Fall 2016 Comp 15

Independent Final Project Specification

Soduku Solver

Julie Jiang

Tufts University

November 11, 2016

1 Overview

This is a Soduku solver. Client will provide a sokudu puzzle in the form of a $n^2 \times n^2$ grid of integers, where n is a postive integer, with the number 0 serving as a placeholder for blank grid cells. Below is an example Soduku of size $3^2 \times 3^2 = 9 \times 9$ input on the left, and its solution output on the right:

```
0 0 3 0 2 0 6 0 0        4 8 3 9 2 1 6 5 7
9 0 0 3 0 5 0 0 1        9 6 7 3 4 5 8 2 1
0 0 1 8 0 6 4 0 0        2 5 1 8 7 6 4 9 3
0 0 8 1 0 2 9 0 0        5 4 8 1 3 2 9 7 6
7 0 0 0 0 0 0 0 8        7 2 9 5 6 4 1 3 8
0 0 6 7 0 8 2 0 0        1 3 6 7 9 8 2 4 5
0 0 2 6 0 9 5 0 0        3 7 2 6 8 9 5 1 4
8 0 0 2 0 3 0 0 9        8 1 4 2 5 4 7 6 9
0 0 5 0 1 0 3 0 0        6 9 5 4 1 7 3 8 2
```

2 Formal Definition

This is a constraint satisfaction problem. The objective is to populate a $n^2 \times n^2$ grid, for some $n \in \mathbb{Z}^+$ with numbers such that each row, column and the n^2 subgrids of size $n \times n$ all contain the numbers 1 to n^2 without repeat.

Let X be a set of $9 \times 9 = 81$ variables such that each variable X_{ij} corresponds to the cell (i, j) in the grid and has a nonempty domain $D(X_{ij}) \subset \{1, 2, ..., n\}$ of possible values. Additionally, let A be the assignment

Additionally, let A be the set of variables that have assigned values, and A' be the set of variables that don't have assigned values. We note that A' and A must be two disjoint sets such that $A' \cup A = X$. For the variables that are assigned a value from the very beginning (i.e. values that client assigned), their domains are singletons, or a set with exactly one element. These variables are automatically added to A.

Let C be a set of directed arcs. An arc $C(A, B)$ for some variables $A, B \in X$ is consistent if for every possible value $x \in D(A)$, there is some value $y \in D(B)$ that is consistent with A. For a typical soduku puzzle of size 9×9, each variable in the grid will have $8 + 8 + 6 = 22$ arcs (8 for its column, 8 for its row, and 6 for the remainder of cells in its subgrid).

3 Algorithms

All the pseudocodes below are taken from chapter 5 of *Artificial Intelligence: A Modern Approach*[2].

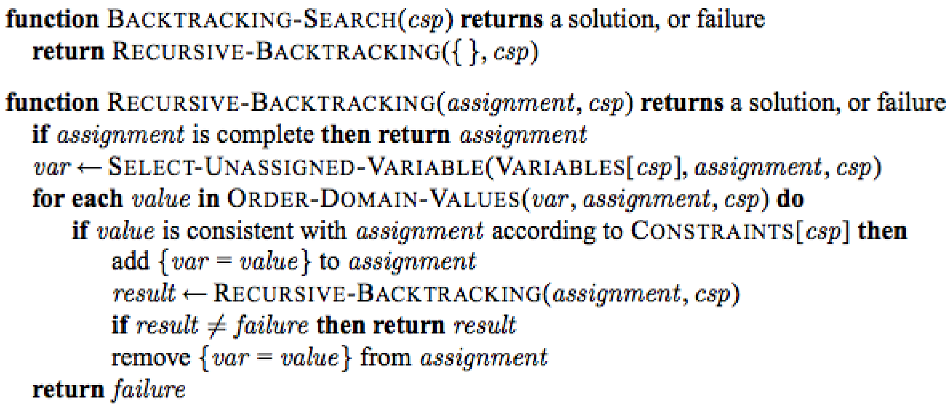
3.1 Preprocessing

1. For each variable in X, add it to set A if it has an assigned value, add it to A' otherwise.

2. For each assigned variable $X_{ij} \in A$, initialize its domain so that $D(X_{ij})$ is a singleton of its assigned value.
3. For each unassigned variable $X_{ij} \in A'$, initialize it's domain so that $D(X_{ij}) = \{1, 2, ..., n^2\}$.
4. For each unassigned varible $X_{ij} \in A'$, remove any value that appears in the domains of the assigned variables in column i, row j, or its $n \times n$ subgrid from its domain.

3.2 Backtracking Search

This search algorithm is a tweak on depth first search. In each iteration, choose a variable X_{ij} from the set of unassigned variables with the smallest number of legal values. This is called the *minimum remaining value* heuristic (see 3.3). This selected variable will then be temporarily assigned a value that rules out the fewest values in the remaining variables, or the *least constraining value*. Once a value has been assigned, enforce arc consistency (see 3.4) for all the variables in its row, column and subgrid. Then recursively repeat this step until either a solution is found or a solution cannot be found, for which we will backtrack.
The pseudocode below is a basic outline of this idea.



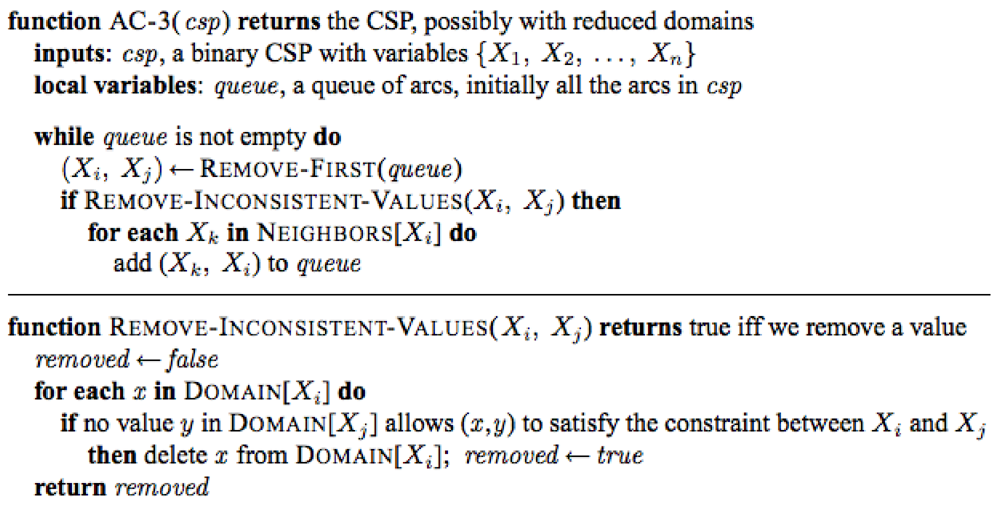
```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
      if value is consistent with assignment according to CONSTRAINTS[csp] then
          add {var = value} to assignment
          result ← RECURSIVE-BACKTRACKING(assignment, csp)
          if result ≠ failure then return result
          remove {var = value} from assignment
  return failure
```

3.3 Heap sort

The variables in the the set A' of unassigned variables are ordered according to the minimum remaining value heuristic. Thus, the set of unasigned variables A' will be stored in a priority queue, which will be implemented as a priority heap.

3.4 Check Arc Consistency

Given a variable, check that arc consistency satisfies for every unassigned variable that lies in its column, row, or subgrid. If any of these variables have to have inconsistent values removed because of the original assignment, check every unassigned variable that lies in its column, row, or subgrid, too. Repeat until there are no arcs left to check.



```
function AC-3(csp) returns the CSP, possibly with reduced domains
  inputs: csp, a binary CSP with variables $\{X_1, X_2, \ldots, X_n\}$
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    $(X_i, X_j) \leftarrow$ REMOVE-FIRST(queue)
    if REMOVE-INCONSISTENT-VALUES($X_i, X_j$) then
      for each $X_k$ in NEIGHBORS[$X_i$] do
        add $(X_k, X_i)$ to queue
```

```
function REMOVE-INCONSISTENT-VALUES($X_i, X_j$) returns true iff we remove a value
  removed $\leftarrow$ false
  for each $x$ in DOMAIN[$X_i$] do
    if no value $y$ in DOMAIN[$X_j$] allows $(x,y)$ to satisfy the constraint between $X_i$ and $X_j$
      then delete $x$ from DOMAIN[$X_i$]; removed $\leftarrow$ true
  return removed
```

4 Data Structures

There are several data structures that I will be using in this project. Most of them will serve as containers for a struct called `cell`. Each `cell` corresponds to a variable X_{ij}, so there are 81 `cells` in total.

```
struct Cell {
   int xcoord;
   int ycoord;
};
```

1. *Priority Queue.* In each iteration of backtracking search, a `cell` will be selected from the priority queue of `cells` that haven't been assigned a value based on the minimum remaining values heuristic. That is, the fewer the number of possible values a variable has, the higher its priority in the queue. To do this, I will store the `cells` in a priority queue in a tree that satisfy the min-heap condition.

2. *Stack.* During backtracking search, a stack is needed to assignments. That is, the value that has been assigned to each variable `cell`.

3. *Queue.* A queue is needed to hold `cells` that have not been checked when enforcing arc consistency.

4. *AVL tree.* Each `cell` contains a set of distinct values that it can take on. A set is implemented as an AVL tree, so that look up time is consistently $O(\log n)$

5. *Hash table.* The hash table will map a set of coordinates to a set of possible values.

5 Design

I will write one class called `Soduku` that takes a set of inputs from the main function and produces a set of solutions (or returns something that indicates failure, if no solutions can be found). Depending on how much leeway I can get with using the STL, I may also have to write one class for each of the data structure I described above.

Last but not least, I will write a main file that contains the main user interface functions.

It goes without saying that I will also write a lot of `cpp` files for testing.

`sodukuSolver.cpp`		
`Soduku.h`	`Soduku.cpp`	
`sodukuGenerator.cpp`	`sodukuGenerator.h`	`testSoduku.cpp`
`PriorityQueue.h`	`testPriorityQueue.cpp`	
`Stack.h`	`testStack.cpp`	
`Queue.h`	`testQueue.cpp`	
`AVLTree.h`	`testAVLTree.cpp`	
`HashTable.h`	`testHashTable.cpp`	

6 Testing

Testing the data structures will be very much like the tests I have written for the data strucutre classes in previous homework – tedious to write but comprehensive in covering corner cases.

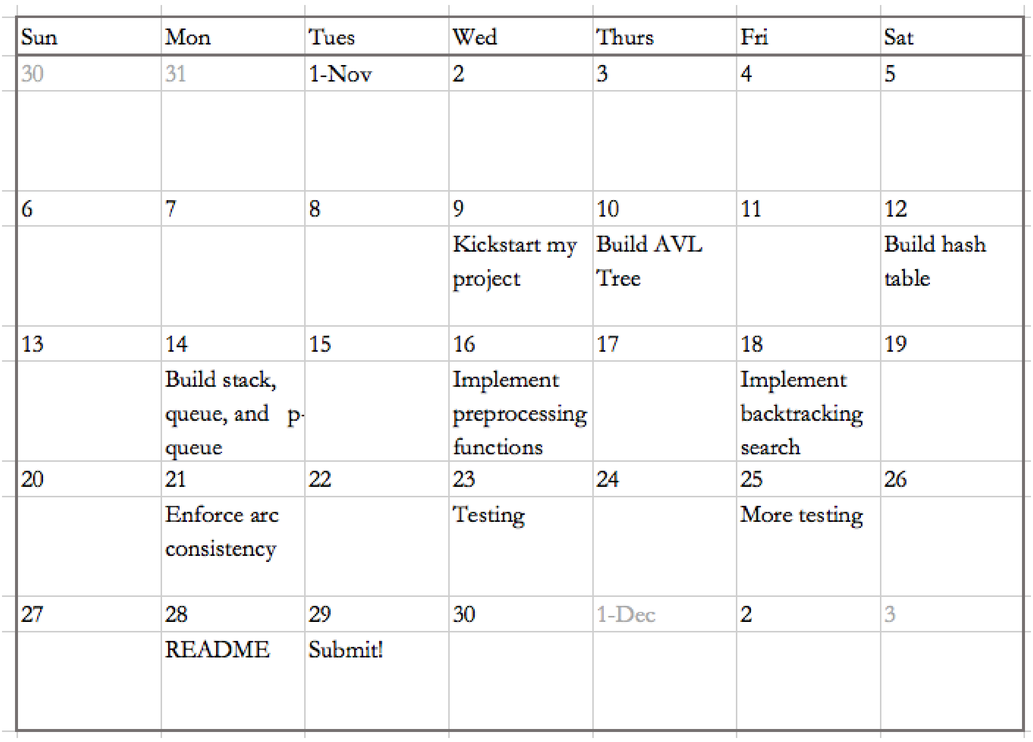
As for testing the soduku solver, I will write a `sodukuGenerator` to generate soduku puzzles of arbitrary size. Checking the validity of a soduku is fairly simple – just make sure that each number $1, 2, ..., n^2$ appears once and only once in every column, row, and $n \times n$ subgrid.

7 Goals

Project Minimum. Solve soduku puzzles of arbitrary size. *Stretch Goal.* Solve soduku puzzles of arbitrary size as fast as possible. *Stretchier Goal.* Come up with a better name for this project.

8 Execution Schedule

Below is my tentative execution plan at a glance:



Sun	Mon	Tues	Wed	Thurs	Fri	Sat
30	31	1-Nov	2	3	4	5
6	7	8	9	10	11	12
			Kickstart my project	Build AVL Tree		Build hash table
13	14	15	16	17	18	19
	Build stack, queue, and p-queue		Implement preprocessing functions		Implement backtracking search	
20	21	22	23	24	25	26
	Enforce arc consistency		Testing		More testing	
27	28	29	30	1-Dec	2	3
	README	Submit!				

References

[1] Klein, Dan and Peter Abbeel. “Constraint Satisfaction Problem”. *UC Berkeley CS188 Intro to AI.* http://ai.berkeley.edu/. Accessed 9 Nov 2016.

[2] Russell, Stuart and Peter Norvig.“Constraint Satisfaction Problem”. *Artificial Intelligence: A Modern Approach.* 3rd ed., Upper Saddle River: Prentence Hall. 2003