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

Teaching constraint programming, need for visualisations

1.2 Background

1.3 Sudoku

Sudoku is a puzzle

1.4 Constraint Programming

Every Constraint Satisfaction Problem (CSP) has variables with domains and constraints associated to them. To start solving Sudoku puzzles, we have to come up with a model representation of the problem. For the purpose of this exercise, there will be 81 variables representing the 9x9 cells in the Sudoku grid, each with an initial domain of integers from 1 to 9. As all the initial puzzles start with a partially completed grid, values that are known will be reflected as variables with an initial domain of one digit, the known value. Such variables will therefore keep their initial domain throughout the algorithm, as no other values are present in their domain.

The constraints for a Sudoku puzzle can be expressed in natural language as such: digits may not appear twice in the same row, column or 3x3 subregions. The rules of the puzzle highlight the need of thinking in terms of 9 rows, 9 columns and 9 subregions, each of them having digits from 1 to 9 with no repetition. We can create 36 not equals constraints between the 9 variables of each row, column, subregions times 27 such regions for a total of 972 constraints between two variables. This would be highly inefficient, hence the need for a better approach. A better approach is to use 27 all-different constraints, one for each row, column and subregions. Each constraint contains 9 variables that should take different values. As each variable is part of a row, column and subregion at the same time, it will therefore be involved in 3 constraints.

Initial puzzles are always partially completed with enough values that by the end of the game there will be a single unique solution to the puzzle.

We get to the solution by running the alldifferent algorithm on rows, columns and subregions to remove values from the domain of the variables. Initially, the known values given at the start of the puzzle will be removed from the domains of possible values in the variables that are part of the same row, column or subregion. As the algorithm progresses, we will eventually remove enough values from the domain of a variable until we are left with a single value that represents the correct answer for that position. Once a value is found, there is a waterfall-like effect across the puzzle as we propagate the knowledge across the grid. The propagation takes place across all the variables that are in the same row, column and subregion with the one just solved. This will keep the 3 constraints in a consistent state, as all the variables will have valid remaining domains not containing any already solved values.

The algorithm is intelligent in a sense that it mimics the thinking of humans when solving a Sudoku puzzle. At some stages of the game we have enough information to know that two particular values will fill two cells, but no information about which value corresponds to which cell. This problem is what makes some Sudoku puzzles so hard to solve.

Humans and the alldifferent algorithm are smart enough to use this information to their advantage and remove the two values from the domains of the adjacent variables. By using this technique, we can ensure the progress of the algorithm towards a solution, even though we still dont know which two variables take which two values. It is good to note that although humans may choose a guessing approach to continue towards a solution, the alldifferent algorithm performs no guessing and works only with the information about the current state of the puzzle.

1.5 The Alldifferent Constraint

Review Regins paper and explain how constraint works [2]

Example of its usage and how it works (maybe 5 variables with domains)

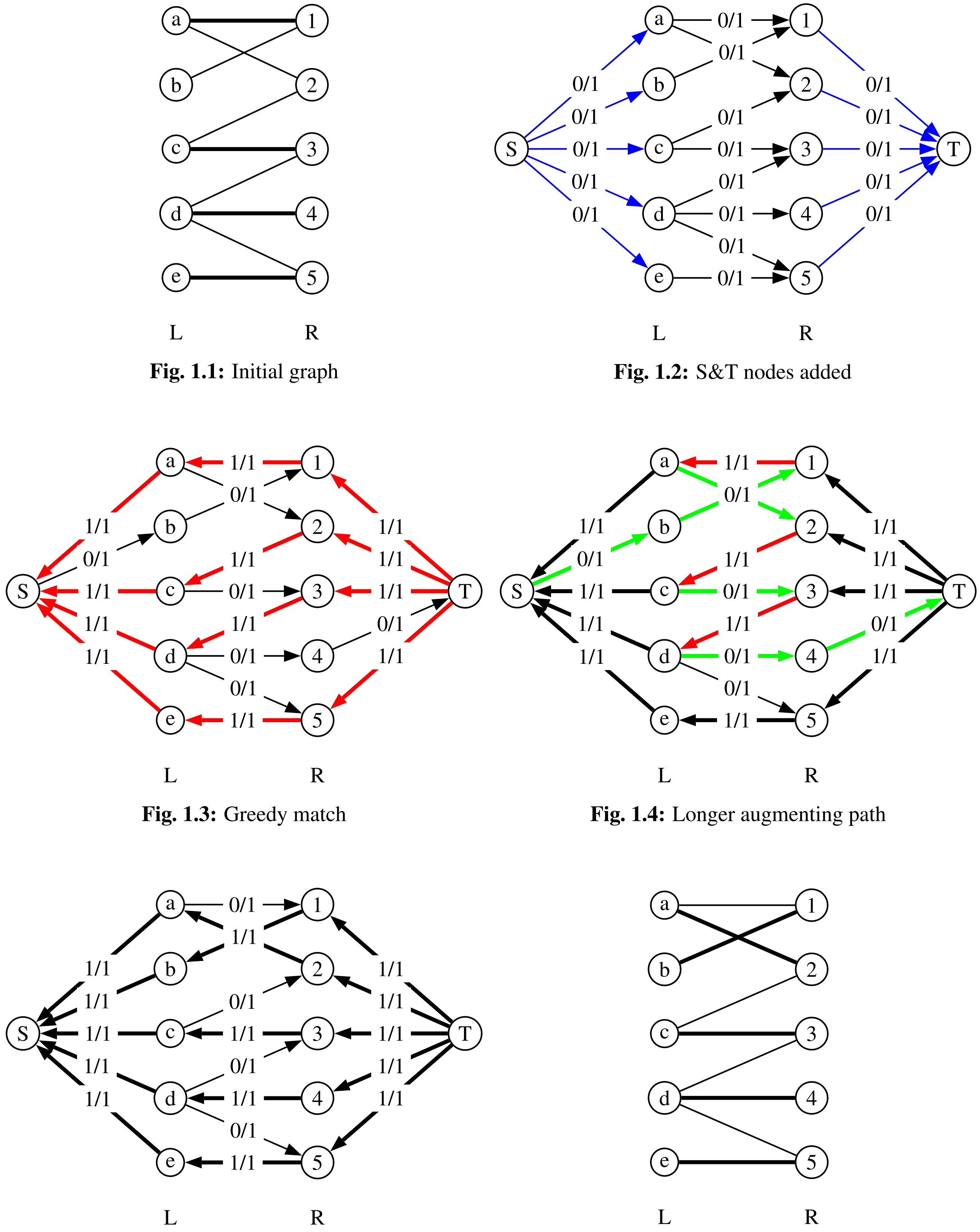


Fig. 1.1: Initial graph

Fig. 1.2: S&T nodes added

Fig. 1.3: Greedy match

Fig. 1.4: Longer augmenting path

Fig. 1.5: New matching

Fig. 1.6: Maximum matching

Fig. 1.7: FordFulkerson algorithm

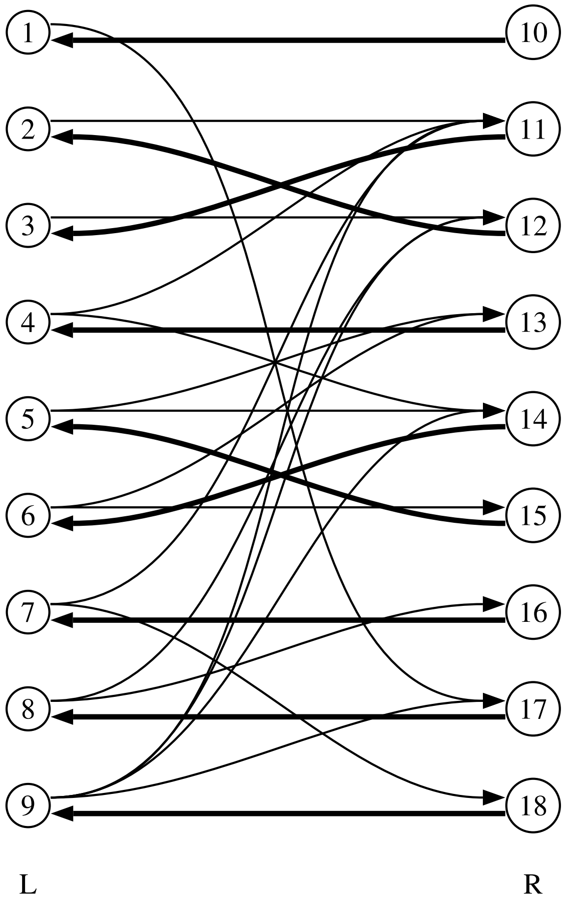


Fig. 1.8: Initial graph

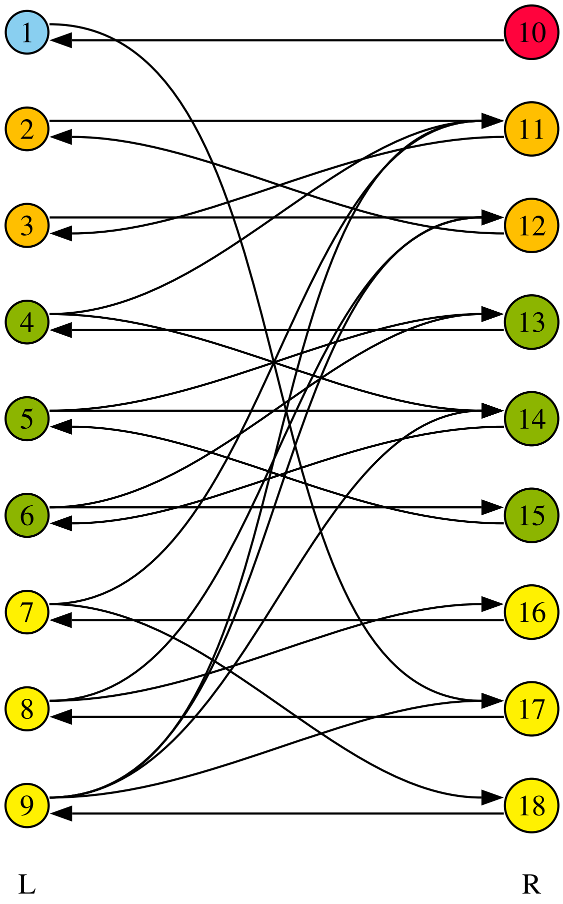


Fig. 1.9: Maximum matching

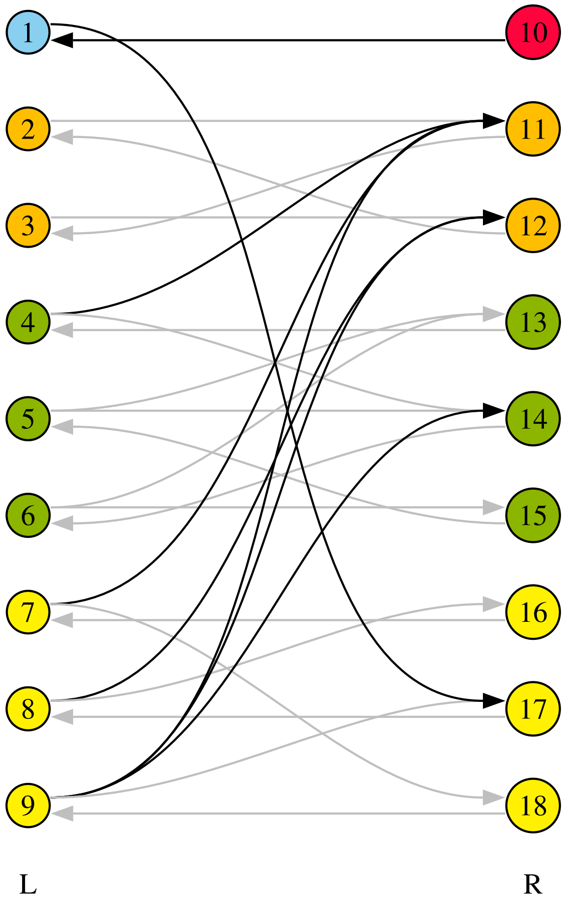


Fig. 1.10: Initial graph

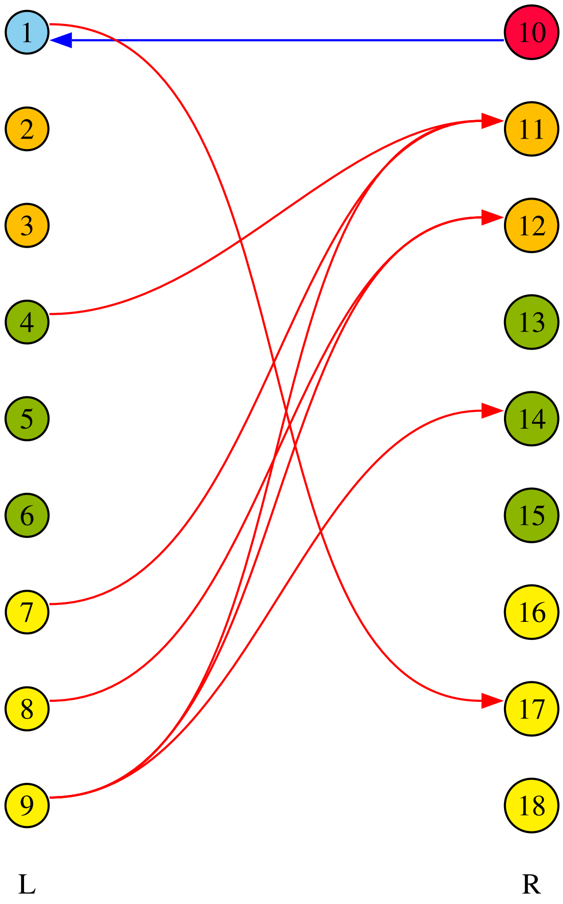


Fig. 1.11: Maximum matching

Fig. 1.12: FordFulkerson algorithm

Chapter 2

Algorithms for AllDiff

2.1 Ford Fulkerson

A flow represents a directed graph with special nodes called the source and the sink. Source nodes have outgoing edges to inner nodes, while sink nodes have incoming edges from inner nodes of the graph. The problem requires sending flows, for example water through a pipe, from the source to the sink, while conserving the equilibrium.

Flows need to get from the source to the sink by following the directed edges that often have different maximum capacity constraints, such as the diameter of a water pipe. Given a flow f in graph G, we build a residual directed graph G_f. Forward edges in the graph havent reached their maximum capacity, while backward edges reached their upper bound.

Ford-Fulkerson algorithm uses the residual graph to achieve a maximum flow in the graph. The algorithm looks for augmenting paths from source S to sink T in a breadth first-search manner. Once a path p is found, we send the weight that will make the bottleneck edge in p fill its maximum capacity. Then, we add (subtract) the weight of each forward (backward) edge in p. Forward edges that reach their maximum capacity turn into backward edges. This results in flow f that has a greater value than before. Berges theorem guarantees that once all the augmenting paths have been successively visited in the algorithm, the end result is a maximum flow of the graph [**?**].

The maximum matching problem is a special case of the maximum flow problem. For the purpose of our example, there will be only one source S that has outgoing edges to the nodes in the left partition L of the bipartite graph and one sink T with incoming edges from the right partition R of the graph. Nodes in L are connected through edges to nodes in R. A matching TODO. A maximum matching TODO. To find a maximum matching for a Graph G, we add a source and sink nodes and set all edges to a maximum capacity of one. We then run the Ford-Fulkerson algorithm on the graph, by finding all the augmenting paths from free nodes in L to free nodes in R.

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

Ford Fulkerson's algorithm for finding a maximum flow in a graph is shown in algorithm 1. [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.

Algorithm 1: Ford Fulkerson

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

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

void $update()$
begin
    int $f \leftarrow minCost()$
    int $v \leftarrow sink$
    while $pred[v] \neq -1$ do
        int $u \leftarrow pred[v]$
        $capacity[u][v] \leftarrow capacity[u][v] - f$
        $capacity[v][u] \leftarrow capacity[v][u] + f$

int $minCost()$
begin
    int $minCost \leftarrow \infty$
    int $v \leftarrow sink$
    while $pred[v] \neq -1$ do
        $minCost \leftarrow minimum(minCost, capacity[pred[v]][v])$
        $v \leftarrow pred[v]$
    return $minCost$
```

2.2 Tarjan algorithm

node	pre	low	stack	new component	color
1	1	1	[1]		🟦
17	2	2	[1, 17]		🟨
8	3	2	[1, 17, 8]		🟨
12	4	4	[1, 17, 8, 12]		🟧
2	5	4	[1, 17, 8, 12, 2]		🟧
11	6	4	[1, 17, 8, 12, 2, 11]		🟧
3	7	4	[1, 17, 8, 12, 2, 11, 3]		🟧
			[1, 17, 8]	[3, 11, 2, 12]	
16	8	2	[1, 17, 8, 16]		🟨
7	9	2	[1, 17, 8, 16, 7]		🟨
18	10	2	[1, 17, 8, 16, 7, 18]		🟨
9	11	2	[1, 17, 8, 16, 7, 18, 9]		🟨
14	12	12	[1, 17, 8, 16, 7, 18, 9, 14]		🟩
6	13	12	[1, 17, 8, 16, 7, 18, 9, 14, 6]		🟩
13	14	12	[1, 17, 8, 16, 7, 18, 9, 14, 6, 13]		🟩
4	15	12	[1, 17, 8, 16, 7, 18, 9, 14, 6, 13, 4]		🟩
15	16	12	[1, 17, 8, 16, 7, 18, 9, 14, 6, 13, 4, 15]		🟩
5	17	12	[1, 17, 8, 16, 7, 18, 9, 14, 6, 13, 4, 15, 5]		🟩
			[1, 17, 8, 16, 7, 18, 9]	[5, 15, 4, 13, 6, 14]	
			[1]	[9, 18, 7, 16, 8, 17]	
			[]	[1]	
10	18	18	[10]		🟥
			[]	[10]	

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) [3]

Tarjan's algorithm for finding strongly connected components is shown in algorithm 2. The $Tarjan$ proceduce 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.

Algorithm 2: Tarjan's strongly connected components

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

void $dfs$(int $u$, Graph $G$)
begin
    $push(u, S)$
    $stacked[u] \leftarrow true$
    $low[u] \leftarrow pre$
    $pre \leftarrow pre + 1$
    int $min \leftarrow low[u]$
    for $v \in N(u, G)$ do
        if $\neg stacked[v]$ then $dfs(v, G)$
        $min \leftarrow minimum(low[v], min)$
    $low[u] \leftarrow minimum(low[u], min)$
    integer $v$
    repeat
        $v \leftarrow pop(S)$
        $id[v] \leftarrow components$
        $low[v] \leftarrow n$
    until $v \neq u$
    $components \leftarrow components + 1$
```

Chapter 3

AllDiff Demo

implementation and usage

Explain how demo is used, what it shows, etc (NOT implementation)

Chapter 4

Implementation

The Graphical User Interface (GUI) is inside a JLayeredPane which is a special swing component that allows components inside it to overlap according to their specified depth. In my implementation, the backmost layer contains a JPanel responsible for the drawn edges in the graph. The layer above contains the rest of the GUI represented by the Sudoku puzzle on the left hand side of the screen, and the vertices of the graph on the right hand side of the screen. By using a layered panel, the edges of the graph can be drawn behind the vertices.

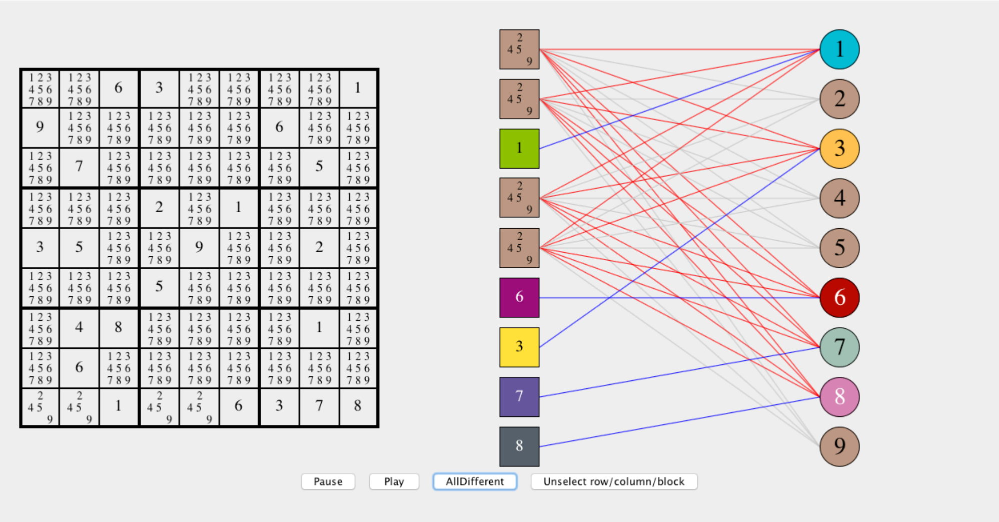
The Sudoku Grid present on the left hand side of the screen is made up of 9 by 9 Sudoku Cells positioned at specific locations. When a new Sudoku Cell instance in made, its ith row and jth column number are passed as arguments to the constructor. Using the information related to its location in the grid, it is now possible to render the border of each cell. Initially all the top, left, bottom and right borders have a width of 1px. To represent the 3 by 3 sub-grids, specific borders are set to have a width of 2px. In the end, the 9 by 9 grid is given an overall thick border by specifying a width of 4px to the cells that are in the first and last columns and rows. All Sudoku Cells contain a JLabel with text retrieved from the Model denoting which possible values are still available for that particular cell.

Not visible to the human eye, a new Sudoku grid instance is drawn on top of the existing one, this time with all borders set to a width of 1px. The Sudoku Cells inside this second grid contain the same information about the state of the puzzle retrieved from the Model. Additional information such as a custom background will be applied to these cells as the algorithm progresses. Cells inside the second grid become visible once the user makes a selection. Once visible, the cells inside the selection are animated into their position inside the graph on the right hand side of the screen. When the user performs deselects his choice, the nice cells are animated back into their original position in the grid and become invisible again.

Particular attention was given to make the software thread safe. It is crucial that the application logic doesnt make the Graphical User Interface unresponsive. To achieve this, the program will run on two threads, one main thread responsible with all the processing need to be done by the program, and the Event Dispatching Thread.

The Event Dispatching Thread (EDT) is a background thread used to invoke Swing methods that change the GUI and listen to events associated to components. Most Swing components are not thread safe, therefore bugs such as race conditions could unexpectedly arise if it was to perform the actions from normal threads. The EDT acts as a queue of events that are performed sequentially.

Throughout the application, Timer objects are used to schedule repeated actions such as the fading out of the components or movement of the cells. The model contains a two-dimensional ArrayList of Timer objects that acts as a queue of animations. Timers on the same level start running at the same time until they all finish. Once all the timers on a level finish, the next level in the two-dimensional ArrayList is started, allowing us to queue a movement of a row after all the cells finish fading out.



1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	6	3	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
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 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	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	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
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	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	5	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	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	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 2 3 4 5 6 7 8 9	4	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	1	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	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
2 4 5 9	2 4 5 9	1	2 4 5 9	2 4 5 9	6	3	7	8

Fig. 4.1: The state of the program after the all-different algorithm finishes on a row, in this case, the last one.

Chapter 5

Conclusion and Future Work

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

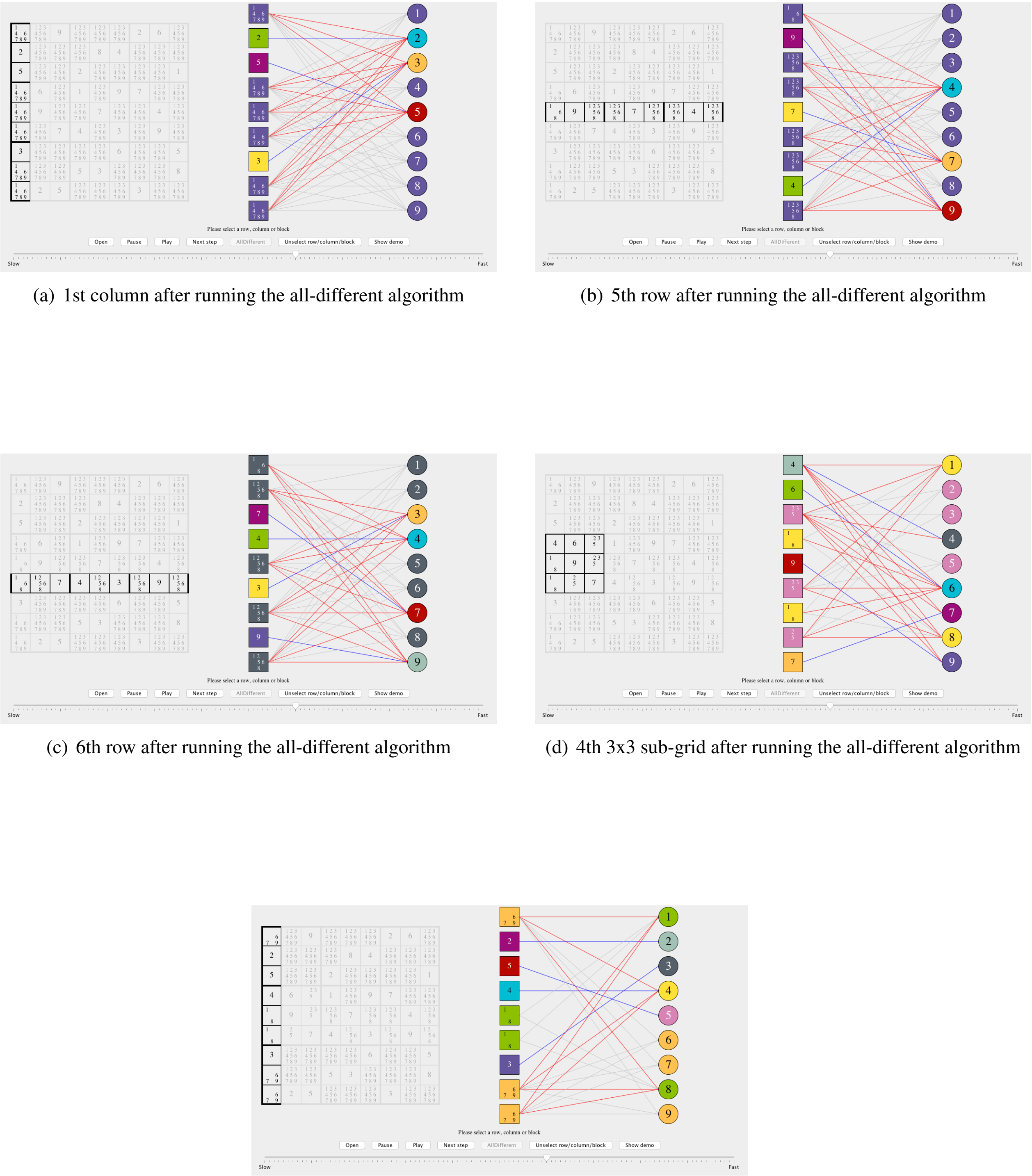
Proof of concept

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.


Please select a row, column or block
Open
Pause
Play
Next step
AllDifferent
Unselect row/column/block
Show demo
Slow
Fast

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

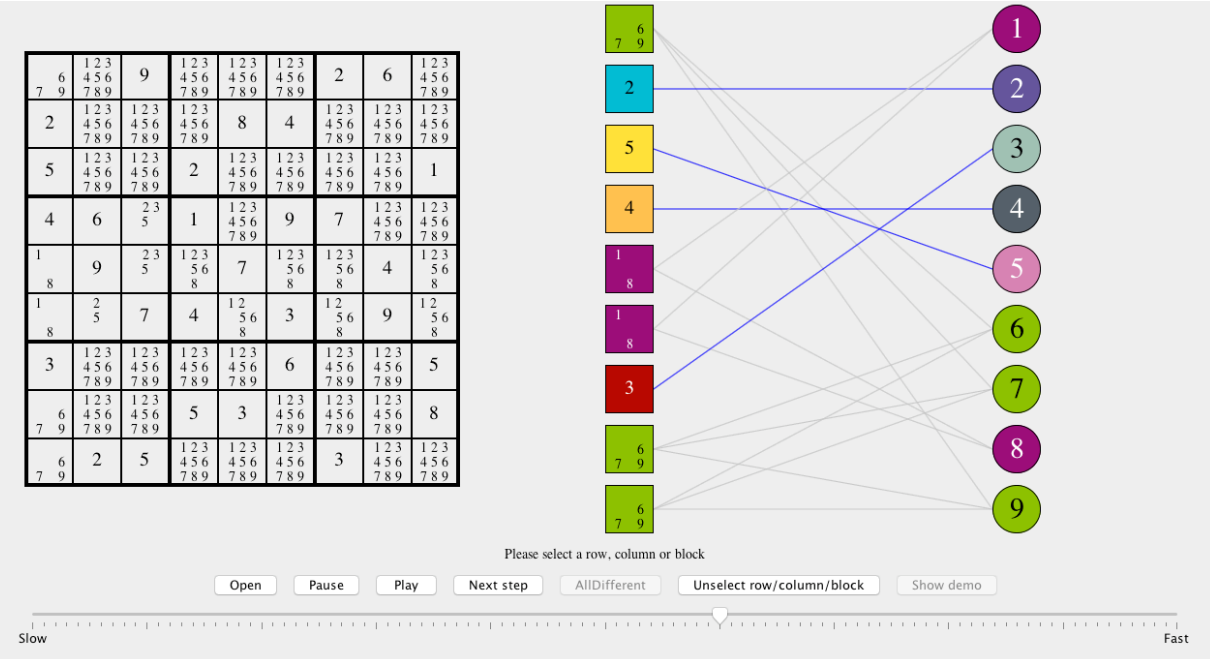
(b) 5th row after running the all-different algorithm

(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



6 7 9	1 2 3 4 5 6 7 8 9	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	2	6	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	8	4	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	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	1
4	6	2 3 5	1	1 2 3 4 5 6 7 8 9	9	7	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9
1 8	9	2 3 5	1 2 3 5 6 8	7	1 2 3 5 6 8	1 2 3 5 6 8	4	1 2 3 5 6 8
1 8	2 5	7	4	1 2 5 6 8	3	1 2 5 6 8	9	1 2 5 6 8
3	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	5
6 7 9	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	5	3	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	8
6 7 9	2	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	3	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9

Please select a row, column or block

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] Lester R Ford and Delbert R Fulkerson. Maximal flow through a network. *Canadian journal of Mathematics*, 8(3):399–404, 1956.

[2] Jean-Charles Régin. A filtering algorithm for constraints of difference in csps. In *AAAI*, volume 94, pages 362–367, 1994.

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