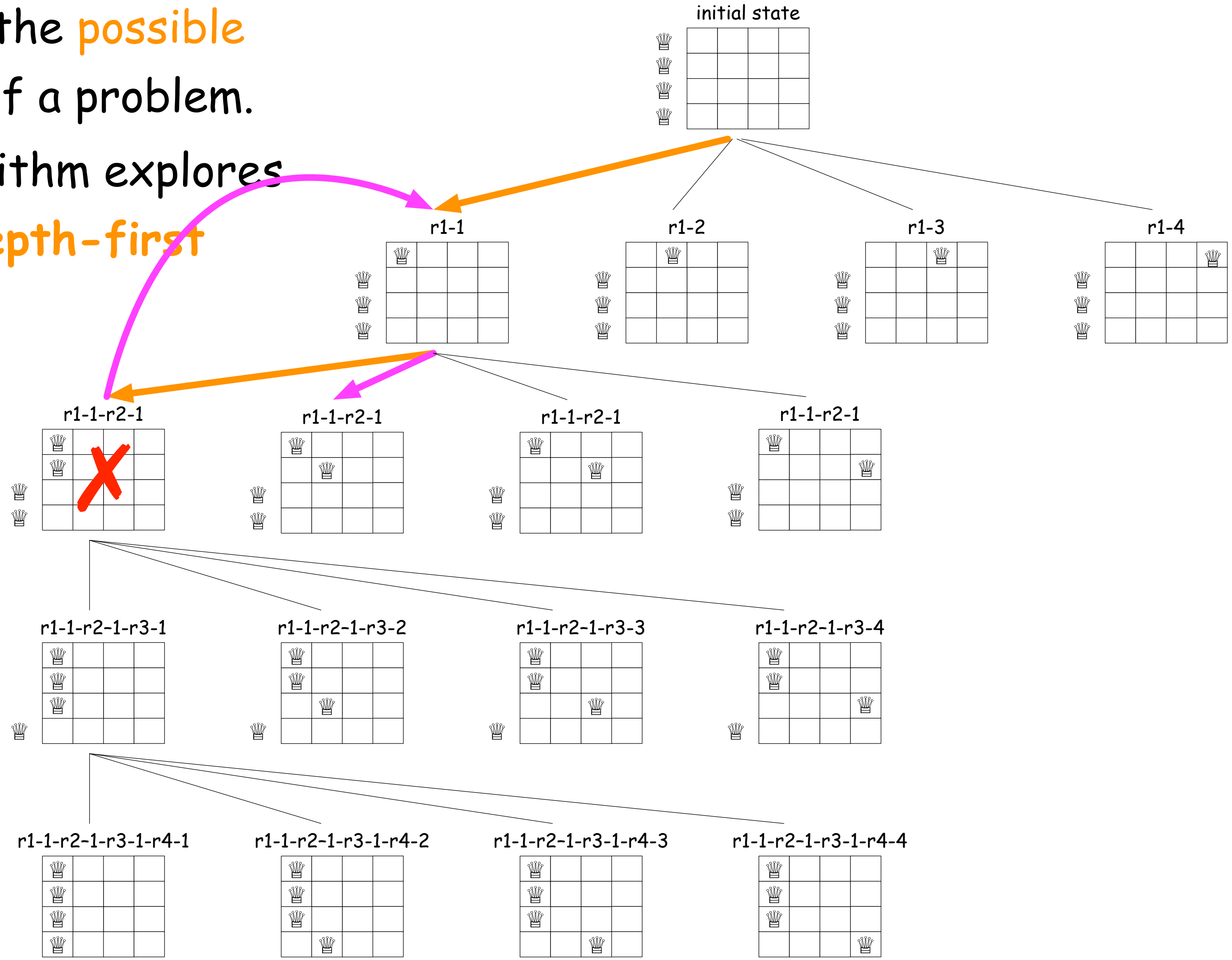
- A methodical way of **searching** through the possible solutions (the "solution/search space") of a problem.
- Starting from an **initial state**, the algorithm explores possible decisions or assignments in a **depth-first** manner.

Core idea 1: Branch by branch



What is Backtracking?

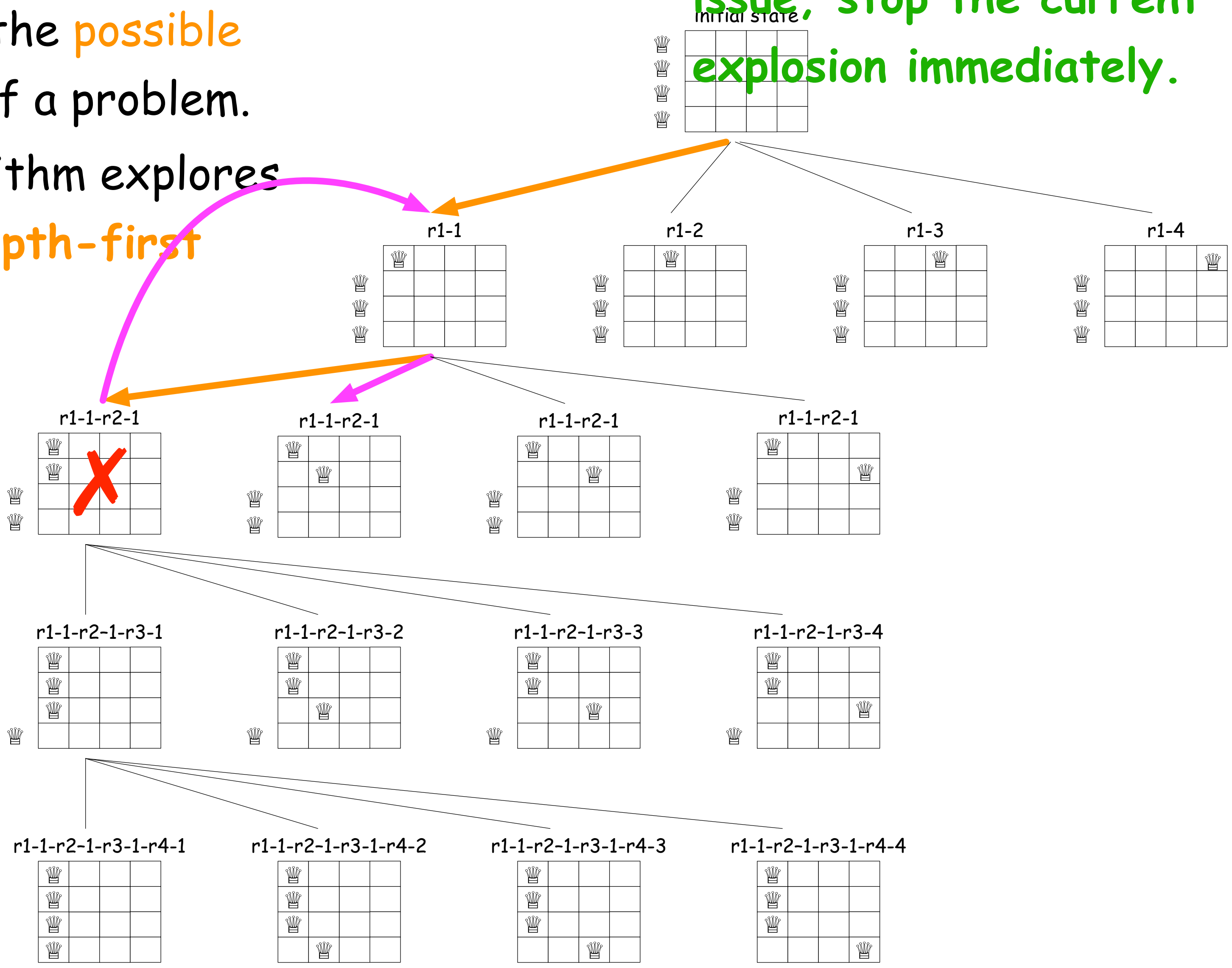
- A methodical way of **searching** through the **possible solutions** (the "**solution/search space**") of a problem.
- Starting from an **initial state**, the algorithm explores possible decisions or assignments in a **depth-first** manner.
- Whenever it becomes clear that a particular path cannot lead to a valid or optimal solution, the algorithm **backtracks**—i.e., it undoes the last decision and tries a different option instead.



What is Backtracking?

- A methodical way of **searching** through the **possible solutions** (the "**solution/search space**") of a problem.
- Starting from an **initial state**, the algorithm explores possible decisions or assignments in a **depth-first** manner.
- Whenever it becomes clear that a particular path cannot lead to a valid or optimal solution, the algorithm **backtracks**—i.e., it undoes the last decision and tries a different option instead.

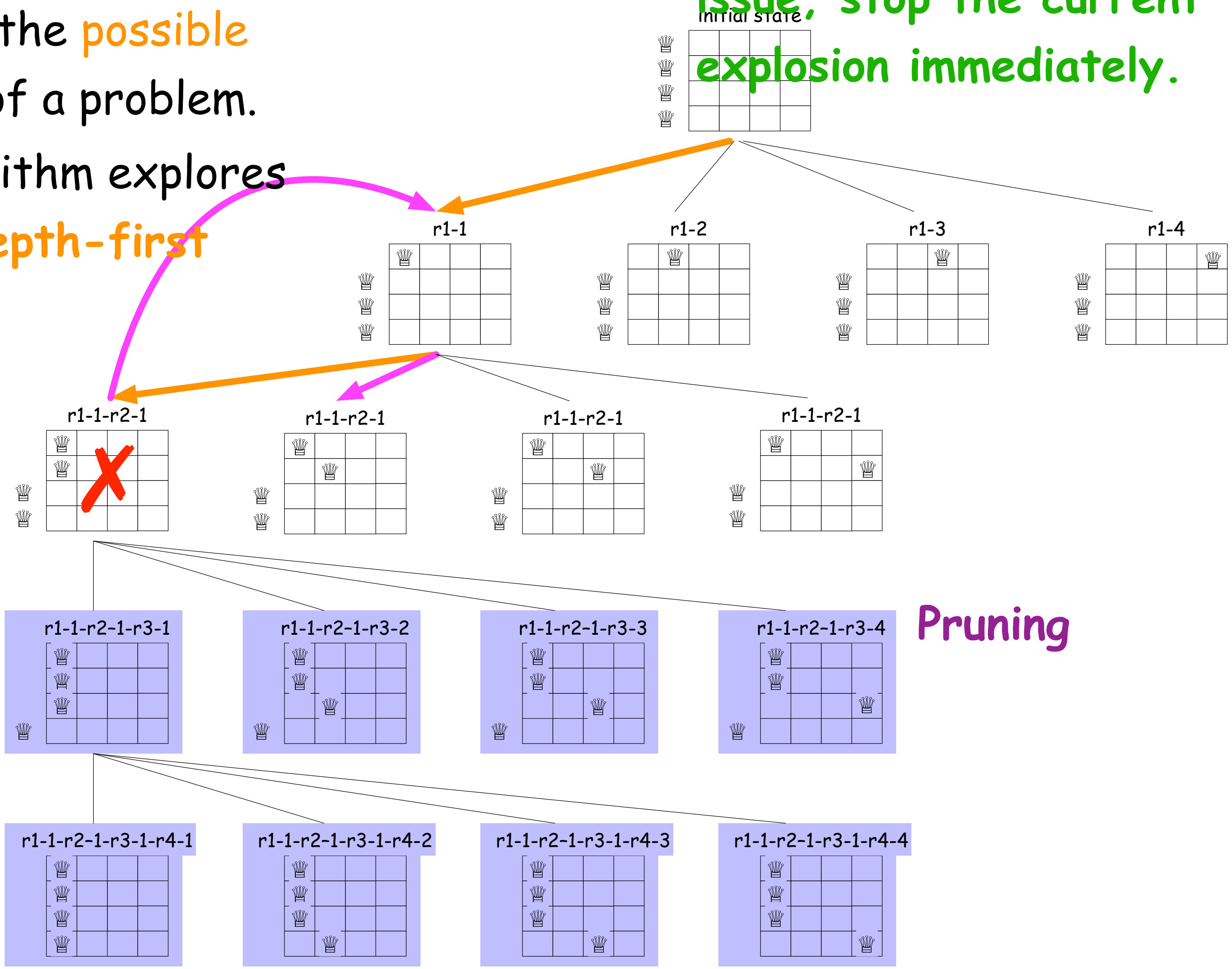
Core idea 2: if arising issue, stop the current explosion immediately.



What is Backtracking?

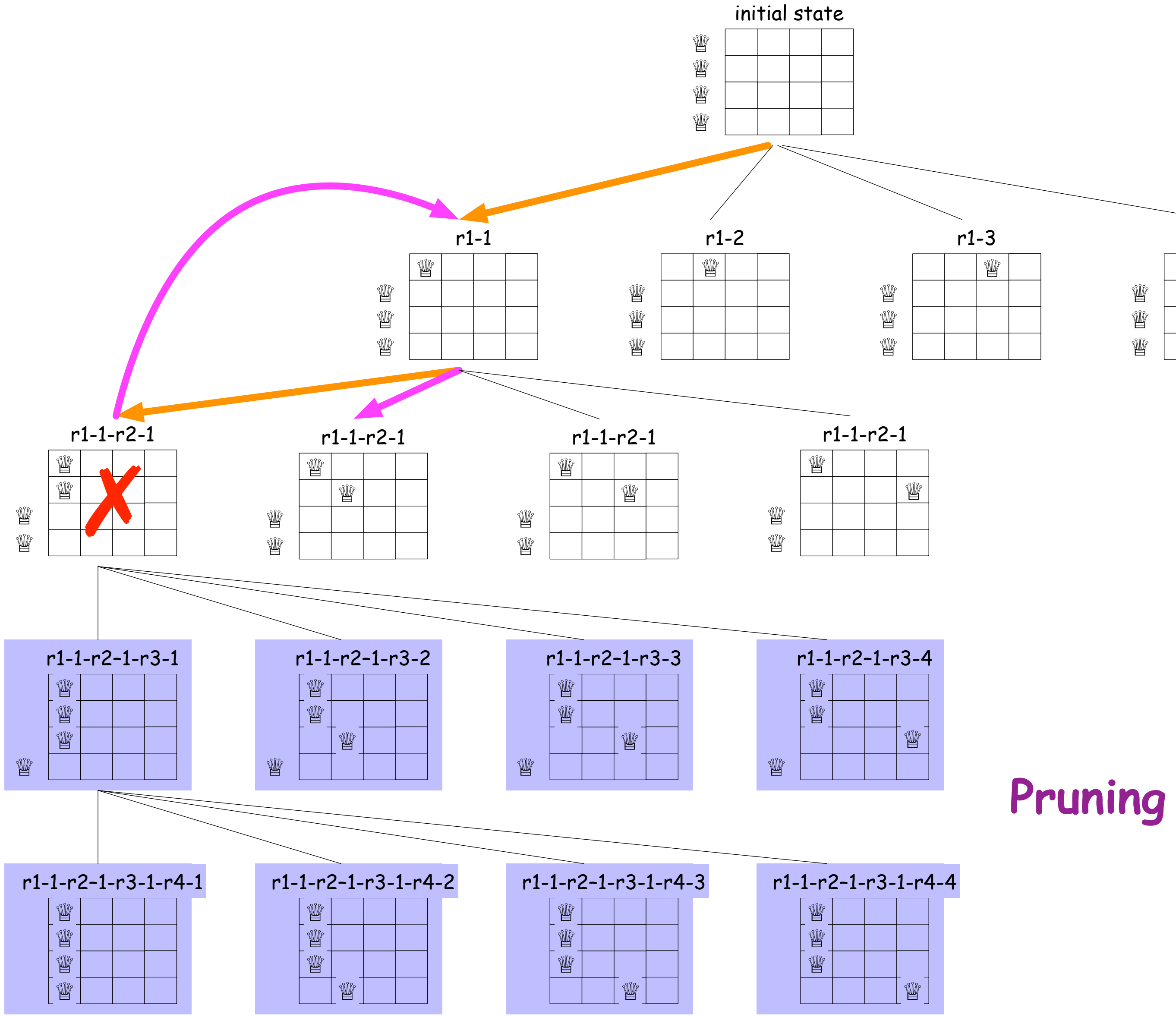
- A methodical way of **searching** through the possible solutions (the "solution/search space") of a problem.
- Starting from an **initial state**, the algorithm explores possible decisions or assignments in a **depth-first** manner.
- Whenever it becomes clear that a particular path cannot lead to a valid or optimal solution, the algorithm **backtracks**—i.e., it undoes the last decision and tries a different option instead.

Core idea 2: if arising issue, stop the current explosion immediately.



What is Backtracking?

- Representing your problem as a tree structure allows you to identify redundant branches, pruning opportunities, and optimization strategies.
- This is a best practice in algorithm design, particularly for backtracking, recursion, and search problems, as it helps minimize unnecessary computations and improve efficiency.



1 - General Backtracking Framework

Generic Pseudo-code - Recursion Style

```
function BACKTRACKING(current_state, current_solution):
    if current_state reaches the required depth or number of steps:
       if objective(current_solution):
            # option 1: record
            # option 2: output current_solution
        return

    # Enumerate all possible options for current_state
    for tentative_option in feasible_option_set:
        if not is_valid(current_solution, tentative_option):
            continue  # Prune the peach space immediately

        # Just do it.(local option is legal)
        current_solution[current_state] = tentative_option

        # Recurse
        BACKTRACKING(current_state + 1, current_solution)

        # Undo the decision (backtrack)
        current_solution[current_state] = None
```

1 - General Backtracking Framework

Generic Pseudo-code - Recursion Style

```
function BACKTRACKING(current_state, current_solution):
    if current_state reaches the required depth or number of steps:
       if objective(current_solution):
            # option 1: record
            # option 2: output current_solution
        return

    # Enumerate all possible options for current_state
    for tentative_option in feasible_option_set:
        if not is_valid(current_solution, tentative_option):
            continue  # Prune the peach space immediately

        # Just do it.(local option is legal)
        current_solution[current_state] = tentative_option

        # Recurse
        BACKTRACKING(current_state + 1, current_solution)

        # Undo the decision (backtrack)
        current_solution[current_state] = None
```

Key Steps:

1. state/decision index
2. options set
3. constraints check
4. how to move to next state
5. backtracking

2 - Sudoku

Sudoku is a logic-based puzzle on a 9×9 grid:

1. Each row must contain the digits 1–9 without repetition.
2. Each column must contain the digits 1–9 without repetition.
3. Each of the nine 3×3 sub-grids (boxes) must contain digits 1–9 without repetition.

The puzzle starts with some cells filled in, and the solver must fill in the rest via logical deduction.

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

2 - Apply Backtracking Framework

```
function BACKTRACKING(current_state, current_solution):
    if current_state reaches goal or boundary:
        if objective(current_solution):
            # option 1: record
            # option 2: output current_solution
            # option 3: just return True
        return

    # Enumerate all possible options for current_state
    for tentative_option in feasible_option_set:
        if not is_valid(current_solution, tentative_option):
            continue  # Prune the peach space immediately

        # Just do it.(local option is legal)
        current_solution[current_state] = tentative_option

        # Recurse
        BACKTRACKING(current_state + 1, current_solution)

        # Undo the decision (backtrack)
        current_solution[current_state] = None
```

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

2 - Apply Backtracking Framework

```
function BACKTRACKING(current_state, current_solution):
    if current_state reaches goal or boundary:
       if objective(current_solution):
            # option 1: record
            # option 2: output current_solution
            # option 3: just return True
        return

    # Enumerate all possible options for current_state
    for tentative_option in feasible_option_set:
        if not is_valid(current_solution, tentative_option):
            continue  # Prune the peach space immediately

        # Just do it.(local option is legal)
        current_solution[current_state] = tentative_option

        # Recurse
        BACKTRACKING(current_state + 1, current_solution)

        # Undo the decision (backtrack)
        current_solution[current_state] = None
```

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

2 - Apply Backtracking Framework

```
function sudoku_solver(current_state, board):
    if all cells are filled:
        return true

    # Enumerate all possible options for current_state
    for tentative_option in feasible_option_set:
        if not is_valid(board, tentative_option):
            continue  # Prune the peach space immediately

        # Just do it.(local option is legal)
        board[current_state] = tentative_option

        # Recurse
        sudoku_solver(current_state + 1, board)

        # Undo the decision (backtrack)
        board[current_state] = None
```

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

2 - Apply Backtracking Framework

```
function sudoku_solver(current_state, board):
    if all cells are filled:
        return true

    # Enumerate all possible options for current_state
    for tentative_option in feasible_option_set:
        if not is_valid(board, tentative_option):
            continue  # Prune the peach space immediately

        # Just do it.(local option is legal)
        board[current_state] = tentative_option

        # Recurse
        sudoku_solver(current_state + 1, board)

        # Undo the decision (backtrack)
        board[current_state] = None
```

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

2 - Apply Backtracking Framework

```
function sudoku_solver(current_state, board):
    if all cells are filled:
        return true

    FIND the next empty (row, col):
        for num in [1:9]:
            if not is_valid(board, tentative_option):
                continue  # Prune the peach space
    immediately

            # Just do it.(local option is legal)
            board[current_state] = tentative_option

            # Recurse
            sudoku_solver(current_state + 1, board)

            # Undo the decision (backtrack)
            board[current_state] = None
```

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

2 - Apply Backtracking Framework

```
function sudoku_solver(current_state, board):
    if all cells are filled:
        return true

    FIND the next empty (row, col):
        for num in [1:9]:
            if not is_valid(board, tentative_option):
                continue  # Prune the peach space
    immediately

            # Just do it.(local option is legal)
            board[current_state] = tentative_option

            # Recurse
            sudoku_solver(current_state + 1, board)

            # Undo the decision (backtrack)
            board[current_state] = None
```

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

2 - Apply Backtracking Framework

```
function sudoku_solver(current_state, board):
    if all cells are filled:
        return true

    FIND the next empty (row, col):
        for num in [1:9]:
            if isValid(board, row, col, num):
                board[row][col] = num
                # Recurse
                sudoku_solver(current_state + 1, board)
                # Undo the decision (backtrack)
                board[current_state] = None
```

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

2 - Apply Backtracking Framework

```
function sudoku_solver(current_state, board):
    if all cells are filled:
        return true

    FIND the next empty (row, col):
        for num in [1:9]:
            if isValid(board, row, col, num):
                board[row][col] = num

                sudoku_solver(current_state + 1, board)

                board[current_state] = None
```

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

2 - Apply Backtracking Framework

```
function sudoku_solver(board):
    if all cells are filled:
        return true

    FIND the next empty (row, col):
        for num in [1:9]:
            if isValid(board, row, col, num):
                board[row][col] = num

                sudoku_solver(board)

                board[row][col] = None
```

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

2 - Apply Backtracking Framework

```
function sudoku_solver(board):
    if all cells are filled:
        return true

    FIND the next empty (row, col):
        for num in [1:9]:
            if isValid(board, row, col, num):
                board[row][col] = num

                sudoku_solver(board)

                board[row][col] = None
    ???
```

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

2 - Apply Backtracking Framework

```
function sudoku_solver(board):
    if all cells are filled:
        return true

    FIND the next empty (row, col):
        for num in [1:9]:
            if isValid(board, row, col, num):
                board[row][col] = num

                sudoku_solver(board)

                board[row][col] = 0
    return false
```

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

2 - Apply Backtracking Framework

```
function sudoku_solver(board):
    if all cells are filled:
        return true

    FIND the next empty (row, col):
        for num in [1:9]:
            if isValid(board, row, col, num):
                board[row][col] = num

                sudoku_solver(board)

                board[row][col] = 0
    return false
```

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

2 - Apply Backtracking Framework

```
function sudoku_solver(board):
    if all cells are filled:
        return true

    FIND the next empty (row, col):
        for num in [1:9]:
            if isValid(board, row, col, num):
                board[row][col] = num

                if sudoku_solver(board):
                    return true

                board[row][col] = 0
    return false
```

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

2 - Apply Backtracking Framework

```
function sudoku_solver(board):
    if all cells are filled:
        return true

    FIND the next empty (row, col):
        for num in [1:9]:
            if isValid(board, row, col, num):
                board[row][col] = num

                if sudoku_solver(board):
                    return true

                board[row][col] = 0
    return false
```

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

2 - Apply Backtracking Framework

```
function is_valid(board, row, col, num):
        for col_in_row in range(9):
            if board[row][col_in_row] == num:
                return False

        for row_in_col in range(9):
            if board[row_in_col][col] == num:
                return False

        box_row_start = (row // 3) * 3
        box_col_start = (col // 3) * 3
        for r in range(box_row_start, box_row_start + 3):
            for c in range(box_col_start, box_col_start +
    3):
                if board[r][c] == num:
                    return False

        return True
```

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

2 - Apply Backtracking Framework

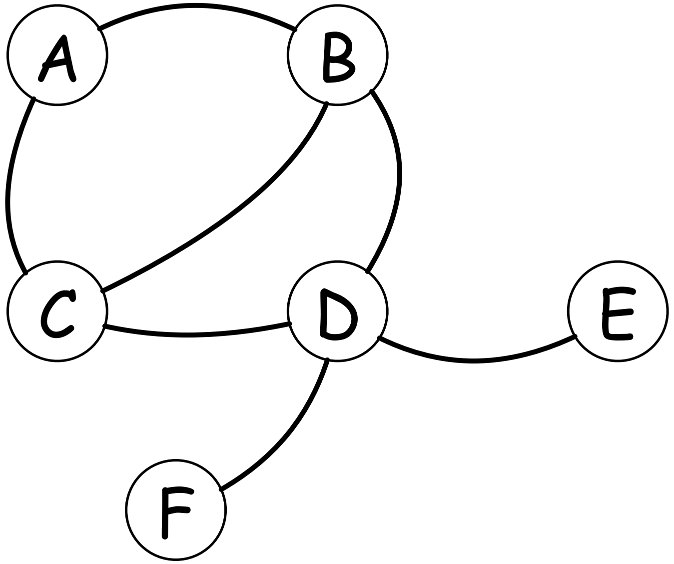
```
function is_valid(board, row, col, num):
        for col_in_row in range(9):
            if board[row][col_in_row] == num:
                return False

        for row_in_col in range(9):
            if board[row_in_col][col] == num:
                return False

        box_row_start = (row // 3) * 3
        box_col_start = (col // 3) * 3
        for r in range(box_row_start, box_row_start + 3):
            for c in range(box_col_start, box_col_start +
    3):
                if board[r][c] == num:
                    return False

        return True
```

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

3-Graph Coloring Problem

Objective: Given an undirected graph $G = (V, E)$ and a total of `K` available colors, assign each vertex a color from $\{1, 2, \ldots, K\}$ such that for every edge (u, v), the vertices u and v do not share the same color.

If successful, we obtain a valid **K-coloring** of the graph; if not, we conclude the graph cannot be colored with `K` colors without violating adjacency constraints.

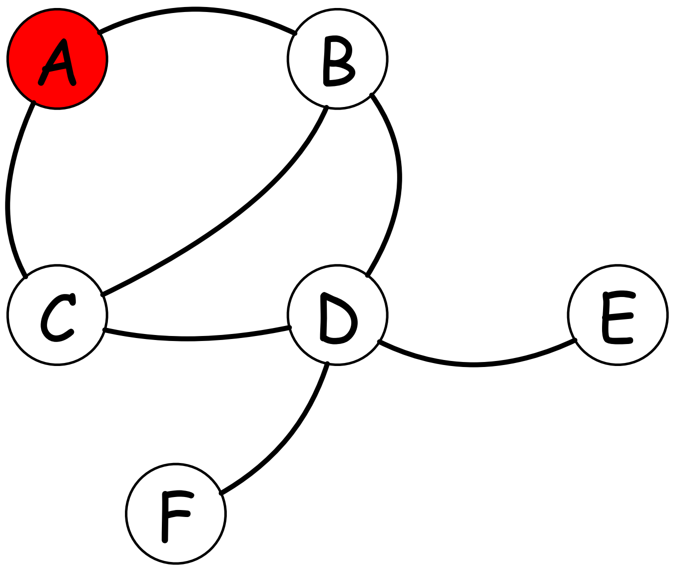



```
function COLOR_GRAPH(G, K):
    # 0 means uncolored
    colors = array of size |V|, initialized with 0
    if assignColor(? ? ? ) == true:
        return colors
    else:
        return "No valid K-coloring found"
```

3-Graph Coloring Problem

Objective: Given an undirected graph $G = (V, E)$ and a total of K available colors, assign each vertex a color from $\{1, 2, \ldots, K\}$ such that for every edge (u, v), the vertices u and v do not share the same color.

If successful, we obtain a valid **K-coloring** of the graph; if not, we conclude the graph cannot be colored with K colors without violating adjacency constraints.

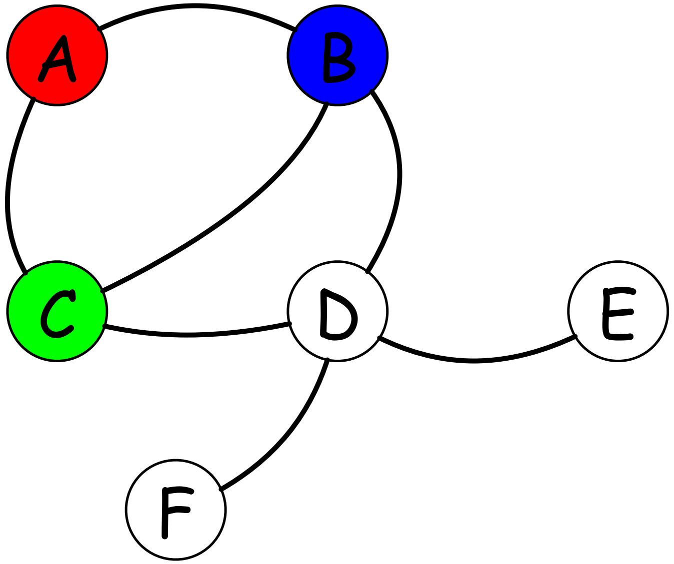



```
function COLOR_GRAPH(G, K):
    # 0 means uncolored
    colors = array of size |V|, initialized with 0
    if assignColor(G, 1, K, colors) == true:
        return colors
    else:
        return "No valid K-coloring found"
```

3-Graph Coloring Problem

Objective: Given an undirected graph $G = (V, E)$ and a total of K available colors, assign each vertex a color from $\{1,2,\ldots,K\}$ such that for every edge (u, v), the vertices u and v do not share the same color.

If successful, we obtain a valid **K-coloring** of the graph; if not, we conclude the graph cannot be colored with K colors without violating adjacency constraints.



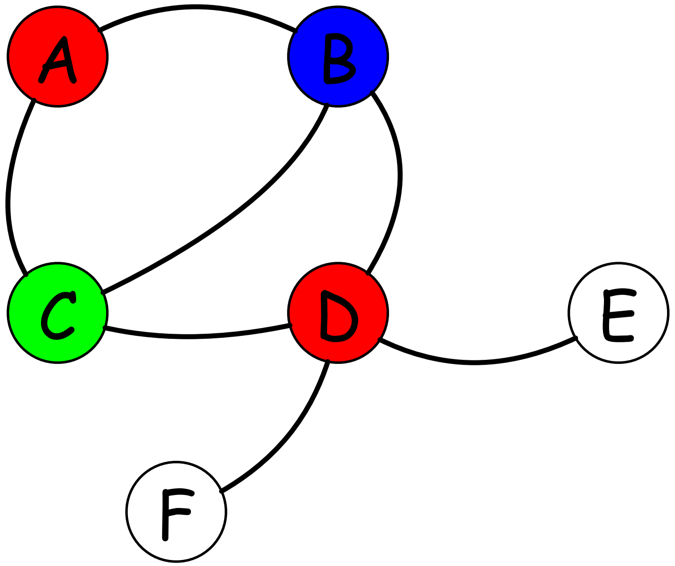

```
function assignColor(G, v, K, colors):
    if v > |V|:
        return true

    for c in 1 to K:
        if isSafe(G, v, c, colors) == true:
            colors[v] = c
            if assignColor(G, v+1, K, colors) == true:
                return true
            colors[v] = 0

    return false
```

```
function isSafe(G, v, color, colors):
    for each neighbor u of v in G:
        if colors[u] == color:
            return false
    return true
```

3-Graph Coloring Problem

Objective: Given an undirected graph $G = (V, E)$ and a total of K available colors, assign each vertex a color from $\{1,2,\ldots,K\}$ such that for every edge (u, v), the vertices u and v do not share the same color.

If successful, we obtain a valid **K-coloring** of the graph; if not, we conclude the graph cannot be colored with K colors without violating adjacency constraints.



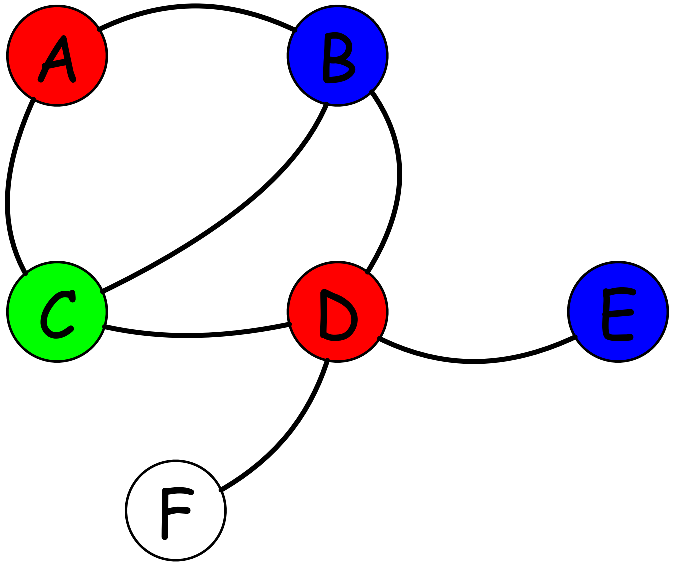

```
function assignColor(G, v, K, colors):
    if v > |V|:
        return true

    for c in 1 to K:
        if isSafe(G, v, c, colors) == true:
            colors[v] = c
            if assignColor(G, v+1, K, colors) == true:
                return true
            colors[v] = 0

    return false
```

```
function isSafe(G, v, color, colors):
    for each neighbor u of v in G:
        if colors[u] == color:
            return false
    return true
```

3-Graph Coloring Problem

Objective: Given an undirected graph $G = (V, E)$ and a total of K available colors, assign each vertex a color from $\{1,2,\ldots,K\}$ such that for every edge (u, v), the vertices u and v do not share the same color.

If successful, we obtain a valid **K-coloring** of the graph; if not, we conclude the graph cannot be colored with K colors without violating adjacency constraints.



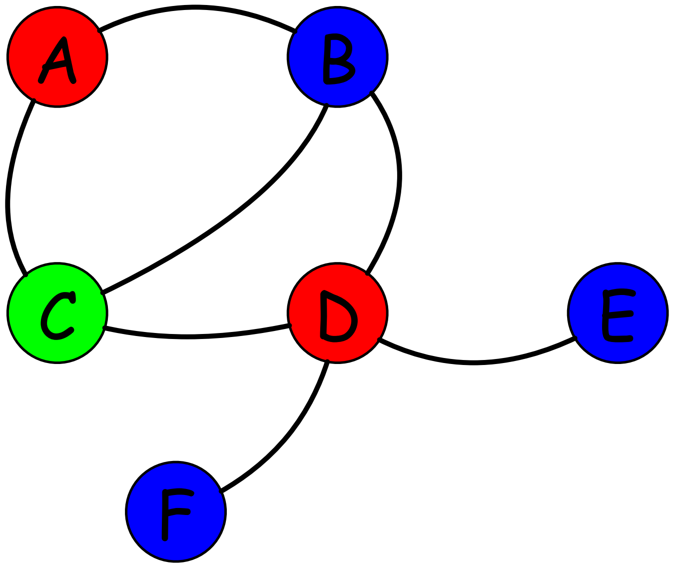

```
function assignColor(G, v, K, colors):
    if v > |V|:
        return true

    for c in 1 to K:
        if isSafe(G, v, c, colors) == true:
            colors[v] = c
            if assignColor(G, v+1, K, colors) == true:
                return true
            colors[v] = 0

    return false
```

```
function isSafe(G, v, color, colors):
    for each neighbor u of v in G:
        if colors[u] == color:
            return false
    return true
```

3-Graph Coloring Problem

Objective: Given an undirected graph $G = (V, E)$ and a total of K available colors, assign each vertex a color from $\{1,2,\ldots,K\}$ such that for every edge (u, v), the vertices u and v do not share the same color.

If successful, we obtain a valid **K-coloring** of the graph; if not, we conclude the graph cannot be colored with K colors without violating adjacency constraints.




```
function assignColor(G, v, K, colors):
    if v > |V|:
        return true

    for c in 1 to K:
        if isSafe(G, v, c, colors) == true:
            colors[v] = c
            if assignColor(G, v+1, K, colors) == true:
                return true
            colors[v] = 0

    return false
```

```
function isSafe(G, v, color, colors):
    for each neighbor u of v in G:
        if colors[u] == color:
            return false
    return true
```

3-Graph Coloring Problem

Objective: Given an undirected graph $G = (V, E)$ and a total of K available colors, assign each vertex a color from $\{1,2,\ldots,K\}$ such that for every edge (u, v), the vertices u and v do not share the same color.

If successful, we obtain a valid **K-coloring** of the graph; if not, we conclude the graph cannot be colored with K colors without violating adjacency constraints.

```
function assignColor(G, v, K, colors):
    if v > |V|:
        return true

    for c in 1 to K:
        if isSafe(G, v, c, colors) == true:
            colors[v] = c
            if assignColor(G, v+1, K, colors) == true:
                return true
            colors[v] = 0

    return false
```

Try all schemes one by one!??

How can we optimize it?

3-Graph Coloring Problem

Objective: Given an undirected graph $G = (V, E)$ and a total of `K` available colors, assign each vertex a color from $\{1,2,\ldots,K\}$ such that for every edge (u, v), the vertices u and v do not share the same color.

If successful, we obtain a valid **`K`-coloring** of the graph; if not, we conclude the graph cannot be colored with `K` colors without violating adjacency constraints.

```
function assignColor(G, v, K, colors):
    if v > |V|:
        return true

    for c in 1 to K:
        if isSafe(G, v, c, colors) == true:
            colors[v] = c
            if assignColor(G, v+1, K, colors) == true:
                return true
            colors[v] = 0

    return false
```

Try all schemes one by one!??

How can we optimize it?

Combine with **heuristics** (vertex ordering, forward checking) for better performance.

4-Summary

1. What is CSP?
2. What is Pruning?
3. What is Backtracking?
4. How to implement backtracking?
5. Exponential Complexity - worst case