Lecture 14

The $n$-Queens Problem

The $n$-queens problem is to place $n$ queens on an $n \times n$ chessboard such that no queens threaten each other. If you aren’t familiar with the rules of Chess: a queen is a chess piece that move horizontally, vertically, or diagonally across a chessboard, which is typically an 8 $\times$ 8 matrix. A queen can “kill” any piece that it can move to, so it is unsafe to be on the same horizontal, vertical, or diagonal line as a queen. The $n$-queens problem is to arrange $n$-queens on a generalized $n \times n$ chessboard, such that no pair of queens can kill each other.

The $n$-queens problem is a canonical example of a constraint-satisfaction problem that can be solved by backtracking search. In this lecture, we’ll begin with a naive implementation of backtracking search, and then refine it to use constraint-propagation, which will make it a lot faster.

A trait for chessboards

Since we are going to go through a few different representations of chessboards, it will help to factor out generic code that prints the representation of chessboards. The ChessBoardLike trait in fig. 24.1 defines a toString method that prints a chessboard of queens, where each queen is printed as Q and each blank space appears as .. This printing method requires the implementing class to have a field that specifies that dimensions of the chessboard and set of coordinates that describe where the queens are placed.

Backtracking Search

The core idea of any solution to the $n$-queens problem is to write a recursive function (called solve) that places 1 new queen on the current board in a position where it does not threaten any existing queen and then recursively calls solve to place the remaining queens. The function terminates successfully when $n$ queens have been placed on the board. The function terminates with an error if there are no positions where the next queen can be safely placed. In any application of solve, there may be several positions where a queen can be safely placed. The key to backtracking is to try a new position if the recursive application produces an error.

Figure 24.2 shows a simple implementation of this idea. The key function is the canPlace predicate which

```
trait ChessBoardLike {
  val dim: Int
  val solution: Set[(Int, Int)]

  override def toString(): String = {
    val builder = new StringBuilder((dim + 1) * dim)
    for (y <- 0 to (dim - 1)) {
      for (x <- 0 to (dim - 1)) {
        if (solution.contains((x, y))) { builder += 'Q' }
        else { builder += '.' }
      }
      builder += \n    }
    builder.toString
  }
}
```

Figure 24.1: A trait for chessboards.
class NaiveQueens(val dim: Int, val solution: Set[(Int, Int)]) extends ChessBoardLike {

  def canPlace(x: Int, y: Int): Boolean = solution forall {
    case (x1, y1) => x != x1 && y != y1 && x + y != x1 + y1 && x - y != x1 - y1
  }

  def solveRec(coords: List[(Int, Int)]): Option[NaiveQueens] = coords match {
    case (x, y) :: rest if canPlace(x, y) => {
      val optSol = (new NaiveQueens(dim, solution + ((x, y)))).solve()
      optSol.orElse(solveRec(rest))
    }
    case Nil => None
    case _ :: rest => solveRec(rest)
  }

  def solve(): Option[NaiveQueens] = (solution.size == dim) match {
    case true => Some(this)
    case false => solveRec(0.until(dim).flatMap(x => 0.until(dim).map(y => (x, y))).toList)
  }
}

Figure 24.2: A naive solution to the n-queens problem.

determines if a new queen maybe placed at coordinates \((x, y)\) by checking if there are any existing queens in the set \(solution\) on the same row, column, diagonal, or antidiagonal.

We can run the solver as follows:

\[
\text{(new NaiveQueens(n, Set())}.solve())
\]

With \(n = 11\), the solver produces a solution in less than a second on my laptop, with \(n = 12\), it takes 55 seconds, and \(n = 13\) would take much longer.

**Constraint Propagation**

Figure 24.3 shows a variant of the naive solver that is substantially faster. The key idea to store a set of locations where a queen can be placed without violating any constraints and then prune the set whenever a new queen is placed.
class OptQueens(val dim: Int, val solution: Set[(Int, Int)],
  available: List[(Int, Int)]) extends ChessBoardLike {

  def place(x: Int, y: Int): OptQueens = {
    new OptQueens(dim,
      solution + ((x, y)),
      available.filter(p => {
        val (x1, y1) = p
        !(x == x1 || y == y1 || x + y == x1 + y1 || x - y == x1 - y1)
      } ))
  }

  def solveRec(coords: List[(Int, Int)]: Option[OptQueens] = coords match {
    case (x, y) :: rest => this.place(x, y).solve.orElse(solveRec(rest))
    case _ :: rest => solveRec(rest)
    case Nil => None
  }

  def solve(): Option[OptQueens] = (solution.size == dim) match {
    case true => Some(this)
    case false => solveRec(available.toList)
  }
}

object OptQueens {
  def empty(dim: Int) = {
    val available = 0 until (dim).flatMap(x => 0 until(dim).map(y => (x, y))).toList
    new OptQueens(dim, Set(), util.Random.shuffle(available))
  }
}

Figure 24.3: A constraint-propagating solution to the n-queens problem.