1 Introduction

Sudoku (a subset of Latin Squares) has become a widely popular game in recent years. A Sudoku puzzle has a simple set of logical rules, yet can require quite a bit of reasoning to solve. We'll see that the game can be formulated as a few other well-defined problems and thus has a fair range of applications.

1.1 Definitions

1.1.1 Latin Squares

A $n \times n$ array which is filled with n distinct symbols (or colors). Each symbol $s \in n$ must appear in each row exactly once and in each column exactly once.

1	2	3
3	1	2
2	3	1

Figure 1: A valid $n = 3$ Latin Square

1	2	3
3	2	1
2	3	1

Figure 2: An invalid $n = 3$ Latin Square

1.1.2 Sudoku

A filled Sudoku array is a valid Latin Square. The set of all valid Sudoku arrays is a subset of all valid Latin Square arrays. The Sudoku array is an $n \times n$ grid with n subarrays of size $\sqrt{n} \times \sqrt{n}$. Like Latin Squares, there are n distinct symbols used to fill the array. In addition to the Latin Squares rules, each symbol $s \in n$ must appear in each subarray exactly once.

(Alternatively, some phrase the array as $n^2 \times n^2$ with n^2 symbols and subarrays of $n \times n$. We will use the former definition to stress that Sudoku is also a Latin Square.)

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

Figure 3: An example $n = 9$ Sudoku

1.1.3 Puzzle

A Latin Squares or Sudoku *puzzle* is a given as a partially filled Latin Squares or Sudoku array. Typically for Sudoku, this is an array with $n = 9$.

To be considered a valid puzzle, the array should have the following properties:

- There should be a solution.
- The solution should be unique.
- The solution should be obtainable through the use of logic (no guessing).

In other words, the solution to the puzzle should not require guessing and backtracking (it would be too tedious to be fun if it did!).

For Sudoku, the number of initially filled squares, k, is typically in the range of $17 - 30$. Puzzles with $k = 17$ are considered extremely difficult while those with $k = 30$ are quite easy.

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

Figure 4: The 'Ocean 17' Sudoku.[1]

1.2 Graph Coloring

Graph Coloring is a form of graph labeling, where no two adjacent vertices have the same color. Graph coloring algorithms are algorithms that take a graph and label the vertices such that adjacent vertices don't have the same color. Graph coloring is a problem applicable, and important, to many domains some of which we will discuss today, and some we won't (coloring countries or states on a map). There has been extensive research into graph coloring, and there exists a lot of theory behind it. Much of it is beyond the scope of this lecture, but we will discuss the applicable areas.

[1] Created by user Oceanh on Wikipedia

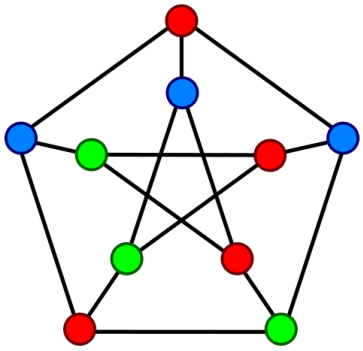

Figure 5: A graph that has been 3 colored.[2]

Latin Square, and thus Sudoku as well, can be formulated as a graph coloring problem. We simply need to define what the vertices are, and how they are connected. In this case, the squares are the nodes, they are not, however, connected to their immediate neighbors. Each square, in the Latin Square problem has an edge to every other square in its row and column. In the case of Sudoku, each square is also connected all of the squares in its $\sqrt{n} \times \sqrt{n}$ subarray.

1.3 Constraint Satisfaction Problems (CSP)

Mathematical problems whose solutions must satisfy a certain set of constraints are called constraint satisfaction problems. Sudoku and Latin Squares are both CSPs. In Latin Squares, all elements in a row or column must have a different color. Additionally, each cell must be colored, i.e. you cannot leave a cell uncolored. Sudoku imposes the additional constraint that boxes must also have n elements, each with different colors. Completing partially filled squares is a CSP because the solution must meet the above constraints.

1.4 NP-Completeness

The problem of filling in a Latin Square, Sudoku or otherwise can be shown to be NP-Complete. We will demonstrate later that you can reduce Latin Squares to a SAT problem which is NP-Complete, thus making this problem NP-Complete [4].

1.5 Fun Facts

- There are TONS of Latin Squares. For a given n, the number of Latin Squares, $L(n)$ is given by: $(n!^{2n})/(n^{n^2}) \leq L(n) \leq \prod_{k=1}^{n}(k!)^{n/k}$ [13].
- It has recently been shown that the smallest number of givens needed to solve a 9×9 Sudoku puzzle is 17 [11].

[2]Created by user `Booyabazooka` on Wikipedia

1.6 Scope of this Lecture

In order to keep this lecture concise and coherent were are limiting its scope. Our primary concern is to cover the problem of filling in a partially specified Latin Square or Sudoku. Additionally, we will focus on examples where the partially filled solution leads to only a single unique solution. (See the definition of *puzzle* in 1.1.3.)

2 Applications

Sudoku and Latin Squares have some interesting mathematical properties. Additionally, they may be phrased as constraint solving problems or encoded as boolean satisfiability problems. Thus, algorithms which solve these puzzles have a wide range of application. Some of these are discussed below (excluding "killing time").

2.1 Design of Experiments

It is very likely that when designing an experiment, you will have several variables to consider. In order to isolate the effects of each variable and avoid testing every combination of all variables, it is possible to use Latin Squares to figure out how to combine various variables.

2.2 Error Correcting Codes (ECC)

One interesting application of Latin Squares is in error correction codes when transmitting broadband internet. They can be used to encode symbols in a unique way that is resistant to noise during transmission.

2.2.1 Example

$$\begin{array}{c|cccc} A & 1 & 2 & 3 & 4 \\ B & 2 & 1 & 4 & 3 \\ C & 3 & 4 & 1 & 2 \\ D & 4 & 3 & 2 & 1 \end{array} \quad \begin{array}{c|cccc} E & 1 & 3 & 4 & 2 \\ F & 2 & 4 & 3 & 1 \\ G & 3 & 1 & 2 & 4 \\ H & 4 & 2 & 1 & 3 \end{array} \quad \begin{array}{c|cccc} I & 1 & 4 & 2 & 3 \\ J & 2 & 3 & 1 & 4 \\ K & 3 & 2 & 4 & 1 \\ L & 4 & 1 & 3 & 2 \end{array}$$

Figure 6: A set of encodings for transmitting characters over a broadband network.[3]

Imagine we had a noisy broadband network with 4 frequencies. To send an A, we first send a signal on frequency 1, then 2, 3 and 4 in successive timeslots. Say there was noise, and that frequencies 1 & 2 were received in both of the first and second slots, but frequencies 3 and 4 were received in their correct timeslot. As the receiver, we don't know if the first two slots were supposed to be frequency (1,2), (2,1), (1,1), or (2,2).

[3] http://en.wikipedia.org/wiki/Latin_square#Error_correcting_codes

However, because of the way we encode the symbols we can know that it was (1,2) because it produces the only valid character.

2.3 Register Allocation

Register allocation can be framed and solved as a graph coloring problem. When compiling a program, and deciding which registers to allocate to which variables, you need to consider which other variables need to be used at the same time. This translates naturally to a graph, where variables are vertices and they are connected if they need to be used at the same time. You can then k-color the graph, where k is the number of registers, to determine register assignment.

2.4 Scheduling

Scheduling is another example of a graph coloring problem. In scheduling you have a set of jobs that need to be completed, at each takes a certain amount of time, and some sort of facility that completes the jobs. A good example of this is assigning classes to classrooms. Classes take up a certain amount of time, and classrooms can only be used by one class at a time. Similar to register allocation, classes would be vertices and they would be connected if their timing conflicts. You then color the graph to assign rooms.

3 Complexity

As we've mentioned, the problem of determining whether a Latin Square has a valid solution is NP-Complete. Here we'll use a small example to demonstrate the proof visually.

We'll use the partial Latin Square below:

1		3
3		2
	3	1

Figure 7: A simple Latin Square

First, we need some definitions for the graph $G = (V, E)$:

Tripartite: A graph G is *tripartite* if its vertices may be divided into three independent sets V_1, V_2, V_3 such that no vertex $v \in V_i$ neighbors any other vertex $u \in V_i$. That is, for all $v, u \in V_i$, the edge (v, u) does not exist.

Triangular Partition: A *triangular partition* of graph G is the division of E into sets of three which form a triangle of vertices.

Uniform: A tripartite graph is *uniform* if each $v \in V_i$ has an equal number of neighbors in V_j and V_k (where $i \neq j \neq k$).

Defect: The *defect* of a partial Latin Square P is a graph $G(P)$ with the following vertices:

- A vertex r_i for each row in P which contains at least one empty cell.
- A vertex c_j for each column in P which contains at least one empty cell.
- A vertex e_k for each symbol in P which does not appear n times.

The defect $G(P)$ is then constructed by drawing the following edges:

- Edge (r_i, c_j) for each empty cell $(i, j) \in P$.
- Edge (r_i, e_k) for each row i which does not contain element e_k.
- Edge (c_j, e_k) for each column j which does not contain element e_k.

Latin Framework: A *Latin framework* is represented as $LF(G; r, s, t)$. LF is an $r \times s$ matrix where each cell contains elements from the set $\{1..t\}$, including $\emptyset$. Additionally:

- If $(r_i, c_j) \in G$ then $LF_{i,j} = \emptyset$.
- If $(r_i, e_k) \in G$ then $e_k \notin LF_{i,\{1..s\}}$
- If $(c_j, e_k) \in G$ then $e_k \notin LF_{\{1..r\},j}$

Intuitively, a partial Latin Square P has a solution if its defect is a tripartite graph with a triangular partition. The defect is mapping of row r_i and column c_j to an used symbol e_k. If the defect is *not* uniform, then P cannot be filled completely.

The proof of NP-completeness then, is a proof that every uniform tripartite graph G is the defect of some partial Latin Square.

Theorem 1. *The problem of determining if a tripartite graph has a triangle partition is NP-complete.* This is proved in [4].

Lemma 2. *For a graph G with $|V| = n$, there is a Latin framework $LF(G; n, n, 2n)$.* The proof is simply to build an LF from P using the rules from the definition of a Latin framework. Additionally, each cell contains the elements $1 + n + ((i + j)\%n)$. The Latin framework will then be $LF(G; n, n, 2n)$. □

$\{1, 3..6\}$	$\emptyset$	$\{1, 3, 4, 5\}$
$\{3, 4\}$	$\emptyset$	$\{2..6\}$
$\emptyset$	$\{3..6\}$	$\{1, 3, 4\}$

Figure 8: The Latin framework for the Latin Square, above

Lemma 3. *For $LF(G; a, b, c)$, $R(k)$ is the number of occurrences of $e_k \in LF$ plus the half degree of e_k in G. If $\forall k \in \{1..t\}.R(k) \geq a + b - c$, then LF can be modified to $LF'(G; a, b + 1, c)$ where the corresponding property holds for $R'(k) \geq a + (b + 1) - c$.* The proof is outlined in [4].

Lemme 4. *$LF(G; a, b, b)$ for a uniform tripartite G can be modified to $LF(G; b, b, b)$.* □

Theorem 5. *$LF(G; 2n, 2n, 2n)$ can be produced from a uniform tripartite graph G in polynomial time.* Starting with $LF(G; n, n, 2n)$, we produce $LF(G; n, 2n, 2n)$ and then similarly $LF(G; 2n, 2n, 2n)$. Each conversion may be performed in polynomial time by [7] in $O(n^{5/2})$. □

Theorem 6. *Deciding if a partial Latin Square P can be completed to a full Latin Square is NP-complete.* Converting P to a triangle partition of tripartite graph is NP-complete by **Theorem 1**. If the $|V| = n$ tripartite graph created from P is not uniform, then there is no solution. Otherwise, we can produce $LF(G; 2n, 2n, 2n)$ in polynomial time by **Theorem 5**. LF then, is a partial Latin Square. If LF can be completed, then so can P. Therefore the problem is reduced to showing that G has a tripartition. This holds because G was built as the *defect* of P. □

4 Summary of Algorithms

4.1 Backtracking

Backtracking is a general class of algorithms that can be applied to certain types of problems, with CSPs being a prominent example. The backtracking algorithm builds solutions to the problem incrementally. At each incremental step the validity of the partial solution is checked. If the partial solution is valid the algorithm progresses building the solution. Otherwise, the previous incremental change is undone and a new one is tried. In the case of finding a full solution, it can either be outputted, or the algorithm can continue if there is more than one valid full solution.

The most straightforward backtracking Sudoku algorithm begins with row 1, column 1. If the entry is not already filled, attempt to fill it with the number 1 and check if any of the constraints are violated. If not, move on to row 1, column 2. Otherwise, fill the entry with the number 2.

This type of algorithm is subject to adverse initial configurations. For example, the following puzzle's solution has the first row: 9, 8, 7, 6, 5, 4, 3, 2, 1:

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

Figure 9: A worst-case puzzle for a simple backtracking algorithm [4].

To avoid this type of attack from an adversary to the simple backtracking algorithm, entries may be chosen at random to be filled. Furthermore, the guess may be chosen at random.

[4]Created by user `lithiumflash` on Wikipedia

With or without randomness, this is an $O(n^{n^2})$ algorithm.

4.2 Logic Based

A logic-based Latin Squares or Sudoku algorithm will mimic what a human does as he or she solves a puzzle. For Sudoku, there are a few configurations to look for which will either

- eliminate a potential symbol from a square, or
- force a square to be a set of symbols.

The first step to a logic based approach would be to fill in the possible symbols that a box can be filled with using the three defining constraints of the puzzle. For each empty cell, start with all possible symbols and eliminate based on the row, column, and subarray. We'll refer to the list of possibilities for a given cell as its *pencilmarks*. Additionally, we'll refer to cells with only one possible symbol remaining as a *single*.

```
function InitPencilMarks(S[1..n][1..n])
    for all (r, c) ∈ (1..n, 1..n) do
        if S[r][c] = BLANK then
            SetPencilMarks(S[r][c], ALL_SYMBOLS)
            for all (r', c') ∈ the row-neighbors of r, c do
                if S[r'][c'] ≠ BLANK then
                    ErasePencilMark(S[r][c], S[r'][c'])
                end if
            end for
            for all (r', c') ∈ the column-neighbors of r, c do
                if S[r'][c'] ≠ BLANK then
                    ErasePencilMark(S[r][c], S[r'][c'])
                end if
            end for
            for all (r', c') ∈ the subarray-neighbors of r, c do
                if S[r'][c'] ≠ BLANK then
                    ErasePencilMark(S[r][c], S[r'][c'])
                end if
            end for
        end if
    end for
end function
```

The outer loop of this algorithm runs over every n^2 cell in the Sudoku array. The inner loop for rows and inner loop for columns each run over the $n-1$ neighboring elements in the row and column respectively. The inner loop for subarrays runs over the $\sqrt{n}-1$ neighboring cells in the subarray. In total, this algorithm runs in $O(n^3)$ time. An implementation may be optimized so as to not check each pair more than once, but as there are $O(n^3)$ neighboring pairs in a Sudoku grid, the asymptotic running time will remain the same.

In addition to the initialization, another trivial algorithm will fill in cells with only a single possibility remaining. Once filled, the symbol used must be removed from the pencil marks of neighboring cells.

```
function RemoveSingles(S[1..n][1..n])
    for all (r, c) ∈ (1..n, 1..n) do
        if S[r][c] = BLANK and NumPencilMarks(S[r][c]) = 1 then
            s ← SinglePencilMark(S[r][c])
            S[r][c] ← s
            for all (r′, c′) ∈ the row-neighbors of r, c do
                if S[r′][c′] = BLANK then
                    ErasePencilMark(S[r′][c′], s)
                end if
            end for
            for all (r′, c′) ∈ the column-neighbors of r, c do
                if S[r′][c′] ≠ BLANK then
                    ErasePencilMark(S[r′][c′], s)
                end if
            end for
            for all (r′, c′) ∈ the subarray-neighbors of r, c do
                if S[r′][c′] ≠ BLANK then
                    ErasePencilMark(S[r′][c′], s)
                end if
            end for
        end if
    end for
end function
```

At first glance, this is an $O(n^3)$ algorithm. However, for a Sudoku *puzzle* there will only be a small number of *singles* after each iteration. Therefore, if we assume the running time is dominated by updating the neighbors of a constant number of *singles* (less than 3, for example), then this is a $\Theta(n^2)$ algorithm. Each of the $\Theta(n^2)$ cells is visited, and an $\Theta(n)$ update is performed for a constant number of them.

Additionally, there may be *singles* due to the fact that no other neighboring cell has particular symbol s in its *pencil marks*.

```
function RemoveConstrainedSingles(S[1..n][1..n])
    for all (r, c) ∈ (1..n, 1..n) do
        for all s ∈ PencilMarks(S[r][c]) do
            for all (r′, c′) ∈ the row-neighbors of r, c do
                if s ∈ PencilMarks(S[r′][c′]) then
                    Continue to next symbol
                end if
            end for
            for all (r′, c′) ∈ the column-neighbors of r, c do
                if s ∈ PencilMarks(S[r′][c′]) then
                    Continue to next symbol
                end if
            end for
            for all (r′, c′) ∈ the subarray-neighbors of r, c do
                if s ∈ PencilMarks(S[r′][c′]) then
                    Continue to next symbol
                end if
```

```
            end for
            S[r][c] ← s
            Continue to next cell
        end for
    end for
end function
```

The RemoveConstrainedNeighbors algorithm runs in $O(n^4)$ time. For each of the n^2 cells, and each of the n potential pencil marks, we must visit each of the $O(n)$ neighbors. If a constrained single is found, the neighbors do not need to be updated (as before) because they necessarily did not have the symbol in their pencil marks to begin with. As with RemoveSingles a legitimate Sudoku puzzle will not have anywhere near n pencil marks per cells. Therefore the running time in practice will be much less than $O(n^4)$.

With initialization and *single* detection out of the way, a set of logical algorithms must be used. Given a well formulated Sudoku *puzzle*, one of the following algorithms *should* result in at least one *single*. That said, no one has proven a minimal set (or any set for that matter) of logical techniques which is guaranteed to solve any Sudoku [5].

The techniques below use a mixture of terminology (ie: pithy names) from [1] and [8].

4.2.1 Locked Candidates (1)

With this technique, if the pencil marks for a particular symbol only appear in one row (or column) of a subarray, then the symbol is removed from the pencil marks of the cells in row (or column) which are outside of the subarray.

In the example below, the symbol 7 (and 8,9) must appear in row 6 of the central subarray. Therefore 7 (and 8,9) should be removed from the pencil marks of all cells in row 6 which are outside of the central subarray.

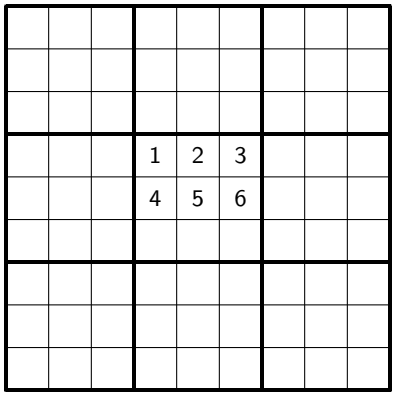

Figure 10: Locked Candidates (1) Example

[5] Or, at least our best efforts have not uncovered a guaranteed set of logical techniques

4.2.2 Locked Candidates (2)

Conversely, symbols outside of a subarray can lock the pencil marks within a subarray. If a symbol appears only in the pencil marks within a single subarray of a particular row (or column), then it may be removed from the marks of the other cells within the subarray.

In the example below, because the symbol 9 in column 4 can only appear somewhere in the central subarray, it should be removed from the pencil marks of columns 5 and 6 within the central subarray.

			1					
	9							
			3					
				3				
			4					
			5					
	9							

Figure 11: Locked Candidates (2) Example

4.2.3 Naked Pairs

Within a particular constraint (row, column, subarray), have exactly the same two symbols in their pencil marks (and only those two), then they can be removed from all other cells in the group.

In the example below, the cells $(5, 5)$ and $(5, 7)$ much each be filled with symbol 8 or symbol 9. Therefore, the pencil marks of the remaining cells in row 5 should be cleared of symbols 8 and 9.

				1		4		
				2		5		
				3				
				4		6		
1			2					3
				5		7		
				6		1		
				7		2		
						3		

Figure 12: Naked Pairs Example

4.2.4 Higher Order Naked Groups

Similar to naked pairs are naked triplets and naked quads. These follow the same principle as naked pairs with one exception. The pencil marks for each cell in the group do not need to contain *all* of the potential symbols in the group. They still, however, cannot contain any additional symbols.

4.2.5 Hidden Pairs and Higher Order Hidden Groups

A set of techniques which are similar to the naked groups are called hidden pairs, triplets, and quads. For example, if two cells within a particular group (row, column, or subarray) contain a pair of symbols which don't appear elsewhere in the group, then all other pencil mark symbols from the pair may be removed. Hidden pairs can then be extended to higher order hidden groups as we did for higher order naked groups.

4.3 Graph Coloring

For a graph coloring algorithm to work with Sudoku or Latin Squares, a fair amount of backtracking will be required. A general graph coloring algorithm called DSATUR is described in [3]. This algorithm uses the term *saturation* to describe the amount of possible colors which a vertex can still be (that is, the set of all possible colors minus the set of neighboring colors). A vertex with a high saturation is constrained to just a few possibilities.

In some applications, such as register allocation, graph coloring does not need to backtrack. Each physical register is assigned a unique color and each variable is considered a vertex. If the algorithm does not arrive a perfect coloring, then variables can be stored in memory.

Sudoku and Latin Squares, however, require a perfect coloring. One way to cut down on the amount of backtracking is to always color the most constrained vertices first.

4.4 Simulated Annealing

Sudoku may be posed as an optimization problem [9]. As such, the global optimum would be 0 entries which violate a constraint and can be solved using *simulated annealing* which we covered in Lecture 13. An annealing algorithm would work as follows:

function SudokuAnnealing($S[1..n][1..n], t_0, \alpha$)
  MarkFixed(S)
  repeat
    $rejects \leftarrow 0$
    $t \leftarrow t_0$
    $S \leftarrow$ RandomFillByBox(S)
    $ni \leftarrow$ NumIncorrect(S)

    while $ni \neq 0$ **do**
      $S' \leftarrow$ Neighbor(S)
      $ni' \leftarrow$ NumIncorrect(S')
      if Accept(ni, ni', t) **then**
        $S \leftarrow S'$
        $ni \leftarrow ni'$
      else
        $rejects \leftarrow rejects + 1$
        if $rejects = n^2$ **then**
          Break
        end if
      end if
      $t \leftarrow \alpha t$
    end while

  until $ni = 0$
end function

function Neighbor($S[1..n][1..n]$)
  repeat
    $i, j \leftarrow rand(1..n), rand(1..n)$
  until $S[i][j]$ is not fixed

  $r_a \leftarrow \lfloor \frac{i-1}{\sqrt{n}} \rfloor \times \sqrt{n} + 1$
  $r_b \leftarrow r_a + \sqrt{n} - 1$
  $c_a \leftarrow \lfloor \frac{j-1}{\sqrt{n}} \rfloor \times \sqrt{n} + 1$
  $c_b \leftarrow c_a + \sqrt{n} - 1$

  repeat
    $k, l \leftarrow rand(r_a..r_b), rand(c_a..c_b)$
  until $S[k][l]$ is not fixed **and** $(k, l) \neq (i, j)$

  swap($S[i][j], S[k][l]$)

end function

The initially empty cells in the Sudoku array are filled such that each subarray contains each of the n symbols. This invariant will hold so that only rows and columns need to be checked for violations to the Sudoku rules.

The NEIGHBOR function chooses two random cells in the same box — neither of which are fixed — and swaps them. This is a bit different than our Ising model annealing algorithm.

The ACCEPT function used in [9] is that from metropolis which we covered in Lecture 13; a neighbor is always accepted if it lowers the number of invalid cells, S. Otherwise, it is accepted with the probability

$$e^{\frac{|ni-ni'|}{t}}.$$

An $\alpha = 0.99$ was found to be acceptable with a $t_0 \approx 20$. Additionally, a reheat quirk was added to start over after too many successive rejects.

4.5 Boolean Satisfiability

By proving that Latin Squares reduces to Sudoku [14], and that Latin Squares is NP-Complete [4], we know that Sudoku can be encoded as a SAT problem [5]. In this encoding, each variable is of the form s_{xyz}. For example $s_{137} = \texttt{TRUE}$ means that the entry at row 1 and column 3 is 7. Alternatively, $s_{243} = \texttt{FALSE}$ means that the entry at row 2 and column 4 is *not* 3.

The minimum encoding [10] required to characterize Sudoku consists of the clauses:

- Each entry must contain at least one number ($n \in \{1..9\}$):

$$\bigwedge_{x=1}^{9} \bigwedge_{y=1}^{9} \bigvee_{z=1}^{9} s_{xyz}$$

- Each number (1..9) is unique to a row:

$$\bigwedge_{y=1}^{9} \bigwedge_{z=1}^{9} \bigwedge_{x=1}^{8} \bigwedge_{i=x+1}^{9} (s_{xyz} \oplus s_{iyz})$$

- Each number (1..9) is unique to a column:

$$\bigwedge_{x=1}^{9} \bigwedge_{z=1}^{9} \bigwedge_{y=1}^{8} \bigwedge_{i=y+1}^{9} (s_{xyz} \oplus s_{xiz})$$

- Each number (1..9) is unique to a 3x3 box:

$$\bigwedge_{z=1}^{9} \bigwedge_{i=0}^{2} \bigwedge_{j=0}^{2} \bigwedge_{x=1}^{3} \bigwedge_{y=1}^{3} \bigwedge_{k=y+1}^{3} \left(s_{(3i+x)(3j+y)z} \oplus s_{(3i+x)(3j+k)z}\right)$$

$$\bigwedge_{z=1}^{9}\bigwedge_{i=0}^{2}\bigwedge_{j=0}^{2}\bigwedge_{x=1}^{3}\bigwedge_{y=1}^{3}\bigwedge_{k=x+1}^{3}\bigwedge_{l=1}^{3}\left(s_{(3i+x)(3j+y)z} \oplus s_{(3i+k)(3j+l)z}\right)$$

A set of redundant clauses may also be added to the encoding [10]. In practice, these are required to solve difficult puzzles [10].

- Each entry contains at most one number ($n \in \{1..9\}$):

$$\bigwedge_{x=1}^{9}\bigwedge_{y=1}^{9}\bigwedge_{z=1}^{8}\bigwedge_{i=z+1}^{9}\left(s_{xyz} \oplus s_{xyi}\right)$$

- Each number (1..9) is in each row:

$$\bigwedge_{y=1}^{9}\bigwedge_{z=1}^{9}\bigvee_{x=1}^{9} s_{xyz}$$

- Each number (1..9) is in each column:

$$\bigwedge_{x=1}^{9}\bigwedge_{z=1}^{9}\bigvee_{y=1}^{9} s_{xyz}$$

- Each number (1..9) is in each 3x3 box:

$$\bigvee_{z=1}^{9}\bigwedge_{i=0}^{2}\bigwedge_{j=0}^{2}\bigwedge_{x=1}^{3}\bigwedge_{y=1}^{3} s_{(3i+x)(3j+y)z}$$

When using a SAT solver to find the solution to a Sudoku (or Latin Squares minus the 3x3 box clauses), the constraints given above are used in addition to the initially filled entries. For example, for the puzzle in Figure 4.1, $s_{263} =$ `TRUE`, $s_{288} =$ `TRUE`, ..., $s_{999} =$ `TRUE` would be necessary.

5 Backtracking

As mentioned before, backtracking is a general algorithm that can be used to solve constraint satisfaction problems. Here we are going to show how it applies to filling in a partially filled Latin Square with a single unique solution.

5.1 Naïve Solution

For the naive solution we use backtracking to come to a correct answer, but do not use any knowledge of the domain to speed things up.

5.1.1 Description

First, check to see if all squares are filled in, if so print out the solution and terminate, you are done. Otherwise for this algorithm, we will consider empty squares in an ordered fashion, say left to right then top to bottom. At each empty square, iterate over the n possible colors and assign them one at a time to the square. After each assignment, validate that it is a valid solution (it is unique in its row, column, and possibly box if you are solving a Sudoku). If it is a valid solution, recursively call your backtracking function with the value filled in. If this function returns to you, there is not a valid solution containing this assignment, so remove it.

5.1.2 Pseudocode

For simplicities sake, we will consider a generic Latin Square here, not a Sudoku.

```
function NaïveBacktracking(PartialLatinSquare)
    if FullSquare(PartialLatinSquare) then
        return PartialLatinSquare
        toFill = FindFirstEmpty(PartialLatinSquare)
        for c in Colors do
            toFill.Color = c
            if IsValid(toFill) then
                NaïveBacktracking(PartialLatinSquare)
            end if
            toFill.Color = NULL
        end for
    end if
end function

function FullSquare(PartialLatinSquare)
    for square in PartialLatinSquare do
        if square.Color == NULL then
            return False
        end if
    end for
    return True
end function

function IsValid(square)
    for x in square.Neighbors do
        if x.Color == square.Color then
            return False
        end if
    end for
    return True
end function

function FindFirstEmpty(PartialLatinSquare)
```

```
    for i in PartialLatinSquare.n do
        for j in PartialLatinSquare.n do
            if PartialLatinSquare[i][j].Color == NULL then
                return PartialLatinSquare[i][j]
            end if
        end for
    end for
end function
```

5.1.3 Correctness and Running Time

This algorithm enumerates every possible incremental solution to a partial Latin Square. These incremental solutions are thrown out if and only if there is no possible way they could lead to a valid Latin Square. Otherwise, the incremental solution is considered in full. Because we consider every possible incremental solution and check it for validity, we are guaranteed to come to a validly filled in Latin Square.

As stated before this problem is NP-Complete, and we are looking for an exact solution. Because of this, we consider on the order of every possible valid Latin Square solutions. The number of possible valid Latin Squares is $O(\prod_{k=1}^{n}(k!)^{n/k})$. More tightly, we have n colors and n^2 squares. Which make n^{n^2} combinations to check. At each combination we have to assign a color, $O(1)$, and check for validity. Assuming an edge list representation to keep track of the row and column neighbors, we have to consider $O(n)$ items when checking for validity, making this $O((n^2)^{n^2})$.

5.1.4 Short Example

	3	2	
	4	3	2
2	1	4	3
3	2	1	4

Table 1: Starting Position

1	3	2	
	4	3	2
2	1	4	3
3	2	1	4

Table 2: Trying a 1. Valid.

1	3	2	**1**
	4	3	2
2	1	4	3
3	2	1	4

Table 3: Moving on. Trying a 1. Invalid.

1	3	2	**2**
	4	3	2
2	1	4	3
3	2	1	4

Table 4: Trying a 2. Invalid.

1	3	2	**3**
	4	3	2
2	1	4	3
3	2	1	4

Table 5: Trying a 3. Invalid.

1	3	2	**4**
	4	3	2
2	1	4	3
3	2	1	4

Table 6: Trying a 4. Invalid. Backtracking!

2	3	2	
	4	3	2
2	1	4	3
3	2	1	4

Table 7: Backtracked. Trying a 2. Invalid

3	3	2	
	4	3	2
2	1	4	3
3	2	1	4

Table 8: Trying a 3. Invalid

4	3	2	
	4	3	2
2	1	4	3
3	2	1	4

Table 9: Trying a 4. Valid

4	3	2	**1**
	4	3	2
2	1	4	3
3	2	1	4

Table 10: Moving on. Trying a 1. Valid.

4	3	2	1
1	4	3	2
2	1	4	3
3	2	1	4

Table 11: Moving on. Trying a 1. Valid.

Done!

5.2 Smarter Backtracking

If we observe the problem more carefully there are a number of optimizations we can make some optimizations. One of the best things can do is change the order in which we consider squares. Rather then considering them blindly, we should consider them in descending order of the number of neighbors they have filled in. This will have a couple of benefits. First, we will discover squares that have only one potential solution, we can fill these in with certainty, and this will further constrain our search. Second, by considering squares with the most number of filled in neighbors first, we are considering the squares with the least number of possible colors first. These squares will have fewer choices to consider and will thus terminate quicker.

Unfortunately these heuristics to not provide an improvement on the asymptotic running time. However, in practice they can help a great deal.

References

[1] Techniques for Solving Sudoku. `www.sudokuoftheday.com/pages/techniques-overview.php`.

[2] Carlos Anstegui, Alvaro Val, Ivn Dot, Csar Fernndez, Felip Many, and Escuela Politcnica Superior. Modeling choices in quasigroup completion: Sat vs. csp. In *In Proceedings of the National Conference on Artificial Intelligence*, pages 137–142, 2004.

[3] Daniel Brélaz. New methods to color the vertices of a graph. *Commun. ACM*, 22(4):251–256, April 1979.

[4] Charles J. Colbourn. The complexity of completing partial latin squares. *Discrete Applied Mathematics*, 8(1):25 – 30, 1984.

[5] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP Completeness*. W. H. Freeman, January 1979.

[6] Carla P. Gomes and David Shmoys. Completing quasigroups or latin squares: A structured graph coloring problem. In *Computational Symposium on Graph Coloring Generalizations*, 2002.

[7] John E. Hopcraft and Richard M. Karp. An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs. *SIAM Journal on Computing*, 2:225–231, 1973.

[8] Angus Johnson. Solving Sudoku, 2005. `http://www.angusj.com/sudoku/hints.php`.

[9] Rhyd Lewis. Metaheuristics can solve sudoku puzzles. *Journal of Heuristics*, 13:387–401, 2007.

[10] Ins Lynce and Jol Ouaknine. Sudoku as a sat problem. In *Proceedings of the 9 th International Symposium on Artificial Intelligence and Mathematics, AIMATH 2006, Fort Lauderdale*. Springer, 2006.

[11] Gary McGuire, Bastian Tugemann, and Gilles Civario. There is no 16-clue sudoku: Solving the sudoku minimum number of clues problem. *CoRR*, abs/1201.0749, 2012.

[12] S. Simonis. Sudoku as a constraint problem. In *CP Workshop on Modeling and Reformulating Constraint Satisfaction Problems.*, pages 13–27, October 2005.

[13] J.H. Van Lint and R.M. Wilson. *A Course in Combinatorics*. Cambridge University Press, 1992.

[14] T. Yato and T. Seta. Complexity and completeness of finding another solution and its application to puzzles. In *Proceedings of the National Meeting of the Information Processing Society of Japan (IPSJ)*, 2002.