Objectives:

This note book is dedicated for students who are interested in Artificial intelligence field, specially in constraint satisfaction problems Algorithms topic. CSP algorithms are one of the most important subjects in AI that can be applied to solve large number of problems. CSP Many problems in AI can b e modeled as constraint satisfaction problems CSPs Hence the development of effective solution techniques for CSPs is an important research problem .

Examples and applications of CSPs can b e found in many areas such as resource allo catio n in scheduling temp oral reasoning natural language pro cessing query optimization in database etc ..

An example problem that can be solved by CSP :

Problem description:

The standard Sudoku puzzle consists of a nine by nine grid, broken into nine three by three boxes. Each of the eighty-one squares must be filled in with a number between one and nine. There are only three rules for filling in these numbers: each row, column and box must contain all of the numbers one through nine. Enough of the numbers are initially filled in to assure that the puzzle has exactly one solution.

– variables are cells– domain of each variable is {1,2,3,4,5,6,7,8,9}– constraints: rows, columns, boxes contain all different numbers

- **How many possible worlds are there?(say,53 empty cells)**

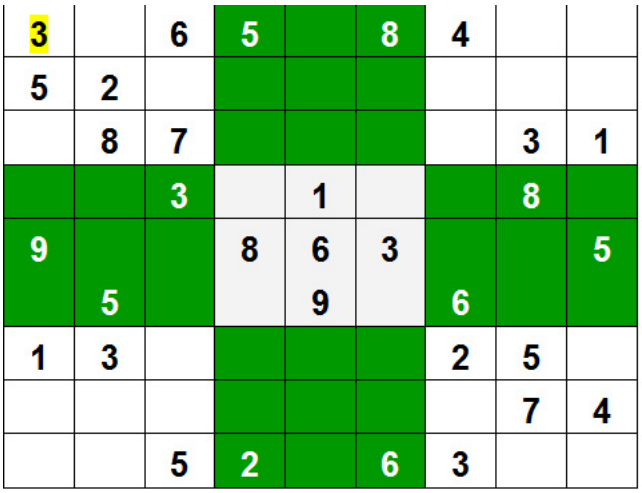
53*9 53^9 9^53

constraints: tasks can't be scheduled in the same location at the same time; certain tasks can't be scheduled in different locations at the same time; some tasks must come earlier than others; etc.

AC3

consistent if for each value x ∈ D X there exists a value y ∈ D y such that the assignments X = x and Y = y satisfy all binary constraints between X and Y. A CSP is arc consistent if all variable pairs are arc consistent. ... Simple consistency algorithm.

State space :



- State space:Sudoku is a classical problem to be modeled as a CSP. The cells of the board can be seen as variables, the numbers 1-9 are the domain and neighboring cells constrain each
- Goal State:value has been assigned to each variable without violating the constraints
- Heuristics function: this heuristic checks if the neighbors of an assigned variable are consistent with the assignment
- Action: all directions “his heuristic checks if the neighbors of an assigned variable are consistent with the assignment “
- Path cost function: the length of a path in meters.
- Solution of the problem: a path from the initial state to the goal state.


```
{
 "cells": [
  {
   "cell_type": "code",
   "execution_count": 5,
   "metadata": {},
   "outputs": [
```

```
    {
     "ename": "SyntaxError",
     "evalue": "invalid syntax (<ipython-input-5-dc2dd581e248>, line 1)",
     "output_type": "error",
     "traceback": [
      "\u001b[1;36m  File \u001b[1;32m\"<ipython-input-5-
dc2dd581e248>\"\u001b[1;36m, line \u001b[1;32m1\u001b[0m\n\u001b[1;33m    AC3
ALGORITHIM\u001b[0m\n\u001b[1;37m
^\u001b[0m\n\u001b[1;31mSyntaxError\u001b[0m\u001b[1;31m:\u001b[0m invalid
syntax\n"
     ]
    }
   ],
   "source": [
    "## AC3 ALGORITHIM\n",
    "AC-3 operates on constraints, variables, and the variables' domains (scopes). ...
AC-3 proceeds\n",
    "by examining the arcs between pairs of variables (x, y). It removes those values
from the domain of x which aren't\n",
    "consistent with the constraints between x and y.💢💯"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "metadata": {},
   "outputs": [],
   "source": [
    "# IMPLEMENTATION OF AC3 ALGORITHM"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": 2,
   "metadata": {},
   "outputs": [],
   "source": [
    "digits =  cols = \"123456789\"\n",
    "rows = \"ABCDEFGHI\"\n",
    "\n",
    "\n",
    "#FINDING THE CROSS PRODUCT OF TWO SETS \n",
    "def cross(A, B):\n",
    "\treturn [a + b for a in A for b in B]\n",
    "\n",
```

```
    "squares = cross(rows, cols)"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": 3,
   "metadata": {},
   "outputs": [],
   "source": [
    "class csp:\n",
    "\n",
    "#INITIALIZING THE CSP\n",
    "def __init__ (self, domain = digits, grid = \"\"):\n",
    "self.variables = squares\n",
    "self.domain = self.getDict(grid)\n",
    "self.values = self.getDict(grid)\n",
    "\n",
    "\n",
    "'''\n",
    "Unitlist consists of the 27 lists of peers \n",
    "Units is a dictionary consisting of the keys and the corresponding lists of peers \n",
    "Peers is a dictionary consisting of the 81 keys and the corresponding set of 20
peers \n",
    "Constraints denote the various all-different constraints between the variables \n",
    "'''\n",
    "\n",
    "self.unitlist = ([cross(rows, c) for c in cols] +\n",
    "            [cross(r, cols) for r in rows] +\n",
    "            [cross(rs, cs) for rs in ('ABC','DEF','GHI') for cs in ('123','456','789')])\n",
    "\n",
    "self.units = dict((s, [u for u in self.unitlist if s in u]) for s in squares)\n",
    "self.peers = dict((s, set(sum(self.units[s],[]))-set([s])) for s in squares)\n",
    "self.constraints = {(variable, peer) for variable in self.variables for peer in
self.peers[variable]}\n",
    "\n",
    "\n",
    "\n",
    "\n",
    "\n",
    "#GETTING THE STRING AS INPUT AND RETURNING THE CORRESPONDING
DICTIONARY\n",
    "def getDict(self, grid=\"\"):\n",
    "i = 0\n",
    "values = dict()\n",
    "for cell in self.variables:\n",
    "if grid[i]!='0':\n",
```

```
  "values[cell] = grid[i]\n",
  "else:\n",
  "values[cell] = digits\n",
  "i = i + 1\n",
  "return values"
 ]
},
{
 "cell_type": "code",
 "execution_count": 6,
 "metadata": {},
 "outputs": [],
 "source": [
  "# APPLYING AC3 ALGORITHM"
 ]
},
{
 "cell_type": "code",
 "execution_count": 4,
 "metadata": {},
 "outputs": [],
 "source": [
  "\n",
  "\n",
  "import queue\n",
  "\n",
  "\n",
  "#THE MAIN AC-3 ALGORITHM\n",
  "def AC3(csp):\n",
  "q = queue.Queue()\n",
  "\n",
  "for arc in csp.constraints:\n",
  "q.put(arc)\n",
  "\n",
  "i = 0\n",
  "while not q.empty():\n",
  "(Xi, Xj) = q.get()\n",
  "\n",
  "i = i + 1 \n",
  "\n",
  "if Revise(csp, Xi, Xj):\n",
  "if len(csp.values[Xi]) == 0:\n",
  "return False\n",
  "\n",
  "for Xk in (csp.peers[Xi] - set(Xj)):\n",
```

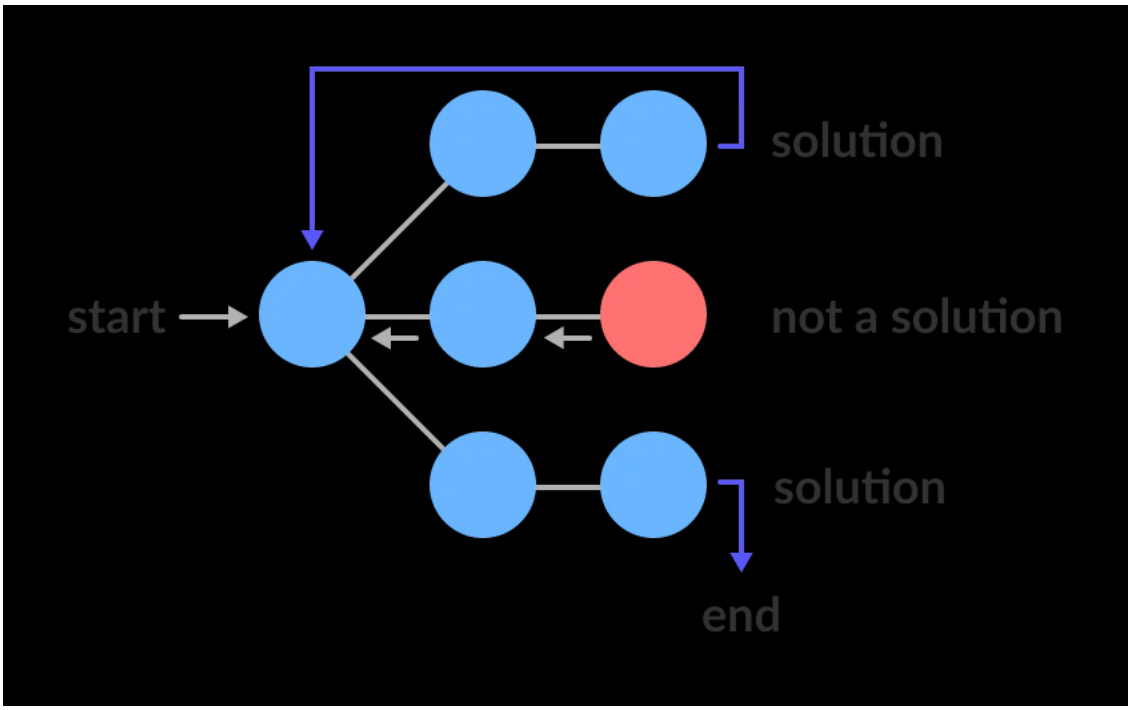
```
    "q.put((Xk, Xi))\n",
    "\n",
    "#display(csp.values)\n",
    "return True \n",
    "\n",
    "\n",
    "\n",
    "#WORKING OF THE REVISE ALGORITHM\n",
    "def Revise(csp, Xi, Xj):\n",
    "revised = False\n",
    "values = set(csp.values[Xi])\n",
    "\n",
    "for x in values:\n",
    "if not isconsistent(csp, x, Xi, Xj):\n",
    "csp.values[Xi] = csp.values[Xi].replace(x, '')\n",
    "revised = True \n",
    "\n",
    "return revised \n",
    "\n",
    "\n",
    "\n",
    "#CHECKS IF THE GIVEN ASSIGNMENT IS CONSISTENT\n",
    "def isconsistent(csp, x, Xi, Xj):\n",
    "for y in csp.values[Xj]:\n",
    "if Xj in csp.peers[Xi] and y!=x:\n",
    "return True\n",
    "\n",
    "return False\n",
    "\n",
    "\n",
    "#DISPLAYS THE SUDOKU IN THE GRID FORMAT\n",
    "def display(values):\n",
    "    for r in rows:\n",
    "        if r in 'DG':\n",
    "            print (\"-------------------------------------------------------------------------------
\")\n",
    "            for c in cols:\n",
    "                if c in '47':\n",
    "                    print (' | ', values[r+c], ' ',end=' ')\n",
    "    else:\n",
    "        print (values[r+c], ' ',end=' ')\n",
    "        print (end='\\n')\n",
    "\n",
    "        \n",
    "\n",
```

```
   "#CHECKS IF THE SUDOKU IS COMPLETE OR NOT\n",
   "def isComplete(csp):\n",
   "\tfor variable in squares:\n",
   "\t\tif len(csp.values[variable])>1:\n",
   "\t\t\treturn False\n",
   "\treturn True\n",
   "\n",
   "\n",
   "#WRITES THE SOLVED SUDOKU IN THE FORM OF A STRING\n",
   "def write(values):\n",
   "\toutput = \"\"\n",
   "\tfor variable in squares:\n",
   "\t\toutput = output + values[variable]\n",
   "\treturn output"
  ]
 },
 {
  "cell_type": "code",
  "execution_count": null,
  "metadata": {},
  "outputs": [],
  "source": []
 }
],
"metadata": {
 "kernelspec": {
  "display_name": "Python 3",
  "language": "python",
  "name": "python3"
 },
 "language_info": {
  "codemirror_mode": {
   "name": "ipython",
   "version": 3
  },
  "file_extension": ".py",
  "mimetype": "text/x-python",
  "name": "python",
  "nbconvert_exporter": "python",
  "pygments_lexer": "ipython3",
  "version": "3.8.5"
 }
},
"nbformat": 4,
"nbformat_minor": 4
```

}

Backtracking
Backtracking is an algorithm that finds the solution to a computational problem by determining all the possible paths to the next state, then testing them one by one for validity by continuing to determine further next states, until either reaching a goal state or failing to produce a valid next state. Upon failing, the algorithm backtracks through the tree and recursively repeats the process again.

State space:



- State space:Sudoku is a classical problem to be modeled as a CSP. The cells of the board can be seen as variables, the numbers 1-9 are the domain and neighboring cells constrain each
- Goal State:value has been assigned to each variable without violating the constraints
- Heuristics function: determining all the possible paths to the next state **By order**: the standard way is choosing an arbitrary order which is decided beforehand. For example, we can go through each row, from left to right.
Minimum remaining values: the next assigned variable will be the one which has the least possible values left for assignment.
In which order should we try the values:
Random order: after choosing a cell, we randomly choose a value from

the possible values left.
Least constraining value: we choose the value which rules out the fewest values in the remaining variables.

- Action: all directions “recursively repeats the process again”
- Path cost function: the length of a path in meters.
- Solution of the problem: a path from the initial state to the goal state.

Code:


```
{
 "cells": [
  {
   "cell_type": "code",
   "execution_count": null,
   "metadata": {},
   "outputs": [],
   "source": [
    "#explanation backtrack algorithm\n",
    "Backtracking is a general algorithm for \n",
    "finding all (or some) solutions to some computational problems, that incrementally builds candidates to the solutions\n",
    ", and abandons each partial candidate (“backtracks”) \n",
    "as soon as it determines that the candidate cannot possibly be completed to a valid solution"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": 1,
   "metadata": {},
   "outputs": [
    {
     "ename": "ModuleNotFoundError",
     "evalue": "No module named 'CSP'",
     "output_type": "error",
     "traceback": [
      "\u001b[1;31m---------------------------------------------------------------------------\u001b[0m",
      "\u001b[1;31mModuleNotFoundError\u001b[0m                       Traceback (most recent call last)",
      "\u001b[1;32m<ipython-input-1-356d0472013a>\u001b[0m in \u001b[0;36m<module>\u001b[1;34m\u001b[0m\n\u001b[0;32m      2\u001b[0m \u001b[1;33m\u001b[0m\u001b[0m\n\u001b[0;32m      3\u001b[0m \u001b[1;32mimport\u001b[0m \u001b[0mqueue\u001b[0m\u001b[1;33m\u001b[0m\u001b[1;33m\u001b[0m\u001b[0m\n\n\u001b[1;32m----> 4\u001b[1;33m \u001b[1;32mfrom\u001b[0m
```

\u001b[0mCSP\u001b[0m \u001b[1;32mimport\u001b[0m
\u001b[1;33m*\u001b[0m\u001b[1;33m\u001b[0m\u001b[1;33m\u001b[0m\u001b[0m\
n\u001b[0m\u001b[0;32m   5\u001b[0m
\u001b[1;33m\u001b[0m\u001b[0m\n\u001b[0;32m   6\u001b[0m \u001b[1;31m#THE
MAIN AC-3
ALGORITHM\u001b[0m\u001b[1;33m\u001b[0m\u001b[1;33m\u001b[0m\u001b[1;33m
\u001b[0m\u001b[0m\n",
      "\u001b[1;31mModuleNotFoundError\u001b[0m: No module named 'CSP'"
     ]
    }
   ],
   "source": [
    "# IMPLEMENTATION OF BACKTRACKING SOLUTION FOR SUDOKU\n",
    "\n",
    "from AC3 import *"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "metadata": {},
   "outputs": [],
   "source": [
    "#applying BACKTRACKING SOLUTION FOR SUDOKU"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "metadata": {},
   "outputs": [],
   "source": [
    "#BACKTRACKING SEARCH INITIALIZES THE INITIAL ASSIGNMENT AND CALLS
THE BACKTRACK FUNCTION\n",
    "def Backtracking_Search(csp):\n",
    "\treturn Backtrack({}, csp)\n",
    "\n",
    "\n",
    "\n",
    "#THE RECURSIVE FUNCTION WHICH ASSIGNS VALUE USING BACKTRACKING
\n",
    "def Backtrack(assignment, csp):\n",
    "\tif isComplete(assignment):\n",
    "\t\treturn assignment\n",
    "\n",

```
"\tvar = Select_Unassigned_Variables(assignment, csp)\n",
"\tdomain = deepcopy(csp.values)\n",
"\n",
"\tfor value in csp.values[var]:\n",
"\t\tif isConsistent(var, value, assignment, csp):\n",
"\t\t\tassignment[var] = value\n",
"\t\t\tinferences = {}\n",
"\t\t\tinferences = Inference(assignment, inferences, csp, var, value)\n",
"\t\t\tif inferences!= \"FAILURE\":\n",
"\t\t\t\tresult = Backtrack(assignment, csp)\n",
"\t\t\t\tif result!=\"FAILURE\":\n",
"\t\t\t\t\treturn result\n",
"\n",
"\t\t\tdel assignment[var]\n",
"\t\t\tcsp.values.update(domain)\n",
"\n",
"\treturn \"FAILURE\"\n",
"\n",
"\n",
"\n",
"#FORWARD CHECKING USING THE CONCEPT OF INFERENCES\n",
"def Inference(assignment, inferences, csp, var, value):\n",
"\tinferences[var] = value\n",
"\n",
"\tfor neighbor in csp.peers[var]:\n",
"\t\tif neighbor not in assignment and value in csp.values[neighbor]:\n",
"\t\t\tif len(csp.values[neighbor])==1:\n",
"\t\t\t\treturn \"FAILURE\"\n",
"\n",
"\t\t\tremaining = csp.values[neighbor] = csp.values[neighbor].replace(value, \"\")\n",
"\n",
"\t\t\tif len(remaining)==1:\n",
"\t\t\t\tflag = Inference(assignment, inferences, csp, neighbor, remaining)\n",
"\t\t\t\tif flag==\"FAILURE\":\n",
"\t\t\t\t\treturn \"FAILURE\"\n",
"\n",
"\treturn inferences\n",
"\n",
"\t\t\n",
"#CHECKS IF THE ASSIGNMENT IS COMPLETE\n",
"def isComplete(assignment):\n",
"\treturn set(assignment.keys())==set(squares)\n",
"\n",
"\n",
"\n",
```

```
    "#SELECTS THE NEXT VARIABLE TO BE ASSIGNED USING MRV\n",
    "def Select_Unassigned_Variables(assignment, csp):\n",
    "\tunassigned_variables = dict((squares, len(csp.values[squares])) for squares in
csp.values if squares not in assignment.keys())\n",
    "\tmrv = min(unassigned_variables, key=unassigned_variables.get)\n",
    "\treturn mrv\n",
    "\n",
    "\n",
    "\n",
    "#RETURNS THE STRING OF VALUES OF THE GIVEN VARIABLE \n",
    "def Order_Domain_Values(var, assignment, csp):\n",
    "\treturn csp.values[var]\n",
    "\n",
    "\n",
    "\n",
    "#CHECKS IF THE GIVEN NEW ASSIGNMENT IS CONSISTENT\n",
    "def isConsistent(var, value, assignment, csp):\n",
    "\tfor neighbor in csp.peers[var]:\n",
    "\t\tif neighbor in assignment.keys() and assignment[neighbor]==value:\n",
    "\t\t\treturn False\n",
    "\treturn True\n",
    "\n",
    "\n",
    "\n",
    "#PERFORMS FORWARD-CHECKING\n",
    "def forward_check(csp, assignment, var, value):\n",
    "\tcsp.values[var] = value\n",
    "\tfor neighbor in csp.peers[var]:\n",
    "\t\tcsp.values[neighbor] = csp.values[neighbor].replace(value, '')\n",
    "\n",
    "\n",
    "\n",
    "#DISPLAYS THE SOLVED SUDOKU IN THE GRID FORMAT\n",
    "def display(values):\n",
    "    for r in rows:\n",
    "    \tif r in 'DG':\n",
    "    \t\tprint (\"-----------------------------------------\")\n",
    "    \tfor c in cols:\n",
    "    \t\tif c in '47':\n",
    "    \t\t\tprint (' | ', values[r+c], ' ',end=' ')\n",
    "    \t\telse:\n",
    "    \t\t\tprint (values[r+c], ' ',end=' ')\n",
    "    \tprint (end='\\n')\n",
    "\n",
    "\n",
```

```
    "#WRITES THE SOLVED SUDOKU IN THE FORM OF A STRING\n",
    "def write(values):\n",
    "\toutput = \"\"\n",
    "\tfor variable in squares:\n",
    "\t\toutput = output + values[variable]\n",
    "\treturn output"
   ]
  }
 ],
 "metadata": {
  "kernelspec": {
   "display_name": "Python 3",
   "language": "python",
   "name": "python3"
  },
  "language_info": {
   "codemirror_mode": {
    "name": "ipython",
    "version": 3
   },
   "file_extension": ".py",
   "mimetype": "text/x-python",
   "name": "python",
   "nbconvert_exporter": "python",
   "pygments_lexer": "ipython3",
   "version": "3.8.5"
  }
 },
 "nbformat": 4,
 "nbformat_minor": 4
}
```


Forward algorithm:

The **forward algorithm**, in the context of a hidden Markov model (HMM), is used to calculate a 'belief state': the probability of a state at a certain time, given the history of evidence. The process is also known as *filtering*. The forward algorithm is closely related to, but distinct from, the Viterbi algorithm.

The forward and backward algorithms should be placed within the context of probability as they appear to simply be names given to a set of standard mathematical procedures within a few fields. For example, neither "forward algorithm" nor "Viterbi" appear in the Cambridge encyclopedia of mathematics. The main observation to take away from these algorithms is how to organize Bayesian updates and inference to be efficient in the context of directed graphs of variables.

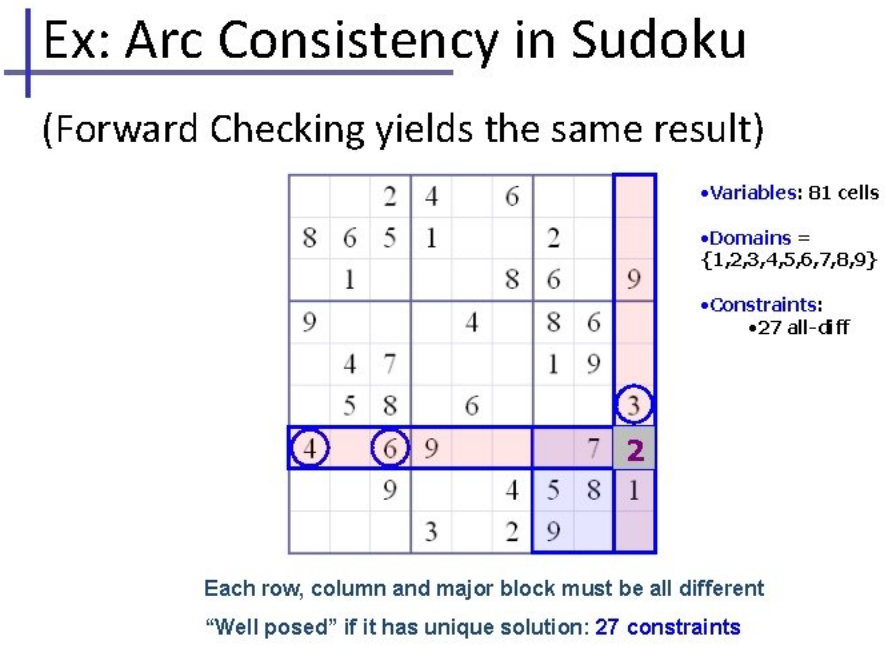
Forward algorithm components:

- a set of N states
- a transition probability matrix A, each a_{ij} represent- ing the probability of moving from state i to state j, s.t. $\sum_{j=1}^{n} a_{ij} = 1 \; \forall \; i$.
- a sequence of T observations, each one drawn from a vocabulary $V = v_1, v_2, ..., v_V$.
- a sequence of observation likelihoods, also called emission probabili- ties, each expressing the probability of an observation o_t being generated from a state i.
- an initial probability distribution over states. π_i is the probability that the Markov chain will start in state i. Some states j may have $\pi_j = 0$, meaning that they cannot be initial states. Also, $\sum_{i=1}^{n} \pi_i = 1$.

How the algorithm works:

Markov Model explains that the next step depends only on the previous step in a temporal sequence. In Hidden Markov Model the state of the system is `hidden` (invisible), however each state emits a symbol at every time step. HMM works with both discrete and continuous sequences of data.

State space :



Code:


```
{
 "cells": [
  {
   "cell_type": "code",
   "execution_count": null,
   "metadata": {},
   "outputs": [],
   "source": [
    "# Sudoku Solver implemented using a constraint-satisfaction formulation\n",
    "# Implements a dumb backtracking method with random cell selection,\n",
    "# backtracking with forward checking, and forward checking with heuristics"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": 9,
   "metadata": {},
   "outputs": [],
   "source": [
    "import math\n",
    "import random\n",
    "import time\n",
    "import sys\n",
    "\n",
    "def main():\n",
    "\n",
    "    if len( sys.argv ) < 3:\n",
    "        print_help()\n",
    "        return 0    \n",
    "    \n",
    "    if len( sys.argv ) == 4:\n",
    "        n_trials = int( sys.argv[3] )\n",
    "    else:\n",
    "        n_trials = 1\n",
    "        \n",
    "    print(\"Running %d trial(s)...\" % n_trials)\n",
    "        \n",
    "    run_times = []\n",
    "      \n",
    "    ### Easy Puzzle\n",
    "    easy   = [[0, 3, 0, 0, 8, 0, 0, 0, 6],\n",
```

```
"             [5, 0, 0, 2, 9, 4, 7, 1, 0],\n",
"             [0, 0, 0, 3, 0, 0, 5, 0, 0],\n",
"             [0, 0, 5, 0, 1, 0, 8, 0, 4],\n",
"             [4, 2, 0, 8, 0, 5, 0, 3, 9],\n",
"             [1, 0, 8, 0, 3, 0, 6, 0, 0],\n",
"             [0, 0, 3, 0, 0, 7, 0, 0, 0],\n",
"             [0, 4, 1, 6, 5, 3, 0, 0, 2],\n",
"             [2, 0, 0, 0, 4, 0, 0, 6, 0]]\n",
"    \n",
"    ### Medium puzzle\n",
"    medium = [[3, 0, 8, 2, 9, 6, 0, 0, 0],\n",
"             [0, 4, 0, 0, 0, 8, 0, 0, 0],\n",
"             [5, 0, 2, 1, 0, 0, 0, 8, 7],\n",
"             [0, 1, 3, 0, 0, 0, 0, 0, 0],\n",
"             [7, 8, 0, 0, 0, 0, 0, 3, 5],\n",
"             [0, 0, 0, 0, 0, 0, 4, 1, 0],\n",
"             [1, 2, 0, 0, 0, 7, 8, 0, 3],\n",
"             [0, 0, 0, 8, 0, 0, 0, 2, 0],\n",
"             [0, 0, 0, 5, 4, 2, 1, 0, 6]]\n",
"\n",
"    ### Hard Puzzle\n",
"    hard   = [[7, 0, 0, 0, 0, 0, 0, 0, 0],\n",
"             [6, 0, 0, 4, 1, 0, 2, 5, 0],\n",
"             [0, 1, 3, 0, 9, 5, 0, 0, 0],\n",
"             [8, 6, 0, 0, 0, 0, 0, 0, 0],\n",
"             [3, 0, 1, 0, 0, 0, 4, 0, 5],\n",
"             [0, 0, 0, 0, 0, 0, 0, 8, 6],\n",
"             [0, 0, 0, 8, 4, 0, 5, 3, 0],\n",
"             [0, 4, 2, 0, 3, 6, 0, 0, 7],\n",
"             [0, 0, 0, 0, 0, 0, 0, 0, 9]]\n",
"\n",
"    ### EVIL Puzzle        \n",
"    evil   = [[0, 6, 0, 8, 0, 0, 0, 0, 0],\n",
"             [0, 0, 4, 0, 6, 0, 0, 0, 9],\n",
"             [1, 0, 0, 0, 4, 3, 0, 6, 0],\n",
"             [0, 5, 2, 0, 0, 0, 0, 0, 0],\n",
"             [0, 0, 8, 6, 0, 9, 3, 0, 0],\n",
"             [0, 0, 0, 0, 0, 0, 5, 7, 0],\n",
"             [0, 1, 0, 4, 8, 0, 0, 0, 5],\n",
"             [8, 0, 0, 0, 1, 0, 2, 0, 0],\n",
"             [0, 0, 0, 0, 0, 5, 0, 4, 0]]\n",
"    \n",
"    puzzles = [easy, medium, hard, evil]\n",
"    \n",
"    puzzle = puzzles[ int( sys.argv[1] ) - 1 ] \n",
```

```
    "    \n",
    "    print(\"Solving the following puzzle: \")\n",
    "    print_puzzle( puzzle )\n",
    "    \n",
    "    start_time = time.time()\n",
    "    \n",
    "    method_type = int( sys.argv[2] )\n",
    "    if method_type == 1:\n",
    "        print(\"Using backtracking search.\") \n",
    "        result = solve_backtrack( puzzle )\n",
    "    elif method_type == 2:\n",
    "        print( \"Using backtracking with forward checking.\" )\n",
    "        result = solve_btfc( puzzle )\n",
    "    else:\n",
    "        print (\"Using backtracking with forward checking and heurisitics.\") \n",
    "        result = solve_btfch( puzzle )\n",
    "        \n",
    "    if result:\n",
    "        print( \"It took\", time.time() - start_time, \"seconds to solve this puzzle.\")\n",
    "    else:\n",
    "        print( \"Failed to find a solution! D:\")\n",
    "   \n",
    "#    for i in range(n_trials):\n",
    "#        start_time = time.time()\n",
    "#        if not solve_backtrack( puzzle ):\n",
    "#            print \"Failed to find a solution! D:\"\n",
    "#        else:\n",
    "#            elapsed = time.time() - start_time\n",
    "#            print \"It took\", time.time() - start_time, \"seconds to solve this 
puzzle.\"\n",
    "#            print \"elapsed:\", elapsed\n",
    "#            run_times.append( elapsed )\n",
    "#    \n",
    "#    run_times\n",
    "#    print \"Average run time:\", sum(run_times)/n_trials\n",
    "\n",
    "    \n",
    "    \n",
    "# Recursive backtracking algorithm to solve puzzle     \n",
    "def solve_backtrack( puzzle ):    \n",
    "    \n",
    "    # store all the possible values remaining for a square\n",
    "    domain = range(1,10) \n",
    "    \n",
    "    # get a list of the empty squares (remaining variables)\n",
    "    empty_squares = get_empty_squares( puzzle )\n",
```

```
"    \n",
"    # if there are no remaining empty squares we're done\n",
"    if len(empty_squares) == 0: \n",
"        print (\"Woohoo, success! Check it out:\")\n",
"        print_puzzle( puzzle )\n",
"        return 1\n",
"    \n",
"    square = get_random_square( empty_squares )\n",
"    row = square[0]\n",
"    col = square[1]\n",
"    \n",
"    while len( domain ) != 0:\n",
"        # get a random value out of the list of remaining possible values\n",
"        value = domain[ int( math.floor( random.random()*len(domain) ) ) ]\n",
"        domain.remove( value )\n",
"        # check the value against theconstraints\n",
"        if check_row( square, value, puzzle ):\n",
"            if check_col( square, value, puzzle ):\n",
"                if check_block( square, value, puzzle ):\n",
"                    puzzle[row][col] = value\n",
"                    if solve_backtrack( puzzle ):\n",
"                        return 1\n",
"                    else:\n",
"                        puzzle[row][col] = 0                    \n",
"    return 0                    \n",
"    \n",
"\n",
"# Backtracking with forward-checking algorithm\n",
"# Stores a list with all the variables and their potentially legal \n",
"# values - stops when there are no empty squares remaining\n",
"# Randomly selects the next cell and the value\n",
"def solve_btfc( puzzle ):    \n",
"   \n",
"    # get a list of the empty squares (remaining variables)\n",
"    empty_squares = get_empty_squares( puzzle )\n",
"\n",
"    # if there are no remaining empty squares we're done\n",
"    if len(empty_squares) == 0: \n",
"        print (\"Woohoo, success! Check it out:\")\n",
"        print_puzzle( puzzle )\n",
"        return 1\n",
"    \n",
"    square = get_random_square( empty_squares )\n",
"    row = square[0]\n",
"    col = square[1]\n",
```

```
    "    \n",
    "    remaining_values = get_remaining_values( puzzle )\n",
    "   \n",
    "    values = list( remaining_values[col+row*9] )\n",
    "    \n",
    "    while len( values ) != 0:        \n",
    "        value = values[ int( math.floor( random.random()*len( values ) ) ) ]\n",
    "        values.remove(value)        \n",
    "        if forward_check( remaining_values, value, row, col ):\n",
    "            puzzle[row][col] = value\n",
    "            if solve_btfc( puzzle ):\n",
    "                return 1\n",
    "            else:\n",
    "                puzzle[row][col] = 0\n",
    "                \n",
    "    return 0\n",
    "\n",
    "# Solves the sudoku puzzle using forward checking and 3 heuristics:\n",
    "# minimum remaining values, degree, and least constraining value heuristics\n",
    "def solve_btfch( puzzle ):\n",
    "     # get a list of the empty squares (remaining variables)\n",
    "    empty_squares = get_empty_squares( puzzle )\n",
    "\n",
    "    # if there are no remaining empty squares we're done\n",
    "    if len(empty_squares) == 0: \n",
    "        print(\" Woohoo, success! Check it out:\")\n",
    "        print_puzzle( puzzle )\n",
    "        return 1\n",
    "    \n",
    "    # find the most constrained square (one with least remaining values)\n",
    "    remaining_values = get_remaining_values( puzzle )\n",
    "    mrv_list = []\n",
    "    [ mrv_list.append( len( remaining_values[ square[0]*9+square[1] ] ) ) for square in
empty_squares ]\n",
    "    # make a list of the squares with the minimum remaining values (mrv)\n",
    "    mrv_squares = []\n",
    "    minimum = min( mrv_list )\n",
    "    for i in range(len(mrv_list)):\n",
    "        value = mrv_list[i]\n",
    "        if value == minimum:\n",
    "            mrv_squares.append( empty_squares[i] )\n",
    "    \n",
    "    # if there are no ties, take the square with the MRV\n",
    "    if len( mrv_squares ) == 1:\n",
    "        square = mrv_squares[0]\n",
```

```
    "    else:\n",
    "        # otherwise, find the most constraining variable (variable with highest
degree)\n",
    "        degree_list = []\n",
    "        for cell in mrv_squares:  \n",
    "            degree = get_degree( cell, puzzle )\n",
    "            degree_list.append( degree )\n",
    "            \n",
    "            max_degree = max( degree_list )\n",
    "            max_degree_squares = []\n",
    "            for i in range(len(degree_list)):      \n",
    "                value = degree_list[i]\n",
    "                if value == max_degree:\n",
    "                    max_degree_squares.append( mrv_squares[i] )\n",
    "            # just take the first square as a tie-breaker      \n",
    "            square = max_degree_squares[0]\n",
    "     \n",
    "    row = square[0]\n",
    "    col = square[1]\n",
    "          \n",
    "    values = list( remaining_values[col+row*9] )\n",
    "    \n",
    "    while len( values ) != 0:        \n",
    "        \n",
    "        lcv_list = get_lcv( values, row, col, remaining_values )\n",
    "        # take the least constraining value\n",
    "        value = values[ lcv_list.index( min( lcv_list ) ) ]\n",
    "        values.remove(value)        \n",
    "        if forward_check( remaining_values, value, row, col ):\n",
    "            puzzle[row][col] = value\n",
    "            if solve_btfch( puzzle ):\n",
    "                return 1\n",
    "            else:\n",
    "                puzzle[row][col] = 0\n",
    "                \n",
    "    return 0\n",
    "\n",
    "# counts the number of times a value appears in constrained cells\n",
    "def get_lcv( values, row, col, remaining_values ):\n",
    "    \n",
    "    lcv_list = []\n",
    "    \n",
    "    for value in values: \n",
    "        count = 0   \n",
    "        for i in range(9):\n",
```

```
"            if i == col:\n",
"                continue           \n",
"            x = remaining_values[row*9+i]                    \n",
"            if value in x:\n",
"                count += 1\n",
"         \n",
"        for i in range(9):\n",
"            if i == row:\n",
"                continue               \n",
"            x = remaining_values[col+9*i]\n",
"            if value in x:\n",
"                count += 1\n",
"\n",
"        block_row = row/3\n",
"        block_col = col/3  \n",
"        for i in range(3):\n",
"            for j in range(3):                \n",
"                if [block_row*3+i, block_col*3+j] == [row, col]:\n",
"                    continue            \n",
"                x = remaining_values[block_col*3+j+(block_row*3+i)*9]\n",
"                if value in x:\n",
"                    count += 1 \n",
"                     \n",
"        lcv_list.append( count )\n",
"                                \n",
"    return lcv_list\n",
"\n",
"# returns the number of variables constrained by the specified square\n",
"def get_degree( square, puzzle ):\n",
"    row = square[0]\n",
"    col = square[1]\n",
"    \n",
"    degree = 0\n",
"    \n",
"    for i in range(9):\n",
"        if i == col:\n",
"            continue               \n",
"        if puzzle[row][i] == 0:\n",
"            degree+=1\n",
"     \n",
"    for i in range(9):\n",
"        if i == row:\n",
"            continue\n",
"        if puzzle[i][col] == 0:\n",
"            degree+=1\n",
```

```
"\n",
"    block_row = row/3\n",
"    block_col = col/3  \n",
"    for i in range(3):\n",
"        for j in range(3):           \n",
"            if [block_row*3+i, block_col*3+j] == [row, col]:\n",
"                continue       \n",
"            if puzzle[block_row*3+i][block_col*3+j] == 0:\n",
"                degree+=1\n",
"\n",
"    return degree         \n",
"    \n",
"\n",
"# checks to see if the value being removed is the only one left\n",
"def forward_check( remaining_values, value, row, col ):    \n",
"\n",
"    for i in range(9):\n",
"        if i == col:\n",
"            continue    \n",
"            \n",
"        x = remaining_values[row*9+i]\n",
"                \n",
"        if len(x) == 1:\n",
"            if x[0] == value:\n",
"                return 0\n",
"     \n",
"    for i in range(9):\n",
"        if i == row:\n",
"            continue\n",
"            \n",
"        x = remaining_values[col+9*i]\n",
"        if len(x) == 1:\n",
"            if x[0] == value:\n",
"                return 0\n",
"\n",
"    block_row = row/3\n",
"    block_col = col/3  \n",
"    for i in range(3):\n",
"        for j in range(3):\n",
"            \n",
"            if [block_row*3+i, block_col*3+j] == [row, col]:\n",
"                continue           \n",
"            \n",
"            x = remaining_values[block_col*3+j+(block_row*3+i)*9]\n",
"            if len(x) == 1:\n",
```

```
    "                if x[0] == value:\n",
    "                    return 0                                \n",
    "    return 1                        \n",
    "\n",
    "\n",
    "# Returns a list of the remaining potential values for each of the 81 squares\n",
    "# The list is structured row by row with respect to the puzzle\n",
    "# Only gets called once, at the beginning of the BT-FC search to initialize\n",
    "def get_remaining_values( puzzle ):\n",
    "    remaining_values = []\n",
    "    # initialize all remaining values to the full domain\n",
    "    [remaining_values.append( range(1,10) ) for i in range(81) ]    \n",
    "    for row in range( len(puzzle) ):\n",
    "        for col in range( len(puzzle[1]) ):\n",
    "            if puzzle[row][col] != 0:\n",
    "                # remove the value from the constrained squares  \n",
    "                value = puzzle[row][col]  \n",
    "                remaining_values = remove_values( row, col, value, remaining_values )
\n",
    "                \n",
    "    return remaining_values     \n",
    "                        \n",
    "                        \n",
    "# Removes the specified value from constrained squares and returns the new
list\n",
    "def remove_values( row, col, value, remaining_values ):\n",
    "    \n",
    "    # use a value of zero to indicate that the square is assigned\n",
    "    remaining_values[col+row*9] = [0]\n",
    "    \n",
    "    # Remove the specified value from each row, column, and block if it's there\n",
    "    for x in remaining_values[row*9:row*9+9]:\n",
    "        try:\n",
    "            x.remove( value )\n",
    "        except ValueError:  \n",
    "           pass \n",
    "            \n",
    "    for i in range(9):\n",
    "        try:\n",
    "            remaining_values[col+9*i].remove( value )\n",
    "        except ValueError:\n",
    "            pass\n",
    "\n",
    "    block_row = row/3\n",
    "    block_col = col/3  \n",
```

```
    "    for i in range(3):\n",
    "        for j in range(3):\n",
    "            try:\n",
    "                remaining_values[block_col*3+j+(block_row*3+i)*9].remove( value )\n",
    "            except ValueError:\n",
    "                pass\n",
    "\n",
    "    return remaining_values\n",
    "   \n",
    "                    \n",
    "    "# return a randomly selected square from the list of empties\n",
    "    "def get_random_square( empty_squares ):   \n",
    "    "    # randomly pick one of the empty squares to expand and return it\n",
    "    "    return empty_squares[ int(math.floor(random.random()*len(empty_squares))) ]
\n",
    "    "    \n",
    "    "    \n",
    "    "# return the list of empty squares indices for the puzzle\n",
    "    "def get_empty_squares ( puzzle ):\n",
    "    "    empty_squares = []\n",
    "    "    # scan the whole puzzle for empty cells\n",
    "    "    for row in range(len( puzzle )):\n",
    "    "        for col in range(len( puzzle[1] )):\n",
    "    "            if puzzle[row][col] == 0:\n",
    "    "                empty_squares.append( [row,col] ) \n",
    "    "    return empty_squares\n",
    "    "    \n",
    "    "    \n",
    "    "# checks the 9x9 block to which the square belongs\n",
    "    "def check_block( square, value, puzzle ):\n",
    "    "    row = square[0]\n",
    "    "    col = square[1]\n",
    "    "    block_row = row/3\n",
    "    "    block_col = col/3\n",
    "    "                  \n",
    "    "    for i in range(3):\n",
    "    "        for j in range(3):\n",
    "    "            if [i,j] == square:\n",
    "    "                continue\n",
    "    "            if puzzle[block_row*3 + i][block_col*3 + j] == value:\n",
    "    "                return 0\n",
    "    "    return 1\n",
    "    "    \n",
    "    "# checks the row of the specified square for the same value\n",
    "    "def check_row( square, value, puzzle ):\n",
```

```
    "    row = square[0]\n",
    "    col = square[1]\n",
    "    for i in range( len( puzzle ) ):\n",
    "        if i == square[0]:\n",
    "            continue\n",
    "        if puzzle[row][i] == value:\n",
    "            return 0\n",
    "            \n",
    "    return 1\n",
    "    \n",
    "    \n",
    "   "# checks the column of the specified square for the same value\n",
    "   "def check_col( square, value, puzzle ):\n",
    "    row = square[0]\n",
    "    col = square[1]\n",
    "    for i in range(len(puzzle[1])):\n",
    "        if i == square[1]:\n",
    "            continue\n",
    "        if puzzle[i][col] == value:\n",
    "            return 0\n",
    "            \n",
    "    return 1\n",
    "   "\n",
    "   "\n",
    "   "def print_puzzle( puzzle ):\n",
    "    for row in puzzle:\n",
    "        print (row) \n",
    "        \n",
    "   "def print_help():\n",
    "    print (\"Usage: python sudoku.py <difficulty=1,2,3,4> <method=1,2,3> [optional:
no. of trials]\"    )      \n",
    "        \n",
    "   "\n",
    "   "#if __name__==\"__main__\":\n",
    "    main()"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "metadata": {},
   "outputs": [],
   "source": []
  }
 ],
```

```
 "metadata": {
  "kernelspec": {
   "display_name": "Python 3",
   "language": "python",
   "name": "python3"
  },
  "language_info": {
   "codemirror_mode": {
    "name": "ipython",
    "version": 3
   },
   "file_extension": ".py",
   "mimetype": "text/x-python",
   "name": "python",
   "nbconvert_exporter": "python",
   "pygments_lexer": "ipython3",
   "version": "3.8.5"
  }
 },
 "nbformat": 4,
 "nbformat_minor": 4
}
```


Conclusions

- After running our program with different parameters, we concluded that using all the heuristics together yields the best result. It is of no surprise, as combining forward checking and arc consistency helps us to detect failure much more quickly than using each one separately. Thus, the program has to backtrack much less.
- We were surprised to discover that forward checking always performed better than arc consistency, even though arc consistency is supposed to detect failures earlier. The reason can be related to our specific implementation or to the problem itself.

Resources :

- http://diuf.unifr.ch/pai/people/juillera/Sudoku/Sudoku.html *Sudoku Explainer by Nicolas Juillerat* (Popular for rating Sudokus in general)

- http://gsf.cococlyde.org/download/sudoku *sudoku by Glenn Fowler* (Popular for rating the hardest Sudokus amongst other things)
- A Pencil-and-Paper Algorithm for Solving Sudoku Puzzles