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. It is present in the bibliography[4] an article that explains it.

Some libraries that were used in the assignment are previous projects of the author published on PyPI, in these cases will be put references.

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 rule as written before and building a method that checks that fact but my implementation works in a different way, it prunes 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.

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 sudoku 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".

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. The peculiarity of this class is that I maintained the triplets of data by rows that simplifies the return of boxes.

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].

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 cell_rtc(self, r, t, c){...}
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 return the absolute indexes of the box.

```
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){...}
```

4 Constraint Propagation

It is done via the method below:

```
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
```

5 BackTracking

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. We take also a snapshot of the Sudodata object in that moment because if at the end no modification has occured then the propagation was useless so the backtracking is needed.

```
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

7 Evaluation

Num. Sudoku	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%

8 Conclusions

I think that in this case the knowledge of the domain was crucial and I was able to devise a good solution since first try because I'm passionate.

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.

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