**Alex Morson**

**May 8, 2018**

Sudoku and Decision Making: Algorithms to Simulate Strategies Used to Solve Logic Puzzles

# Introduction

The challenge of a logic puzzle is based on the decision-making abilities of the player. Although the answer sets to logic puzzles can be arrived at through multiple decision-making methods, some strategies are better than others; the skill of a player is determined by not only their ability to solve a puzzle at all, but how efficiently they can do so compared to other players capable of solving the puzzle. I wanted my project to involve a computer making decisions from a set of options within a grid-based environment, like Chess or some sort of war game; I chose Sudoku for the logic puzzle because it’s grid-based and has a simple, but well-defined set of rules, putting a strong emphasis on strategy.

# Problem Description

## *Sudoku Problem*

A standard Sudoku puzzle can be defined mathematically as:

* A set *S* of 81 grid squares (Provan 702-703).
* A domain set *I* = {1, 2, 3, 4, 5, 6, 7, 8, 9} (Provan 702-703)
* A collection *G* of cell groups (the rows/columns/3x3 subgrids), with each cell group *B ε G* consisting of a set of exactly nine squares in *S*. (Provan 702-703)
* An initial set of clues , with square , with fill value (Provan 703), and (Taalman 8).
* (Provan 702-703)

To solve the puzzle, the player must fill in all the cells on the grid with values from *I* such that each cell group *B ε G* has a set of cells with exactly one cell assigned a value of *i* for all *i ε I*, andthe final configuration of the grid is consistent with *A* (Provan 702). Because of these constraints, a Sudoku puzzle has only *one* possible configuration of cells and integer values that satisfy these requirements, thereby giving it only one possible solution state. However, the sequence of actions to reach this solution state varies. A player’s skill at solving Sudoku puzzles is determined by the sequence of actions he performs to reach that state.

A Sudoku problem can essentially be thought of as a *k*-Coloring problem, where *k* = 9; each color corresponds to each value *i* in I, and the grid can be reimagined as a board with 81 vertices, one for each cell, with an edge connecting each cell to one that is in the same row, column, and subgrid. In this way, nodes connected by an edge cannot share the same color, just as no cell in a grid can share a value with a cell in the same row, column, and/or subgrid. In this manner, verifying whether a completely filled board is the solution to the puzzle becomes a simple task.

## *Artificial Intelligence*

All artificial intelligence agents are, by nature, problem solvers, with each serving as the solution to a particular problem, also known as a *task environment* (Russel & Norvig 40). An agent perceives its environment through the use of *sensors* (Russel & Norvig 34). From this information, the agent is able to change the state of the environment through the use of *actuators* (Russel & Norvig 34). The agent’s perceptual input(s) at any given instance is encoded as the percept(s) (Russel & Norvig 34), and the agent’s behavior is determined by the *agent function*, which maps a percept to a specific action (Russel & Norvig 35). An agent acts continuously until either the environment reaches its goal state or until the agent is forcibly shut down (Russel & Norvig 66).

## *Synthesis*

In the context of artificial intelligence, a Sudoku puzzle serves as the environment that the agent acts upon. By extension, this means that the goal state of the environment is the same as it is in the puzzle: a grid where every cell is filled and no row, column, or subgrid contains two or more cells with the same value or a cell with a value outside the domain *I* = {1, 2, 3, …, 9}.

As was mentioned earlier, the goal of the project is to solve the puzzle utilizing different strategies the way a human would. This means understanding what exactly human players do when solving a puzzle. The number of things a player can actually do to a cell are limited to:

1. Assigning a value to the cell
2. Removing a value from the cell’s list of possible values.

The list of possible values that a cell can have is dependent on the values that are assigned to its neighbors (all the cells within a cell’s row/column/subgrid). When a player finds a cell that has only one possible value, they will fill the cell with that value. Once there are no more cells with only one possibility, the player will observe the relations between cells on the board and based on the specifics of these relations, will eliminate choices from a cell’s list of possible values until a cell is found to have only one possible value. After this, the process restarts. A strategy is performed by searching for a specific, corresponding type of relation and either assigning a value or removing a possibility from all the cells that are found to possess that relation.

# Specific Methods & Implementation

## *Sudoku Puzzle Data Structure(s)*

Since the puzzle needs to be referenced constantly, and because there is never more than one puzzle instantiated at the same time, the puzzle object is implemented as a singleton class called SudokuPuzzle. A singleton is a class where the program/project/solution contains only one instance of that class (Sarcar 17). SudokuPuzzle is made up of several parts:

* Board: A data structure that represents the actual board of cells. The class’s fields include:
  + cellList: A list collection that contains all the cells on the board.
  + rowList: A list collection that contains all the rows on the board.
  + columnList: A list collection that contains all the columns on the board.
  + subgridList: A list collection that contains all the 3x3 subgrids on the board.
  + puzzleId: A string id value.
  + frequencyCounter: A dictionary-like collection that keeps track of all the times a value is assigned to a cell on the board.
* DiscardedValuesTable: A specialized dictionary that is meant to keep track of all of the integer values that were eliminated from a cell’s possibilities: the cells in grid are the keys, and the lists of discarded possibilities are the values.
* CellGroup: A collection type that represents a row, a column, or a subgrid in the puzzle. Each one possesses the following fields:
  + UnitType: An enumeration that indicates whether a cell is a row, column, or subgrid
  + Members: The list of cells in the group.
  + Index: An integer value that identifies which exact row/column/subgroup its supposed to represent.
  + IsFull: returns a boolean based on whether all the cells are filled.

Its functions include:

* + GetOpenMembers: Returns all the members with unfilled values.
  + GetRemainingFills: Returns the list of values in domain *I* that have not been assigned to any members of the group.
  + GetOpenCellCount: Return the number of members with unfilled values.
  + GetCellsWithPossibility(n): Returns all members that has some number *n* in their list of possibilities.
  + ReturnSharedNeighbors:
  + TargetValueUsed(n): Returns a boolean indicating if any of the members have a number *n* assigned as a value.
* CellData: A class containing the basic fields of a cell, namely the fillValue, rowNumber, columnNumber, and subgridNumber. All clues are deserialized into this class before being converted into actual SudokuCell objects.
* SudokuCell: Child class of CellData. This class handles all the actual functionality that each cell is supposed to have. Its additional fields include:
  + isFilled: returns true if the cell is assigned a value and false if not.
  + CellType: An enumeration indicating whether or not the cell is a clue.
  + isClue: returns true if CellType == CellType.Clue.
  + rowPointer: Pointer to the row in the puzzle that this cell belongs to.
    - rowNeighbors: Returns all the cells in its row other than itself.
  + columnPointer: Pointer to the column in the puzzle that this cell belongs to.
    - columnNeighbors: Returns all the cells in its column other than itself.
  + subgridPointer: Pointer to the subgrid in the puzzle that this cell belongs to.
    - subgridNeighbors: Returns all the cells in its subgrid other than itself.
  + Neighbors: The union of the row, column, and subgrid neighbors.
  + puzzlePointer: A pointer to the puzzle singleton.
  + NeighborFills: The set of all fill values assigned to the cell’s neighbors.
  + DiscardedFills: The pointer to the cell’s entry in the DiscardedValuesTable, which lists all the possibilities for the cell that have been ruled out.
  + Possibilities: The list of all possible values that can be assigned to this cell. If the cell is filled, this field is empty. Otherwise, it is equal to .
* ClueSerializationUtility: The class that is responsible for building the puzzles. It takes a string input from the user, finds the json file whose file name matches, and then deserializes the file into CellData objects. The class’s deserialization method is invoked when the puzzle/board is instantiated.

## *Agent Program Flow*

### *Strategy Pattern*

The strategy pattern is a common behavioral design pattern used in software. A family of algorithms is defined, and each one is encapsulated as an object whose main method can be invoked (Sarcar 39). All the strategies are derived from an abstract class known as Strategy<T>, which has a single abstract method that takes in a parameter. Because the parent here class uses generics, and the type of generic can differ depending on the exact strategy, the strategy objects cannot be stored in a collection, although there is no particular need for such a collection anyway. Because they share a parent class though, the algorithms of all the strategies are executed in the same fashion, even if the algorithms are very different.

### *Sensors and Actuators*

In the context of the project, each strategy object serves as an actuator of the agent. Each strategy takes in a collection of items that contain the information on the cells that are to operated on and the necessary parameters as a single parameter. The strategy algorithm then either assigns a value to the cell or eliminates one or more possible values for each cell passed in. This raises two important questions though: how does the agent know when to execute a certain strategy and how does it obtain the necessary cell information for the strategy to operate on? The answer to these questions lie in the use of *contexts*.

The contexts are essentially the sensors of the agent, each one associated with a particular strategy. Much like in the strategy pattern, contexts are encapsulated as objects that inherit from a common abstract class. This abstract class, Context<T>, is a bit more complicated than the strategy abstract class, as it contains two cornerstone functions:

* GetContext: An abstract function that checks whether or not the associated strategy can be executed. The means of doing so varies greatly between contexts, but regardless, it always returns a boolean value on whether or not the strategy can be executed, and if so, also returns the necessary information to feed into the strategy function.
* ContextAlgorithm: Unlike the previous function, this algorithm for this function is constant for all the contexts. It executes GetContext and checks the returned boolean value. If it is true, the function returns the associated Percept enum, and otherwise, returns Percept.None.

In addition to the contexts, another of the agent’s sensors (known as the *PuzzleCertifier*) determines whether or not the puzzle is solved, which it does by checking the following:

* Are all the cells in the puzzle filled?
* Does the sum of all the cell values add up to exactly ?
* Do none of the rows, columns, or subgrids contain two or more cells whose values are the same?

If all three conditions are true, then the puzzle has reached its goal state and is therefore solved.

### *Agent Function*

The agent function is executed upon instantiation of the puzzle; it loops continuously until the puzzle is solved. At the start of each iteration, the agent checks each context in order of priority until it finds one that is returned as viable, setting the agent’s percept field to the context’s corresponding enum, and records the parameters that will be used as inputs for the strategy. Once the agent executes the corresponding strategy, it checks the state of the puzzle, exiting the loop if the puzzle has reached the goal state.

As you may have noticed, the agent function only acts upon a single strategy per iteration of the loop. This is because of how deeply connected the cells are to each other. Every cell’s list of possibilities is based on the contents of its neighbors and on its entry in the DiscardedValuesTable, so when a cell is assigned a value or has its possibility list changed, it affects the possibility lists of all its neighbors. The other reason there is only a single strategy executed per iteration is because of how the strategies are prioritized. The simplest strategy is to locate cells with only one possible value and assign each one their sole possible value; the other strategies are performed when the agent cannot locate such cells. In this way, the agent function executes the simplest strategies as often as possible, maximizing efficiency and minimizing the chance of an error. In fact, simple enough puzzles can be solved only using the Only Choice and Single Possibility rules, the optimal set of decisions for an agent when solving those puzzles.

## *Context and Strategy Algorithms*

### *Only Choice Rule*

**Explanation:** The agent searches for a list of cell groups that contain exactly one open cell. If such a list is found, the agent goes through each group and fills the cell with its sole remaining value (n.d.-a 2-3, n.d.-b).

**Context Algorithm(out List<CellGroup> otcgroups):**

// *Algorithm uses a LINQ query on the puzzle to select all the cell groups who have exactly one open cell.*

List<CellGroup> onlyChoiceCellGroups = puzzle.GetUnifiedCellGroups().Select(y => y).Where(z => z.GetRemainingFills().Count == 1).ToList();

//  *The algorithm sets the outgoing parameter as the list of cell groups found (which if there are none, will be a null value), while the context returns a boolean on whether the puzzle has at least one group that qualifies.*

otcgroups = onlyChoiceCellGroups;

return (onlyChoiceCellGroups != null && onlyChoiceCellGroups.Count >= 1);

**Strategy Algorithm(List<CellGroup> params):**

// *For each cell group in the parameters, fill its empty cell with the remaining value.*

foreach CellGroup cg in params:

SudokuCell pnlyCell = cg.GetOpenMembers().FirstOrDefault(); onlyCell.fillCell(onlyCell.Possibilities.FirstOrDefault());

### *Single Possibility Rule*

**Explanation:** The agent searches for a list of all cells on the board that only have one possibility. If this list contains at least cell, the agent then fills each cell in the list with its sole possible value. It differs from the *Only Choice Rule* in that the cells returned do not have to be the only open members in their row, column or subgrid (n.d.-a 2-3, n.d.-c).

**Context Algorithm(out List<SudokuCell> outcells):**

// *The algorithm uses a LINQ query to grab all the cells on the board with exactly one possibility.*

List<SudokuCell> singlePossCells = puzzle.CellColl.Select(y => y).Where(z => z != null).Where(omega => omega.Possibilities?.Count == 1).ToList();

*// The algorithm sets the outgoing parameter to the list of cells found, and returns a boolean on whether or not the list of cells is not null and at least has one value inside.*

outcells = singlePossCells;

return (singlePossCells != null && singlePossCells.Count > 0);

**Strategy Algorithm(List<SudokuCell> params):**

foreach SudokuCell sc in params:

sc.fillCell(sc.Possibilities.FirstOrDefault());

### *Subgroup Exclusion Rule*

**Explanation:** For each number *i* in the domain *I*, the agent checks all the subgrids in the puzzle that contain *i* as a remaining value. For each of these subgrids, check through each of the cells that contain *i* as a possibility. If the only cells in the current cell’s row or column that have *i* as a possibility are only found in the current subgrid, then a new *SGUnitIntersectKey* is created. An *SGUnitIntersectKey* is a structure that has the following fields:

* intersectValue: A value *i* from *I*.
* subgrid: An open subgrid from the puzzle that contains intersectValue as an unused fill value.
* intersectCells: A list of cells where the only cells in their row or column with intersectValue as a possibility are also in the subgrid. Thus it is a list of subgrid intersections.
* elimCells: A list of cells in the subgrid where each entry has at least one neighbor in their row and column that is not located in the subgrid and has intersectValue as a possibility.

Once a list of *SGUIntersectKeys* is made, the agent goes through each item in the list and adds the *SGUintersectKey*’s intersectValue to all of the cells in elimCells entries in the DiscardedValuesTable, thus eliminating it from their lists of possible values (n.d.-a 2-3, n.d.-d).

**Context Algorithm(out List<SGUnitIntersectKey> sgintersections):**

foreach *i* ϵ *I*

List<CellGroup> subgridCands = puzzle.GetOpenSubgrids().Where(osg => osg.GetRemainingFills().Contains(i)).ToList();

if (subgridCands != null && subgridCands.Count > 0){

foreach (CellGroup sgrid in subgridCands)

List<SudokuCell> sgMems = sgrid.GetOpenMembers.Where(sgmem => sgmem.Possibilities.Contains(i)).ToList();

if (sgMems != null && sgMems.Count > 0){

List<SudokuCell> sgExclusivePossCells = sgMems.Where(sgc => SoloSGNs(sgc,i)).ToList();

List<SudokuCell> nonExclusiveCells = sgMems.Where(sgc => !SoloSGNs(sgc,i)).ToList();

if (sgExclusivePossCells.Count >= 1)

sgIntersections.Add(new SGUnitIntersectKey(sgrid, sgExclusivePossCells, nonExclusiveCells, i));

}

}

}

}

sgIntersections = sgIntersections.Where(a => a.elimCells != null && a.elimCells.Count > 0).ToList();

return sgIntersections != null && sgIntersections.Count > 0;

private bool SoloSGs(SudokuCell scell, int fN){

return (scell.GetNeighborsWithFillValue(fN, UnitType.Row).Select(s => s.sgNumber).Distinct().Count() == 1) || scell.GetNeighborsWithFIllValue(fN, UnitType.Column).Select(s => s.sgNumber).Distinct().Count() == 1);

}

**Strategy Algorithm(List<SGUnitIntersectKey> params):**

foreach (SGUnitIntersectKey sgKey in params){

CellGroup targetGroup = sgKey.subgrid;

List<SudokuCell> exclusionTargets = sgKey.elimCells();

if (exclusionTargets != null && exclusionTargets.Count > 0){

foreach(SudokuCell exTarget in exclusionTargets){ sudoku.DiscardedValuesTable.AddDiscardedValue(exTarget,sgKey.intersectValue);

}

}

}

### *Naked Twin Exclusion Rule*

**Explanation:** The agent searches for all cells that are open and have at least two open neighbors. For each cell in this list, the agent gets all of the cell’s neighbors that has exactly two possible values and then compares its possibility set to each one. For each one with an identical possibility set, a new TwinNode is created and added to a list. A TwinNode is a data structure with the following fields:

* twinCells: The pair of cells with identical possibility sets (e.g the cells that make up the naked twin).
* possSet: A hash set of integers that contains the possibility values of the twins.

Each naked twin is stored in a special collection known as a TwinNodesCollection, which contains two lists: a list of all the TwinNodes (called twinNodes) and a list of the cells that appear in any of the naked twins (called keyNodes). The agent goes through each twin that has at least one neighboring cell that shares a value with the twin’s possSet and for each of these neighbors, adds the intersect of the possSet and the neighbor’s possibilities to the neighbor’s entry in the DiscardedValuesTable (n.d.-a 2-3, n.d.-e).

**Context Algorithm(out TwinNodesCollection ntwins):**

TwinNodesCollection twinNodesCollection = new TwinNodesCollection();

var potTwins = from c in puzzle.cellColl

where !c.isFilled && c.neighbors != null && c.neighbors.Where(n => !n.isFilled).Count() > 2

select c;

potTwins = potTwins.ToList();

foreach (SudokuCell potTwin in potTwins ?? new List<SudokuCell>()){

List<SudokuCell> a = potTwin.neighbors.Where(acell => !acell.isFilled && acell.Possibilities.Count == 2).ToList();

foreach(SudokuCell ac in a ?? new List<SudokuCell>()){

HashSet<int> hashA = new HashSet<int>(potTwin.Possibilities);

HashSet<int> hashB = new HashSet<int>(ac.Possibilities);

if (hashA.setEquals(hashB)){

TwinNode tNode = new TwinNode(potTwin, ac, hashA, TwinEnum.Naked);

twinNodesCollection.Add(tNode);

}

}

}

ntwins = twinNodesCollection;

return twinNodesCollection.TwinNodes.Count > 0 && CheckContextTwo(twinNodesCollection);

private bool CheckContextTwo(TwinNodesCollection twins) {

foreach(TwinNode nTwin in twins.TwinNodes){

List<SudokuCell> universalNeighbors = nTwin.keyA.neighbors.Union(nTwin.keyB.neighbors).Except(new List<SudokuCell>(){nTwin.keyA, nTwin.keyB }).ToList();

List<SudokuCell> debugNeighbors = universalNeighbors.Select(h => h).Where(uf => uf.Possibilities != null && uf.Possibilities.Intersect(nTwin.possSet) != null && uf.Possibilities.Intersect(nTwin.possSet).Count() > 0).ToList();

if (debugNeighbors.Count > 0)

return true;

}

return false;

}

**Strategy Algorithm(TwinNodesCollection params):**

if (params != null && params.Count > 0){

foreach(TwinNode nTwin in params.TwinNodes){

List<SudokuCell> universalNeighbors = nTwin.keyA.neighbors.Union(nTwin.keyB.neighbors).Except(new List<SudokuCell>(){nTwin.keyA, nTwin.keyB}).ToList();

List<SudokuCell> debugNeighbors = universalNeighbors.Select(h => h).Where(uf => uf.Possibilities != null && uf.Possibilities.Intersect(nTwin.possSet)!= null && uf.Possibilities.Intersect(nTwin.possSet).Count() > 0).ToList();

foreach(SudokuCell neigh in universalNeighbors ?? new List<SudokuCell>()){

if (params.ConfirmTwinKey(neigh)){

List<TwinNode> nakedTwins = params.GetTwinNodesWithKey(neigh);

var untouchs = nakedTwins.SelectMany(neightw.possSet).Distinct();

HashSet<int> neighHash = new HashSet<int>(untouchs);

List<int> touchables = nTwin.possSet.Except(neighHash).ToList();

if (touchables != null && touchables.Count > 0)

sudoku.discardedValuesTable.AddDiscardedValues(neigh, touchables);

else {

if (neigh.Possibilities != null && neigh.Possibilities.Intersect(nTwin.possSet).Distinct() != null)

sudoku.discardedValuesTable.AddDiscardedValues(neigh, neigh.Possibilities.Intersect(nTwin.possSet).Distinct().ToList());

}

}

}

}

}

### *Hidden Twin Exclusion Rule*

**Explanation:** The agent searches for pairs of cells that are in the same cell group such that the intersect of both of their possibility set is equal to one of their possibility sets, and the only cells that the values are possibilities of in the group are the two cells. The agent then goes through all such hidden twins and takes the cell with the larger possibility set, adding all the values that are not in the twins’ intersect to its entry in the DiscardedValuesTable (n.d.-a 2-3, n.d.-f).

The algorithm was unable to be debugged into a usable condition in time. For this reason, while it has a context and strategy class as well as a corresponding percept, the strategy is not called in the current build. It is, however, a good example of a strategy that’s needed to solve difficult Sudoku puzzles.

# Results

As of this writing, the agent is able to successfully read in all puzzles and can consistently solve puzzles of easy to moderate difficulty. The only strategy that has not been fully debugged is the Hidden Twin Exclusion Rule, which means that puzzles that require it currently cannot be solved.

There is a good deal of room for expanding the program. In addition to finishing debugging and implementing the Hidden Twin Exclusion Rule, the Swordfish (n.d.-a 21-22) and X-Wing (n.d.-a 19-20) rules can be added to the agent, allowing it to solve the most difficult puzzles. Additionally, while the actions of the agent are recorded on the console window, a graphical interface would be a significant boon, especially one that updates as the agent continues making decisions.

# 

# References and Sources

Kleinberg, Jon, and Éva Tardos. *Algorithm Design*. 1st ed., Pearson Education, Inc., 2006.

Taalman, Laura. “Taking Sudoku Seriously.” *Math Horizons*, vol. 15, no. 1, 2007, pp. 5–9. *JSTOR*, JSTOR, www.jstor.org/stable/25678701.

Provan, J. Scott. “Sudoku: Strategy versus Structure.” Vol. 116, no. 8, 2009, pp. 702–707., www.jstor.org/stable/40391196. Accessed 24 Oct. 2017.

Russell, Stuart J., and Peter Norvig. *Artificial Intelligence: a Modern Approach*. 3rd ed., Prentice-Hall, 2010.

Sarcar, Vaskaran. *Java Design Patterns : A Tour of 23 Gang of Four Design Patterns in Java*. Berkeley, California]: Apress, 2016. Expert's Voice in Java. Web.

Nagel, Christian, Jay Glynn, and Morgan Skinner. *Professional C# 5.0 and . Net 4. 5. 1*. Somerset: John Wiley & Sons, Incorporated, 2014. Web.

(n.d.-a) “Sudoku Strategy”. *Sudoku Dragon*, Siluian Software Limited, [www.sudokudragon.com/sudokustrategy.htm](http://www.sudokudragon.com/sudokustrategy.htm). Web.

(n.d.-b) “Using the 'Only Choice' puzzle solving strategy.” *Sudoku Dragon*, Siluian Software Limited, [www.sudokudragon.com/tutorialonlychoice.htm](http://www.sudokudragon.com/tutorialonlychoice.htm). Web.

(n.d.-c) “Using the ‘Single Possibility’ puzzle solving strategy.” *Sudoku Dragon*, Siluian Software Limited, [www.sudokudragon.com/tutorialsingleposs.htm](http://www.sudokudragon.com/tutorialsingleposs.htm). Web.

(n.d.-d) “Using the ‘Shared Subgroup’ puzzle solving strategy.” *Sudoku Dragon*, Siluian Software Limited, [www.sudokudragon.com/tutorialsharedsubgroup.htm](http://www.sudokudragon.com/tutorialsharedsubgroup.htm). Web.

(n.d.-e) “Using the ‘Naked Twin Exclusion’ puzzle solving strategy. *Sudoku Dragon*, Siluian Software Limited, [www.sudokudragon.com/tutorialnakedtwins.htm](http://www.sudokudragon.com/tutorialnakedtwins.htm). Web.

(n.d.-f) “Using the ‘Hidden Twin Exclusion’ puzzle solving strategy. *Sudoku Dragon*, Siluian Software Limited, [www.sudokudragon.com/tutorialhiddentwins.htm](http://www.sudokudragon.com/tutorialhiddentwins.htm). Web.

*Super Sudoku: A Mind-Boggling Medley of Sensational Sudoku Puzzles*. Modern Publishing, Unisystems.Inc, 2005.