UNICAM
Università di Camerino
1336

Constraint Programming with Google OR-Tools

Google OR-Tools

What is OR Tools?

OR-Tools is an open source software suite for **optimization, for solve problems in vehicle routing, network flows, integer and linear programming, and constraint programming.**

Google OR-Tools

OR Tools: Identify the type of Solvers

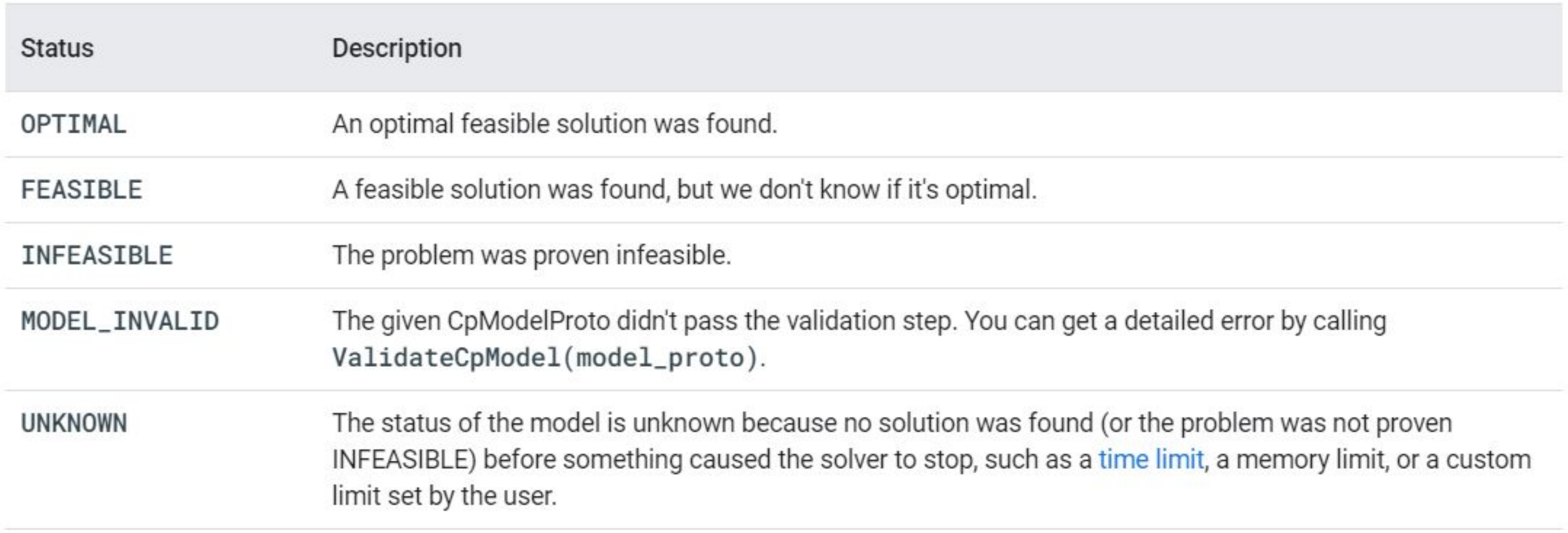
There are **many different types of optimization problems in the world**. For each type of problem, there are **different approaches and algorithms for finding an optimal solution**. Before you can start writing a program to solve an optimization problem, you need to **identify what type of problem you are dealing with, and then choose an appropriate solver** — an algorithm for finding an optimal solution.

These are the types of problems that OR-Tools solves:

- Linear optimization
- Mixed-Integer optimization
- Constraint optimization
- Network flows/ Routing
- Assignment
- Scheduling

The Employee Scheduling problem:CP-SAT Solver

The CP-SAT Solver returns one of the status values shown in the table below:



Status	Description
`OPTIMAL`	An optimal feasible solution was found.
`FEASIBLE`	A feasible solution was found, but we don't know if it's optimal.
`INFEASIBLE`	The problem was proven infeasible.
`MODEL_INVALID`	The given CpModelProto didn't pass the validation step. You can get a detailed error by calling `ValidateCpModel(model_proto)`.
`UNKNOWN`	The status of the model is unknown because no solution was found (or the problem was not proven INFEASIBLE) before something caused the solver to stop, such as a time limit, a memory limit, or a custom limit set by the user.

OR-Tools: Constraints Programming

Employee Scheduling example:

The problem arises when companies that operate continuously — such as factories — need to create weekly schedules for their employees.

The company runs three 8-hour shifts per day and assigns three of its four employees to different shifts each day, while giving the fourth the day off.

Actors:

- Company
- 3 runs every day (8-hours)
- 4 Employees
- Every day a different employee has a day off

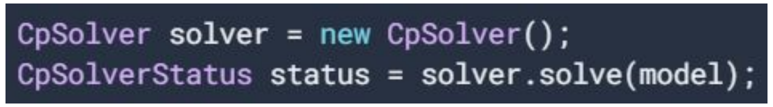
The Employee Scheduling problem

Organizations whose employees work multiple shifts need to schedule sufficient workers for each daily shift. Typically, the schedules will have constraints, such as:

“No employee should work two shifts in a row“

Finding a schedule that satisfies all constraints can be computationally difficult.

OR-Tools provide the CP-SAT Solver for solve such problems:



```
CpSolver solver = new CpSolver();
CpSolverStatus status = solver.solve(model);
```

The Employee Scheduling problem

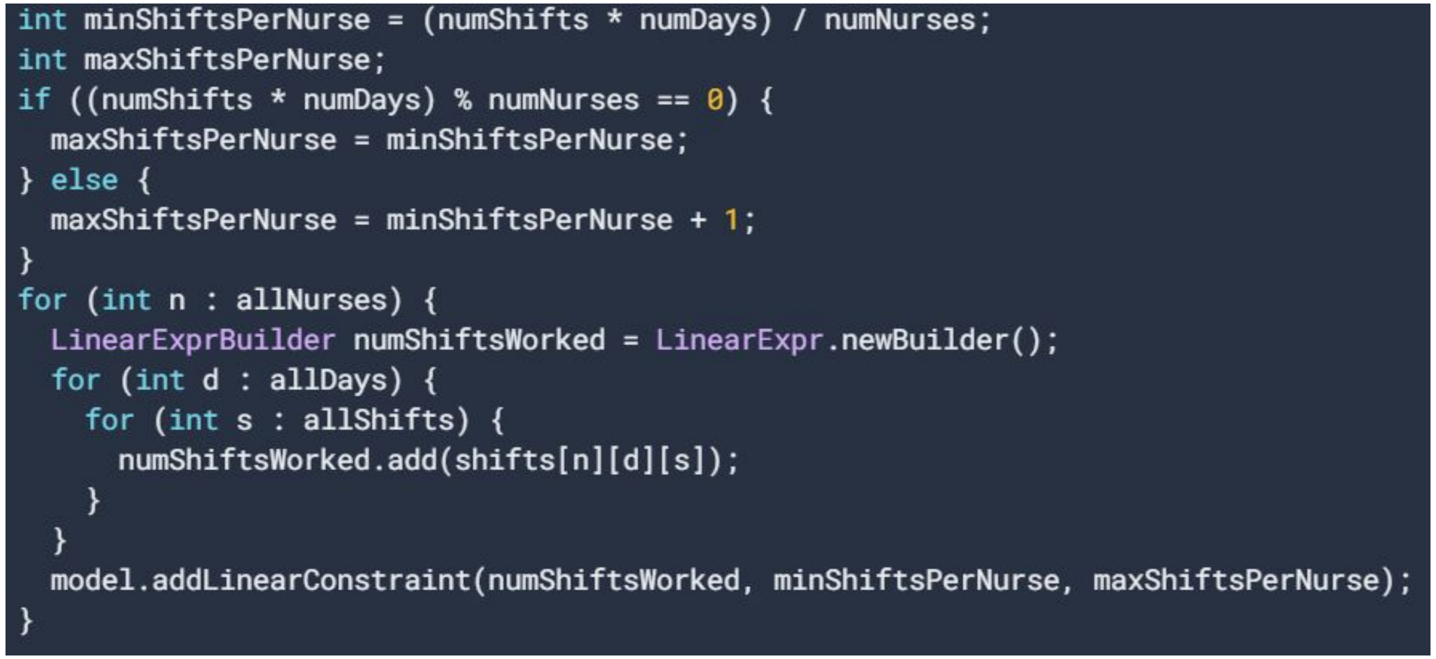
In particular the example is about a **nurse scheduling problem.** A **hospital supervisor needs to create a schedule** for **four nurses** over a **three-day period**, subject to the following conditions:

- **Each day is divided into three 8-hour shifts**
- **Every day, each shift is assigned to a single nurse, and no nurse works more than one shift.**
- **Each nurse is assigned to at least two shifts during the three-day period.**

The Employee Scheduling problem: step by step

Next, we need to **assign shifts to nurses as evenly as possible:**

Since there are **nine shifts over the three-day period**, we can assign **two shifts to each of the four nurses**. After that there will be **one shift left over**, which can be assigned to any nurse.

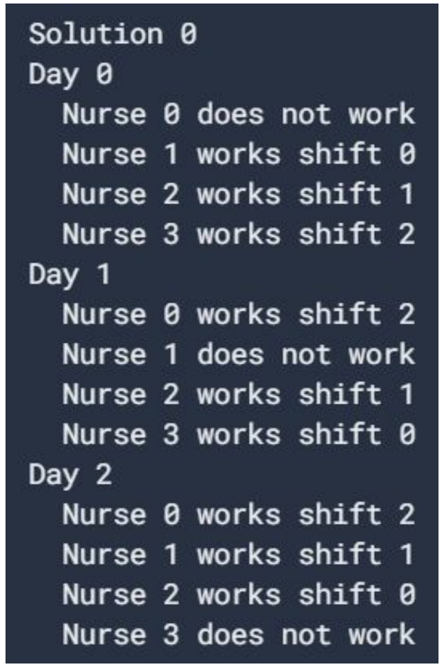


```
int minShiftsPerNurse = (numShifts * numDays) / numNurses;
int maxShiftsPerNurse;
if ((numShifts * numDays) % numNurses == 0) {
  maxShiftsPerNurse = minShiftsPerNurse;
} else {
  maxShiftsPerNurse = minShiftsPerNurse + 1;
}
for (int n : allNurses) {
  LinearExprBuilder numShiftsWorked = LinearExpr.newBuilder();
  for (int d : allDays) {
    for (int s : allShifts) {
      numShiftsWorked.add(shifts[n][d][s]);
    }
  }
  model.addLinearConstraint(numShiftsWorked, minShiftsPerNurse, maxShiftsPerNurse);
}
```

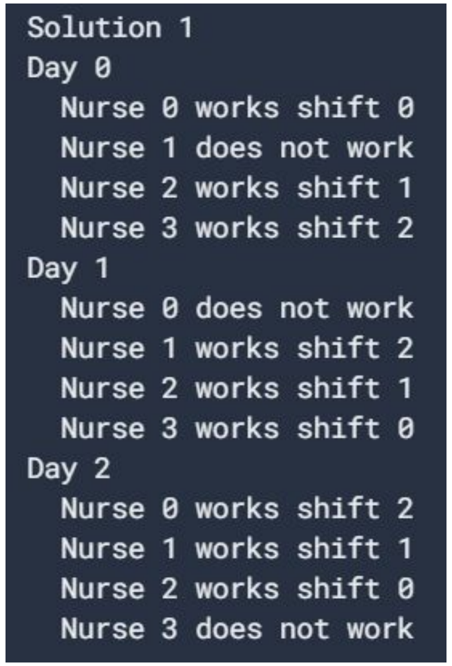
minShiftsPerNurse: used to distribute the shifts evenly.

Note: *If this is not possible, because the total number of shifts is not divisible by the number of nurses, some nurses will be assigned one more shift.*

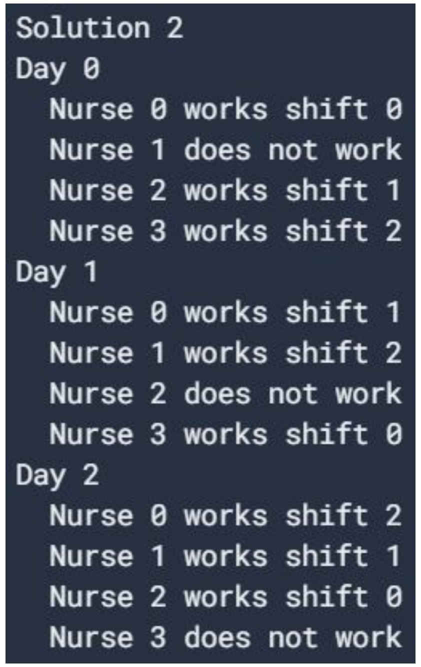
The Employee Scheduling problem: Solutions



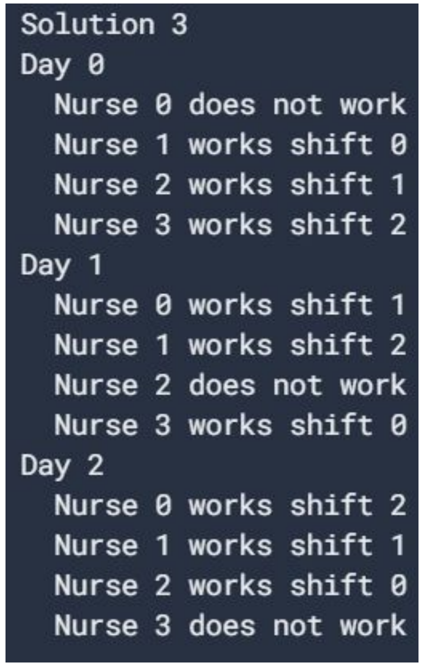
```
Solution 0
Day 0
   Nurse 0 does not work
   Nurse 1 works shift 0
   Nurse 2 works shift 1
   Nurse 3 works shift 2
Day 1
   Nurse 0 works shift 2
   Nurse 1 does not work
   Nurse 2 works shift 1
   Nurse 3 works shift 0
Day 2
   Nurse 0 works shift 2
   Nurse 1 works shift 1
   Nurse 2 works shift 0
   Nurse 3 does not work
```



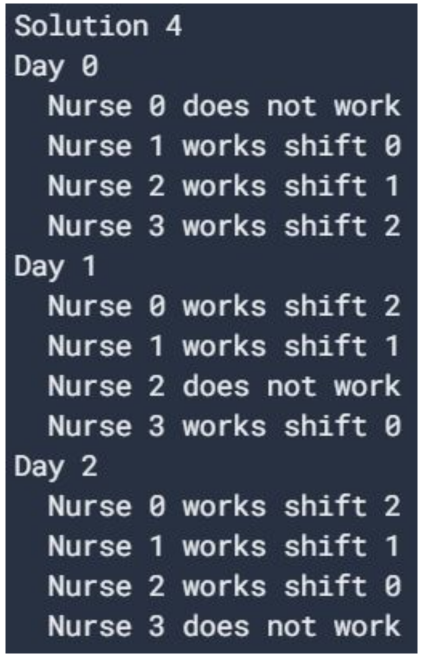
```
Solution 1
Day 0
   Nurse 0 works shift 0
   Nurse 1 does not work
   Nurse 2 works shift 1
   Nurse 3 works shift 2
Day 1
   Nurse 0 does not work
   Nurse 1 works shift 2
   Nurse 2 works shift 1
   Nurse 3 works shift 0
Day 2
   Nurse 0 works shift 2
   Nurse 1 works shift 1
   Nurse 2 works shift 0
   Nurse 3 does not work
```



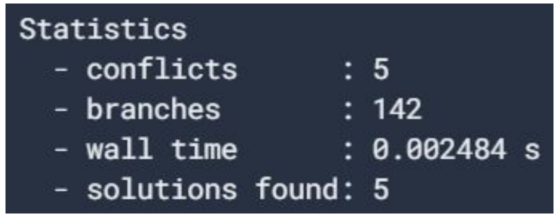
```
Solution 2
Day 0
   Nurse 0 works shift 0
   Nurse 1 does not work
   Nurse 2 works shift 1
   Nurse 3 works shift 2
Day 1
   Nurse 0 works shift 1
   Nurse 1 works shift 2
   Nurse 2 does not work
   Nurse 3 works shift 0
Day 2
   Nurse 0 works shift 2
   Nurse 1 works shift 1
   Nurse 2 works shift 0
   Nurse 3 does not work
```



```
Solution 3
Day 0
   Nurse 0 does not work
   Nurse 1 works shift 0
   Nurse 2 works shift 1
   Nurse 3 works shift 2
Day 1
   Nurse 0 works shift 1
   Nurse 1 works shift 2
   Nurse 2 does not work
   Nurse 3 works shift 0
Day 2
   Nurse 0 works shift 2
   Nurse 1 works shift 1
   Nurse 2 works shift 0
   Nurse 3 does not work
```



```
Solution 4
Day 0
   Nurse 0 does not work
   Nurse 1 works shift 0
   Nurse 2 works shift 1
   Nurse 3 works shift 2
Day 1
   Nurse 0 works shift 2
   Nurse 1 works shift 1
   Nurse 2 does not work
   Nurse 3 works shift 0
Day 2
   Nurse 0 works shift 2
   Nurse 1 works shift 1
   Nurse 2 works shift 0
   Nurse 3 does not work
```



```
Statistics
  - conflicts      : 5
  - branches       : 142
  - wall time      : 0.002484 s
  - solutions found: 5
```

There are 4 choices for the one nurse who works an extra shift. Having chosen that nurse, there are 3 shifts the nurse can be assigned to on each of the 3 days, **so the number of possible ways to assign the nurse with the extra shift is 4 · 33 = 108**. After assigning this nurse, there are two remaining unassigned shifts on each day..

The Job Shop problem

One common scheduling problem is the ***job shop***, in which **multiple jobs are processed on several machines**. **Each job consists of a sequence of tasks, which must be performed in a given order, and each task must be processed on a specific machine**.

The problem is to schedule the tasks on the machines so **as to minimize the *length* of the schedule**—the time it takes for all the jobs to be completed.

There are several constraints for the job shop problem:

- **No task for a job can be started until the previous task for that job is completed;**
- **A machine can only work on one task at a time;**
- **A task, once started, must run to completion.**

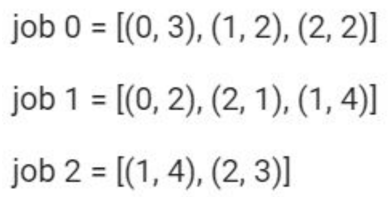
The Job Shop problem

Each task is labeled by a pair of numbers **(m,p)** where:

- **m** is the number of the machine the task must be processed on;
- **p** is the processing time of the task (the amount of time it requires).

In the example, job 0 has three tasks. The first, (0, 3), must be processed on machine 0 in 3 units of time. The second, (1, 2), must be processed on machine 1 in 2 units of time, and so on.

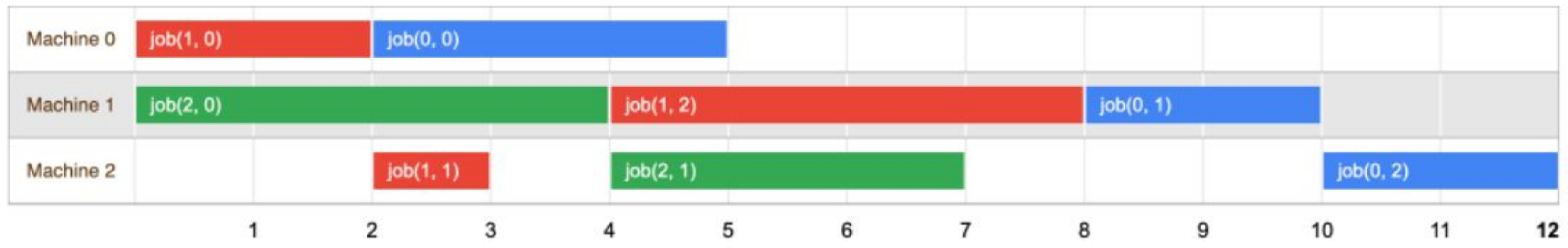
Altogether, there are eight tasks:



```
job 0 = [(0, 3), (1, 2), (2, 2)]

job 1 = [(0, 2), (2, 1), (1, 4)]

job 2 = [(1, 4), (2, 3)]
```

The Job Shop problem: A solution for the problem

A solution to the job shop problem is an assignment of a start time for each task, which meets the constraints given above. The diagram below shows one possible solution for the problem:



job 0 = [(0, 3), (1, 2), (2, 2)]

job 1 = [(0, 2), (2, 1), (1, 4)]

job 2 = [(1, 4), (2, 3)]

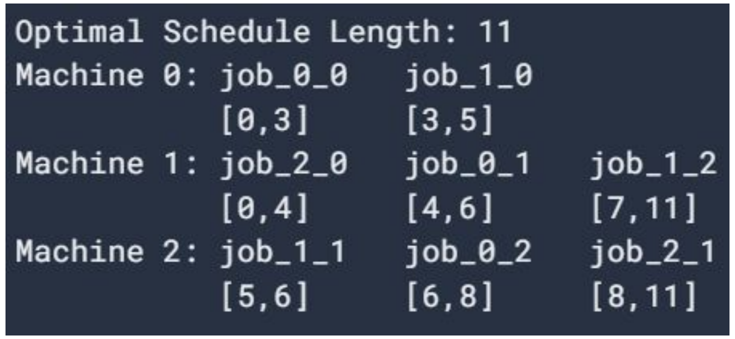
All the tasks for each job are scheduled at non-overlapping time intervals, in the order given by the problem.

The length of this solution is 12, which is the first time when all three jobs are complete.

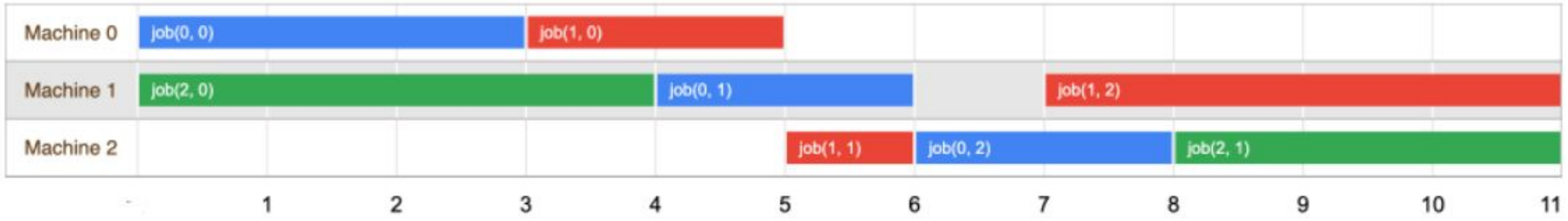
However, as you will see below, **this is not the optimal solution to the problem!!**

The Employee Scheduling problem: Result

8. Display the results:



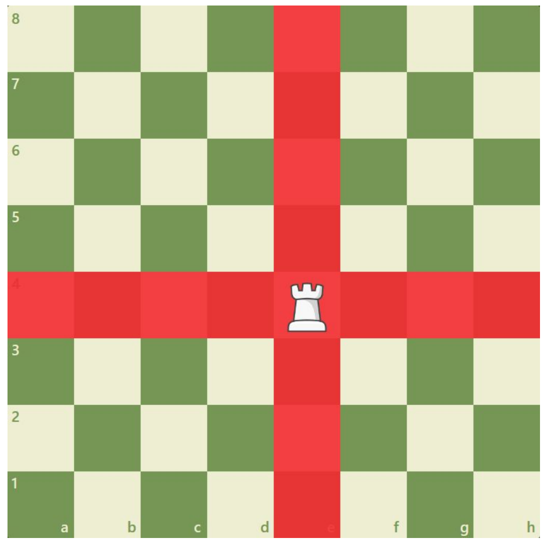
```
Optimal Schedule Length: 11
Machine 0: job_0_0   job_1_0
           [0,3]     [3,5]
Machine 1: job_2_0   job_0_1   job_1_2
           [0,4]     [4,6]     [7,11]
Machine 2: job_1_1   job_0_2   job_2_1
           [5,6]     [6,8]     [8,11]
```



Machine 1 might wonder why job_1_2 was scheduled at time 7 instead of time 6. Both are valid solutions, but **the objective is to minimize the makespan**. Moving job_1_2 earlier wouldn't reduce the makespan, **so the two solutions are equal from the solver's perspective**.

The n-Towers Problem

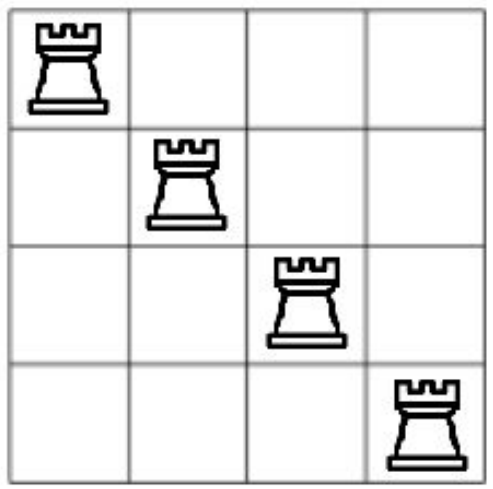
The n-Towers problem is a constraint programming (CP) by a combinatorial problem based on the game of chess. In chess, a **tower** can attack horizontally and vertically. The n-Towers problem asks:



*How can **N towers** be placed on an **NxN** chessboard so that **no two of them attack each other**?*

The n-Towers Problem: a possible solution

Below, you can see one **possible** solution to the N-queens problem for N = 4.



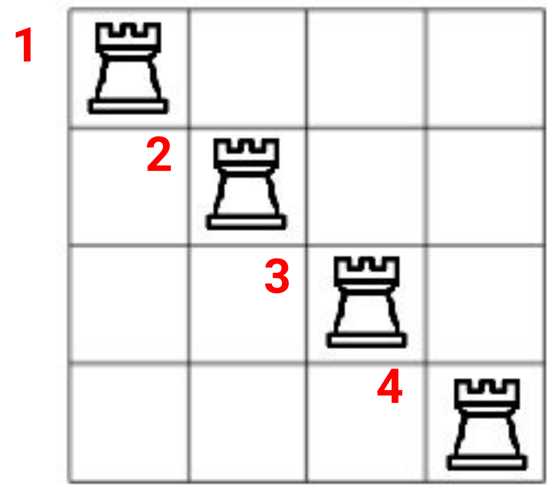
No two towers are on the same row or column

Note that this isn't an optimization problem: we want to find all possible solutions, rather than one optimal solution, which makes it a natural candidate for constraint programming.

CP approach to the N-towers problem

A **CP solver** works by systematically **trying all possible assignments of values to the variables in a problem, to find the feasible solutions**.

In the 4-towers problem, **the solver starts at the leftmost column and successively places one tower in each column**, at a location that is not attacked by any previously placed towers.



CP approach to the N-towers problem

Constraints:

1. There must be one tower in each column;
2. There must be one tower in each row;
3. N must be equal for number of towers, rows and columns.

Propagation and backtracking

There are **two key elements** to a constraint programming search:

- **Propagation — Each time the solver assigns a value to a variable, the constraints add restrictions on the possible values of the unassigned variables**. These restrictions propagate to future variable assignments. For example, in the 4-towers problem, each time the solver places a tower, it can't place any other queens on the row and column the current tower is on. **Propagation can speed up the search significantly by reducing the set of variable values the solver must explore**.

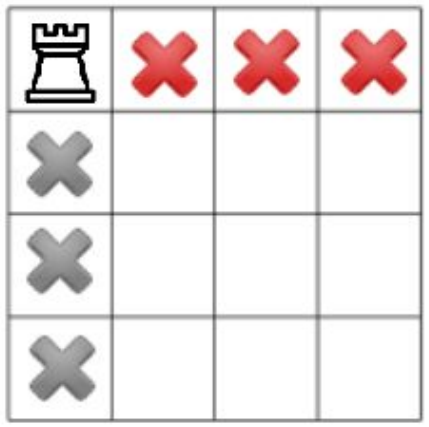
- **Backtracking occurs when either the solver can't assign a value to the next variable, due to the constraints, or it finds a solution.** In either case, the solver backtracks to a previous stage and changes the value of the variable at that stage to a value that hasn't already been tried. In the 4-towers example, this means moving a tower to a new square on the current column.

Propagation and backtracking

How constraint programming uses propagation and backtracking to solve the 4-towers problem?

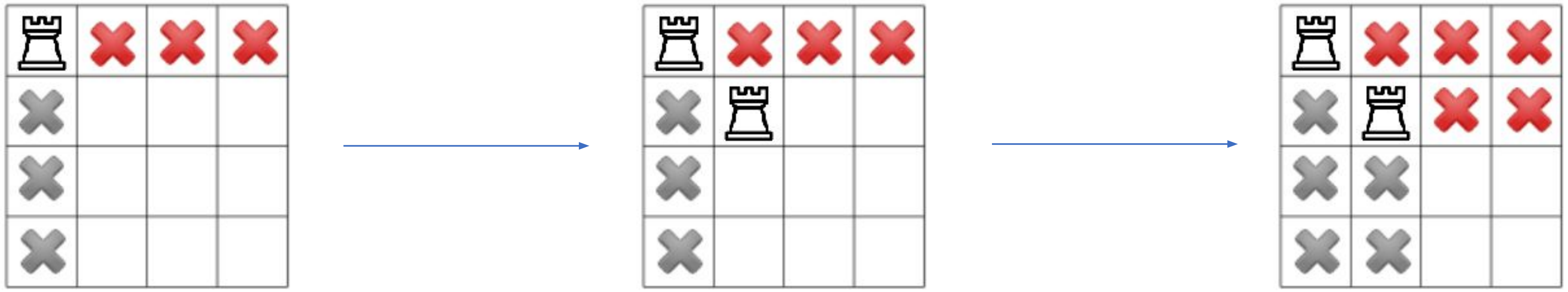
Let's suppose the solver starts by arbitrarily placing a tower in the upper left corner. That's a hypothesis of sorts; perhaps it will turn out that no solution exists with a tower in the upper left corner.

Given this hypothesis, what constraints can we propagate? One constraint is that there can be only one tower in a column (the gray Xs below), and another constraint prohibits two towers on the same row (the red Ys below).



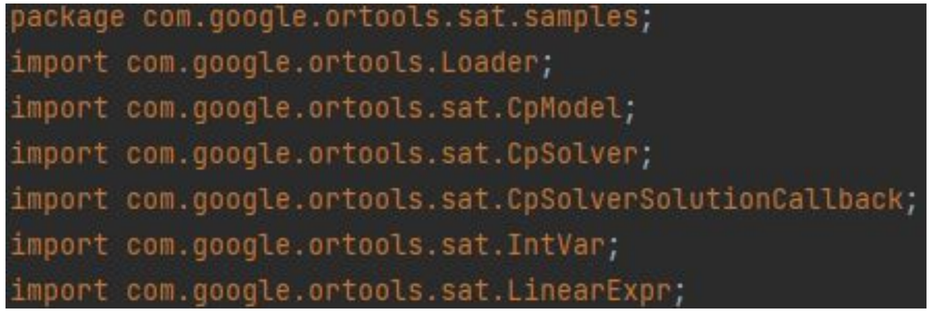
Propagation and backtracking

Our constraints propagated, we can test out another hypothesis, and place a second tower on one of the available remaining squares. Our solver might decide to place in it the first available square in the second column:



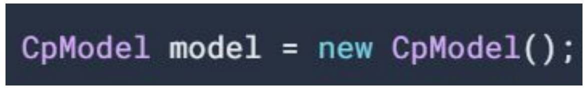
The n-Towers Problem

1. **Import Libraries**



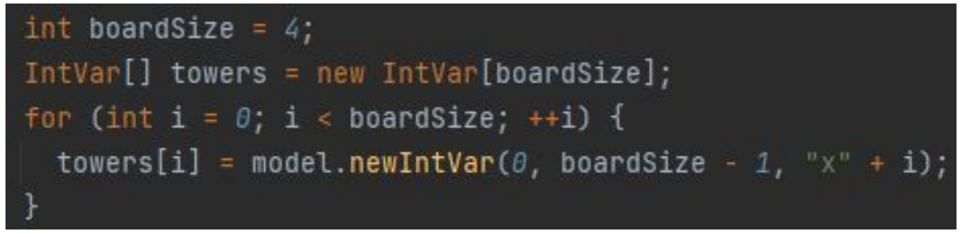
```java
package com.google.ortools.sat.samples;
import com.google.ortools.Loader;
import com.google.ortools.sat.CpModel;
import com.google.ortools.sat.CpSolver;
import com.google.ortools.sat.CpSolverSolutionCallback;
import com.google.ortools.sat.IntVar;
import com.google.ortools.sat.LinearExpr;
```

2. **Declare the model**



```java
CpModel model = new CpModel();
```

3. **Create the variables**



```java
int boardSize = 4;
IntVar[] towers = new IntVar[boardSize];
for (int i = 0; i < boardSize; ++i) {
  towers[i] = model.newIntVar(0, boardSize - 1, "x" + i);
}
```

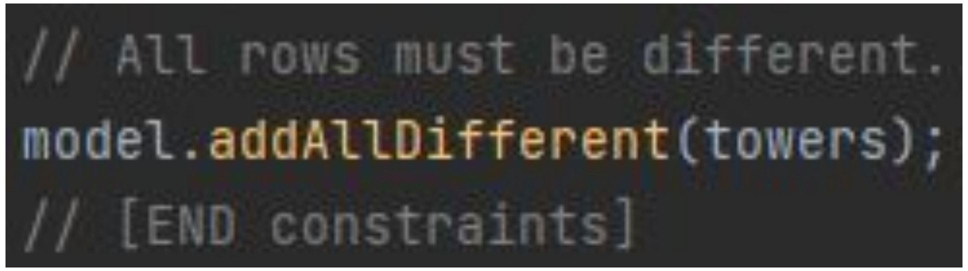
Here we assume that towers[j] is the row number for the queen in column j. In other words, **towers[j] = i means there is a tower in row i and column j.**

Note: When drawing a diagram of a solution, you will get a different picture depending on whether the rows are ordered from bottom to top or top to bottom (and whether the columns are ordered from left to right or vice versa). However, the ordering doesn't change the set of all possible solutions, just the way they are represented in a diagram.

The n-Towers Problem

4. Create the constraints

The code uses the ***AddAllDifferent*** method, which requires all the elements of a variable array to be different.



```
// All rows must be different.
model.addAllDifferent(towers);
// [END constraints]
```

No two towers on the same row

Applying the solver's **AllDifferent** method to towers forces the values of towers[j] to be different for each j, which means that all towers must be in different rows.

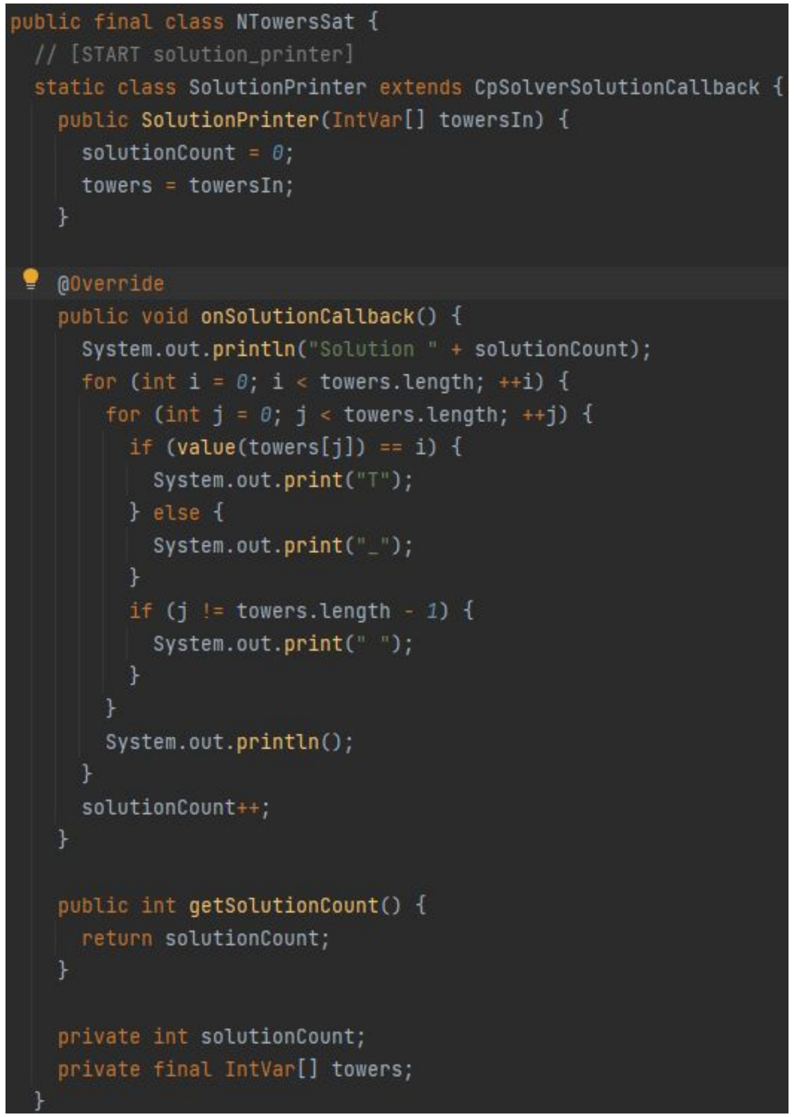
No two towers on the same column

This constraint is implicit in the definition of towers. Since no two elements of towers can have the same index, no two towers can be in the same column.

The n-Towers Problem

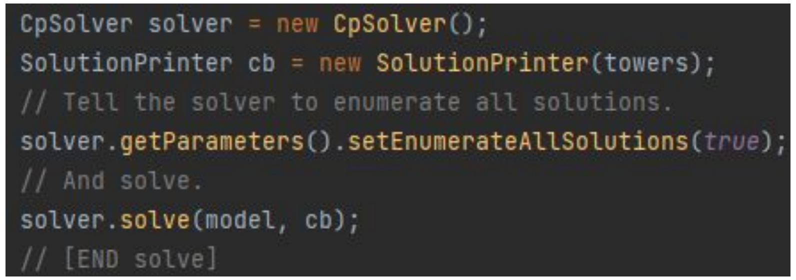
5. Create a solution pointer

To print all solutions to the n-Towers problem, you need to pass a callback, called a *solution printer*, to the CP-SAT solver. The callback prints each new solution as the solver finds it.

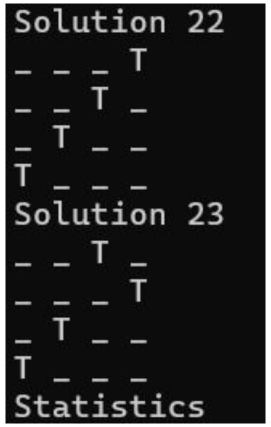


```java
public final class NTowersSat {
  // [START solution_printer]
  static class SolutionPrinter extends CpSolverSolutionCallback {
    public SolutionPrinter(IntVar[] towersIn) {
      solutionCount = 0;
      towers = towersIn;
    }

    @Override
    public void onSolutionCallback() {
      System.out.println("Solution " + solutionCount);
      for (int i = 0; i < towers.length; ++i) {
        for (int j = 0; j < towers.length; ++j) {
          if (value(towers[j]) == i) {
            System.out.print("T");
          } else {
            System.out.print("_");
          }
          if (j != towers.length - 1) {
            System.out.print(" ");
          }
        }
        System.out.println();
      }
      solutionCount++;
    }

    public int getSolutionCount() {
      return solutionCount;
    }

    private int solutionCount;
    private final IntVar[] towers;
  }
```

6. Call the solver and display the results



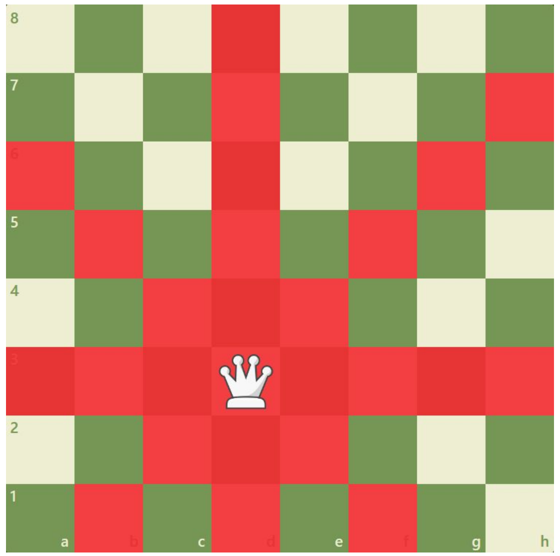
```java
CpSolver solver = new CpSolver();
SolutionPrinter cb = new SolutionPrinter(towers);
// Tell the solver to enumerate all solutions.
solver.getParameters().setEnumerateAllSolutions(true);
// And solve.
solver.solve(model, cb);
// [END solve]
```



```
Solution 22
_ _ _ T
_ _ T _
_ T _ _
T _ _ _
Solution 23
_ _ T _
_ _ _ T
_ T _ _
T _ _ _
Statistics
```

The n-Queens Problem: Assignment

The n-Queens problem is a constraint programming (CP) by a combinatorial problem based on the game of chess. In chess, a **queen** can attack horizontally, vertically, and **diagonally**. The n-Queens problem asks:



*How can **N queens** be placed on an **NxN** chessboard so that **no two of them attack each other**?*

CP approach to the N-queens problem

Some advices for the assignment:

Constraints:

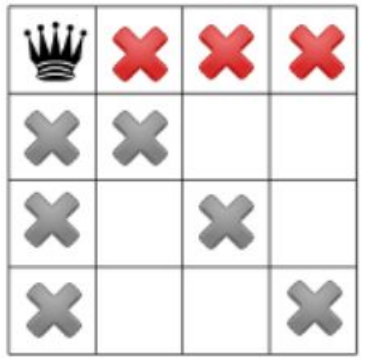
1. There must be one queen in each column;
2. There must be one queen in each row;
3. N must be equal for number of towers, rows and columns;
4. **There must be one queen in each diagonal.**

Propagation and backtracking

How constraint programming uses propagation and backtracking to solve the 4-queens problem?

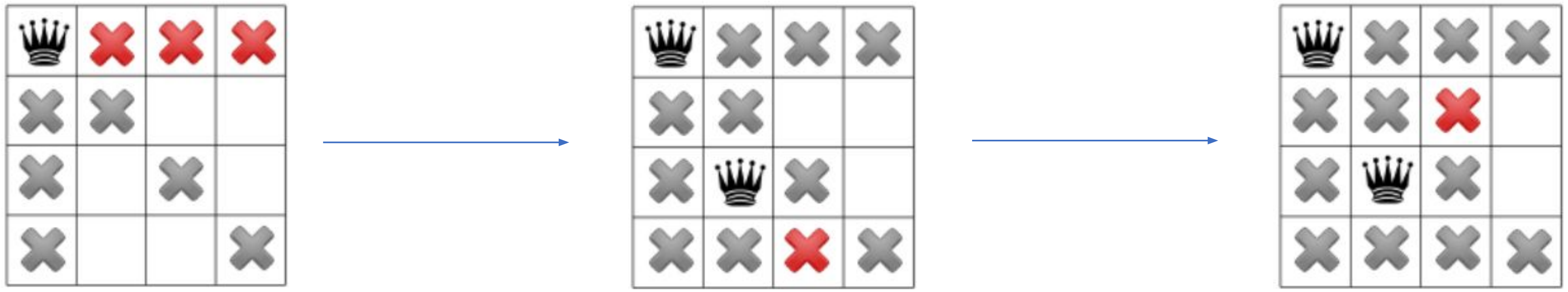
Let's suppose the solver starts by arbitrarily placing a tower in the upper left corner. That's a hypothesis of sorts; perhaps it will turn out that no solution exists with a tower in the upper left corner.

Given this hypothesis, what constraints can we propagate? One constraint is that there can be only one queen in a row (the red Xs below), and another constraint prohibits two queens on the same diagonal and column (the grey Xs below)



Propagation

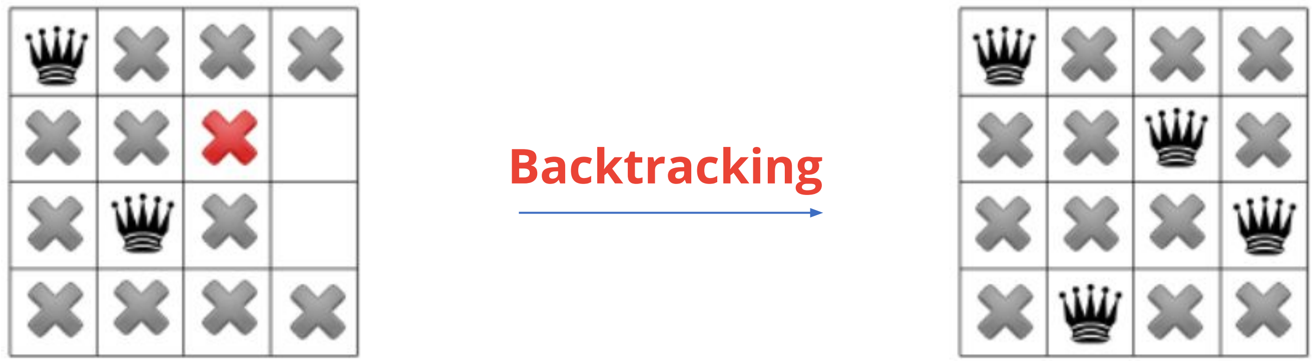
Our constraints propagated, we can test out another hypothesis, and place a second queen on one of the available remaining squares. Our solver might decide to place in it the first available square in the second column:



After propagating the diagonal constraint, we can see that it leaves no available squares in either the third column or last row.

Backtracking

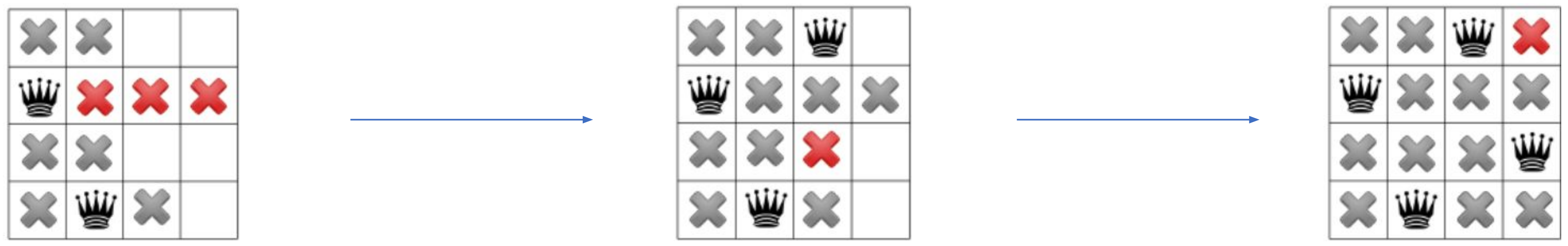
With no solutions possible at this stage, we need to **backtrack**. One option is for the solver to choose the other available square in the second column. However, constraint propagation then forces a queen into the second row of the third column, leaving no valid spots for the fourth queen:



After propagating the diagonal constraint, we can see that it leaves no available squares in either the third column or last row.

Another Backtracking

And so the solver must backtrack again, this time all the way back **to the placement of the first queen**. We have now shown that no solution to the queens problem will occupy a corner square. Since there can be no queen in the corner, the solver moves the first queen down by one, and propagates, leaving only one spot for the second queen:



Propagating again reveals only one spot left for the third queen.