CS21120 Assignment, 2020-2021

(Corrected version)

James Finnis (jcf12@aber.ac.uk)

Release date: 26th October 2020

Hand in deadline: 20th November 2020, 13:00 via Blackboard

Feedback date: 11th December 2020

Introduction

This assignment will test your ability to implement a depth-first recursive algorithm to solve a puzzle and think about aspects of its performance in terms of Big-O complexity. It will also test your ability to write this code to conform to existing interfaces – much programming in the real world requires this. You will need to show that you can test your code: while some basic tests are provided, you should write more. You should also be able to say why you wrote the tests as you did.

Finally, you will need to write a report describing how you went about implementing each part of the assignment, including why you wrote the algorithms the way you did and what the complexities of the different algorithms are. Each task has a description of what you should write in the report.

Problem Description

For this assignment, you will be writing a program to solve Sudoku puzzles. These puzzles consist of a 9x9 grid of numbers divided into 3x3 sub-grids, with only some of the numbers provided. Here's an example:

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

Solving the puzzle involves working out how to complete the grid, which must obey the following rules:

- Each row contains each digit once only
- Each column contains each digit once only
- Each 3x3 sub-grid contains each digit once only

A valid solution for the puzzle above is:

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

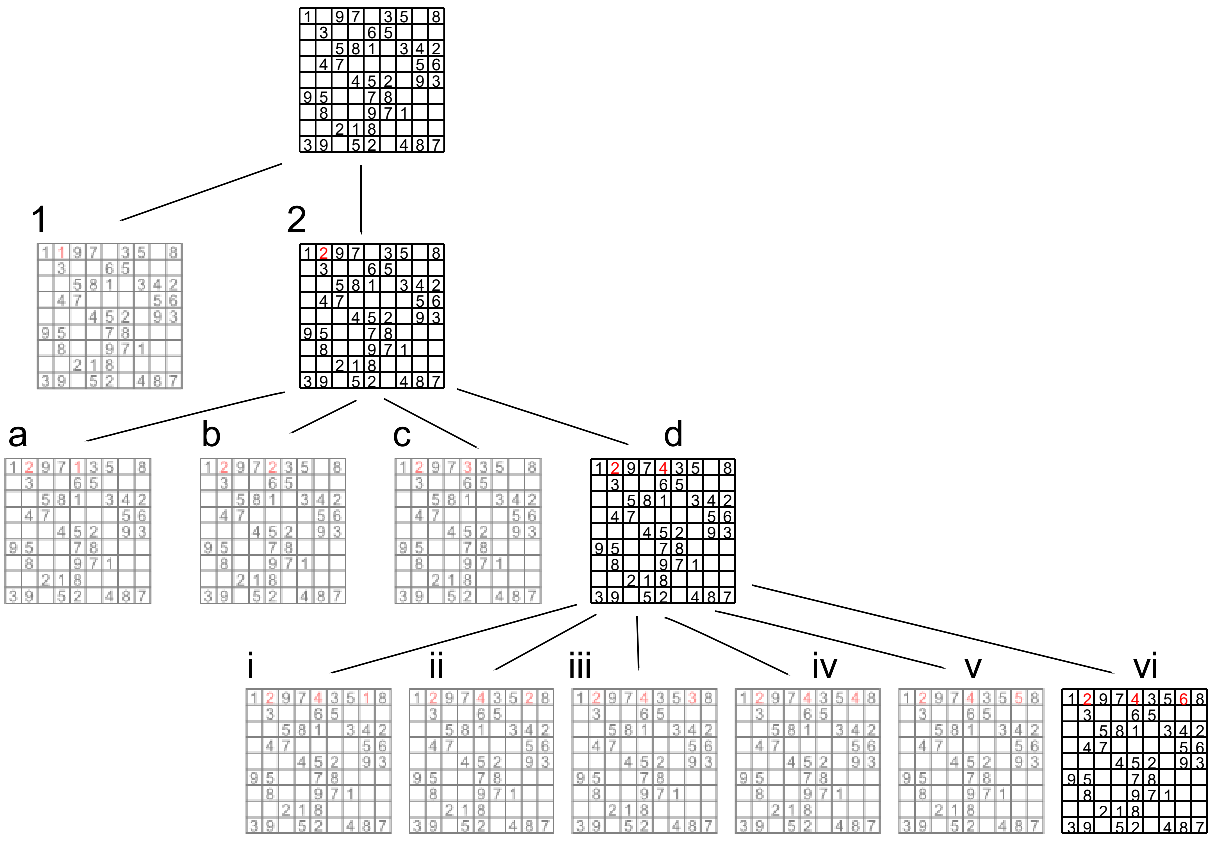
There are many Sudoku (or Su Doku) generators online, for example https://www.sudokuweb.org/ - if you're unfamiliar with the puzzle I recommend trying a few by hand to see the challenge involved.

Solving the Puzzle

For the assignment, you will be writing an in-order recursive backtracking depth-first search for solutions:

- **In-order**: the algorithm will try to fill each cell in turn, working from top-left to bottom-right, and trying each digit from 1 up to 9.
- **Recursive:** the algorithm works by finding a single valid digit to place, and then calling the algorithm on the resulting grid (which is a simpler version of the puzzle because it has an extra cell filled in).
- **Backtracking:** if the algorithm can't find a valid digit to place in the next cell, the digit it placed previously must be incorrect, so we need to go back a step (up a level in the recursion) and try the next digit in the previous cell. If that fails, we go up another level still. If this fails up to the top level, there is no solution.
- **Depth-first:** If you consider the possible ways to fill in the grid as a tree (see the next page), the tree will be explored in a depth-first manner, moving along each branch as far as possible before finding a solution or backtracking.

Look at the diagram below, which shows part of the tree of possible solutions. Grids which are invalid are greyed out.



Our algorithm should do the following:

1. Try to put a 1 in the first available place. That's invalid, there's a 1 in the same row.
2. Try to put a 2 in the same place. That's fine, try to solve the resulting puzzle by recursing:
    a. Try to put a 1 in the first available place. That's invalid, there's a 1 in the same column.
    b. Try a 2 in the same place. Invalid, there's already a 2 in the same column.
    c. Try a 3 in the same place. Invalid, there's a 3 in the same row.
    d. Try a 4 in the first available place. That's valid, so try to solve that puzzle.
        i. Try a 1 in the first available place. Invalid, there's a 1 in the same row.
        ii. Try a 2 in the same place. Invalid, there's a 2 in the same row.
        iii. Try a 3 in the same place. Invalid, there's a 3 in the same sub-grid.
        iv. Try a 4 in the same place. Invalid, there's a 4 in the same column.
        v. Try a 5 in the same place. Invalid, there's a 5 in the same row.
        vi. Try a 6 in the same place. That's valid, so try to solve that puzzle. Eventually this path leads to a solution – we never get as far as step 3.

The small portion of the tree shown above doesn't show most of the possible digits and doesn't show any backtracking. This happens when all the possible digits are invalid in a position, meaning that puzzle can't be solved. To fix this, we go up to the previous level, remove the digit created there, and try a different one.

This video shows the process in full detail – you can see that backtracking doesn't start to happen until the fourth row: https://www.youtube.com/watch?v=e9y7nHuHrEs

The Algorithm

The algorithm itself is as follows, in pseudocode. Make sure you understand the logic behind it and follow the indenting carefully. **(This is the corrected algorithm.)**

Function **solve**, which returns a boolean (true if it solved the puzzled, false if it failed):

```
• For each grid cell:
    o If the cell is empty:
        ▪ For each digit 1-9:
            • Insert the digit into the cell
            • If the resulting grid is valid:
                o Result=solve() (i.e. try to recursively solve the new grid with a digit filled in)
                o If result is true, return true (we solved it!)
        ▪ set  the cell to empty
        ▪ return false – we tried all the digits, none led to a solution. It's a dead end.
• No empty cell was found, return true – we solved the grid!
```

Provided Code

You are provided with a Zip archive containing two packages:

- **uk.ac.aber.cs21120.interfaces** contains the interfaces you must implement in your solution. If you do not implement these interfaces, or write code which does not conform to them, your solution will fail my marking tests.
- **uk.ac.aber.cs21120.tests** contains classes which test your program. The *GridTests* class can be included once you have written *Grid,* and the *SolverTest* class can be included once you have written *Solver.* It also contains *Examples:* a utility class with 400 example puzzles to solve.

These are in two folders inside the archive called **interfaces** and **tests.** When prompted by the assignment, copy the files into packages of the correct names inside your project.

Tasks

Setting Up

Before starting work, set up a new empty project and create three packages:

- **uk.ac.aber.cs21120.solution**
- **uk.ac.aber.cs21120.interfaces**
- **uk.ac.aber.cs21120.tests**

Task 1: The Grid Class

Your first task is to create a class for the grid. This encapsulates the underlying 9x9 grid, and also handles checking the grid for validity. This must be created in the **uk.ac.aber.cs21120.solution** package, and should implement the **IGrid** interface provided in the **uk.ac.aber.cs21120.interfaces** package.

Before you start work, copy the files from the provided **uk.ac.aber.cs21120.interfaces** package into the same package in your program. Leave the remaining packages empty for now.

Now create a **Grid** class inside your solutions package which implements *IGrid* with the following methods:

- **public int get(int x, int y) throws IGrid.BadCellException**
    - this returns the value of the cell at x,y in the grid as an integer from 1-9, or 0 if the cell is empty. If x or y are out of range, **IGrid.BadCellException** (an exception nested inside *IGrid*) should be thrown.
- **public void set(int x, int y, int digit) throws IGrid.BadCellException, IGrid.BadDigitException**
    - this sets the cell at x,y in the grid to an integer from 1-9, or 0 for empty. If x or y are out of range, **IGrid.BadCellException** (an exception nested inside *IGrid*) should be thrown. If the value itself is not in the range 0-9, **IGrid.BadDigitException** should be thrown.
- **public boolean isValid()**
    - This returns true if the grid is valid. In a valid grid, each (non-zero) digit must only appear once in each row, each column and each 3x3 sub-grid (see the rules in the problem description above). Note that zero cells are empty and do not count.

Your grid class must also have a constructor

- **public Grid()**

This is not specified in the interface (because Java does not permit that), but my tests will not run without it. Also, note that the two exceptions I have defined are subclasses of *RuntimeException* – this means that you don't have to provide try/catch blocks every time you use *get()* and *set()*. The most difficult part of this task is the *isValid()* method for checking the grid according to the rules.

Testing

I have added a fairly comprehensive set of tests to the **uk.ac.aber.cs21120.tests** package to help you debug. Copy only the **GridTests** class into your *tests* package as you work on this task – if you copy other files your solution will not compile at this stage (as these require a solver).

In Your Report

Your report should describe how your *isValid()* method works in some detail. You should also talk about how easy or difficult you found the task, and how you went about solving it.

Task 2: The Solver Class

This task involves writing the actual Sudoku solver class, **Solver.** This class must be in the **uk.ac.aber.cs.21120.solution** package, and must implement the **ISolver** interface. The interface describes the following single method:

- **public boolean solve()** will attempt to solve the grid passed into the constructor. It will modify this grid as it does so, and will call itself recursively to solve simpler versions of the grid. If it succeeds, it will return true and the grid will be complete. If it fails, this grid cannot be solved and the solver must backtrack.

See the algorithm described above for more details. Your class must also provide a constructor of the form

- **public Solver(IGrid g)**

which again is not specified in the interface itself but is required for testing. This should simply store the grid in a member variable, which *solve()* can then modify as it runs.

To help with debugging, you may find it useful to write a method which outputs the grid as a string. Here is a simple example you can use:

```
public String toString(){
    StringBuilder b = new StringBuilder();
    for(int y=0;y<9;y++){
        for(int x=0;x<9;x++){
            b.append(get(x,y));
        }
        b.append('\n');
    }
    return b.toString();
}
```

I would also suggest adding some *System.out.println()* messages to show the progress of the algorithm. It might also be helpful to add an *int depth* member variable, which is incremented whenever *solve()* is entered and decremented when it returns. You can then print out the depth of the recursion with your other messages. Finally, note that the algorithm usually finishes very quickly, but if a lot of backtracking is required it can take a little while to solve a puzzle.

Testing

Inside the **uk.ac.aber.cs21120.tests** package you will find a class called **SolverTests** which you should copy into your own *tests* package and run. This tests progressively more difficult puzzles, starting from those with just one or two digits to fill in, then some with three or four gaps which require backtracking, before testing puzzles with 20 and 40 gaps. I will be using these tests to mark your assignment.

You will find also find an **Examples** class which you can copy into your *tests* package. This class contains 400 example puzzles, with a varying number of empty cells: low numbered examples have fewer empty cells, which vary from 30 for example 0 up to 69 for example 399. This class requires that you have a working *Grid* in your solution.

Examples has the following public static methods:

- **public static int getGapCount(int n)**: return the number of empty cells in puzzle *n*
- **public static IGrid getExample(int n)**: return an *IGrid* containing puzzle *n* ready to be solved. Note that the object itself will be an instance of your own *Grid* class.

These are static methods which don't require an *Examples* object, so to call them use (for example)

Examples.getExample(100)

If you write your own test make sure to add them to your copy of the *tests* package and submit them as part of the assignment.

In Your Report

You should describe the problems you had, how you tested the code, and give a worst case Big-O complexity in terms of *n* where *n* is the number of empty spaces (hint: it's the same as for a brute-force search – just consider how many branches the algorithm must explore).

For flair marks, can you think of any ways to make the algorithm more efficient? Hint: the way *isValid()* is used isn't very efficient, and there is a better way to do it. There are also better ways to pick the next cell to try to put a digit into, rather than just the next empty one. You don't need to write any code for this part, just write down your ideas and the thinking behind them.

You are encouraged to research algorithms on the Internet or elsewhere for this task – but remember to include your references when you discuss your ideas. Not doing so counts as unfair academic practice!

Task 3: Solving Puzzles

Add a **Main** class to your *solution* package, with a *main* method. This should try to solve all 400 examples in the *Examples* class (in the *tests* package). For each puzzle, record the number of gaps and how long each puzzle took to solve. This may take a few minutes to run, and possibly several tens of minutes.

You can time the puzzles by making use of the **System.currentTimeMillis()** method, which returns the current time in milliseconds as a long:

```java
long start = System.currentTimeMillis();
// do something in here
long timeTaken = System.currentTimeMillis()-start;
```

The easiest way to record the times is simply to print *puzzlenumber,gaps,time* using *System.out.println()* or *System.out.format().* Then cut and paste into Excel or a text file. Please don't include the table in your report, it will be quite big!

Can you see a pattern in the results? Does it confirm your Big-O from task 2? For flair marks, include a plot of the results (time against number of gaps) – this may show the pattern more clearly, particularly if you plot the log of the time rather than just the time.

Task 4: Generating Puzzles

How could you use your Sudoku solver, combined with a random number generator, to write a program for generating Sudoku puzzles? Write a few lines of pseudocode to describe your ideas – or for flair marks actually write a class to do it and include it in your assignment. Again, you are free to research puzzle generators on the Internet, but remember to cite your references.

Writing the report

Your report should be around 1000-1500 words, containing the following sections:

- Task 1: The Grid Class
- Task 2: The Solver Class
- Task 3: Solving Puzzles
- Task 4: Generating Puzzles
- Self-evaluation: a paragraph describing which parts you found difficult and why, which parts you think you did well, and what mark you expect to get

Before writing the first two sections remember to read the "In Your Report" part of each task. Sections 1-4 should contain descriptions of how you went about the task and descriptions of any complex algorithms not described in this assignment brief.

The report must be provided as a PDF document.

Submission format

The submission must be a ZIP file called **cs21120_<userid>.zip** with <userid> replaced with your Aberystwyth user ID (for example cs21120_jcf12.zip). It must contain:

- A directory called **solution** containing the files from your **solution** package (*Grid*, *Solver* and *Main*).
- A directory called **test** containing any additional test classes you wrote
- A PDF file containing your report.

There will be an element of automated marking – I will copy your files into my own project and run them against my own tests. It is therefore important that you follow the submission requirements precisely, particularly the correct naming of classes and packages.

Marking

Marks are based on the functionality of your code and the contents of your report, as follows:

- Task 1: The Grid Class – 30%
    - 15% for the functionality of the code (including an element of automated testing)
    - 15% for the contents of the corresponding section of the report.
- Task 2: The Solver Class – 30%
    - 15% for the functionality of the code (including automated testing)
    - 15% for the contents of the corresponding section of the report.
- Task 3: Solving Puzzles – 20%
    - 10% for the functionality of the code
    - 10% for the report, describing how the code works and the findings of the experiment
- Task 4: Generating Puzzles – 10% for a good description of the algorithm
- Self-Evaluation and Formal Aspects – 10%. This includes the submission format (PDF, ZIP) as well as readability and format of your report, referencing and the quality of your code including comments. All figures, tables and diagrams should include captions, and be numbered and cited where appropriate in the main text.
- Up to 10% extra marks will be added (although the assignment will be capped at 100%) for flair, including:
    - Puzzle solver code
    - Plotting the gaps/time curve
    - Improvements to the solver algorithm

Academic conduct

As with all such assignments, the work must be your own. Do not share code with your colleagues and ensure that your own code is securely stored and not accessible by your classmates. Any sources you use should be properly credited with a citation, and any help you get should be acknowledged.

Your report must accurately reflect what you have achieved with your code, any discrepancies between your code and report could be treated as academic fraud. Finding undue similarities between your work and others' could be treated as unacceptable academic practice. Your attention is drawn to the University regulation on unacceptable academic practice at

https://www.aber.ac.uk/en/academic-registry/handbook/regulations/uap/

Support

If you have any problems with understanding or completing the assignment please ask for help, either by email (jcf12@aber.ac.uk) or via the Blackboard forum for the programming assignment. If there are any circumstances that affect your work, please visit

https://impacs-inter.dcs.aber.ac.uk/en/cs-undergraduate/resources/special-circumstances

to see what to do. You should also contact your year tutor (Laurence Tyler, lgt@aber.ac.uk, for second year Computer Science students.)