Backtracking Lab
Statistics 650/750
Week 11 Thursday

Alex Reinhart and Christopher Genovese

9 Nov 2017

Announcements

- Challenge 1 revisions are due Tuesday, November 14
- Remember you want 12 Mastered assignments (not including projects) for an A in the course, along with both projects
- Challenge 2 deadlines:
  - First draft and peer review Tuesday, November 28
  - Submissions due Tuesday, December 5
  - Final Challenge 2 revisions due **Thursday, December 14**
- Also note that the last homework submissions and revisions will be accepted on **Tuesday, December 12**

Challenge 2

Challenge 2 starts today.

There are five challenges to choose from, three of which are new:

crime-data Write code that uses SQL to load data and automatically generate weekly reports.

shazam Identify songs in your library from brief snippets of audio, using hashing, SQL, and signal processing techniques.

r-tree Create and use a specialized data structure for spatial data analysis.

You can also choose to do either the `word-clouds` or `anomaly-speed` challenge from Challenge 1, provided you didn't already do it. No, you can't do the same challenge twice.

Remember to start early, use good naming and style, and write plenty of tests *from the beginning!*

Backtracking

Backtracking is a general strategy for solving *constraint satisfaction* problems: we have a bunch of constraints on possible solutions, and we must try possible solutions until we find one that satisfies all the constraints.

There are many problems that fit this mold. Some examples:

Room assignments We have k classes that have to be fit into n rooms. Each class can only fit into rooms large enough for it. There are a limited number of time slots available for each room. We must assign classes to rooms so all classes get a time slot and fit in their room.

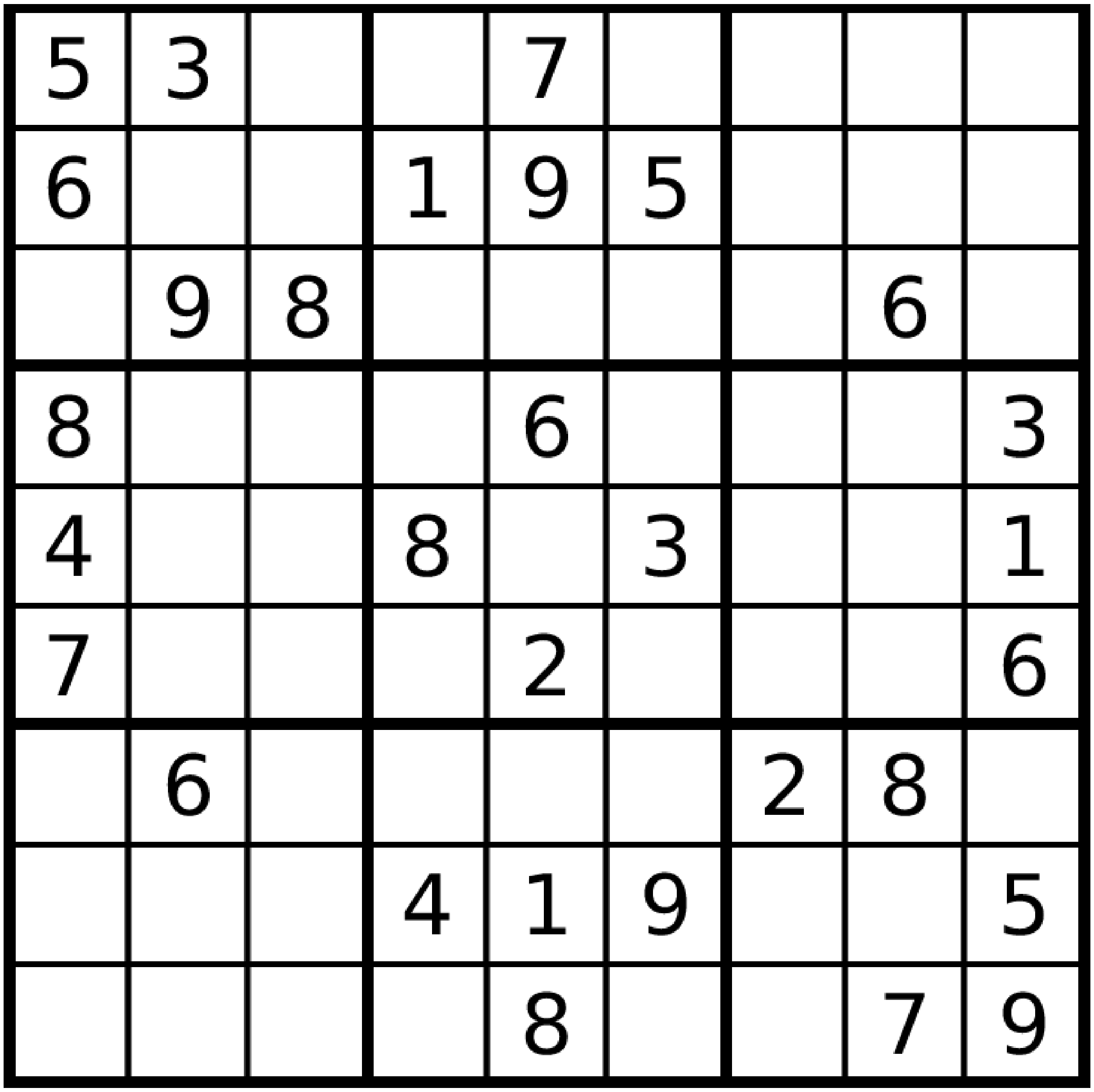
Graph coloring Color each node in a graph with one of k colors, such that no node is connected to another node of the same color. (Includes map coloring.)

Logic programming Specify a problem as a set of logical expressions or rules and find values of variables which make the expressions true.

One prominent example is also a popular puzzle.

Sudoku

One canonical example where backtracking algorithms shine is Sudoku, which you may already be familiar with:

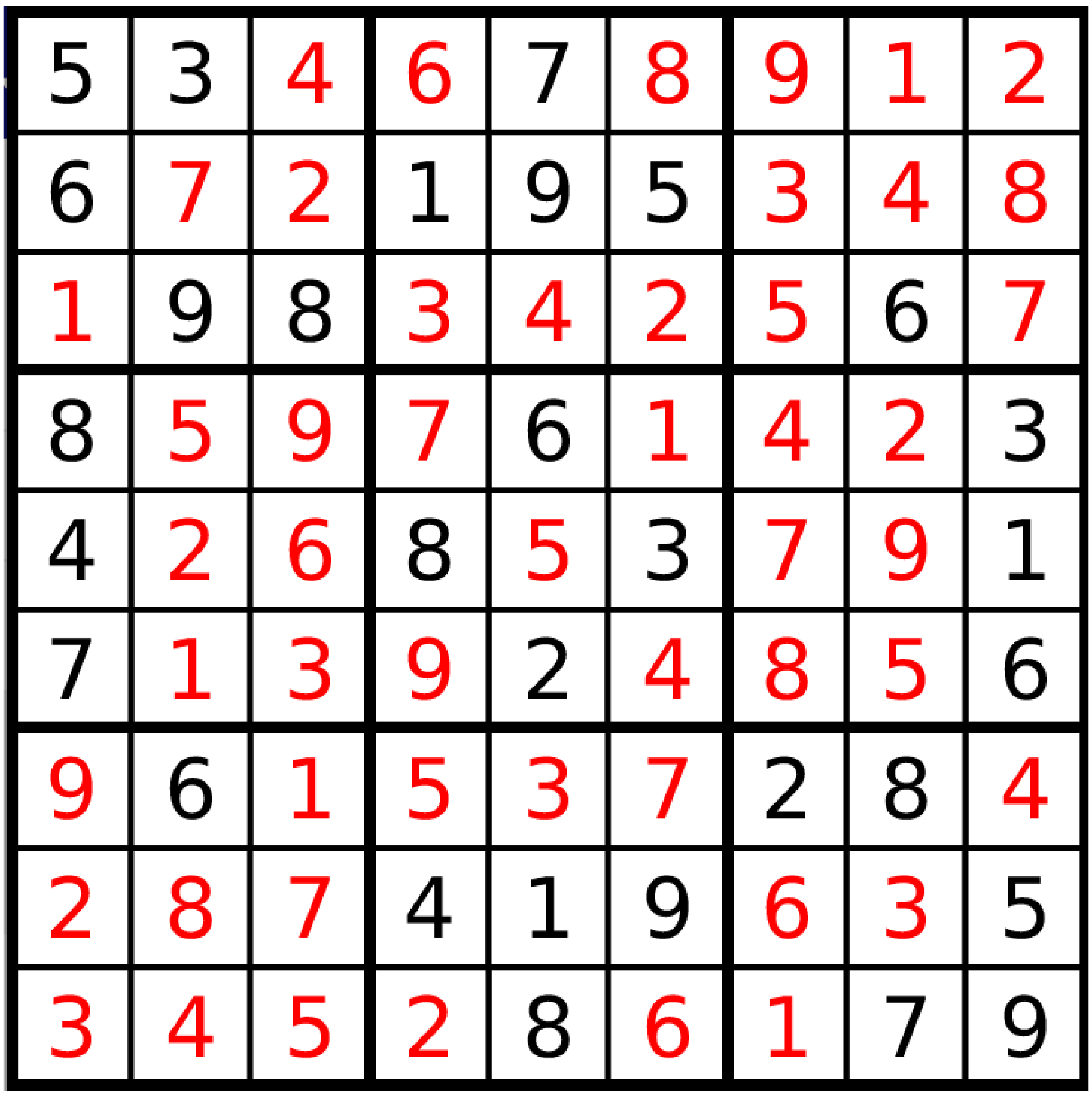


5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

(example from `https://commons.wikimedia.org/wiki/File:Sudoku-by-L2G-20050714.svg`)

A Sudoku grid is a 9 x 9 grid, divided into separate 3 x 3 blocks. It starts with numbers in some of the cells, as in the example above. Your goal is to fill in the empty cells with digits from 1 to 9, so that the same digit does not appear twice in any row, column, or 3 x 3 block. You can't change numbers already provided in the grid above.

For example, the above grid can be solved as follows:



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

Sudoku is a popular puzzle to solve by hand. We can, however, take all the fun out of it by writing a program to solve it.

One possible method is to simply try every combination of numbers and see which combinations meet the rules. However, this grid has 51 open cells, meaning the number of possible ways we can fill those cells is $9^{51} = 4.64 \text{ x } 10^{48}$. That would be stunningly inefficient. Search time would be exponential in the number of cells.

Of course, nearly all of those combinations are invalid: multiple cells break the rules. Suppose, instead of enumerating *complete* grids, we start with *partial* solutions – say, just filling in one cell – and gradually fill in more and more cells, until we either (a) solve the puzzle or (b) realize we are stuck. If we get stuck, we don't bother filling in the rest of the grid and trying all the possibilities for it – we *backtrack*, erasing the previous thing we did and trying a different option.

[Tree diagram]

The key idea is that *backtracking prunes the tree*: when we backtrack, we throw away large numbers of candidate solutions, meaning we never have to look at any of them. This dramatically cuts down the search time.

Implementation

We need several pieces to implement this.

1. A data structure to represent the Sudoku grid as it is currently filled in – the current partial solution.
2. A way to determine which moves are possible, e.g. which cells we can fill and with which numbers.
3. A function to choose which cell we should fill next.
4. A function to choose which digit, of possibly several possibilities, we should choose to fill a cell.
5. The core backtracking algorithm.

Our choices for 3 and 4 matter quite a lot: if we choose correctly, we can rapidly reduce the search space by eliminating impossible grids; if we choose unwisely, we may spend a lot of time exploring a portion of the space that has no possible solutions. We will need to design our algorithm so we can easily try different choices, to determine which performs the best.

For simplicity, we'll combine 1 and 2 into one: our representation for the grid will be a matrix, where every grid cell contains a vector of all currently possible values for that cell. (In Python, we'll use a dictionary of sets.)

This leads to a few core functions:

- `assign` assigns a value to a single cell, eliminating that possible value from all peer cells
- `eliminate` eliminates a possible value from a cell. For consistency, if it results in a cell having only one possible value, that value is eliminated from all peer cells. This is *constraint propagation*.

Finally, we have `solve_sudoku`:

```r
solve_sudoku <- function(grid, next_cell, order_choices) {
    if (identical(grid, FALSE)) {
        ## Must have died earlier
        return(FALSE)
    }

    if (solved(grid)) {
        ## We have solved the grid.
        return(grid)
    }

    #### INSERT CODE HERE ####
}
```

Requirements

Using the R or Python code in `documents/ClassFiles/weekB/`:

1. Finish `solve_sudoku`. Do not look at the code in `sudoku-grid-util.R` (or `.py`).
2. If you get `solve_sudoku` working, uncomment `assign` and `eliminate` below and try to fill in their implementations.
3. Provide several different functions which can be used for `next_cell`:
   - `most_constrained`, which picks the cell with the fewest possible digits
   - `least_constrained`, which does the opposite
4. Provide several different functions for `order_choices`:
   - `ascending_order`, which picks the smallest digits first
   - `descending_order`, which picks the largest digits first
   - `random_order`, which picks them in random order
5. Using the `count_calls` function provided, test which of these methods solves the provided grids most efficiently.

Example Sudoku grids, and their solutions, can be found in the `documents/ClassFiles/weekB/grids/` directory. You can use the `read_grid` function, with a filename, to get a grid object. Your code should work like this:

```r
grid <- read_grid("grids/grid1.txt")

solved.grid <- solve_sudoku(grid, most_constrained, smallest_first)

print(solved.grid)
```

or like this:

```python
grid = read_grid("grids/grid1.txt")

solved_grid = solve_sudoku(grid, most_constrained, smallest_first)

solved_grid
```

These notes are posted in `documents/Lectures/` for your reference.