Homework1 - Federico Ponzi - 1532793

A journey to parallelize a Sudoku solver

Starting to solve the homework, I've started trying the most naive & simple solution:

```
     solutions = 0 #the number of solutions
     table = the table of the sudoku (in an appropriate
datastructure)
     while there are empty cells:
          e = getEmptyCell(), #which, in the ds, it's denoted
as -1
          cs = findCandidates(e) #suitable numbers for this
cell
```

```
        for every candidate c in cs:
            table2 = Copy of the table
            table2.setValue(e, c)
            solutions += spawnThread(table2)
    return solutions
```

This implemenation got ~10 seconds in the multithreaded way. The problem with this solutions is that the findCandidates function, needs to:

- scan column = 9
- scan row = 9
- scan square = 9

Needs to iterate every time to update the number of candidates. Also, the getEmptyCells, needs to scan the whole 9*9 table to find empty cells.

We can think sudoku's empty cells in a different way: using a **graph**.

For example:

```
   012 345 678
0  574 382 169
1  683 194 752
2  921 675 843

3  765 243 018
4  438 917 526
5  192 856 437

6  856 731 294
7  34. 529 6.1
8  21. 468 3.5
```

(first column and first row are indexes. Dots are empty cells.)

We do a scan of this table: search for empty cells, and do a list of possible candidates for that cell.

Also, we save the relation between empty cells. So, we have:

```
For the node [7,2]
1 candidate: [7,2] = {7}
2 relations: [7,2] -> [7,7], [8,2]
Where: -> is links-to relation
```

This basically means that, when I set the cell `[7,2]= 7` I should remove `7` as candidate from the linked cells.

The other 3 cells are:

```
[7,7] = {7,8}
[7,7] -> [7,2], [8,7]

[8,2] = {7, 9}
[8,2] -> [7,2], [8,7]

[8,7] = {7, 8}
[8,7] -> [7,7] [8,2]
```

We can think the overall structure, as a graph where two neighbor nodes are linked together if and only if they fall in the same row, in the same column, or in the same square. The graph is represented with an adjacency list (for space efficiency reasons).

So this is how the new algorithm looks like:

```
solutions = 0
ecg = Empty Cells Graph. Every cell has a list of
candidates and an hashset of linked nodes.
```

```
    while ecg has nodes:
        e = getRandomNode(ecg) //fast
        cs = getCandidates(ecg) //fast
        for every candidate c in cs:
            if using c doesn't lead to a good
configuration:
                continue
            table2 = Copy of the table
            table2.setValue(e, c)
            solutions += spawnThread(table2)
    return solutions
```

Optimization 1: pruning by wisely choose next random node

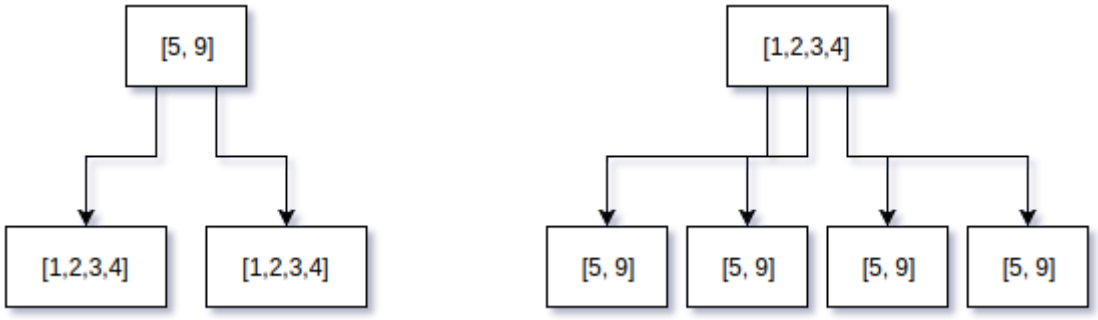
The DAG looks very different, based on the call on getRandomNode. If we have this graph (with only 1 solution):

```
    [0,1] = {1}
    [0,1] -> [0,2]

    [0,2] = {1,2}
    [0,2] -> [0,1]
```

If we choose [0,2] before, we would need 2 threads (one per candidate), to find out 1 solution for the sudoku.

If we choose [0,1] before, we would update [0,2], and than spawn just another thread to find out 1 solution for the sudoku (or zero with halve threads optimization).



This is a very basic example, because we can check if we end up in a good configuration before spawning the thread. But for bigger graph, this would lead to an efficient execution.

To overcome this, we can better choose the next node by selecting the node with **less candidates**. This cost a little more, but in the end it's very much more convenient.

Optimization 2: pruning before spawing the thread

Adding a check before spawning the thread, can save us from the hassle of spawning useless threads.

In `game3.txt`, there are **6846** less spawned threads.

Optimization 3: Halve threads optimization

We can use the halve threads optimization, to reduce the number of spawned threads.

In game3.txt, there are **702951** (with halveThreads flag setted) vs **1285081** spawned threads.

Optimization 4: Using the right ds

We have a graph, on which every node has:

- A name: his coordinates in the table.
- A list of coordinates: His neighbours
- A list of candidates: possible values for this element.

The operations needed for this DS are:

- Get the list of neighbours and check if a value is in this list
- Get the list of candidates and check if a value is in this list

For this reason, I've choosen to implement this DS using two maps

```
HashMap<Coordinates, HashSet<Coordinates>> graph;
HashMap<Coordinates, HashSet<Integer>> candidates;
```

After implementing the equals and hashcode method of the entity `Coordinates`, we can have:

- `graph.get()`: O(1)
- `graph.remove()`: O(1)
- `candidates.get()` : O(1)
- `candidates.remove()`: O(1)

The Hashset looks perfect for this task, because the remove operation's complexity it's just O(1) vs ArrayLists's O(n) [1]. But there is a drawback. Let's do a simple benchmark with ArrayList creation vs HashSet (which internally uses hashmaps) instantiation with numbers from 0-9 in 1 second. [0]:

- 15872981 ArrayLists in 1 second
- 7286756 HashSets in 1 second

After the change from HashSets to ArrayLists (using the file `test1/test1_d.txt`):

- Using HashSets: Done in: 39187ms
- Using ArrayLists: Done in: 28244ms

This initialization it's done while making a copy of the graph for the new threads.

Benchmarks results

In brief

Filename	Parallel	Sequential	Speedup
test0/game0.txt	1ms	1ms	0.33
test0/game1.txt	11ms	11ms	1.0
test0/game2.txt	14ms	16ms	1.1
test0/game3.txt	2473ms	3811ms	1.5
test1/test1_a.txt	23ms	34ms	1.4

test1/test1_b.txt	1417ms	1687ms	1.19
test1/test1_c.txt	9918ms	18671ms	1.88
test1/test1_d.txt	26285ms	48204ms	1.83
test1/test1_e.txt	115558ms	233368ms	2.01
test2/test2_a.txt	44ms	63ms	1.43
test2/test2_b.txt	442ms	545ms	1.23
test2/test2_c.txt	4465ms	7672ms	1.71
test2/test2_d.txt	34091ms	66910ms	1.96
test2/test2_e.txt	230341ms	451989ms	1.96

Complete

Filename	Parallel	Sequential	Speedup	Empty Cells	Fill Factor	Solutions	Workers	Search space
test0/game0.txt	1ms	1ms	1.0	4	4%	1	1	4 (len: 1)
test0/game1.txt	11ms	11ms	1.0	45	55%	1	1	5441955840000000000 (len: 18)
test0/game2.txt	10ms	16ms	1.6	49	60%	1	1	6823269127183128330240 (len: 22)
test0/game3.txt	2473ms	3811ms	1.5	59	72%	50142	63834	2304433152000000000000000000000000000000 (len: 40)

test1/test1_a.txt	23ms	34ms	1.4	53	65%	1	170	4312979991503409512448000 (len:26)
test1/test1_b.txt	1417ms	1687ms	1.19	59	72%	4715	155810	19477518632563507200000000000000000 (len:37)
test1/test1_c.txt	9918ms	18671ms	1.88	61	75%	132271	291117	13980445502865408000000000000000000000 (len:41)
test1/test1_d.txt	26285ms	48204ms	1.83	62	76%	587264	10121979	47784725839872000000000000000000000000 (len:42)
test1/test1_e.txt	115558ms	233368ms	2.01	63	77%	3151964	52939640	234091637722432143360000000000000000000 (len:44)
test2/test2_a.txt	44ms	63ms	1.43	58	71%	1	313	24563768857859261988864000000000 (len:32)
test2/test2_b.txt	442ms	545ms	1.2	60	74%	276	16227	26171801548441 4301673

			3					8816000000000 (len:36)
test2/test2_c.txt	4465ms	7672ms	1.71	62	76%	32128	974012	5546527766851092480000000000000000000 (len:40)
test2/test2_d.txt	34091ms	66910ms	1.96	64	79%	1014785	15340410	5436619103789835275678515200000000000000000 (len:44)
test2/test2_e.txt	230341ms	451989ms	1.96	65	80%	7388360	110076409	4281337544234495279596830720000000000000000 (len:46)

As we can see, the speedup it's always greater then 1. This is because we're doing multiple DFS (Depth-first search). In the case we have one straight line to the solution, using the HalveThreads optimization we can end up with the same run time.

As we can see in the table, there is no strict correlation between the fill factor and the execution time, while it's correlated with the space factor. This is more evident with these two test cases which have the same fill factor, but very different execution time and solution space:

test0/game3.txt	2473ms	3811ms	1.5	59	72%	50142	63834	2304433152000000000000000000000000000 (len: 40)

test1/test1_b.txt	1417ms	1687ms	1.19	59	72%	4715	155810	19477518632563507200000000000000000 (len:37)

As the benchmarks also suggests, with a big search space we end up with an highly connected graph. The more the graph is connected, the bigger the search space and the execution time.

About the testing environment

Intel(R) Core(TM) i7-4500U CPU @ 1.80GHz (4 processors)

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                4
On-line CPU(s) list:   0-3
Thread(s) per core:    2
Core(s) per socket:    2
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 69
Stepping:              1
CPU MHz:               2863.781
BogoMIPS:              4788.89
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              4096K
NUMA node0 CPU(s):     0-3
```

More info: http://cpuboss.com/cpu/Intel-Core-i7-4500U

Complete code

The complete code it's available on Github: https://github.com/FedericoPonzi/SudokuSolver.

To compile the code:

```
javac ponzi/federico/homeworkone/Main.java
```

To run the code:

```
java ponzi.federico.homeworkone.Main
```

References

[0] Benchmark arraylist vs hashsets:

```
import java.util.*;
public class ProvaArr{
    public static int n = 0;
    public static void main(String[] args)
    {
        new Thread()
        {
                public void run()
                {
                    try
                    {
                            Thread.sleep(1000);
                    }
                    catch (InterruptedException e)
                    {
                            e.printStackTrace();
                    }
                    System.out.println(n);
                    System.exit(0);
```

```
                    }
                }.start();
                HashSet<Integer> z;
                while(true){
                    z =  new HashSet<Integer>();
                    for (int i = 0; i < 10; i++){
                        z.add(i);
                    }
                    n++;
                }
            }
        }
```

[1] ArrayList's remove in the OpenJDK (from: http://hg.openjdk.java.net/jdk7/jdk7/jdk/file/00cd9dc3c2b5/src/share/classes/java/util/ArrayList.java)

```
public E remove(int index) {
    rangeCheck(index);
    modCount++;
    E oldValue = elementData(index);
    int numMoved = size - index - 1;
    if (numMoved > 0)
        System.arraycopy(elementData, index+1, elementData, index,
                         numMoved); //*
    elementData[--size] = null; // Let gc do its work
    return oldValue;
}
```

The System.arraycopy complexity it's O(N).