Distributed Sudoku Solver

Author: Riccardo Bernardi

E-mail: 864018@stud.unive.it

Useful Info

The code and the report were entirely developed only by Riccardo Bernardi.

The author developed a library to program in a more functional way in python called pygraham so it is possible to find "list(...).map(...).reduce(...)". This construct is correct and it is introduced thanks to pygraham library, freely available on PyPI. The library overrides the builtin list class. It is present in the bibliography[4] an article that explains it.

In general: some libraries that were used in the assignment are previous projects of the author published on PyPI, in these cases will be put references to the docs.

Table of Contents

1 Background

The sudoku is a challenging game about finding the proper numbers that can stay in a certain cell of a 9x9 square. A number is correct when it respects the constraints of:

- 1 only one number with same value on the same row, column and 3x3 box
- 2 on the same row, column and 3x3 box have to be every number from 1 to 9 in only one copy

Some numbers at the start of the game are already provided on the sudoku and cannot be modified, you can only insert new numbers.

Another important notice about sudokus is that a sudoku that is correct should have exactly one solution, this is very important because asserting this fact will allow us to search for a solution with the certainty of converging.

Constraint Propagation is about taking the rules as written before and building a method that checks that fact, pruning wrong solutions from possible ones. In particular at the very start every 0 cell(void cells to be calculated) are substituted by an array [1...9] and the **propagate_constraints** function will prune wrong ones recursively.

Constraints can be applied directly or indirectly. Directly means that we impose no equal digits on the same row/col/box. Indirect means that we impose every number from 1 to 9 have to be present in every row/col/box. In our case we exploit semantics of the sets in python to enforce both the constraints at the same time.

In general we can implement the constrains propagation using the most-constrained strategy or the least one. It is known that the most-constrained strategy helps pruning the tree of recursion in the backtracking so it is the best choice[5].

Both **most-constrained** and **least-constrained** are implemented because of completeness but it is obvious and it is also proven by the empirical results that **most-constrained** is the best strategy(also used in the real game when done using pencil) because in the recursion you will find with more cells completed. This is because the only case in which you use backtracking is when the constraints become **cyclic** so no solution can be found with only the **propagation**, in this case solving one little constraint means that at least another one or more cells will be easily completed by the propagation in the recursion. Instead if you use the least constrained strategy probably you will complete only one cell with probably the wrong answer and you will need to have many and many recursions before knowing that your initial choice was wrong and so you need to close the stack and retry the same non-efficient strategy.

The easiest puzzles will be solved by just doing the steps before but for the hardest ones you will need to apply **backtracking** that is about searching for all possible solutions in the puzzle until the only one solution is found. If a wrong one is found it is ignored.

2 Web Scraping

This part is not requested but was really important to check the performances of the algorithm and to improve them. I mean that training my algorithm on a thousand of images is very different between training it on a batch of ten. The webscraper works connecting to some predetermined pages that is known contain links to ".txt" files with puzzles or also directly one page that contains many puzzles. Obviously this means that the webscraper is able to accept different formats of data and encode it correctly.

The code below connects to the sites with the most difficult sudokus that i was able to find:

```
def download_sudokus(){...}
```

The function get_txt works as below:

```
def get_txt(url){...}
```

Given an url it tries to parse it in different ways. In the first case if the file is a webpage and not a txt file it will identify all the links to ".txt" files, in this case we assume that every file contains only one sudoku per file. Otherwise if the file is unique but contains many sudokus divided by some characters identified by mean of regular expressions we divide the sudokus into many different files.

Data is not only retrieved on the web but can also be loaded via a simple ".txt" file that contains many files, an example are:

```
def load_qqwing_sudokus(){...}
```

You can load a bunch of ".txt" all at the same time putting them in a precise folder. In this case the sudokus come from the website QQWing that created for me 1000 in a file and other 500 sudokus in another file. the settings that I put are that the puzzles should be divided into difficult and easy in a fair way, this is important because from this fact depends relevance of my next evaluations.

After creating a repository with **1899** sudokus I thought that samples were sufficient.

Next question i answered was how many of them were very similar to others. This seems unimportant but if too many puzzles are too similar to other ones the performances can be affected by the same puzzles presented too many times.

To conduct this test I used a software that I previously developed, it is called "antiplagiarism" and can be found in the bibliography[3] an article that describes its use.

The test said that ~150 sudokus were very similar, so at least 90% of similarity but since it represents only less than 10% of the entire dataset I continued the analysis.

The downloading of the data can be activated or deactivated using: "DOWNLOAD_DATA = False".

All the data above can be activated by putting "WEBSCRAPED" constant equal to True.

After some other searches I've found an immense repository of sudokus provided by the very famous Kaggle website, also this is available to be used in this project, it is activated by putting to True the constant "LOAD_KAGGLE". I noticed that this dataset performs many times better than the webscraped one because all the puzzles reside on a single ".csv" file so the cost of IO is low w.r.t. to the webscraped data that is segmented into many litlle files.

Other function that are used are:

```
def squeze_all(matrix){...}
def show(matrix){...}
def parse_sudoku(f){...}
```

3 Data Model

From the very first time I thought that a proper data model can be very helpful so I modelled a class that was able to mainly give me rows, columns and boxes easily but also that provided the transformes to change values of the sudoku passing a callback/lambda.

It is important to clarify that I approached the problem substituting the 0's in the matrix passed by the user with a list [1...9] for every 0's. So propagating constraints for me is done via the elimination of the wrong choices in the list [1...9].

The fields of the Sudodata are here below, they are only 2, the necessary ones. The. matrix is the raw matrix that was only parsed to be a correct matrix with 9 rows and 9 columns. The parsing is done because the sudokus are received from different sources in many different formats(eg: csv, txt, web, many sudokus in a txt, links to txts, etc). In case you are using the solution is provided and at the end there will be a further check about correctness of my solution, if no assertion is thrown then my solution is congruent to the one provided by Kaggle.

```
def __init__(self, matrix, sol=-1):
  self.data = matrix
  self.sol = sol
```

Some basic methods, the method with no indication returns the value(it is a getter) instead if it is named "assign" then it is a setter.

```
def cell_rc(self, r, c){...}
def assign_cell_rc(self, r, c, v){...}
def column(self, c){...}
def assign_column(self, c, v){...}
def row(self, r){...}
def assign_row(self, r, v){...}
def triplet(self, r, t){...}
```

Some iterators and transformers, the iterators actually are generators so they **yield** values, instead the transformers accepts as parameter only a **lambda** to modify the values. The most difficult transformer to be created was the box_transformer because it was more difficult to apply a function and then to remap all modification on the original matrix so as a workaround it will give the absolute indexes of the box to the lambda that will use them.

```
def cell_iter(self){...}
def row_iter(self){...}
def col_iter(self){...}
def box_iter(self){...}
def cell_transformer(self, l){...}
def row_transformer(self, l){...}
def col_transformer(self, l){...}
def box_transformer(self, l){...}
```

Some useful overrides are the ones here, in particular the iter will iterate by rows, this is useful for routines operations that do not need a strict ordering(e.g.: checking if a sudoku is solved can be more easily done via __iter__ since no particular order is required).

```
def __iter__(self){...}
def __eq__(self, other){...}
def __repr__(self){...}
def __str__(self){...}
```

Some other methods that are needed are here, the only cryptic name is "void_elems" and this checks if there are void lists internally to the sudoku, if instead of a number is present a void list this means that all the possibilities were discarded so the recursion guessed the wrong value and the sudoku can be discarded. Hash do what everyone could expect, it produces a string hash not-unique for the specific matrix. For not-unique I mean that if a value is not found instead of a 0 there is a list [1...9] and it is hashed as the sum of itself.

```
def is_solved(self){...}
def void_elems(self){...}
def duplicates(self){...}
def hash(self){...}
```

The is_solved method is very important because based on the status of the sudoku returns one of the states in this tuple: ("CORRECT","WRONG","CONTINUE"). If the sudoku is completed then the "CORRECT" is returned, if there are duplicates or a cell have no number but also no available solutions(there is an empty list) then the "WRONG" word is returned and this means that you are in the wrong branch of the recursion that have to be dropped. If instead the sudoku is not solved but you are on the good way(no duplicates and no empty lists) then a "CONTINUE" is returned.

To speed-up the computation of the most expensive operations(box operations in general) I precomputed the values and here is the signature of the function:

```
def precompute_box_indexes(){...}
```

4 Constraint Propagation

The constraint strategy that i use is both direct and indirect in the sense that at the same time I check that one number is present in each row, column and box in only one copy. This is not going to apply an higher cost because I use the implicit semantic of the sets in python. The modelling is about creating a set of a row/col/box for the vector [1...9] and one for the reference row/col/box and i check that their intersection length is equal to 9.

All the job of constraints propagation is done through the function below taht takes the Sudodata and returns it modified. All the mechnism works creating proper callbacks that will do the simplifications on the data and then it is called into a proper transformer that will apply such modifications to the existent sudoku.

```
def propagate_constraints(data):
  def box_prop(box_indexes){...}
  data.box_transformer(box_prop)
```

```
    def col_prop(col){...}
    data.col_transformer(col_prop)

    def row_prop(row){...}
    data.row_transformer(row_prop)

    def squeeze(row){...}
    data.row_transformer(squeeze)

    return data
```

One important part is the squeezing operation. At a certain moment in a certain cell will remain only one value and so you want that such value trasforms from a vector of possible values into the actual value.

Example:

1) We start by substituting every 0 with [1...9].

```
[3, 7, [1, 2, 3, 4, 5, 6, 7, 8, 9], 5, [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3,
4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9],
6]
[[1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6,
7, 8, 9], 3, 6, [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], 1,
2]
[[1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6,
7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], 9, 1, 7, 5, [1, 2, 3, 4, 5, 6, 7, 8,
9]]
[[1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6,
7, 8, 9], 1, 5, 4, [1, 2, 3, 4, 5, 6, 7, 8, 9], 7, [1, 2, 3, 4, 5, 6, 7, 8,
9]]
[[1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], 3, [1, 2, 3, 4, 5,
6, 7, 8, 9], 7, [1, 2, 3, 4, 5, 6, 7, 8, 9], 6, [1, 2, 3, 4, 5, 6, 7, 8, 9],
[1, 2, 3, 4, 5, 6, 7, 8, 9]]
[[1, 2, 3, 4, 5, 6, 7, 8, 9], 5, [1, 2, 3, 4, 5, 6, 7, 8, 9], 6, 3, 8, [1, 2,
3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8,
9]]
[[1, 2, 3, 4, 5, 6, 7, 8, 9], 6, 4, 9, 8, [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2,
3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8,
9]]
[5, 9, [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], 2, 6, [1, 2,
3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8,
9]]
[2, [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5,
6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], 5, [1, 2, 3, 4, 5, 6, 7, 8, 9], 6,
4]
```

2) We start the first round of constraint propagation. Every constraint propagation round is about eliminating as many wrong choices as possible by row, col and box. At the end we do a squeezing[12].

```
[3, 7, [8, 1, 2, 9], 5, 4, 2, [8, 9, 4], [8, 9, 4], 6]
[[8, 9, 4], [8, 4], [8, 9, 5], 3, 6, 7, [8, 9, 4], 1, 2]
[[8, 4, 6], [8, 2, 4], [8, 2, 6], [8, 2, 4], 9, 1, 7, 5, [8, 3]]
[[8, 9, 6], [8, 2], [8, 9, 2, 6], 1, 5, 4, [8, 9, 2, 3], 7, [8, 9, 3]]
[[8, 1, 4, 9], [8, 1, 2, 4], 3, 2, 7, [9, 2], 6, [8, 9, 2, 4], [8, 1, 5, 9]]
[[1, 4, 9, 7], 5, [1, 2, 9, 7], 6, 3, 8, [1, 2, 4, 9], [9, 2, 4], [1, 9]]
[[1, 7], 6, 4, 9, 8, [3, 7], [1, 2, 3, 5], [2, 3], [1, 3, 5, 7]]
[5, 9, [8, 1, 7], [4, 7], 2, 6, [8, 1, 3], [8, 3], [8, 1, 3, 7]]
[2, [8, 1, 3], [8, 1, 7], 7, 1, 5, [8, 1, 3, 9], 6, 4]
```

3) we continue propagating

```
[3, 7, [8, 1, 9], 5, 4, 2, [8, 9], [8, 9], 6]
[[8, 9, 4], [8, 4], [8, 9, 5], 3, 6, 7, [8, 9, 4], 1, 2]
[[8, 4, 6], [8, 2, 4], [8, 2, 6], 8, 9, 1, 7, 5, [8, 3]]
[[8, 9, 6], [8, 2], [8, 9, 2, 6], 1, 5, 4, [8, 9, 2, 3], 7, [8, 9, 3]]
[[8, 1, 4, 9], [8, 1, 4], 3, 2, 7, 9, 6, [8, 9, 4], [8, 1, 5, 9]]
[[1, 4, 9, 7], 5, [1, 2, 9, 7], 6, 3, 8, [1, 2, 4, 9], [9, 2, 4], [1, 9]]
[[1, 7], 6, 4, 9, 8, 3, [1, 2, 3, 5], [2, 3], [1, 3, 5, 7]]
[5, 9, [8, 1, 7], 4, 2, 6, [8, 1, 3], [8, 3], [8, 1, 3, 7]]
[2, [8, 3], 8, 7, 1, 5, [8, 9, 3], 6, 4]
```

[...] excluded some computed values

10)

```
[3, 7, 1, 5, 4, 2, 8, 9, 6]
[9, 8, 5, 3, 6, 7, 4, 1, 2]
[6, 4, [2, 6], 8, 9, 1, 7, 5, 3]
[[8, 6], 2, 6, 1, 5, 4, 3, 7, 9]
[4, 1, 3, 2, 7, 9, 6, 8, [1, 5]]
[7, 5, 9, 6, 3, 8, [1, 2], 4, 1]
[1, 6, 4, 9, 8, 3, 5, 2, [5, 7]]
[5, 9, 7, 4, 2, 6, 1, 3, [8, 1]]
[2, 3, 8, 7, 1, 5, 9, 6, 4]
```

11) After some rounds(11 rounds) the puzzle is solved. The time displayed here was successively improved. This puzzle is only an example because it is really easy but the real performances are computed on the most challenging ones. The program displays some statistics those were improved in the last version of the program providing also the forecasted duration of solving a pool of sudokus.

```
[3, 7, 1, 5, 4, 2, 8, 9, 6]
[9, 8, 5, 3, 6, 7, 4, 1, 2]
```

```
[6, 4, 2, 8, 9, 1, 7, 5, 3]
[8, 2, 6, 1, 5, 4, 3, 7, 9]
[4, 1, 3, 2, 7, 9, 6, 8, 5]
[7, 5, 9, 6, 3, 8, 2, 4, 1]
[1, 6, 4, 9, 8, 3, 5, 2, 7]
[5, 9, 7, 4, 2, 6, 1, 3, 8]
[2, 3, 8, 7, 1, 5, 9, 6, 4]

 [SEQUENTIAL] Solved sudokus: 1 out of 1
Tot of correct over all: 1 / 1
Accuracy is: 100.00 %
Elapsed: 0.00016804933547973633 min
```

5 BackTracking

Before doing backtracking you probably would like to choose your strategy, you can do this by setting the constant "MOST-CONSTRAINED" to True that is the default choice and the most recommended one or you can swap its value to obtain the least constrained strategy. The two strategies are implemented with two functions, here below the signature. Each one will take in the possible values that can be put into the sudoku to complete it in the form of a vector and will output "x, y, choices" that are the row, the column and a vector chosen with the preferred strategy(with most constrained the choices vector is as short as possible). The choices vector represents the branches that will have the recursion tree.

```
def get_most_constrained_choice(possibles){...}
def get_least_constrained_choice(possibles){...}
```

The "possibles" are calculated by the method of Sudodata class called get_possibles that localizes only the values that instead of being an integer are vectors and based on the strategy chosen before will output only the best choice fitting the strategy.

```
def get_possibles(self)
```

Obviously if "possibles==0" is True then no solution is available but since the sudoku is not solved(because few lines before this fact is checked) then you are in a wrong branch of the recursion so the only thing to do is dropping that branch with a "return -1".

The core of the backtracking is done via the function solve, it takes as input a matrix that is the sudoku board that was formatted by the parse_sudoku function. After that the matrix is transformed into a Sudodata.

The mechanism of this recursion is very easy, when the function is called it is checked if the sudoku is solved, this is useless if this is the first call of the function but it is useful because when the recursion is called then if the sudoku is wrong the branch is killed as fast as possible. If the sudoku is not solved and doesn't contain errors then you are going to propagate the constraints. This last passage is straight-forward because comes directly from the rules of the game, the only interesting bit that I add is that I obtained better results with lower values of PROPAGATION_TRIES(eg: from 3 to 7), so the number of propagations. This is because in few cases the puzzle is simple and it is solved with 12 moves but for all the other complex puzzles you are going to put 12loops inside many and many recursions. This is going to kill your perfomances. I mean that is worthless to gain some seconds on few easy puzzles instead that having a better resolution on heavier ones. After the propagation you are going to check which are the possible moves to be done in the recursion. If there are no possible moves you are in a recursion(or the puzzle is wrong) and you have to drop it. Then from all the possible moves you take the one whose fits best your chosen strategy(MOST or LEAST). I implemented both but it is important to say that the MOST is the strongly recommended one, this is because in the best/average case you are going to choose a cell in which the possibility is going to be X or Y, in this case you have a branching factor that is 2 but for every tree of recursion you are going to simplify at least another cell, why? Just because if you have one cell that have a cyclic behavior(not solvable with propagation because swaps with another cell the values) then you absolutely solve this behaviour of the other one blocking the first. So choosing the MOST strategy means that in a recursion you fill at least 2 cells instead this is not guaranteed with LEAST nor excpectable.

```python
def solve(data):
  check = data.is_solved()
  if check == CORRECT:
    return data
  if check == WRONG:
    return -1

  for i in range(PROPAGATION_TRIES):
    data = propagate_constraints(data)

  possibles = data.get_possibles()

  if len(possibles) == 0:
    return -1

  if MOST_CONSTRAINED:
    x, y, choices = get_most_constrained_choice(possibles)
  else:
    x, y, choices = get_least_constrained_choice(possibles)

  for k in choices:
    to_pass = copy.deepcopy(data)
    to_pass.assign_cell_rc(x, y, k)

    result = solve(to_pass)

    if result != -1:
      return result

  return -1
```

6 Distributed Computations

The pool of available sudokus are distributed over a finite set of raspberry pi that use the **solve** function shown above to compute the result. The computation is started by putting the constant "DISTRIBUTE" to True, by default it is False

```
DISTRIBUTE = true
```

The commands available are the ones here below, in the "hosts" variable are listed the hostnames of the slaves that you can access to make them working. All these commands work on the entire pool at the same time. The "upload" will put your files into the raspberries, be aware that if you import a library into your files then you need to also install them into the raspberries and this can be done via the function "setup". Shutdown halts all the slaves/hosts. Start will put all the slaves listening on the port and hostname of the laptop that hosts the redis server. The redis server is needed as a storage of the jobs that are managed via the software called "RQ".

```
hosts = [
    'Rpi1',
    'Rpi2',
    'Rpi3',
    'Rpi4',
]

def shutdown(){...}
def setup(){...}
def upload(){...}
def start(){...}
```

In the way explained here the job is enqueued into on the queue "q" and the argument of the function solve is the Sudodata with the matrix of the sudoku and its solution.

```
if DISTRIBUTE:
   jobs.append(q.enqueue(solve, Sudodata(curr_sudoku,sol_sudoku)))
```

Here the code to check if a job is done and to retrieve its returning value from the RQ queue. The "job.return_value" will give you the resulting matrix computed on one of the N hosts.

```
excluded = set()
while any(not job.is_finished for job in jobs):
   for i in jobs:
      if i.is_finished and (i.get_id() not in excluded):
         excluded.add(i.get_id())
         distributed_finished += 1
      if distributed_finished != 0:
            print("\r [DISTRIBUTED] Finished sudokus:", distributed_finished,
"out of", num_sudoku_avail, end='')

    sleep(0.01)

for jj in jobs:
    result = jj.return_value
    if (result != -1) and (result is not None) and (type(result) != str):
        solved += 1
```

The library which provides me functionalities to connect to many slaves on-demand is called fabric[6].

In case we use the kaggle library performances are worst if distributed but in case of webscraped ones performances are really better, this is because webscraped ones are less fast due to the IO operations neeeded to load every sudoku.

7 Evaluation

This is the best result:

Num. Sudokus	Time in mins	Description of improvement	Constants	seconds per sudoku	Accuracy
10000sudokus	0.43mins	polished the code, switched to Kaggle sudoku dataset, using pandas lowered IO ops time	DISTRIBUTED, KAGGLE, PROPAGATION_TRIES = 5	0.0026secs	100%

I achieved it by doing:

- The equality between two matrices is very costly in my case so i discarded that type of approach.
- Strangely putting a low number of loops for the propagation enhanced the performance, I think that this is because most of the puzzles are difficult and a long loop is very heavy inside recursions. Optimal values are between 3 and 5.
- using pandas
- precomputing indexes of the boxes
- cutting out checks inside the recursion, for example there is only one "data.is_solved", this was studied in a way that the tradeoff cost/gain was as maximum as achievable .
- many other smaller studies was conducted to take out best perfoarmances taking care of spatial, time complexity and related time of execution. The proof is here below, reading from the first row in the table until the last you will be able to track many of my experiments.

This is the history of many of the attempts that I've tried to improve performances:

Num. Sudokus	Time in mins	Description of improvement	Constants	seconds per sudoku	Accuracy
1899sudokus	26mins	This was the first try	MOST_CONSTR	0.89secs	100%
1899sudokus	12mins	[ADD-1] pruning tree, adding more returns if data.void or data.duplicates	MOST_CONSTR	0.38secs	100%
1899sudokus	7.9mins	[ADD-2] pruning tree, adding more propagation just before the recursion begins	MOST_CONSTR	0.25secs	100%
1899sudokus	5mins	[ADD-3] distributing on the raspberry pi cluster	MOST_CONSTR, DISTRIBUTED	0.16secs	100%
1899sudokus	a lot	This is done for completeness of the assignment but the time is awful	LEAST_CONSTR	a lot	n.a.
1899sudokus	8.50mins	[ADD-4] equality of sudoku is performed using a custom hashing, not so efficient	HASH-COMPARISON	0.26secs	100%
1899sudokus	12mins	[MOD-5] polished and simplified the code to be more readable, performances are worst	PROPAGATION_TRIES = 12	0.38secs	100%
1899sudokus	17mins	[MOD-5] polished and simplified the code to be more readable, performances are worst	PROPAGATION_TRIES = 20	0.38secs	100%
1899sudokus	8.1mins	[MOD-5] polished and simplified the code to be more readable, performances are worst	PROPAGATION_TRIES = 8	0.38secs	100%
1899sudokus	6.58mins	[MOD-5] polished and simplified the code to be more readable, performances improved	PROPAGATION_TRIES = 5	0.20secs	100%
1899sudokus	6.92mins	[MOD-5] polished and simplified the code to be more readable, performances improved	PROPAGATION_TRIES = 4	0.22secs	100%
1899sudokus	6.79mins	[MOD-5] polished and simplified the code to be more readable, performances improved	PROPAGATION_TRIES = 3	0.21secs	100%
1899sudokus	5.65mins	[MOD-5] polished and simplified the code to be more readable, deleted one check	PROPAGATION_TRIES = 5	0.18secs	100%
1899sudokus	3.86mins	[MOD-5] polished and simplified the code to be more readable, deleted one check	DISTRIBUTED, PROPAGATION_TRIES = 5	0.12secs	100%
10000sudokus	0.56mins	polished the code, switched to Kaggle sudoku dataset, using pandas lowered IO ops time	DISTRIBUTED, KAGGLE, PROPAGATION_TRIES = 5	0.003secs	100%
10000sudokus	0.43mins	polished the code, switched to Kaggle sudoku dataset, using pandas lowered IO ops time	KAGGLE, PROPAGATION_TRIES = 5	0.0026secs	100%
10000sudokus	0.46mins	polished the code, switched to Kaggle sudoku dataset, using pandas lowered IO ops time	KAGGLE, LEAST_CONSTRAINED, PROPAGATION_TRIES = 5	0.00276	100%

7.1 Statistics on data

In this deeper analysis we detached ourselves from the time that is machine-dependent and we preferred the parameters that are dependent only about the implementation. In the table below the reader can see the number of sudokus on which the tests are conducted and the relative dataset from which they belong. The name of the dataset is really really important because the WEBSCRAPED one is more and more difficult with respect to the kaggle's one. I just calculated the mean difficulty of the datasets and the reader can interpret it as "higher means easier". So it's easy to see that KAGGLE is 8 points easier than WEBSCRAPED. This 8 points of difference have to be interpreted as "8 cells more are already given", this means that KAGGLE puzzles are a lot easier with respect to WEBSCRAPED ones because they have 8 more cells yet filled. After this the other measures are average of recursions per sudoku, and average rounds of contraint propagations per sudoku. The strategy columns indicates which is the strategy that was used, default uses no special ordering so it is faster from the algorithmic complexity but it requires probably more recursions. The last column is about how many will be the rounds of propagation that will be applied to every sudoku at every recursion, this is a critical value because if it is too high than all the nested recursions will be very very slower but a too little value will put all computations into recursions. the reason why the number of propagations is fixed a-priori is becuase in my trials showed int he table above I proved that this fact enhances the performances! But I admit that the "adaptive" way of propagating constraints in more elegant, It is available obviously but not activated by default.

We are going to talk about the tests on the KAGGLE1M dataset and this is because in that dataset the LEAST strategy is applicable because the dataset is very easy instead on the WEBSCRAPED (that contains the most difficult ones, up to [14], [15],[16] and [17]). This fact can be checked in the table below because the number of recursions and contraints are the highest ones:

Num. Sudokus	Dataset	Recursions /sudoku	Constr.props /sudoku	Mean Difficulty	Strategy	Prop. Tries
10	WEBSCRAPED	18249.1	91250.5	26.3	LEAST	5

the table above means that using the LEAST strategy on the most difficult puzzles it took on average 18249 recursions and 91250.

For what concerns the KAGGLE1M dataset we can see that the algorithm is efficient because on the average on 100000 it takes only 11 constraint propagations to complete the puzzle and exaclty 0 recursions. We can also note that in many cases the DEFAULT strategy is more efficient with respect to the LEAST strategy and this is simple to be explained becuase we know that the LEAST ordering is not nice from the perspective of the recursion instead using the DEFAULT ordering is similar to the fact of having a random order because we can say that the 0-filled cells are in a certain sense randomly ordered.

Num. Sudokus	Dataset	Recursions /sudoku	Constr.props /sudoku	Mean Difficulty	Strategy	Prop. Tries
100000	KAGGLE1M	0.00122	11.01342	33.81248	MOST	11

All the tests here below achieved 100% accuracy on every test.

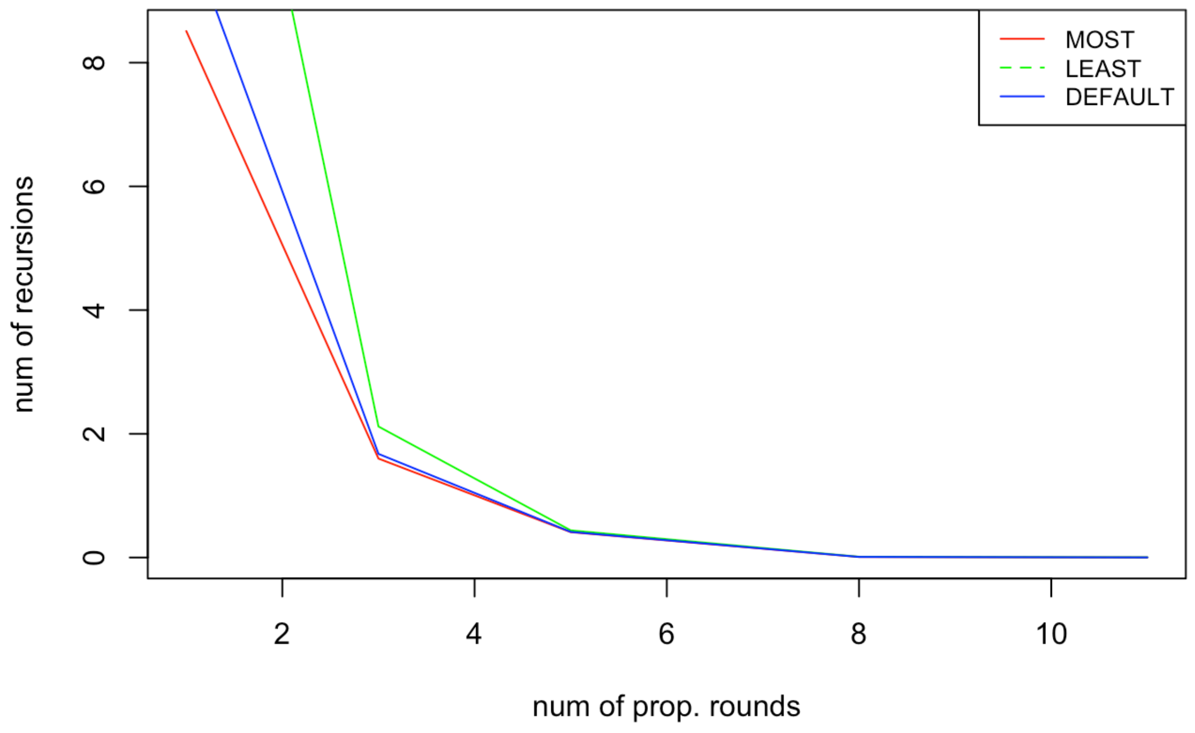
Num. Sudokus	Dataset	Recursions /sudoku	Constr.props /sudoku	Mean Difficulty	Strategy	Prop. Tries
1899	WEBSCRAPED	174.148	875.742	25.737	MOST	5
100'000	KAGGLE1M	0.41	7.05	33.812	MOST	5
30	WEBSCRAPED	105.67	533.34	25.8332	MOST	5
30	WEBSCRAPED	248.867	1249.33	25.8332	DEFAULT	5
30	WEBSCRAPED	-	-	25.8332	LEAST	5
10	WEBSCRAPED	18249.1	91250.5	26.3	LEAST	5
30	KAGGLE1M	0.333	6.666	33.63	MOST	5
30	KAGGLE1M	0.333	6.666	33.63	DEFAULT	5
30	KAGGLE1M	0.333	6.666	33.63	LEAST	5
1000	KAGGLE1M	0.388	6.94	33.753	MOST	5
1000	KAGGLE1M	0.414	7.07	33.753	LEAST	5
1000	KAGGLE1M	0.39	6.95	33.753	DEFAULT	5
10000	KAGGLE1M	0.4013	7.0065	33.8165	MOST	5
10000	KAGGLE1M	0.4263	7.1315	33.8165	LEAST	5
10000	KAGGLE1M	0.4062	7.031	33.8165	DEFAULT	5
100000	KAGGLE1M	0.41041	7.05205	33.81248	MOST	5
100000	KAGGLE1M	0.43952	7.1976	33.81248	LEAST	5
100000	KAGGLE1M	0.4144	7.072	33.81248	DEFAULT	5
100000	KAGGLE1M	0.00927	8.07416	33.81248	MOST	8
100000	KAGGLE1M	0.01215	8.0972	33.81248	LEAST	8
100000	KAGGLE1M	0.00964	8.07712	33.81248	DEFAULT	8
100000	KAGGLE1M	0.00122	11.01342	33.81248	MOST	11
100000	KAGGLE1M	0.00374	11.04114	33.81248	LEAST	11
100000	KAGGLE1M	0.00155	11.01705	33.81248	DEFAULT	11
100000	KAGGLE1M	1.60018	7.80054	33.81248	MOST	3
100000	KAGGLE1M	2.11849	9.35547	33.81248	LEAST	3
100000	KAGGLE1M	1.67568	8.02704	33.81248	DEFAULT	3
100000	KAGGLE1M	8.51034	9.51034	33.81248	MOST	1
100000	KAGGLE1M	17.06548	18.06548	33.81248	LEAST	1
100000	KAGGLE1M	10.15919	11.15919	33.81248	DEFAULT	1

7.2 Plotting results

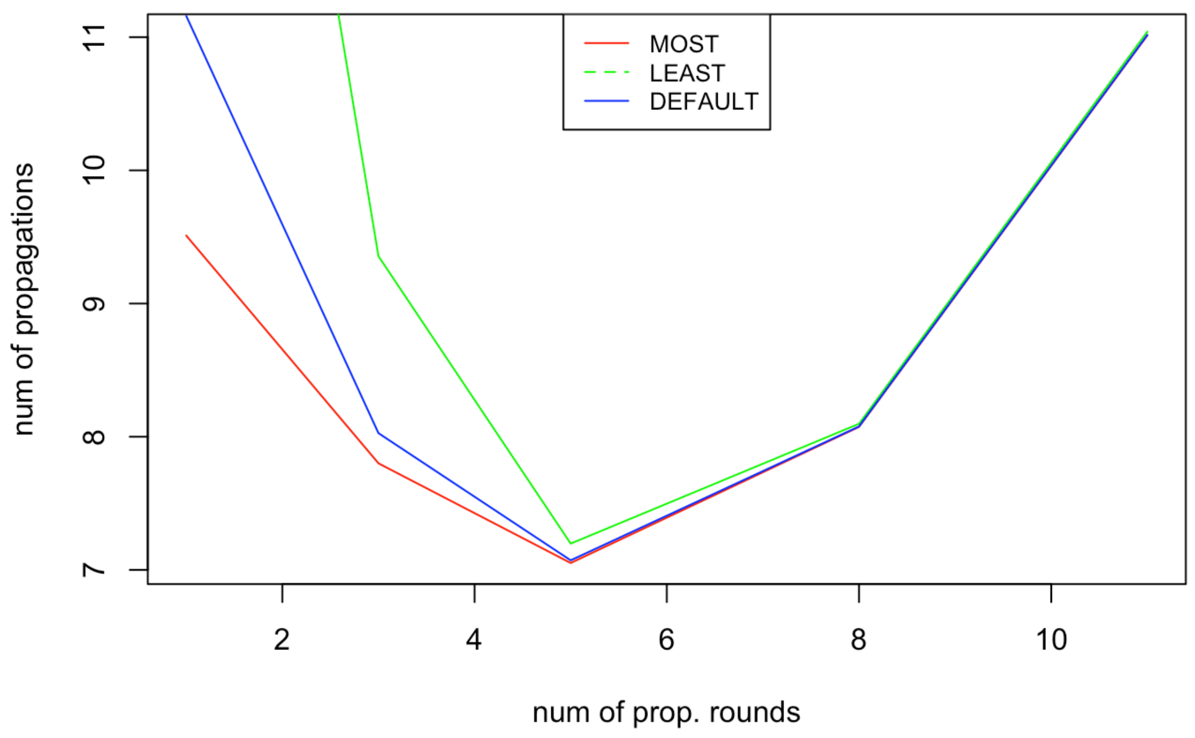
Data below is taken from the above table with num sudoku = 100'000, dataset = KAGGLE1M and obviously difficulty = 33.81248.

From the images below we can see that the MOST strategy beats all the others for a small number of propagation rounds. This can be seen from the graphs because the red line is the lower one, this is because lower line means less recursions and propagations that is obviously better. In general it beats all of them until the number of propagation rounds is small. When the number of propagation rounds is higher then the performances decrease for all the methods and no difference can be found.

I would like to remember the reader that these tests are done on the sudokus provided by kaggle that seems to be very easy because otherwise on very difficult puzzles such as WEBSCRAPED the LEAST strategy takes too much time to complete. Also the x-axis is about the number of a-priori fixed propagation rounds. The number of propagation rounds can be also inferred or adaptively induced by the program but it is expensive and we decided for this better performing strategy. The other strategy is also available.



Also here the same thing as above can be seen.



8 Conclusions

In conclusion I can say that was very interesting and I think that probably the program can be faster distributing it using the python multiprocessing library natively. I didn't do that because actually it is already possible by putting the machine that is the master in the queue of the slaves. This is because the slaves natively works in multiprocessing. Other improvements are available such as using genetic algorithms and applying them can be very interesting.

9 Bibliography

- [1] Berggren, “A study of Sudoku solving algorithms.”
- [2] Simonis, “Sudoku as a Constraint Problem.”
- [3] https://medium.com/@machine.learning.language/anti-plagiarism-python-e5f259eb3f46
- [4] https://medium.com/@machine.learning.language/pygraham-functional-methods-in-python-d5921208416a
- [5] slides of the professor, part of "Local Search", slide number 34 of 61
- [6] http://www.fabfile.org
- [7] https://python-rq.org
- [8] https://pypi.org/project/redis/
- [9] https://www.crummy.com/software/BeautifulSoup/bs4/doc/
- [10] https://www.kaggle.com/bryanpark/sudoku
- [11] https://www.kaggle.com/rohanrao/sudoku
- [12] https://www.w3resource.com/numpy/manipulation/squeeze.php

- [13] https://docs.python.org/3/library/re.html
- [14] http://lipas.uwasa.fi/~timan/sudoku/
- [15] http://norvig.com/easy50.txt
- [16] https://raw.githubusercontent.com/dimitri/sudoku/master/sudoku.txt
- [17] https://projecteuler.net/project/resources/p096_sudoku.txt