University of Glasgow | School of Computing Science

# Animating a Sudoku solver

Gabriel I. Stratan

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

Level 4 Project — November 18, 2015

**Abstract**

Sudoku is a popular puzzle played all over the world. It consists of filling in a 9x9 grid such that every row, column and 3x3 sub-grids have different digits from 1 to 9. Solving the puzzle will make use of the all-different algorithm from Constraint Programming for which an implementation will be provided. Finally, the program will animate all the steps done by the algorithm.

## Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: ______________________ Signature: ________________________

# Chapter 1

# Introduction

## 1.1 Aims

## 1.2 Background

## 1.3 Motivation

## 1.4 Report Content

- Chapter 2
- Chapter 3
- Chapter 4

# Chapter 2

# Requirements

## 2.1 Problem Analysis

## 2.2 Proposed Solution

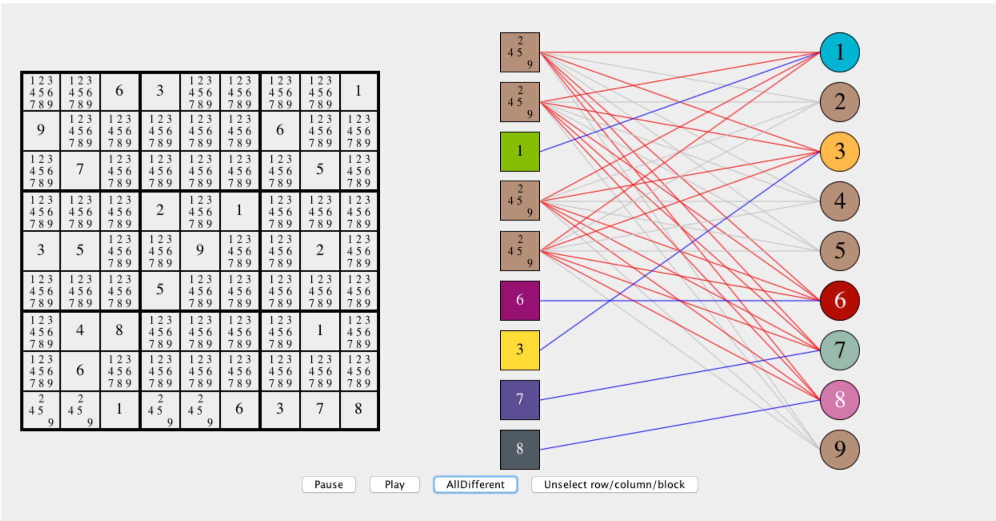



**Fig. 2.1:** The state of the program after the all-different algorithm finishes on a row, in this case, the last one.

# Chapter 3

# Design and Implementation

## 3.1 High-Level Overview

Model-View-Controller

### 3.1.1 Problem Instance Representation

Sudoku files

## 3.2 FordFulkerson's algorithm for computing a maximum matching

#### Origins

#### Description

#### Implementation

#### Visualisation

#### Discussion

## 3.3 Tarjan's algorithm for finding strongly connected components

#### Origins

#### Description

#### Implementation

#### Visualisation

#### Discussion

**Algorithm 1:** Ford Fulkerson

```
void FordFulkerson(Graph G)
begin
    Global int capacity[][] ← getCapacity(G)
    Global int n ← |V(G)|
    Global int source ← 0
    Global int sink ← n − 1
    Global pred ← new int[n]
    Global visited ← new boolean[n]
    Global Q ← new Queue()
    while bfs(G) do
        update()

boolean bfs(Graph G)
begin
    clear(Q)
    fill(pred, −1)
    fill(visited, false)
    enqueue(source, Q)
    visited[source] ← true
    while (¬isEmpty(Q)) do
        int v ← dequeue(Q)
        if v = sink then return true
        for w ← 0 to n do
            if ¬visited[w] and capacity[v][w] > 0 then
                pred[w] ← v
                enqueue(w, Q)
                visited[w] ← true
    return false

void update()
begin
    int f ← minCost()
    int v ← sink
    while pred[v] ≠ −1 do
        int u ← pred[v]
        capacity[u][v] ← capacity[u][v] − f
        capacity[v][u] ← capacity[v][u] + f

int minCost()
begin
    int minCost ← ∞
    int v ← sink
    while pred[v] ≠ −1 do
        minCost ← minimum(minCost, capacity[pred[v]][v])
        v ← pred[v]
    return minCost
```

Having a maximum matching for our graph, the next step in the alldifferent algorithm is to turn the previously undirected graph, into a directed one. The matches resulted from the Ford Fulkerson algorithm are assigned a direction from the 1 to 9 values to the corresponding cells of the Sudoku row. The edges that remain unused in the matching are given a direction from left to right (i.e. from the Sudoku cells of the row to the 1-9 values).

// write that we delete S/T nodes and corresponding edges

Now that all the edges from the graph are directed, the next step in the alldifferent algorithm is to find the strongly connected components of the graph. In order to do this, we introduce now Tarjans algorithm for finding strongly connected components (SCCs) in a given graph G.

The algorithm starts by visiting every node in the directed graph in a depth first search manner. During the search, nodes are added to a stack in the order they are discovered only if they were not already part of the stack.

Backtracking is triggered when we reach a node that is upper compared to our previous node (if(min ¡ low[u])). We know this by keeping record of the upmost node reachable from node u, including node u itself during each branch of the depth first search. We use low to denote the minimum index representing the upmost node in the branch.

If the current node is less than the upper node is less than the current index, it means that we have

If the upper node is equal to the node we are currently visiting, then the algorithm just found a strongly connected component that contains all nodes on the stack starting from the top of it, until encountering the current node. The nodes are popped out of the stack and a SCC id/index is assigned to it for later use. Once the current node is reached, we increment the count of the SCCs to start filling a new SCC.

// write that there are no self loops / self edges (u!=i) [1]

**Algorithm 2:** Tarjan's strongly connected components

```
void Tarjan(Graph G)
begin
    Global int pre ← 0
    Global int components ← 0
    Global int n ← |V(G)|
    Global S ← new Stack()
    Global stacked ← new boolean[n]
    Global id ← new int[n]
    Global low ← new int[n]
    for u ∈ V(G) do
        if ¬stacked[u] then dfs(u, G)

void dfs(int u, Graph G)
begin
    push(u, S)
    stacked[u] ← true
    low[u] ← pre
    pre ← pre + 1
    int min ← low[u]
    for v ∈ N(u, G) do
        if ¬stacked[v] then dfs(v, G)
        min ← minimum(low[v], min)
    low[u] ← minimum(low[u], min)
    integer v
    repeat
        v ← pop(S)
        id[v] ← components
        low[v] ← n
    until v ≠ u
    components ← components + 1
```

Ford Fulkerson's algorithm for finding a maximum flow in a graph is shown in algorithm 1. The $FordFulkerson$ procedure on line 1 takes a graph $G$ as an argument and represents the start of the algorithm. The graph passed as an argument holds information about the capacity of each edge. Line 3 stores the capacity of the graph $G$ in a global two-dimensional integer array. The capacity of an edge between vertices $u$, $v$ can be found out in $capacity[u][v]$. Line 4 declares an integer $n$ used to store the number of vertices in the given graph. Lines 5 and 6 declare two integers called $source$ and $sink$ representing the first and the last vertices in the given graph.

In line 7 we declare a vertex-indexed array $pred$ that is used to store references to the previous vertex discovered in the breadth-first search tree. Line 8 declares a vertex-indexed array $visited$ used to keep track if a vertex has already been visited in the breadth-first search procedure. Line 9 declares a queue of integers $Q$ used to store **XXX**.

The algorithm consists of repeated calls to the breadth-first search procedure (line 10). In line 11, as long as the breadth-first search procedure finds **shorter augmenting paths?** we call the update procedure to update the capacities of the edges.

The $breadth - firstsearch$ procedure starts at line 12 and takes a graph $G$ as an argument. In lines 14 to 16, the queue $Q$, $pred$ and $visited$ array are reset to have default values.

The breadth-first search calls always start from the first vertex, representing the source. This vertex is enqueued on queue $Q$ in line 17 and is marked as visited in line 18.

Lines 19 to 26 contain a loop performing changes on the queue $Q$ that is used to store an augmenting path. The loop starts on line 20 by dequeuing a vertex from the queue $Q$ and storing it in $v$. Line 21 forces the breadth-first search procedure to return $true$ when we reach the last vertex in the graph, the sink. Lines 22 to 26 contain a loop over **all the vertices** $w$ in the graph. In line 23 we check to see if the discovered vertex $w$ hasn't been already visited and we consider it if it still has available capacity. Line 24 stores the parent value of $w$ in the augmenting path. Line 25 enqueues the discovered vertex $w$ on the queue $Q$ and marks it as visited in line 26. Finally, the breadth-first search procedure returns $false$ in case **of no augmenting path found?**.

The $update$ procedure at line 28 is used to update the capacities in the graph to reflect the found paths. Line 30 declares an integer $f$ used to store the **difference/delta**. Line 31 declares an integer $v$, initially representing the last vertex in the graph called the sink. Lines 32 to 35 contain a loop that updates the capacities of the edges **in the augmenting path?**. Line 33 uses the integer $u$ to store a vertex that is the neighbour of vertex $v$ **in the path**, so that together they make an edge. Lines 34 and 35 updates the flow along the edge from $u$ to $v$ and $v$ to $u$ by the found difference.

The $minCost$ procedure at line 36 is used to find the minimum cost, in terms of capacity used, from the sink **to the source?**. Line 38 declares an integer $minCost$ used to represent the current value of the minimum cost, and it has a very large number at the time of its initialization. Line 39 declares an integer variable $v$ used to store vertices. The variable $v$ is initialized as the last vertex in the graph called the sink. Lines 40 to 42 contains a **loop over the parent vertices** of $v$. Line 41 updates the value of the minCost variable **to the minimum**. The $minimum$ procedure returns the minimum of two given integers. Line 42 updates the value of integer $v$ to **its parent**. At the end of the $minCost$ procedure, we return the value of the currently found $minCost$ variable.

Tarjan's algorithm for finding strongly connected components is shown in algorithm 2. The $Tarjan$ procedure in line 1 takes a Graph $G$ as an argument and represents the start of the algorithm. The global integer $pre$ declared in line 3 is used to store **XXX**. The integer $components$ in line 4 keeps track of how many components were identified in the given graph. The integer $n$ in line 5 represents the number of vertices in the given graph.

In line 6 we introduce a Stack data structure that will hold all the vertices that are part of the same strongly connected component. Line 7 contains a declaration for the $stacked$ vertex-indexed array of booleans used to keep track if a vertex is present or not on the stack $S$.

An integer vertex-indexed array $id$ is declared in line 8 to store the id of the strongly connected component of each vertex. Line 9 declares a vertex-indexed array $low$ used to store the topmost reachable vertex in the depth-first search tree through a back edge.

Lines 10 and 11 contain a for loop that calls a procedure for performing depth first search on every vertex in the given graph that is not currently stacked.

Line 12 declares a recursive procedure for performing a depth-first search starting from vertex $u$ in graph $G$.

The procedure starts by pushing the vertex $u$ on the stack $S$ (line 14) and updating the value in the $stacked$ array (line 15) to reflect the change. Line 16 sets the vertex's $v$ value for $low$ to $pre$ **XXX** in line 16 to 17. Line 18 declares an integer $min$ that has an initial value **XXX**.

Lines 19 to 21 loop through every vertex $v$ that is a neighbour of vertex $u$. The procedure $N(u, G)$ on line 19 returns a list of neighbouring vertices of a given vertex, not including the given vertex. If the discovered neighbour $v$ is not currently on the stack $S$, then we proceed to make a recursive call to the depth-first search function on the found vertex.

After the recursive call finishes processing the sub-tree of the neighbouring vertex $v$, $min$ gets updated on line 21 to store the topmost reachable vertex that could be the same, or even higher in the tree if a higher back-edge from $v$ was discovered.

Line 22 updates the topmost reachable vertex of vertex $u$ to reflect any changes after processing all the neighbouring vertices.

Line 23 declares an integer $v$ used to store a vertex that for the repeat-until loop in lines 24 to 28. The loop begins by popping vertices out of the stack $s$ on line 25. Each vertex $v$ found on the stack is assigned the current component id. Line 26 assigns the last leaf node of the graph as a topmost reachable vertex of vertex $v$ in order to mark it as already part of a component. The repeat-until loop in line 24 to 28 is runs until finding **XXX self edge?**.

The depth-first search procedure finishes on line 29 by updating the total number of strongly connected components discovered.

# Chapter 4

# Conclusion

We have shown how to implement the all-different constraint.

# Appendices

# Appendix A

# Running the Program

An example of running from the command line is as follows:

```
> javac *.java
> java Sudoku
```

This will open the application loaded with the hard Sudoku problem $/herald20061222H.txt$.

**TODO**: what about the Choco3 library? add it to path? remove it?

# Appendix B

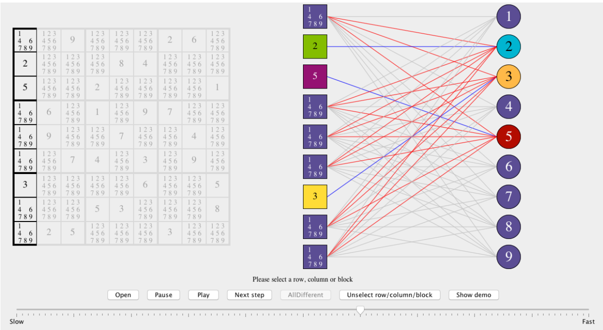
# Proof of concept

The following sequence of steps will provide a visual proof of the all-different algorithm. The following figures capture the user running the all-different algorithm on 5 predetermined rows, columns or 3x3 sub-grids. For demonstration purposes, the Sudoku instance used is $lockedset.txt$.

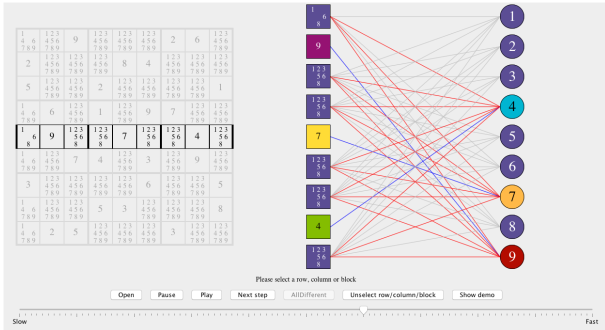
The sequence is as follows:

```
> Open lockedset.txt Sudoku instance
>
> Select the 1st column
> Run the all-different algorithm
> Unselect
>
> Select the 5th row
> Run the all-different algorithm
> Unselect
>
> Select the 6th row
> Run the all-different algorithm
> Unselect
>
> Select the 4th 3x3 sub-grid
> Run the all-different algorithm
> Unselect
>
> Select the 1st column
> Run the all-different algorithm
> Unselect
```

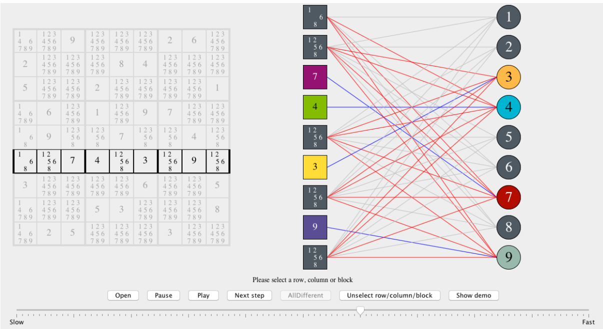
Alternatively, the user can press the $ShowDemo$ button to run the same steps.



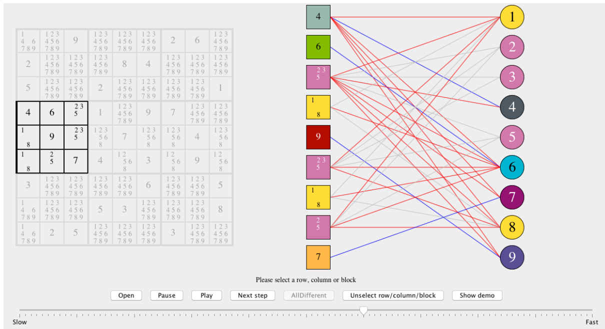

(a) 1st column after running the all-different algorithm



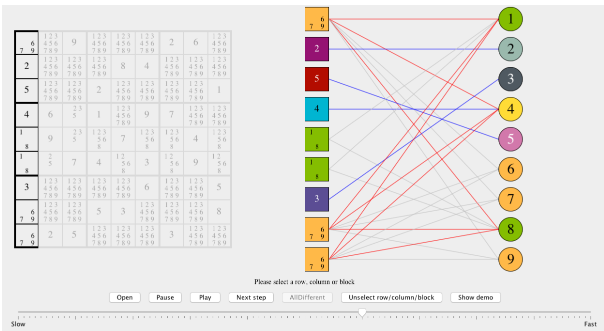

(b) 5th row after running the all-different algorithm




(c) 6th row after running the all-different algorithm




(d) 4th 3x3 sub-grid after running the all-different algorithm




(e) 1st column after running the all-different algorithm

**Fig. B.1:** Steps in the demo

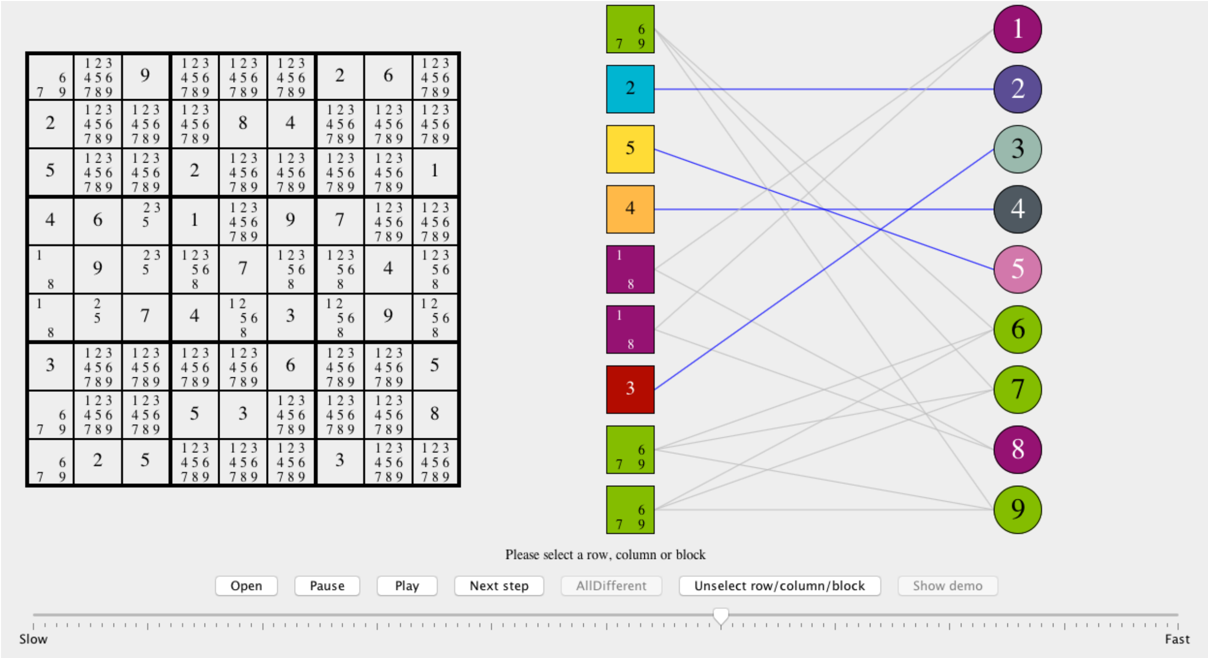



**Fig. B.2:** State of the program after running the demo

As seen in Fig. B.1, the all-different algorithm performs changes only to the selected 9 cells inside a row, column or 3x3 sub-grid.

The end result of following the above steps can be seen in Fig. B.2. This state mimics the human thinking when solving a Sudoku. In the 1st column there are two cells with a domain of $1, 8$, but we don't know yet which cell will take which digit. What we do know though, is that the digits $1, 8$ will be distributed in that two particular cells, therefore in the last step, after running the all-different algorithm on the 1st column we can observe that the digits $1, 8$ disappear from the domains of the rest of the cells in the selection.

# Bibliography

[1] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.