IP Lecture 20: Solving Sudoku Problems

Joe Pitt-Francis

—with thanks to Gavin Lowe—

Sudoku problems

In a Sudoku problem, you are given a 9 by 9 grid of squares, with some squares containing digits between 1 and 9 (e.g. the left-hand grid below). You have to fill in the rest of the squares so the same digit does not appear twice in the same row, column or 3 by 3 block (e.g. the right-hand grid below).

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

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

The idea

We will maintain a collection of partial solutions, each of which extends the initial position, in a way consistent with the rules. We start with just the initial position. We will repeatedly:

- Pick one of the partial solutions;
- If it is complete, print it out;
- Otherwise, pick a blank position (i, j) in the partial solution;
- For each digit d, if d can legally be played in position (i, j), then create a new partial solution by adding that play to the current partial solution, and add it to the collection.

Partial solutions

We will represent each partial solution by an object, keeping track of the digits placed so far. We will need the following operations on partial solutions.

- An operation to initialise a partial solution in the starting position, say based on a description held in a file;
- An operation to print out a completed solution;
- An operation to test whether a partial solution is complete;
- An operation to pick a blank position in which we can play next;
- An operation to test whether we can legally play digit d in position (i, j);
- An operation to create a new partial solution by adding digit d in position (i, j).

Partial solutions

Each partial solution will correspond to the following trait.

```
/** state:  board : {0..8} × {0..8} ⇸ {1..9}
  * DTI:  ∀(i, j), (i, j′) ∈ dom board • j ≠ j′ ⇒ board(i, j) ≠ board(i, j′) ∧
  *       ∀(i, j), (i′, j) ∈ dom board • i ≠ i′ ⇒ board(i, j) ≠ board(i′, j) ∧
  *       ∀(i, j), (i′, j′) ∈ dom board •
  *           (i, j) ≠ (i′, j′) ∧ i div 3 = i′ div 3 ∧ j div 3 = j′ div 3 ⇒
  *               board(i, j) ≠ board(i′, j′)
  */
trait Partial{
  ...
}
```

The partial function *board* records which digits have been placed so far. The DTI captures the rules of the puzzle. We'll write $DTI_A(board)$ to indicate that *board* satisfies the DTI.

Partial solutions

```
trait Partial{
  /** Initialise from a file.  Assume file presents starting
    * position, using "." to represent a blank position.
    * pre:  fname contains 9 lines, each containing 9 characters
    *       from {1..9} or ".", and obeying the rules of the DTI.
    * post: ∀i, j ∈ {0..8} •
    *          ∀d ∈ {1..9} • fname(i, j) = d ⇔ board(i, j) = d ∧
    *          fname(i, j) = "." ⇔ (i, j) ∉ dom board
    * where fname(i, j) denotes jth character of line i of fname.
    */
  def init(fname: String)

  /** Is the partial solution complete?
    * post: board = board₀ ∧ returns  dom board = {0..8} × {0..8} */
  def complete : Boolean
```

Partial solutions

```
trait Partial{
  ...
  /** Print partial solution.
    * pre:  complete
    * post:  board = board₀ ∧ prints 9 lines,
    *        with line i containing board(i,0)...board(i,8). */
  def printPartial

  /** Find a blank position.
    * pre:  ¬complete
    * post:  board = board₀ ∧ returns (i,j) s.t. (i,j) ∉ dom board. */
  def nextPos : (Int,Int)
```

`complete` is a pure function (i.e. deterministic with no side effects) so we can use it in preconditions to stand for the value it returns.

Partial solutions

```
trait Partial{
  ...
  /** Can we play value d in position (i,j)?
    * pre: i,j ∈ {0..8} ∧ d ∈ {1..9}
    * post: board = board0 ∧ returns DTI_A(board ⊕ {(i,j) → d}). */
  def canPlay(i: Int, j: Int, d: Int) : Boolean

  /** Create a new partial solution, extending this one by
    * playing d in position (i,j).
    * pre: i,j ∈ {0..8} ∧ d ∈ {1..9} ∧ canPlay(i,j,d)
    * post: board = board0 ∧
    * returns p s.t. p.board = board ⊕ {(i,j) → d}.
    */
  def play(i: Int, j: Int, d: Int) : Partial
}
```

The main algorithm

Later, we will implement `Partial` using a class `SimplePartial`. Once we have done this, we can create the initial partial solution as follows:

```
val p0 = new SimplePartial
p0.init(fname)
```

Recall that we will need to keep a collection of partial solutions. We will do this using a stack. We initialise the stack as follows:

```
val stack = new scala.collection.mutable.Stack[Partial]
stack.push(p0)
```

The main algorithm

```
while(stack.nonEmpty){
  val p = stack.pop
  if(p.complete){  // done!
    p.printPartial
  }
  else{
    // Choose position to play
    val (i, j) = p.nextPos
    // Consider all values to play there
    for(d <- 1 to 9; if p.canPlay(i, j, d)){
      val p1 = p.play(i, j, d); stack.push(p1)
    }
  }
} // end of while
```

Correctness

Let's write $\mathit{completions}(p)$ for all ways of completing p:

$$\mathit{completions}(p) = \{p' \mid p.\mathit{board} \subseteq p'.\mathit{board} \wedge \texttt{complete}(p')\}$$

We want to show that the program prints all of $\mathit{completions}(\texttt{p0})$.

The invariant is

$$\mathit{completions}(\texttt{p0}) = \bigcup\{\mathit{completions}(p) \mid p \in \texttt{stack}\} \cup \text{those solutions printed so far.}$$

Initially, the stack contains just `p0` (and nothing has been printed), so the invariant holds.

Correctness

Consider when we remove `p` from the stack. This, in effect, removes *completions*(`p`) from the right-hand side of the invariant.

If `complete(p)`, then *completions*(`p`) $=$ {`p`}; we print `p`, thereby adding *completions*(`p`) back to the right-hand side of the invariant.

Otherwise, suppose $p' \in$ *completions*(`p`). Let `(i, j) = p.nextPos`; suppose p' has value `d` in position `(i, j)`; and let `p1 = p.play(i, j, d)`. Then $p' \in$ *completions*(`p1`), so when `p1` is pushed onto the stack, p' is added back to the right-hand side of the invariant.

Conversely, any completion of one of the `p1` pushed onto the stack is also a completion of `p`.

Finally, at the end the stack is empty, so we've printed all completions of `p`.

Why a stack?

Nothing in the correctness argument depends upon the fact that we used a stack: it would still work if we used a different set-like datatype.

However, using a stack affects the order in which we search the state space. Consider a graph whose nodes are partial solutions, and where there is an edge from p to p' if p' can be obtained from p by making a single move. Then using a stack means that we will perform a depth-first search.

Alternatively, if we had used a queue we would have performed a breadth-first search. This would probably have required more memory, because more partial solutions would have been held in intermediate states.

The Artificial Intelligence course will describe more searching strategies.

Partial solutions

We still need to implement the code to represent partial solutions. We will implement a class `SimplePartial` to extend the `Partial` trait.

We use a 9 by 9 array of integers, `contents`, to store the digits held in each square of the partial solution.

```
class SimplePartial extends Partial{
  private val contents = Array.ofDim[Int](9,9)
  ...
}
```

`contents(i)(j)` will store the special value 0 to represent an empty square.

Abs: $board = \{(i,j) \to \texttt{contents}(i)(j) \mid i,j \in \{0..8\} \land \texttt{contents}(i)(j) > 0\}$

DTI: $(\forall i,j \bullet \texttt{contents}(i)(j) \in \{0..9\}) \land DTI_A(board)$

Printing a partial solution

```
def printPartial = {
  for(i <- 0 until 9){
    for(j <- 0 until 9) print(contents(i)(j))
    println
  }
  println
}
```

Testing if a partial solution is complete

```
def complete : Boolean = {
  for(i <- 0 until 9; j <- 0 until 9)
    if(contents(i)(j) == 0) return false
  true
}
```

Finding a position to play

We will always play in the first blank square.

```
def nextPos : (Int,Int) = {
  for(i <- 0 until 9; j <- 0 until 9){
    if(contents(i)(j) == 0) return (i,j)
  }
  throw new RuntimeException("nextPos: No blank position")
}
```

We'll see a better solution later.

Testing whether a play is possible

```
def canPlay(i: Int, j: Int, d: Int) : Boolean = {
  // Check if d appears in row i
  for(j1 <- 0 until 9) if(contents(i)(j1) == d) return false
  // Check if d appears in column j
  for(i1 <- 0 until 9) if(contents(i1)(j) == d) return false
  // Check if d appears in this 3x3 block
  val basei = i/3*3; val basej = j/3*3
  for(i1 <- basei until basei+3; j1 <- basej until basej+3)
    if(contents(i1)(j1) == d) return false
  // All checks passed
  true
}
```

Extending a partial solution

```
def play(i: Int, j: Int, d: Int) : Partial = {
  // Clone this
  val p = new SimplePartial
  for(i1 <- 0 until 9; j1 <- 0 until 9){
    p.contents(i1)(j1) = contents(i1)(j1)
  }
  // And add d
  p.contents(i)(j) = d
  p
}
```

Initialising from a file

```
def init(fname: String) = {
  val lines = scala.io.Source.fromFile(fname).getLines
  for(i <- 0 until 9){
    val line = lines.next
    for(j <- 0 until 9){
      val c = line.charAt(j)
      if(c.isDigit) contents(i)(j) = c.asDigit
      else { assert(c == '.'); contents(i)(j) = 0 }
    }
  }
}
```

`lines` is an `Iterator[String]` (see next term); abstractly, it represents a sequence of `String`s. `lines.next` returns the next line.

A better implementation of `Partial`

The implementation using `SimplePartial` works pretty well. But we can do better. `SimplePartial.nextPos` always chooses to play in the first empty square. A better tactic is to choose the empty square that has the fewest legal plays. In order to find this square efficiently, we store, for each position `(i, j)`, the set of digits that can be played in `(i, j)`.

AdvancedPartial

```
class AdvancedPartial extends Partial{
  private val contents = Array.ofDim[Int](9,9)

  // pos(i)(j) is a list of all values
  // that can be placed in square (i,j)
  private val pos = Array.ofDim[List[Int]](9,9)
  ...
}
```

(Note that we might have used an array of `BitMapSets` for `pos`.)

The abstraction function is as for `SimplePartial`. The DTI is extended to describe `pos`.

Abs: $board = \{(i, j) \to \texttt{contents}(i)(j) \mid i, j \in \{0..8\} \wedge \texttt{contents}(i)(j) > 0\}$

DTI: $(\forall i, j \bullet \texttt{contents}(i)(j) \in \{0..9\}) \wedge DTI_A(board) \wedge$

$\forall i, j \bullet pos(i)(j) = [d \mid d \leftarrow [1..9], DTI_A(board \oplus \{(i, j) \to d\})]$

The unchanged operations

The `printPartial`, `complete` and `canPlay` operations are identical to as in `SimplePartial`.

How could we have avoided writing this code twice? See next term.

Making a play

To avoid repeated code, it's useful to define an operation to make a particular play in current partial, updating `pos` to maintain the DTI.

```
/** Play d in position (i,j), updating pos.
  * Pre: canPlay(i, j, d)
  * Post: board = board_0 (+) {(i,j) -> d}. */
private def makePlay(i: Int,j: Int,d: Int) = {
  contents(i)(j) = d; pos(i)(j) = d :: Nil
  // Remove d from row i
  for(j1 <- 0 until 9; if j1 != j) pos(i)(j1) = pos(i)(j1).filter(_ != d)
  // Remove d from column j
  for(i1 <- 0 until 9; if i1 !=i ) pos(i1)(j) = pos(i1)(j).filter(_ != d)
  // Remove d from this 3x3 block
  val basei = i/3*3; val basej = j/3*3
  for(i1 <- basei until basei+3;
      j1 <- basej until basej+3; if (i1,j1) != (i,j))
    pos(i1)(j1) = pos(i1)(j1).filter(_ != d)
}
```

init and play

`init(fname)` initialises each entry of `pos` to `(1 to 9).toList`, and then uses `makePlay` to add the digits defined in the file.

`play` is a simple adaptation from `SimplePartial`.

```
def play(i: Int, j: Int, d: Int) : Partial = {
  // Make a copy of this in p
  val p = new AdvancedPartial
  for(i1 <- 0 until 9; j1 <- 0 until 9){
    p.contents(i1)(j1) = contents(i1)(j1)
    p.pos(i1)(j1) = pos(i1)(j1)
  }
  // Now play d in (i,j)
  p.makePlay(i,j,d); p
}
```

`nextPos`

To implement `nextPos`, it is useful to have a function that returns a "score", using a heuristic to estimate how good it is to play in. We choose a heuristic that gives a higher score to a square with fewer legal moves.

```
/** Return measure of how good it would be to play in position
  * (i,j), with 0 representing a square that has already been
  * played in. */
private def score(i: Int, j: Int) : Int =
  if(contents(i)(j) != 0) 0 else 10-pos(i)(j).length
```

It is interesting to consider different heuristics. The Artificial Intelligence course considers such questions further.

nextPos

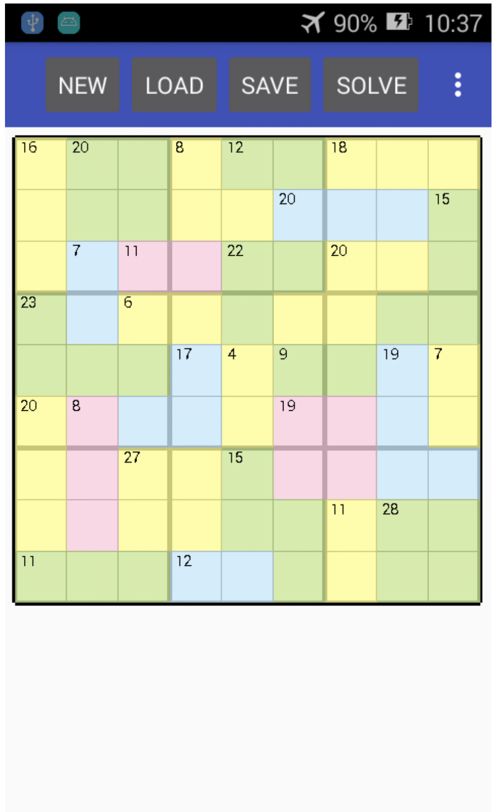
nextPos then finds the square with the highest score.

```
/** Find a blank position to play in; Pre: !complete. */
def nextPos : (Int,Int) = {
  var bestScore = 0; var bestPos = (0,0)
  for(i <- 0 until 9; j <- 0 until 9){
    val thisScore = score(i,j)
    if(thisScore > bestScore){
      bestPos = (i,j); bestScore = thisScore
    }
  }
  assert(bestScore > 0)
  bestPos
}
```

Results

The improved version is between about 15 and 40 times faster. The number of states explored is reduced by a similar ratio.

The depth-first search method gives us plenty of opportunities for extensions and variations. One extension is to solve Killer Sudoku.



Where we are

Part one: Programming with state. Loop-based programs

- How to program in an imperative style;
- how to reason mathematically about programs that use loops;
- how to implement some important algorithms imperatively.

Part two. Data structures and encapsulation. Specifying, programming and correctness with abstract datatypes.

- How to specify abstract datatypes;
- how to implement some important data structures;
- how to formalise relationship between abstract datatype and implementation.

Part three. Programming in the large. Object-oriented techniques and design patterns.

Reminders

Abstract:

- Abstract datatype (ADT) gives the generic description;
- Corresponds well with Scala `trait`;
- State, init, preconditions and postconditions.

Concrete:

- Concrete datatype gives an implementation;
- Corresponds well with Scala `class`;
- Obeys preconditions and postconditions;
- Datatype invariant (DTI): how variables maintain consistency.

Connection:

- Abstraction function shows the correspondence. A function from the concrete implementation to the abstract state: $a = abs(c)$.

Abstract datatypes and concrete data-structures

ADTs

- Set
- Bag
- Map (`PhoneBook`)
- Stack
- Queue
- Dequeue
- Priority queue

Concrete data structures

- Array (ordered/unordered)
- Pair of arrays
- Array of pairs/objects
- Circular array
- Linked list (ordered/unordered)
- Binary search tree
- Hash table

Part two motto

How about

> *"Never break the wall of abstraction."*

?

Summary

Solving Sudoku puzzles.

- An initial abstract search algorithm…
- … giving us the requirements for the representation of partial solutions…
- … allowing us to implement the algorithm and argue for its correctness.
- A simple implementation of a partial solution…
- … and then a more sophisticated one, giving faster results.