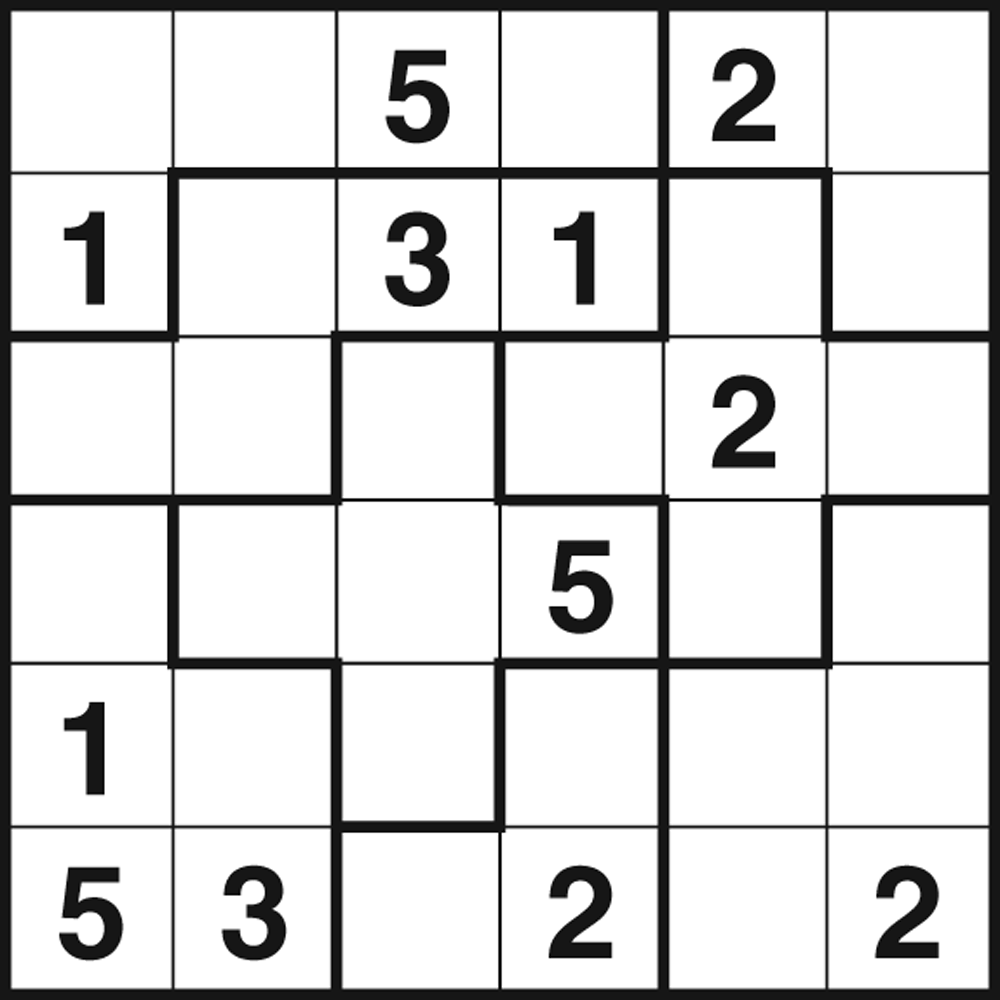
Algorithmics SAT: Suguru Solver

Suguru is a Japanese puzzle invented by Naoki Inaba in 2001 under the name “ナンバーブロック” meaning ‘number block’, but was given the name Suguru, which is a contraction of ‘Suji gurupu’ meaning ‘number group’ was given by Puzzle Media Ltd (UK) as they believed it reflected what the puzzle was better. [1]

This is because the puzzle is divided into a grid, where regions (or groups) are outlined in a thicker line. When given a puzzle to solve, only a limited amount of the numbers are given, however they are enough to figure out the rest of the puzzle by following the two rules:

1. For a region of size N, every number 1 to N must be contained inside of it, and

*Fig. 1 A Suguru puzzle [4]*

1. No two of the same number can be adjacent, including diagonally.

An example of a puzzle can be seen in Figure 1. The goal of the puzzle is to fill every cell inside the grid with a valid number; one that follows the aforementioned rules. Knowing this, the puzzle can be broken up into a few key features:

* The grid itself, including how many cells there are and its layout,
* The values of the cells, and the cells that are adjacent to it;
  + Note: within this report, adjacent includes diagonal at all times,
* The regions that the cells are within, outlined by the thicker lines.

Pseudocode

Pseudocode Start

import suguru ADT referred to as suguru //pre-constructed ADT representing a suguru puzzle now including regions property

//ASSUMES: the suguru abides by all the rules of the puzzle, and has one valid solution

function updatePossibleValues(node)

//input: a node wanting to have possibleValues updated; void output

for region\_node in suguru.regionNodes(node) do //update based on region relation nodes

if region\_node.value is not None and region\_node.value in node.possibleValues then

remove region\_node.value from node.possibleValues

end if

end for

for adjacent\_node in suguru.adjacentNodes(node) do //update based on adjacent related nodes

if adjacent\_node.value is not None and adjacent\_node.value in node.possibleValues then

remove adjacent\_node.value from node.possibleValues

end if

end for

function nodeSolvable(node)

//input: a node, to be checked if solvable; outputting either the value to be updated to, or null if not able to

//if only one possible value

if length of node.possibleValues = 1 then

return node.possibleValues[0]

end if

//if final posisble value within the region

for possibleValue in node.possibleValues do

is\_unique <-- True

for region\_node in suguru.regionNodes(node)

if possibleValue in region\_node.possibleValues then

is\_unique <-- False

break

end if

end for

if is\_unique is True do

return possibleValue

end if

end for

return None

function updateStack(stack, node, sn<--false)

//input: the current stack state, and the node which has just had a change which impacts surroundings; outputs the updated state of the stack

//the sn Boolean value dicatates whether to update the adjacent or just region

for region\_node in suguru.regionNodes(node) do //add all region related nodes to the stack

if region\_node.value is None and region\_node not in stack then

stack.append(region\_node)

end if

end for

if sn not true:

for adjacent\_node in suguru.adjacentNodes(node) do //add all the adjacent related nodes

if adjacent\_node.value is None and adjacent\_node not in stack then

stack.append(adjacent\_node)

end if

end for

end if

return stack

function sharedNeighbours(nodeList)

//input: a list of all the nodes which contains a certain possibleValue (currently being checked)

//output: the overlapping neighbours of the nodes within the nodeList

neighbour\_count <-- empty dictionary

shared\_neighbours <-- empty list

//count the occurences of each adjacent node

for node in nodeList do

for adjacent\_node in suguru.adjacentNodes(node) do

if adjacent\_node not in nodeList then

if adjacent\_node not in neighbour\_count then

neighbour\_count[adjacent\_node] <-- 0

end if

neighbour\_count[adjacent\_node] <-- neighbour\_count[adjacent\_node] + 1

end if

end for

end for

//identify the shared neighbours

for node, count, in neighbour\_count.items() do

if count = length of nodeList then

shared\_neighbours.append(node)

end if

end for

return shared\_neighbours

function suguruSolver(suguru)

//Main suguru solving algorithm, requiring the suguru to be parsed in, and outputting the final state

//initialization phase

for node in suguru.nodes do

if node.value is None

region\_size = length of suguru.regionNodes(node) + 1 //have to add one because regionNodes returns every OTHER node in region

node.possibleValues <-- [from 1 to region\_size]

end if

end for

last\_reg\_visited <-- suguru.regions[0]

//main loop

do

update\_made <-- False

stack <-- suguru.nodes //add all nodes to stack

//deplete the stack whenever not empty

while stack is not empty do

node <-- stack.pop()

if node.value is None then

updatePossibleValues(node)

solvable\_value <-- nodeSolvable(node)

if solvable\_value is not None then //if node solvable

node.value <-- solvable\_value

node.possiblValues <-- empty list

stack <-- updateStack(stack, node)

update\_made <-- True

end if

end if

end while

//shared neighbour strategy

breaker <-- False

for regionNodes in suguru.regions starting after last\_reg\_visited do

for possibleValue in range from 1 to length of regionNodes do

shared\_possibleValue <-- empty list

for region\_node in regionNodes do

if region\_node.value is None and possibleValue in region\_node.possibleValues then

add region\_node to shared\_possibleValue

end if

end for

//get the shared neighbours then filter down the nodes

shared\_neighbour\_options <-- sharedNeighbours(shared\_possibleValue)

for option in shared\_neighbour\_options do

if option.value is None and possibleValue in option.possibleValues then

remove possibleValue from shared\_neighbour.possibleValues

update\_made <-- True

breaker <-- True

last\_reg\_visited <-- region

updateStack(region\_node, sn<--true) //fill the stack with cells that have new information based on them

end if

if breaker is True then //break here if an update is made to increase efficiency

break

end if

end for

while update\_made is True

return suguru

display(suguruSolver(suguru))

Pseudocode End

# How does the algorithm work?

What is the Suguru ADT?

The Suguru is a variation of a graph ADT, with extra specific details included for precision, efficiency and fitness for purpose. Every cell within the Suguru puzzle is modelled as node, with properties for its face value, and its possible values that it could be. It is then interconnected within the graph, to every node that it’s adjacent to, and every node that it shares a region with, in respective, different edge types. Within the pseudocode, the following functions are utilised:

|  |  |
| --- | --- |
| **Signature** | **Description** |
| createSuguru: gridValues x regionValues 🡪 Suguru (ADT) | Requires the values of the cells and the region placements to create a new Suguru ADT |
| nodes: Suguru 🡪 list; | Returns a list of all nodes inside of the Suguru ADT |
| regions: Suguru 🡪 list of lists; | Returns a list of lists containing nodes in respective regions |
| adjacentNodes: Suguru x node 🡪 list; | Returns a list of all nodes that are adjacent to given node |
| regionNodes: Suguru x node 🡪 list; | Returns a list of all nodes that are within the same region of as the given node |
| possibleValues: Suguru x node 🡪 list; | A property, attached to every node containing a set of values which the node could be (updated manually) |
| value: Suguru x node 🡪 int; | A property, used in place of nodePossibilites if the value of the node is known (updated manually after creation) |

What does each module do?

|  |  |
| --- | --- |
| **Function** | **Description** |
| Update Possible Values | Given a node, every node within that node’s adjacent nodes and region nodes, if it has a value associated with it, it’s removed from the node’s possible value set. It is a void function, meaning it only updates the Suguru and doesn’t return anything. |
| Node Solvable | Given a node, it will be checked for the two conditions where it can be updated:   1. The node only has one possible value 2. The node is the last node within it’s region that contains a possible value   If it can, then it returns what that value is, and if not, it returns null/ none |
| Update Stack | Given the current state of the stack, and a node in focus, it will add every adjacent and region node of the given node provided it doesn’t have a value yet and is not already inside of the stack.  It will then return the new state of the stack for it to be officially updated.  If sn is true, then it is being called from the shared neighbours sub algorithm, which means only the region nodes are being effected by the update and should be the only ones to join the stack. |
| Shared Neighbours | Given a list of nodes, often a subset of a region all containing the same possible value, the function will return a list of overlapping adjacent neighbours between all the nodes within the given list. |

How does the main solving algorithm work?

Firstly, it will initialise the possible values for every cell. Then it will queue up a full stack.

The rest of the solving algorithm will continually either update cell values, or cell possible values until it’s solved. This is guaranteed to be the case evident in the provided proof. Within every stack call, the topmost item will be removed and check if it has a value already (as solved cells are useless to work on). The cell in focus will then have it’s possible values updated so it can be accurately checked if it’s solvable. If it is solvable, it will update that cell’s value, and run updateStack to queue in all cells effected by this update.

Once the stack is depleted, the shared neighbours strategy will find an update for the possible values of some cells within the puzzle, and hence append effected cells to the stack to run check for solving possibilities again. The shared neighbours strategy essentially checks for overlapping adjacent cells for cells with the same possible value inside a region, as it is guaranteed that the cell will then not be able to include that possible value no matter where that possible value is placed within the options.

(rough) Call graph:

A diagram of a software system

Description automatically generated

# Time complexity analysis

Context

For the following time complexity analysis, the goal of the investigation is to find the upper bound/ worst case time complexity of the algorithm. Therefore, some metrics/ techniques/ parameters are overlooked and/or simplified and could not be so easily justified for a specific time complexity analysis but doesn’t hurt the integrity for a worst case (WC) scenario. To understand the analysis, the following parameters will be useful to know:

Units that grow with the puzzle size:

n = number of cells within the puzzle/ nodes within the graph

m = number of regions within the puzzle/ different connected sections within the graph

Units that do not grow with the puzzle size:

r = largest region size within the puzzle

s = smallest region size within the puzzle

S = stack operations running on topmost item (S1 means a successful update, S2 means not)

SN = Shared neighbours strategy running

Time complexity of modules

**Shared neighbours**

**A screenshot of a computer program

Description automatically generated**

**Update possible values**

**A computer screen shot of a code

Description automatically generated**

**Node solvable**

A computer screen shot of a code

Description automatically generated

**Update stack**

**A screenshot of a computer program

Description automatically generated**

Worst case scenario

Time complexity:

To get this time complexity, the analysis can be broken down into sections, and the diagram below can assist.

Firstly, the algorithm initialises the Suguru at the beginning of the solver.

Then it will run through the solving process, in the worst case solving one value at a time, not even every loop, but intermittently between many loops of SN calls. This is because, ultimately, every node value update gets the solver significantly closer to solving the entire puzzle. Therefore, for the solver to update values as irregularly as possible, prolonged periods would have to occur between updates, which can be achieved through running the shared neighbours strategy as many times as possible so that the algorithm will continue to run but make as least progress as possible.

To achieve as least progress as possible, not only would the algorithm have to run through every region to find an update for the possible values, but theoretically it could not lead to an immediate update of a cell value, and instead multiple inferences could be required to do so.

It should be noted that, it is extremely unlikely to need to go through every region to find an update, and multiple inferences are even less likely. The chance of two run throughs begin required is unlikely to find in any regular Suguru, but technically more than two could be needed, up to a theoretical m times, which is infinitesimally small chance but must be accounted for. In any regular puzzle, some update would be found before reaching this point, and it would never have to happen to this extent repeatedly as there’s not enough possible values to remove total, but for the sake of finding the complete upper bound – it has been factored in.

General time complexity diagram

A math equations on a white paper

Description automatically generated

**Stack time complexity**

A computer screen shot of a code

Description automatically generated

S1:

S2:

**Shared neighbours time complexity**

A screenshot of a computer program

Description automatically generated

Final worse case time complexity

(factor out and remove r because it’s a constant, therefore, not contributing to final time complexity)

(In the worst case, m would be equal to to be as large as possible)

(because s is also a constant, it can be factored out and removed too)

(this produces a very rough time complexity excluding many constant factors, but can provide an accurate upper bound/ big O time complexity)

(therefore, solves the puzzle, clearly in polynomial time)

# Further understanding

Understanding the polynomial solving nature of the time complexity

Suppose this algorithm is being compared to a backtracking variant, a known NP solution. It is important to think of backtracking as branching at every decision. Although it can utilise heuristics and constraints, it will still be running the same algorithm on different instances/ states of the puzzle, making the solution non-polynomial. The difference inside of this solution, is the linear nature, it never calls itself/ running the same solving technique within itself, therefore, never branching and remains polynomial.

Inside a backtracking solution, choices are made at ever uncertainty; meanwhile a choice is never made unless certain within this algorithm thereby never have to test different possibilities and can just continue to make progress knowing that the rest of the state it’s in is accurate.

Certainty to solve; Proof by contradiction

Inside Suguru puzzles, it is assumed that there is only one valid solution for a given puzzle, and it is known that every loop the algorithm will attempt to solve as much as it can, through updating cell values and possible values. This is an important aspect that makes this algorithm viable and able to work.

Suppose it were to reach a point where it could not make any more progress. This would mean that the current state of the Suguru has multiple options for what it could do next, this would result in two cases:

Case 1: Only one of those options ends up working out.

This is a contradiction as it means that there were not multiple options, and therefore the one singular option would’ve been found.

Case 2: Multiple of those options ends up working out.

This means that the solver ends up finding multiple solutions, which is also a contradiction as the puzzle is not meant to have multiple solutions.

# Further application

Sudoku integration

The ramifications of this solving approach extent beyond sugurus. Theoretically it can be applied to any problem where only one valid solution is possible. As an example, Sudokus, a similar puzzle to sugurus will be used to explain the way the techniques can be integrated.

In a sudoku, cells and regions are used similarly to constrict values, and similar rules are applied to the entirety of the puzzle therefore, not many adjustments are needed for the algorithm to be effective on sudokus too.

In a prototype model, used to solve sudokus, the new constraints are inputted, and the stack works the same. The main difference is how to remove the possible values. Technically it works on the same methodology, in which cell possible values can be removed based on guaranteed knowledge inferred from surrounding information. The difference in sudokus though is that they have more complicated relationships to the surrounding cells and make guaranteeing such knowledge harder.

Utilising some primitive techniques to do so, the prototype model is able to solve practically all common sudokus (including extreme difficulty on sudoku.com, and hard difficulty on New York Times). It becomes complicated when sudokus have been crafted with the intention that they require advanced techniques to deduce the outcomes, in a way they need specific solving techniques which the primitive ones do not cover. Theoretically if every possible value eliminating method was built in, or a general one could be built, it would be able to solve every sudoku puzzle too.

If a more complete version was created, it would likely have a worse case time complexity integer higher than the NP backtracking version for most puzzles as the sheer amount of methods required or checks made would be overwhelming, but it would still be solve in polynomial time, and the average solve would be much more efficient.

This is crucial because sudokus are currently generally solved via backtracking methods, which are notoriously slow, and inefficient, therefore, if this technique can be integrated into it, or other similar problems, it could revolutionize the way in which these types of problems are approached.

NP problem solving potential

To start with, consider a problem containing a definite solution. There will be reasons as to why there is only one solution, for example within a CSP puzzle like a sudoku, altering values causes contradictions within the rules. Generally, as explained the solving process works towards reaching the solution using these constraints, but it can reach a point where no more direct decisions can be made. However, provided that the state is guaranteed to be correct, then inferences can be made about between the current and goal state which doesn’t

Essentially, given the current state in a puzzle is guaranteed to be correct, inferences can be made that

Consider a problem containing a definite solution. In theory it can apply to optimisation problems too, where there is one best solution, however will be overlooked for the following explanation. Provided every other state of the problem (every other attempt) is not a valid solution, it means the final goal state can be worked towards using inferences and connections between this knowledge. For example, the solving method would continually check for updates to the state, either locked in values, or possibilities for each option

and it is known to contain a solution as there are constraints within the problem which eliminate other possible solutions and validate the correct one. In any scenario like this, the constraints and initial state can be used to eliminate possible states that branch from the current. There could be scenarios where the constraints have to be inferred, but it should still be able to apply properly provided they ensure/ guarantee an outcome.

Then the next step is to make some sort of progress towards the goal state, ideally updating a piece of information within the problem based on guaranteed factors which let further deductions be made surely.