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

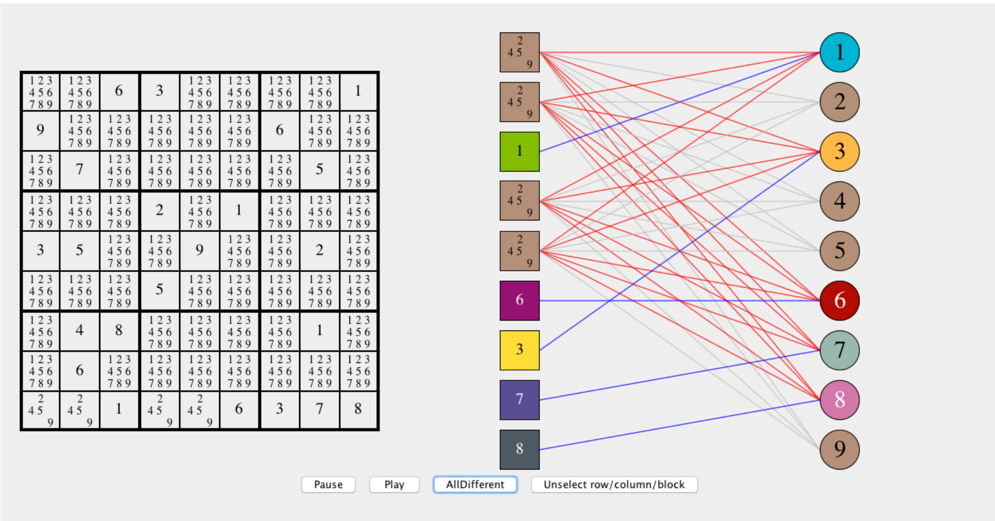


Fig. 2.1: The state of the program after the all-different algorithm finishes on a row, in this case, the last one.

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, Graph R)
begin
    Global int n ← |V(G)|
    Global pred ← new int[n]
    Global visited ← new boolean[n]
    Global Q ← new Queue()
    while bfs(G) do
        update(R)

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

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

int minCost()
begin
    int minCost ← Integer.MAX_VALUE
    int v ← n − 1
    while pred[v] ≠ 1 do
        minCost ← min(minCost, R[pred[v]][v])
        v ← pred[v]
    return

int min(int a, int b)
begin
    if a ≤ b then return a
    return b
```

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

```
¡¡¡¡¡¡¡ Updated upstream void Tarjan(integer A[][], int n)
======= ¡¡¡¡¡¡¡ HEAD void Tarjan(Graph G)
======= void Tarjan(integer A[][], int n)
¿¿¿¿¿¿¿ origin/master ¿¿¿¿¿¿¿ Stashed changes 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)
        if low[v] < min then min ← low[v]
    if min < low[u] then
        low[u] ← min
        return
    integer v
    repeat
        v ← pop(S)
        id[v] ← components
        low[v] ← |V(G)|
    until v ≠ u
    components ← components + 1
```

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

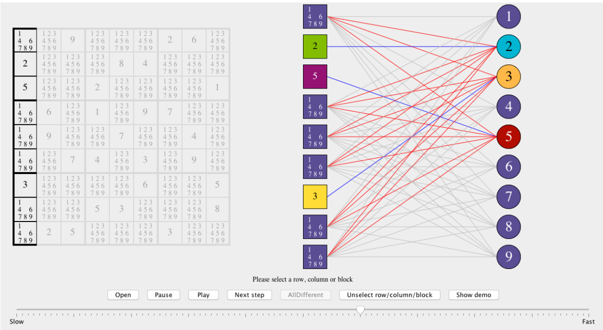
Proof of concept

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 $lockedset.txt$.

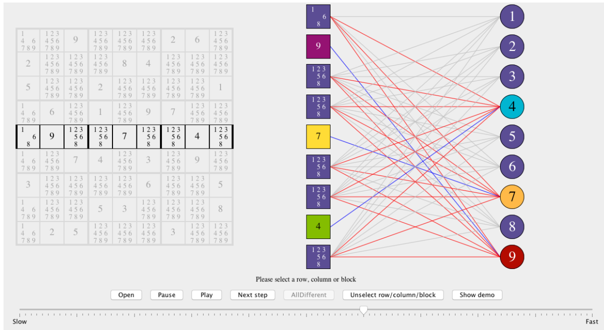
The sequence is as follows:

```
> 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
```

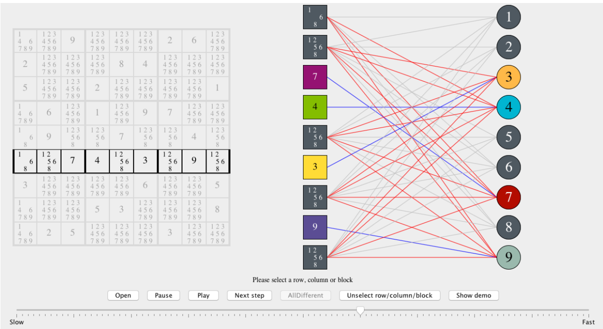
Alternatively, the user can press the $ShowDemo$ button to run the same steps.



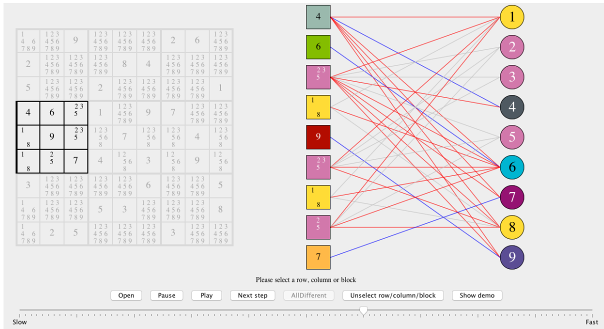
(a) 1st column after running the all-different algorithm



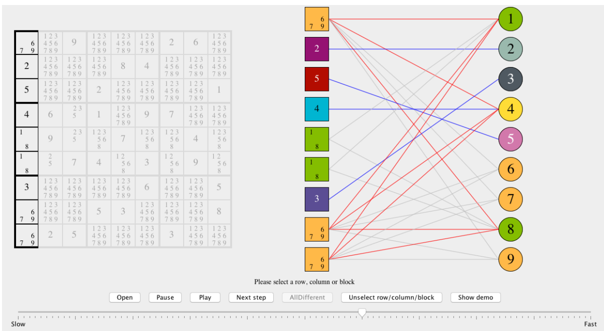
(b) 5th row after running the all-different algorithm



(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

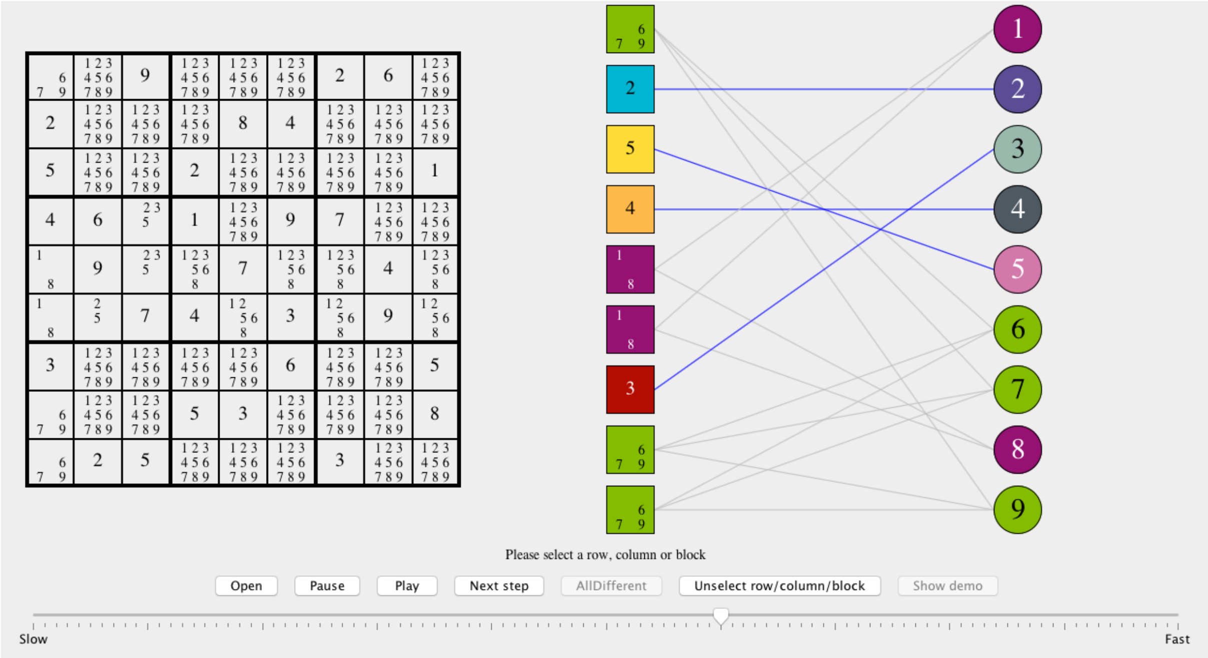


Fig. B.2: State of the program after running the demo

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.