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

Constraint Programming (CP) is a technique originating from Artificial Intelligence used to solve hard real life combinatorial problems.

Constraints can be found everywhere in our daily life, and represent mathematical abstractions of the dependencies in the physical world. More formally, a constraint is a logical relationship among a number of variables, each with its own domain. The constraint therefore imposes restrictions on the values the variables can take. It is good to note that the values of the variables are not always numeric, a heterogeneous constraint would require a word (string) to have a specific length (numeric). The constraints are declarative as they only specify a relationship that must hold between variables and do not provide an algorithm to enforce the relationship.

Constraint Programming is the area of Computing Science that solves problems by specifying a list of constraints and then finding the solutions that satisfy all the constraints. A Constraint Satisfaction Problem (CSP) can be defined as a triple $\mathcal{P} =< X, D, C >$, where X represents a set of n variables $X =< x_1, x_2, ..., x_n >$, D represents the n domains associated to the n variables $D =< D_1, D_2, ..., D_n >$ such that $x_i \in D_i$, and finally C represents a set of t constraints $C =< C_1, C_2, ..., C_t >$. A constraint C_i restricts the values one or more variables can simultaneously take.

A solution for a CSP $\mathcal{P}$ is a set of values $S =< s_1, s_2, ..., s_n >$, such that each variable gets assigned a value from its domain and all the constraints are satisfied at once. The set of all the solutions to the problem is noted

as $sol(\mathcal{P})$ for a CSP that has multiple solutions. One may also be interested in finding an optimal solution to the problem, one that for example minimizes the path of a traveling salesman. When $sol(\mathcal{P})$ is equal to the empty set, it means that the original CSP is unsatisfiable. This simple representation can be used to model complex real world problems such as planning, scheduling, timetabling, supply chain management and more.

There are two strategies for solving CSPs: inference and search. If all the variables in the problem have finite domains, this means the search space for a solution is finite and represented by all the possible combinations of the values of the variables. Although one can, in theory, enumerate all the possible combinations, we use inference and search to reduce the search space. Inference techniques remove large subspaces of the search tree through local constraint propagation on the basis that no solutions may be found using a particular value from the domain of a variable. Search techniques, such as backtracking, are used to systematically explore the search space, usually with the use of some heuristics, and reduce the search space whenever a single failure is found. Both strategies are frequently used together and work because a solution to a CSP must have all its constraints satisfied, therefore a local inconsistency on a subset of the variables will not result in a solution.

Backtracking search algorithms look for solutions to a CSP by traversing the search tree in a depth-first search manner. At each node of the search tree, a variable is assigned a value out of its domain, and the node gets extended with branches for the remaining values in the variables domain that need to be considered later, as they could be part of a solution. For example, consider a simple CSP problem with three variable x, y and z. Suppose that the search algorithm already assigned a value to the variable x out of its domain, and it reaches a node representing the variable y with the domain $D_y = \{1, 2, 3\}$. The backtracking search algorithm will generate three branches, each one of them representing the variable y getting instantiated to a value from its domain. For every value assigned to the variable y, all the constraints tied to this variable and the previously visited variables are checked to see if they still hold. When the constraints are not satisfied by a selected value for a variable, the algorithm proceeds in assigning and testing the rest of the values in the domain of y. Once all the values from the domain of the variable y have been tried, the algorithm will backtrack to the previous variable, x in this case, and assign it the next value from its domain. Note that every time backtracking occurs, subtrees of the search tree will no longer be generated and visited since they will not contain any solutions to the CSP. In the problem mentioned above, the subtree for the values of z is no longer generated and tested as the partial solution made out of some specific values for x and y already violated the constraints of the problem. A complete solution is found once all the variables are assigned values that are allowed by the constraints. If the user is interested in only one solution to the problem, the search will terminate, otherwise the algorithm will continue to explore the search tree for other solutions. In the case the problem has no solutions, the algorithm will terminate after visiting all the possibilities in the search tree.

The efficiency of the backtracking search algorithm can be improved by performing constraint propagation to maintain consistency between the values of the variables and their associated constraints. Every time a variable is assigned a value from its domain, constraint propagation is made on the constraints the variable was part of. Propagation will check all the variables that were in a constraining relationship with the original variable and will remove values from their domain that are inconsistent with the problem requirements. By reducing the domains of the variables, the search tree is pruned and the efficiency of the search increases. The domain of a variable may be reduced to the empty set after propagation, which means that there is no value left for that variable that will satisfy the constraints and therefore backtracking should be initiated. On other occasions, the domain of a variable will be reduced to a single value, which means that the value for the variable is now known and removing the need to search through values that would not lead to a solution. The backtracking search algorithm will resume when the recursive propagation results in no changes to the domains of the variables. Information from constraint propagation is usually incorporated in the search algorithm in form of variable and value ordering heuristics that improve the efficiency of the search even more.

Variable and value ordering heuristics are ways to improve the backtracking search algorithm by prioritizing the instantiation of some variables and by choosing some values from their domain first. Ordering heuristics are essential to effectively solve hard combinatorial problems. The most popular heuristic is the fail-first approach,

something Haralick and Elliot described as To succeed, try first where you are most likely to fail. [**?**]. The principle is to prioritize the instantiation of variables with the smallest domains of values. The assumption behind this is that branches in the search tree that have no solutions will be discovered earlier and pruned to bring a noticeable increase in efficiency later in the search. Consider the worst-case scenario when performing a backtracking search on a problem that has no solutions, the search tree will be much smaller as earlier failure is encouraged, making it faster to prove that a problem has no solutions. In the case of problems that start off with equally sized domains, the fail-first principle can still be applied by prioritizing the instantiation of the variables that take part in many constraints. Once a variable was selected for instantiation, some values from its domain could lead to a solution faster than others. The order values are chosen for assignment to the variables is only relevant when trying to find a single solution to the problem. The reason behind this is that trying to find all the solutions to a problem or proving that there are no solutions would have to traverse all the search space anyway. A heuristic to prioritize values from a domain is to assess the likelihood each value has in getting closer to a solution, a choice that will have minimum impact on the domain of other variables after propagation, therefore increasing the chances of finding a solution faster.

Some solutions to a problem are better than others. Consider the Traveling Salesman Problem (TSP), where the shortest possible path between some cities should be found. The problem is NP-hard as the running time to find the solution increases exponentially with the number of cities. The general approach to the optimization problem is to introduce a constraint that minimizes a variable representing an objective, the length of the path in the case of the TSP. Finding an optimal solution is often hard, and proving optimality of a solution might be impossible for complex combinatorial problems.

Constraint Programming can be regarded as a form of declarative programming, where a user specifies a problem in terms of its variables, their domains and the constraints that should be satisfied. The computer then solves the given problem using a range of techniques that are general approaches to problem solving which can be applied to all the problems that can be modeled as constraint satisfaction problems.

The original exercise of finding an algorithm to solve a specific problem is now turned into an exercise of finding the best way to model the problem as a CSP, which can later be solved by the computer. The model can make the difference between efficient solving of a problem or a problem that cannot be solved due to its combinatorial complexity. There are usually many ways in which one can model the same problem, but the efficiency of the solution could be different. Consider a problem in which a number of g guests should be offered a seat at one of the t tables, with the constraint that couples should sit together. One could decide to represent each of the g guests as a variable associated with an initial domain representing a listing of the t tables, from which a choice for the table should be made based on the constraints. Another way to model the same problem is to have a number of t variables representing the tables, each with an initial domain representing a listing of all the guests. The search for a solution would reduce the domains of the variables in both models until there is only a single table in the domains of the variables (for the first model where guests are variables), or in the case of the second model, until all the variables representing tables have the cardinality of their domains reduced to their sitting capacity. Although both models are logically equivalent, one may perform worse than the other because of how its represented in the memory of the computer.

When modeling a problem, it is important to avoid symmetries in the form of equivalent solutions. Symmetries represent waste computational resources during the backtracking search that is used to explore symmetrical assignments. One should avoid such symmetries by adding new constraints that will filter out the symmetrical solutions. Such constraints are problem specific. If it is necessary to find all the solutions to a problem, including the equivalent ones, one can easily perform permutations of the variables outside of the search space once the main solutions are found, thus avoiding exploring similar branches in the search tree. In the example problem mentioned above, there are symmetries in the solutions, as the same group of guests will be seated at different tables throughout the solutions. As we know tables are the same in real life and can be easily swapped without violating the constraints, such computations should not be made to improve efficiency of the search.

Real world problems are often large and over-constrained, which may raise difficulties in modeling them. Most of the time there are conflicting objectives such as trying to maximize the efficiency of a production facility, while minimizing the operational costs. Tradeoffs between the constraints should be made to come up with a solution to the problem. Problem decomposition should also used to break a large real world scenario into smaller problems that are easier to solve. Finally, one should consider which model is easier to upgrade when more constraints should be added.

1.5 The Alldifferent Constraint

1.5.1 Review of Jean-Charles Regin's paper

Review Regins paper and explain how constraint works [2]

1.5.2 Usage example

To demonstrate the usage and effects of the alldifferent constraint, we are going to use a simple example. The CSP problem will be modeled in Java and will make use of Choco3, a powerful free and open-source Java library used to solve Constraint Satisfaction Problems. The implamentation of the alldifferent constraint in the Choco3 library is based on Jean-Charles Regin's paper[2] reviewed above.

The problem has three variables x, y, z with the following initial domains: $x = \{1, 2\}$, $y = \{1, 2\}$ and $z = \{2, 3\}$. The constraint that should be satisfied by all the solutions of the problem is that x, y and z should all be different. The listing TODO shows how the problem is modeled and implemented using the Choco3 library. Line 1 defines a Choco3 solver object, that will store the variables, their domains and constraints associated to them. Lines 3 to 5 define the three integer variables in the problem and their associated initial domains. The reference to the $solver$ object makes sure the variables are added to the model. Line 7 introduces the $alldifferent$ constraint between the three variables. The constraint is then posted to the model and the CSP is now fully modeled.

Before trying to find a solution, a call to the $propagate$ method is made on the model on line 10. The propagation algorithm will make all the constraints consistent by looping thorugh all the constraints in the model (only one in this case) with the aim of reducing the domains of the variables by removing values that will never part of a solution. When a value is removed from the domain of a variable, this might trigger a recursive propagation call on all its dependant variables to make the constraints consistent. The recursive propagation calls end when the problem is in a consistent state. It is good to note that propagation will preserve all the original solutions to the problem, while greatly reducing the search space.

The $ContradictionException$ on line 11 is an exception that is raised by Choco3 whenever a variable is left with an empty domain or is instantiated to a value that is out of its initial domain. This usually happens in overconstrained problems, where there are no values in the domains of the variables that satisfy all the constraints at once. In such cases, there would be no valid solutions to the problem. This is not the case with the example showcased here, as there are three variables with three different values in total.

Once the propagation is finished, a call is made on line 15 to output the updated domains of the variables. The updated domains after the call to the $propagation$ method are as follows: $x = \{1, 2\}$, $y = \{1, 2\}$ and $z = \{3\}$. The variable z lost the value 2 from its domain, while to the domains of x and y stayed the same. There are two reasons behind removing the value 2 from the domain of variable z. Firstly, there would no solutions with z taking the value of 2 as this would leave either x or y with an empty domain after one of them takes the value of 1. Secondly, two variables (x and y) have the same domain of $\{1, 2\}$, therefore a solution that is consistent with

```
1  Solver solver = new Solver("AllDifferentExample");
2
3  IntVar x = VF.enumerated("x", new int[]{1, 2}, solver);
4  IntVar y = VF.enumerated("y", new int[]{1, 2}, solver);
5  IntVar z = VF.enumerated("z", new int[]{2, 3}, solver);
6
7  solver.post(ICF.alldifferent(new IntVar[]{x, y, z}));
8
9  try{
10     solver.propagate();
11 } catch (ContradictionException e){
12     // Handle the exception
13 }
14
15 System.out.println(x + "\n" + y + "\n" + z);
16
17 if(solver.findSolution()){
18     do{
19         int x_val = x.getValue();
20         int y_val = y.getValue();
21         int z_val = z.getValue();
22
23         System.out.println(x_val + "; " + y_val + "; " + z_val);
24     } while(solver.nextSolution());
25 }
```

Listing 1.1: Usage of the alldifferent constraint in Choco3

the alldifferent constraint will have x taking the value of 1 and y taking the value of 2 (or vice-versa), making it impossible for z to take the value of 2. Since we know that the values 1 and 2 will be attributed to x and y, the value 2 is removed from the domain of z to reflect this change and reduce the search space.

Lines 17 to 25 will iterate over the two solutions of the problem and print the values for the variables. The solutions to the problem are $S_1 = \{1, 2, 3\}$ and $S_2 = \{2, 1, 3\}$ and they both contain values for the three variables that satisfy the alldifferent constraint.

The solutions are

```
x: 1 y: 2 z: 3
x: 2 y: 1 z: 3
solutions:2
```

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).

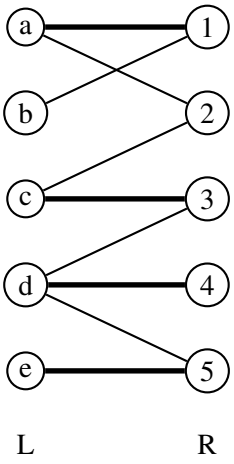


Fig. 2.1: Initial graph

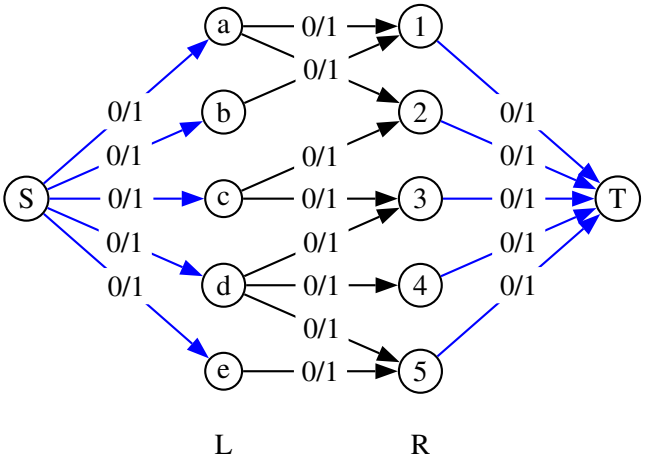


Fig. 2.2: S&T nodes added

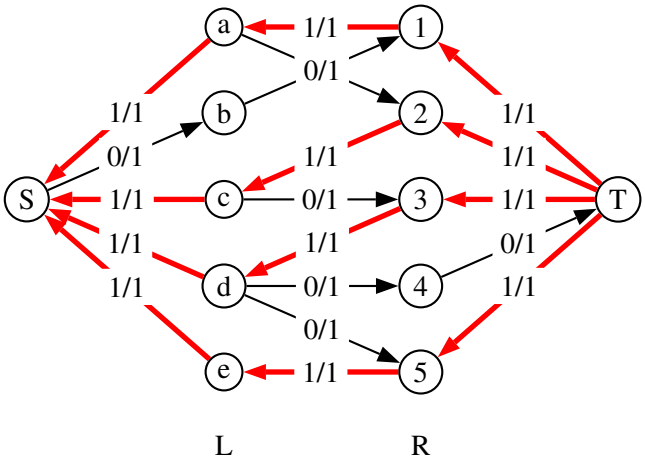


Fig. 2.3: Greedy match

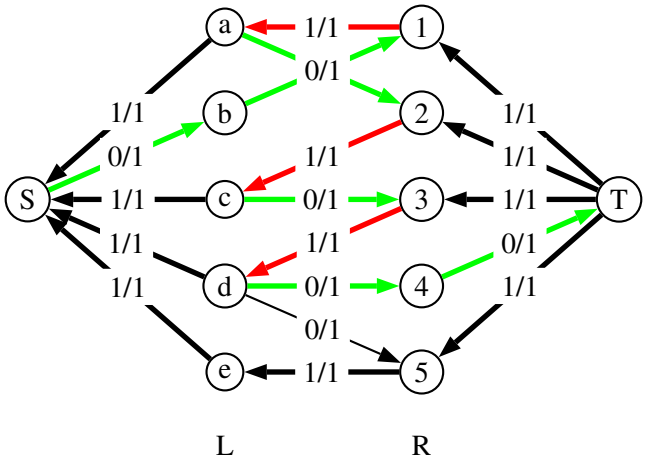


Fig. 2.4: Longer augmenting path

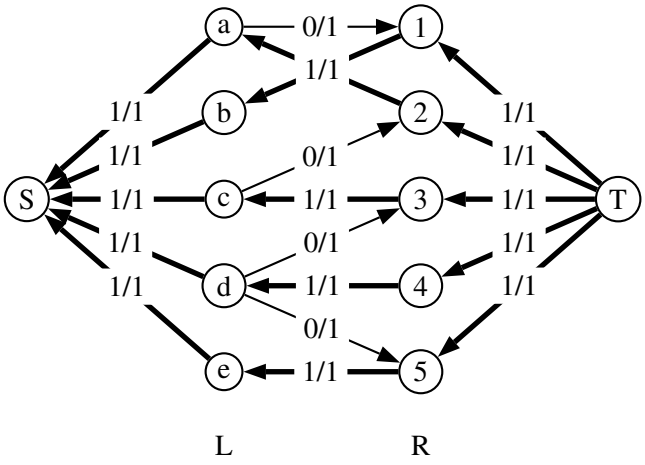


Fig. 2.5: New matching

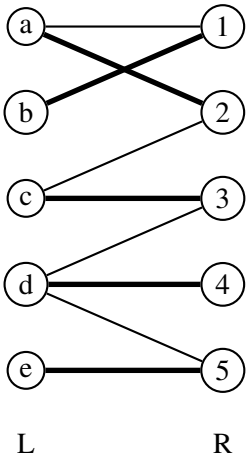


Fig. 2.6: Maximum matching

Fig. 2.7: FordFulkerson algorithm

Algorithm 1: Ford Fulkerson

```
1  void FordFulkerson(Graph G)
2  begin
3      Global int capacity[][] ← getCapacity(G)
4      Global int n ← |V(G)|
5      Global int source ← 0
6      Global int sink ← n − 1
7      Global pred ← new int[n]
8      Global visited ← new boolean[n]
9      Global Q ← new Queue()
10     while bfs(G) do
11         update()

12 boolean bfs(Graph G)
13 begin
14     clear(Q)
15     fill(pred, −1)
16     fill(visited, false)
17     enqueue(source, Q)
18     visited[source] ← true
19     while (¬isEmpty(Q)) do
20         int v ← dequeue(Q)
21         if v = sink then return true
22         for w ← 0 to n do
23             if ¬visited[w] and capacity[v][w] > 0 then
24                 pred[w] ← v
25                 enqueue(w, Q)
26                 visited[w] ← true
27     return false

28 void update()
29 begin
30     int f ← minCost()
31     int v ← sink
32     while pred[v] ≠ −1 do
33         int u ← pred[v]
34         capacity[u][v] ← capacity[u][v] − f
35         capacity[v][u] ← capacity[v][u] + f
36         v ← u

37 int minCost()
38 begin
39     int minCost ← ∞
40     int v ← sink
41     while pred[v] ≠ −1 do
42         minCost ← minimum(minCost, capacity[pred[v]][v])
43         v ← pred[v]
44     return minCost
```

Ford Fulkerson's algorithm for finding a maximum flow in a graph is shown in algorithm 1. [1]. The $Ford\ Fulkerson$ 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 is stored 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 references to 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 vertices in the order they are visited.

The algorithm consists of repeated calls to the breadth-first search procedure (line 10). In line 11, we call the update procedure to update the capacities of the edges after finding augmenting paths.

The $breadth\text{-}first\ search$ 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 index value of w's predecessor 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 $false$ when no more augmenting paths are found.

The $update$ procedure at line 28 is used to update the capacities of the edges in the graph to reflect the updated flow. Line 30 declares an integer f used to store the amount of flow we are adding along the found path. The value for f is calculated in the $minCost$ procedure at line 37. Line 31 declares an integer v, initially representing the sink which is the last vertex in the graph. 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 the index of a vertex that is the predecessor of vertex v along the path. Lines 34 and 35 updates the flow along the edge from u to v and v to u by the found difference. Finally, v is updated on line 36 to take the value of u such that the while loop continues to change the capacities of the remaining edges along the path.

The $minCost$ procedure at line 37 is used to find the amount of flow to be sent along the new augmenting path from the sink to the source. The value is equal to the minimum capacity that is still available along the edges of the path. Line 39 declares an integer $minCost$ used to represent the current value of the minimum cost, initialized to a big value. Line 40 declares an integer variable v initially used to store a reference to the sink vertex. Lines 41 to 43 contain a loop over the edges along the path, starting from the sink toward the source. Line 42 updates the value of $minCost$ if the current edge in the loop has a smaller capacity left than what was previously discovered. The $minimum$ procedure returns the minimum of two given integers. Line 43 updates the value of integer v to its predecessor so the loop can continue along the path until it reaches the $source$. At the end of the $minCost$ procedure, we return the value of the currently found $minCost$ variable.

2.2 Tarjan algorithm

A strongly connected component SCC of a directed graph G is a maximal set of vertices such that every vertex is reachable from every other vertex by following the directed edges.

There are four kinds of edges in a depth-first traversal: tree edges, forward edges, back edges and cross edges. Tree edges are edges between a vertex and vertices that were not previously visited. Forward edges are from ancestor vertices to descendant vertices that were already visited during the search. Back edges link descendant vertices back to already visited ancestor vertices. Lastly, cross edges represent edges between two vertices that are neither an ancestor nor a descendant for each other, such as the edges between different components of the graph.

For each vertex in a graph G, the algorithm keeps track of two properties. The first property we remember is the $index$ of the node, its consecutive order of discovery during the depth-first search, a value that will remain constant throughout the algorithm. The second property associated with all the nodes is named low and keeps track of the lowest (oldest) ancestor reachable from their position in the graph. As each node of the graph gets discovered in a depth-first search manner, the initial value that low gets assigned is the same as the $order$ the node was discovered. The value for low may get changed during the search if an ancestor for the node is discovered by following a back edge. After the recursive depth-first search finishes visiting all of a nodes neighbours, the nodes value for low gets updated to the lowest index of its oldest reachable ancestor. If the value for low of a node stays the same even after visiting all of its neighbours, then it means it is the root of a strongly connected component or its an individual vertex that will be a component on its own. [3]

For the purpose of finding SCCs, every time a node is discovered in the search, the node is put on a stack S. After returning from the recursive DFS calls on all of the adjacent neighbours of vertex, if both $index$ and low have the same value, it means weve finished discovering a SCC. The vertices in the newly discovered strongly connected component will be popped from the stack S until we reach the current vertex representing the root of the SCC. Vertices that are part of other SCCs are still left on the stack as they have different roots represented by the value of low.

Figure 2.8 TODO shows a bipartite directed graph with 18 vertices. There are 9 backward edges that represent the maximum matching found using the Ford-Fulkerson algorithm in section TODO. The rest of the possible edges are assigned a forward direction. This graph is similar to the graphs that will be generated to represent a row, column or one of the nine sub-grids of the Sudoku puzzle. For example, the left partition of the graph will contain the nine variables of a Sudoku row, and the right partition will contain the digits from 1 to 9. The edges will then link each variable of the row to values that are still available in their domain.

The Table 2.1 TODO contains the output after running the Tarjans algorithm on the graph in figure 2.8 TODO. As previously discussed, vertices are visited in a depth-first search manner, in this case starting from the first vertex. Each node has two properties associated to it: its order of discovery or $inded$, and its oldest reachable ancestor. Initially, both properties take the value of the index, but as the algorithm progresses the values for low may become lower after visiting all of the neighbours of the node and finding the oldest (lowest) reachable ancestor. All vertices are pushed on a stack in the order they are visited. Once a back edge is found in the graph, vertices that are currently on the stack get popped including the root of the newly discovered SCC. Note that vertices may still remain on the stack as they will be part of different components that have other roots. As each vertex is made part of a component, its value for low gets updated to a large value to mark it as already part of a component such that it will not become the root of another component in the future.

The graph in figure 2.9 TODO shows the 5 strongly connected components identified after running Tarjans algorithm on the initial graph in figure 2.8 TODO. Vertices that have the same background are part of the same component; meaning that one vertex can reach every other vertex in the component by following the directed edges. Note that some vertices make a component on their own as there is no way to reach them again in a cycle.

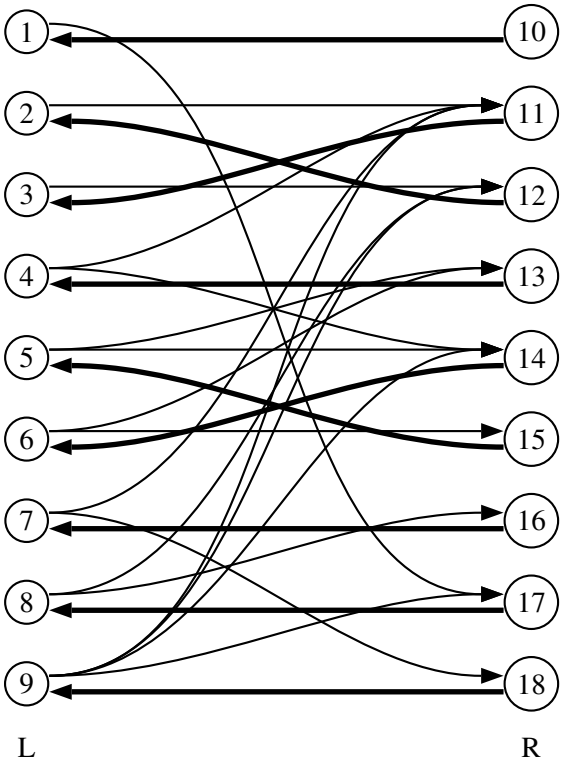


Fig. 2.8: Initial graph

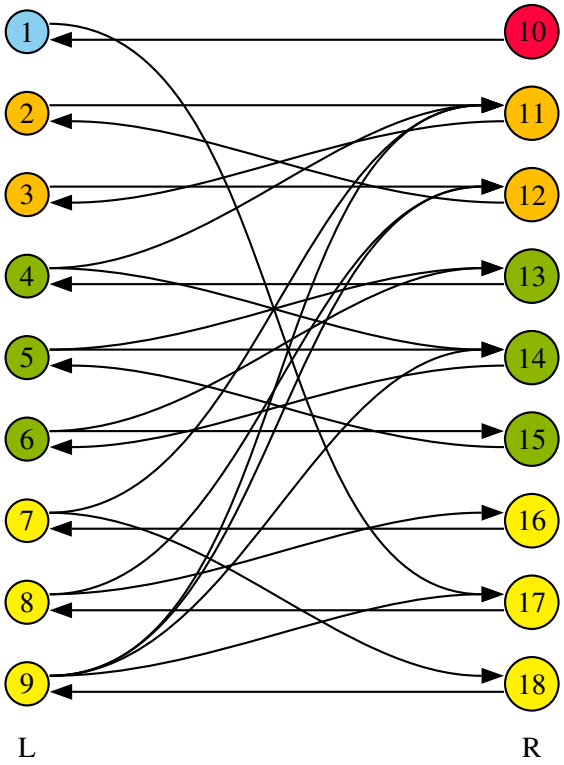


Fig. 2.9: Maximum matching TODO RENAME

node	index	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]	

Table 2.1: Steps Tarjan's algorithm TODO

Algorithm 2: Tarjan's strongly connected components

```
1  void Tarjan(Graph G)
2  begin
3      Global int index ← 0
4      Global int components ← 0
5      Global int n ← |V(G)|
6      Global S ← new Stack()
7      Global stacked ← new boolean[n]
8      Global id ← new int[n]
9      Global low ← new int[n]
10     for u ∈ V(G) do
11         if ¬stacked[u] then dfs(u, G)

12 void dfs(int u, Graph G)
13 begin
14     push(u, S)
15     stacked[u] ← true
16     low[u] ← index
17     index ← index + 1
18     int min ← low[u]
19     for v ∈ N(u, G) do
20         if ¬stacked[v] then dfs(v, G)
21         min ← minimum(low[v], min)
22     low[u] ← minimum(low[u], min)
23     integer v
24     repeat
25         v ← pop(S)
26         id[v] ← components
27         low[v] ← n
28     until v ≠ u
29     components ← components + 1
```

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 $index$ declared in line 3 stores the current order in the depth-first search, and gets incremented each time a vertex is discovered. 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 then updating the value in the $stacked$ array (line 15) to reflect the change. Line 16 sets the vertex's v value for low to $index$ as every vertex assumes to have

no ancestors until a back edge to one is found. In line 17, the value for $index$ is incremented by one to keep count of the current number of discovered vertices. Line 18 declares an integer min used to hold information about the oldest, lowest ancestor that will be discovered later in the search and is initialized to take the same value of the current vertex.

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 reaching the root of the component.

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

The final step in the all-different algorithm uses the knowledge gained after finding the strongly connected component in the graph using Tarjans algorithm. For this step, we are going to be interested only the edges that link two vertices in different strongly connected component. Figure 2.10 TODO shows a graph highlighting the edges we are interested in. It is important to remember that we chose a graph with 18 vertices to highlight how the all-different algorithm will run on the 9 variables of a Sudoku row. The variables of the Sudoku row are shown in the left L partition of the graph and are assigned letters from A-I, each having edges to vertices in the right R partition of the graph representing digits in the domain of the variable. The direction of the edges that are left indicate two actions. Forward edges (coloured in red) indicate that the digits from the right R partition should be removed from the domain of the variables in the L partition to achieve consistency TODO. Backward edges (coloured in blue) indicate finding the solution to a variable, in this case the first variable A of a selected Sudoku row will certainly take the value of 1 as all other options were removed from its domain.

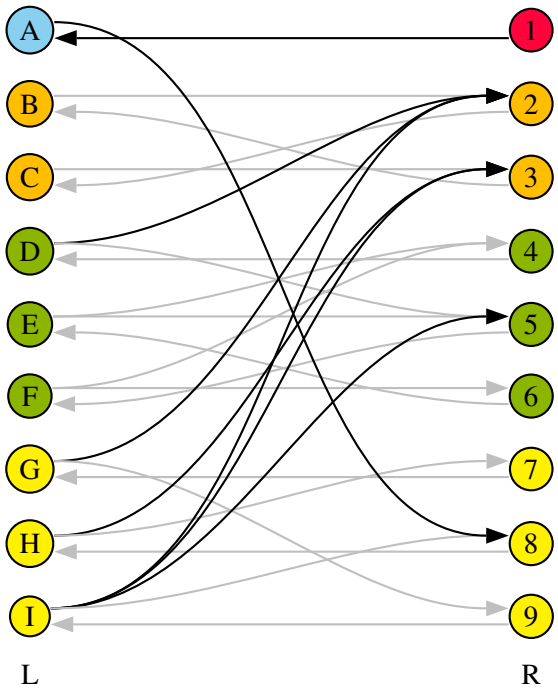


Fig. 2.10: Initial graph

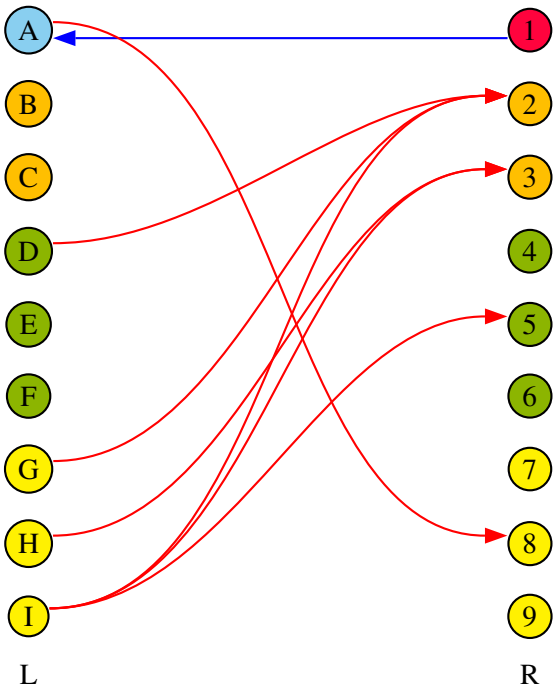


Fig. 2.11: Maximum matching

Fig. 2.12: Tarjan's algorithm next steps TODO: rename

18	23	23	245	456	456	279	378	23589

Fig. 2.13: A Sudoku row used to illustrate the alldifferent algorithm

18	23	23	245	456	456	279	378	23589

Fig. 2.14: Removal of values from the domain (in red) and assignments (in blue) in a Sudoku row after the alldifferent algorithm

Chapter 3

AllDiff Demo

implementation and usage

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

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.

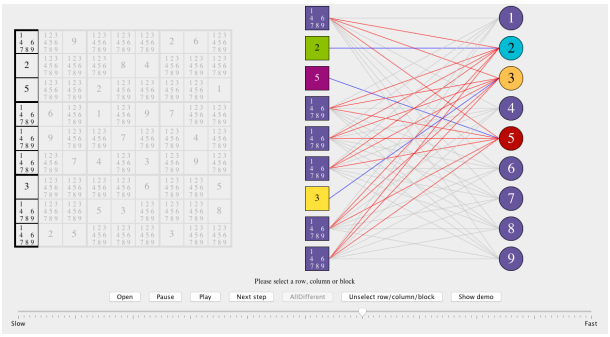
The constraints for a Sudoku puzzle can be expressed in natural language as such: digits may not appear twice in the same row, column or 3x3 subregions. The rules of the puzzle highlight the need of thinking in terms of 9 rows, 9 columns and 9 subregions, each of them having digits from 1 to 9 with no 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.

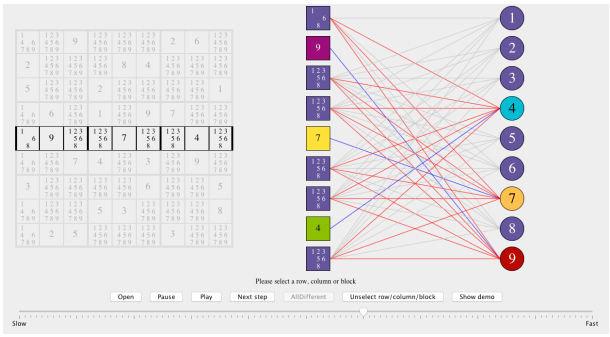
We get to the solution by running the alldifferent algorithm on rows, columns and subregions to remove values from the domain of the variables. Initially, the known values given at the start of the puzzle will be removed from the domains of possible values in the variables that are part of the same row, 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.

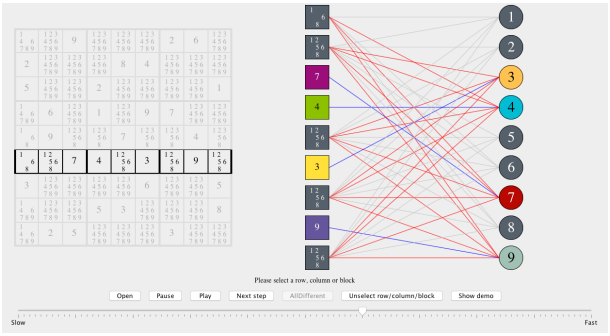
Humans and the alldifferent algorithm are smart enough to use this information to their advantage and remove the two values from the domains of the adjacent variables. By using this technique, we can ensure the progress of the algorithm towards a solution, even though we still dont know which two variables 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.



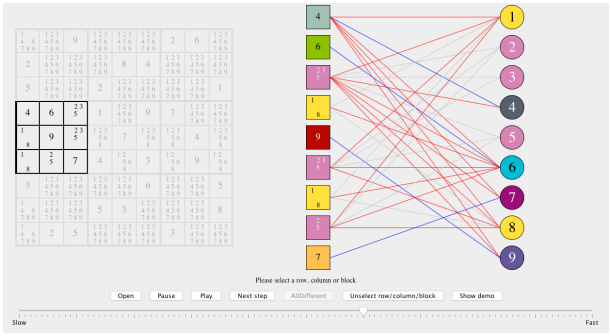
(a) 1st column after running the all-different algorithm



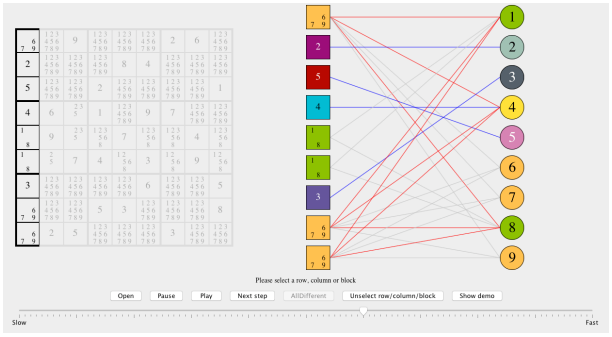
(b) 5th row after running the all-different algorithm



(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. 3.1: Steps in the demo

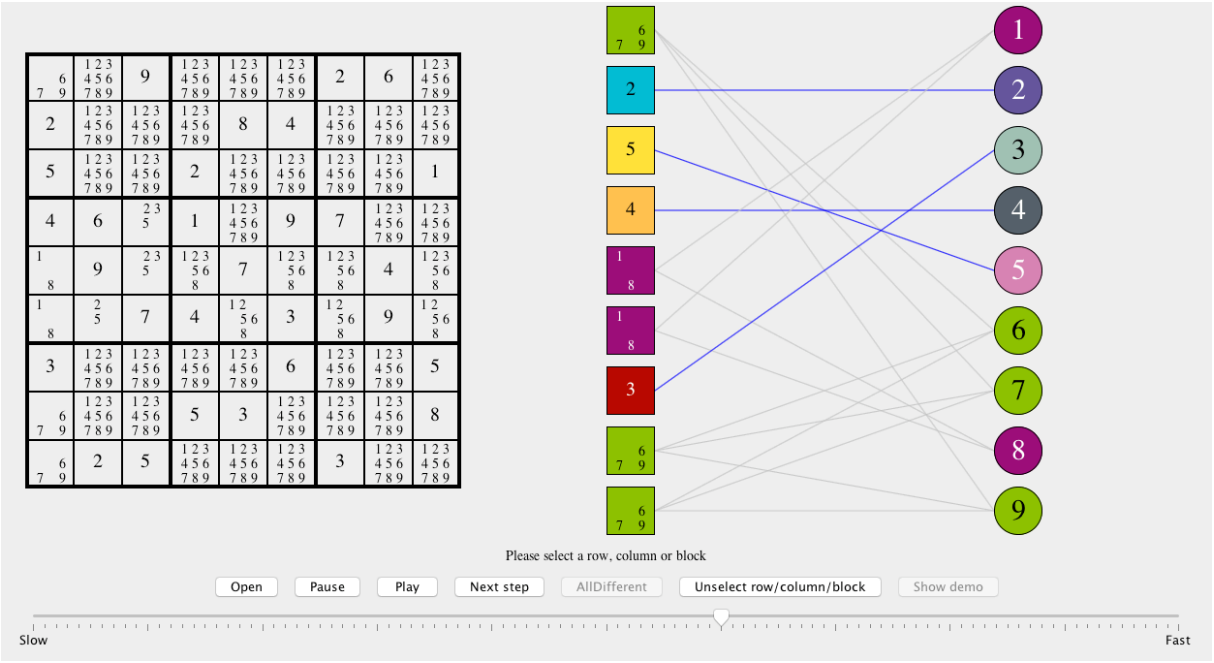


Fig. 3.2: State of the program after running the demo

As seen in Fig. 3.1, 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. 3.2. 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.

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.

Chapter 5

Conclusion and Future Work

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

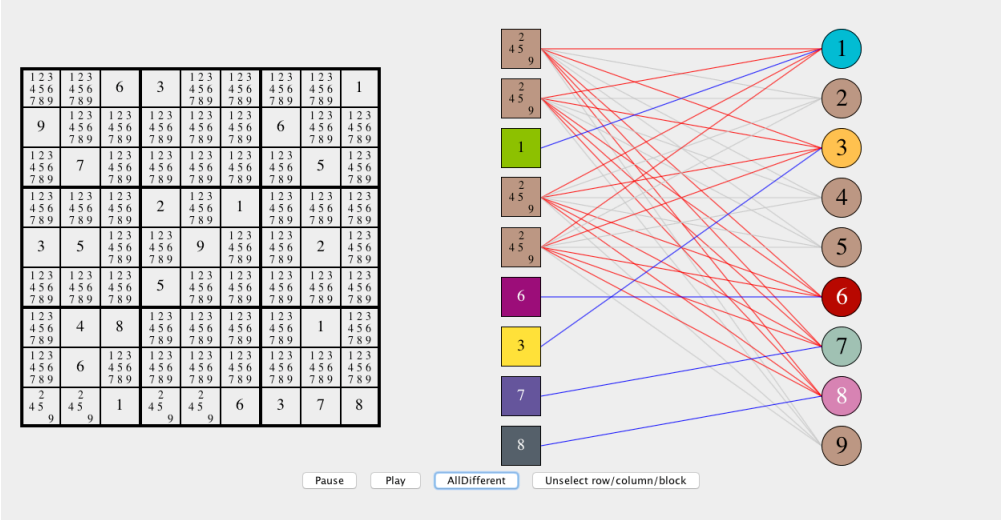


Fig. 5.1: The state of the program after the all-different algorithm finishes on a row, in this case, the last one.

Move this from here

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.

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.