Answer Set Programming with Sudoku

GitHub Repository: https://github.com/ricardoaxelbaeza/CHOBS-Project

Alec Ortiz
Alejandro Hernandez
Ricardo Baeza
Steven Chan

Table of Contents

I. Abstract

Answer Set Programming is a one kind of declarative programming approach which is useful to solve mainly NP-hard search problems. The NP-hard problem we attempted to solve was the nine by nine Sudoku board. Unlike traditional programming approaches, we do not program a set of instructions, but rather define the problem and the search space to search from. The set of instructions the computer performs is handled by the clingo solver which can ground and solve search problems. As expected, the clingo solver is able to outperform any human, including Sudoku professionals.

Throughout the development of the Sudoku problem, the concept of answer set programming was used as it allowed for the use of a declarative programming approach. In using declarative programming, the programmer is able to describe the problem to the computer, and the computer solves the problem based on the programmer's description. This methodology was used in the language AnsProlog, where the Sudoku problem was able to be modeled (instantiating a 9x9 Sudoku board) and then defined based on the rules of Sudoku. For example, a Sudoku board cannot have repeating values in the same row, column, and subquare. These rules that describe Sudoku, are then explained to the computer, through the use of constraints.

The test cases shown throughout this paper, define the process of how the sudoku problem was solved with the methodology of answer set programming and through the language of AnsProlog. Furthermore, the process of additional Sudoku variants such as Diagonal Sudoku will also be presented throughout the paper.

II. Introduction

The objective of this project is to solve a nine by nine Sudoku board using Answer Set Programming. In case of unfamiliarity the ultimate objective of solving a Sudoku board is to populate the board with values ranging from one to nine, where they do not repeat for each nine rows, nine columns and nine subsquares. Answer Set Programming provides a different technique of solving problems where the solution can be described as a set rather than a program or its output. In layman's terms, we won't be solving the Sudoku board with algorithms or programs, rather defining constraints and allowing clingo to ground facts and generate answer sets. The main problem we tackled solving sudoku with AnsProlog, is how to model the game and its constraints. We took the approach of creating 81 cells, a cell defined as an object with a row, column, and value, then gave a set of constraints to govern how the cells could relate to each other. Defining the constraint relations differently allowed us to create different variants of Sudoku as well.

Considering the topics that are to be discussed in this report, we have found it helpful to clearly define some vocabulary, so as to minimize confusion and maximize consistency. Throughout this paper, we provide formal definitions to words for which it is necessary to understand. As you will see, these definitions are concise and accompanied with examples if appropriate.

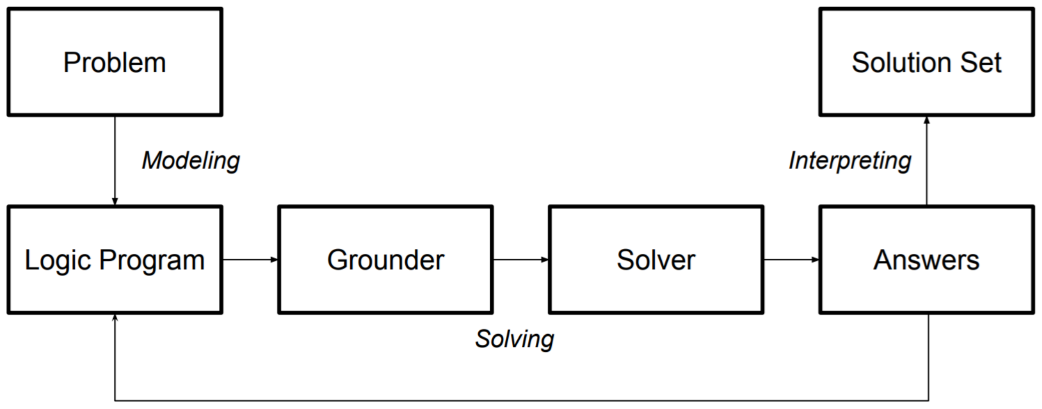
III. Methods/Concepts

The overarching methodology we have implemented to solve Sudoku is a problem-solving process known as *Answer Set Programming*. This strategy begins with a concrete understanding of the problem and follows a set procedure to output a desired solution. As you will see in this section, the two main tools we used to practice Answer Set Programming are *AnsProlog*, a declarative programming language, and *clingo*, a grounding and solving process implemented as a single system.

A. Answer Set Programming

Answer Set Programming (ASP) is a form of programming that uses a declarative programming approach; this essentially means a programmer describes what the problem is to a computer and the computer solves the problem based on the programmer's description. This form of programming is different from conventional programming methods (e.g. imperative programming with Java, Python, etc.) where a program is written as explicit instructions a computer follows to solve a problem. Below we discuss how ASP solves a problem, from start to finish.

1. The Problem Solving Process



Modeling the Problem. Just like other programming methodologies, Answer Set Programming is problem solving. The first step is to begin with a problem. The next step is to describe this problem to the computer. This description must be written, or encoded, so that a computer can understand it. For ASP, this description is often called a *model* of the problem and is written as a *logic program*. A logic program contains a description of the problem, but more importantly, it contains a description of the solution space. Ultimately, the computer will use the problem description to create instances of possible answers, from which it utilizes the solution space description to eliminate instances that do not match the solution descriptions.

Solving. The actual solving is a two-step process. First, the computer *grounds* the logic program. The logic program contains general descriptions of the problem; this grounding creates a single, specific instance of the problem. Second, the computer *solves* that specific instance of the problem by verifying that there are no contradictions with the description of valid solutions. If there are contradictions, that instance is excluded from the final set of solutions. The computer repeats the grounding and solving step until all instances have been reviewed. Together, this process is done with Grounder and Solver softwares.

Interpretation and Output. Finally, each solution is interpreted by the computer according to some specifications provided by the programmer and the set of solutions is outputted. This interpretation does not change the set of solutions, just format the output in some visual manner.

The remainder of this paper demonstrates the specific tools and techniques we used to implement this Answer Set Programming process. Following this method, we are able to model and solve Sudoku, as well as analyze the performance of our implementation.

B. Declarative Programming: AnsProlog

Recall that the second step of the Answer Set Programming procedure is to write a program that accurately models the logical constraints of the problem. The implemented logic programs for this project were written in the syntax of AnsProlog. Short for "Answer Set Programming in Logic", AnsProlog is a declarative programming language similar to Prolog. Before examining the encoded logic in our final AnsProlog programs, we must first cover the general syntax of this language.

Definition 1: A **Fact** is something that is true (often called valid) and stands as a logical proposition. Visually, they are denoted as lowercase and ending with a period.
EXAMPLE: Say we want to declare four people by their names. We can write the following AnsProlog:

```
mary.
lucia.
eddy.
sebastian.
```

Facts are necessary to define the logical truth of relationships between *predicates*.

Definition 2: A **Predicate** defines the relationship between a symbol and its fields. Predicates are denoted as such: *<symbol>(<fields...>)*.
EXAMPLE: We can define a relationship between our aforementioned people and their school year:

```
junior(mary).
freshman(lucia).
sophomore(eddy).
freshman(sebastian).
```

The kinds of relationships we can define with predicates are virtually limitless. Additionally, AnsProlog allows us to set fields as *variables*, which are not assigned concrete values.

Definition 3: **Variables** allow predicates to define relationships with unspecified fields.
EXAMPLE: As seen below, variables begin with a capital letter.

```
funny(Person).
tall(X).
athletic(Student.)
```

Variables are very useful when we create *rules*.

Definition 4: A **Rule** is used to define the logical relationships between predicates and/or facts. Rules are denoted as *<head> :- <body>*.
EXAMPLE: We can write rules to create valid instances of the following predicates:

```
popular(X) :- funny(X), smart(X)
popular(X) :- athletic(X).
likes(X, Y) :- tall(Y), handsome(Y).
```

A rule is read as <body> *implies* <head>. **One can understand the body as some condition(s) that once satisfied, imply the head as true.**

With these features, we can essentially encode any kind of logic with AnsProlog! It is important that you understand these concepts very well, as they are required to follow the intuition of rules with greater complexity- particularly the *Choice Rule* and *Integrity Constraints*. With these more complex rules, we are able to provide both a description of the Sudoku puzzle and how it is played to the computer- all within AnsProlog.

More information on AnsProlog syntax can be found in the Supplementary.

Recall that after the Solving process of ASP, we obtain Answers to generate a Solution Set. But what are Answers and Solution Sets? Below are definitions and examples of both, explained with AnsProlog syntax.

Definition 5: An **Answer** is a set of predicates that satisfy all logical constraints.
EXAMPLE: If we initialize the following predicates:

```
a(1). a(2). a(3).
```

And apply the following rule:

```
b(X) :- a(X), X > 1.
```

Then we can expect our Answer to be

```
Answer:        a(1). a(2). a(3). b(2). b(3).
```

Definition 6: A **Solution Set** is a set of unique Answers.
EXAMPLE: If we write an AnsProlog program that instantiates a single predicate

```
a(X).
```

so that 0 < X > 10, then we can expect our Solution Set to be

```
Solving...
Answer 1:       a(1).
Answer 2:       a(2).
. . .
Answer 9:        a(9).
SATISFIABLE
```

C. Grounding & Solving: Clingo

Once the logic program has been written, describing what the problem is, the next step entails a Grounder software, known as *gringo*, which is able to translate programs into propositional logic programs. After this translation, *CLASP*, an answer set Solver software, computes the answer set of the propositional logic program and outputs a solution. The solver that was used in this Sudoku project was *clingo*, a combination of gringo and CLASP. The reader should note however, that clingo offers more control over the grounding and solving process than gringo and CLASP can offer individually; an example of this would be incremental grounding and solving.

D. Complex AnsProlog Rules

At this point, it is possible to generate a Sudoku board with only facts and rules. However, the facts and rules need to be analyzed carefully to make sure the content is generated in the right combination. Instead, the *Choice Rule* and *Integrity Rule* are usually used to generate content. The Choice Rule is used to plan the design space while the Integrity rule is used to nullify parts of the unwanted design space.

Definition 7: A **Choice Rule** is used to specify the initial design space and allows AnsProlog to output multiple solutions. The rule allows the solver to arbitrarily choose what rules are created. The rule can be constrained with numerical values and implication.

EXAMPLE: Consider the following AnsProlog code.

```
store(1..3).
item(banana). item(orange). item(cherry). item(tomato).
1 {has(S,I) : item(I)} 4 :- store(S).     % Choice Rule
#show has/2.
```

In this example, the choice rule generates an arbitrary amount of *has(..)* predicates between 1-4 for each store. These *has(S,I)* facts can only include an item and nothing else. In short, each store has at least one item and at most four items.
With choice rules, AnsProlog programs can produce more numerous and complex solution sets. In this example, it outputs 3375 answers.

```
Solving...
Answer: 1
has(1,tomato) has(2,tomato) has(3,tomato)
Answer: 2
has(1,tomato) has(2,tomato) has(3,cherry)
...
Answer: 3373
has(1,banana) has(1,orange) has(1,cherry) has(2,banana) has(2,orange)
has(2,cherry) has(2,tomato) has(3,banana) has(3,orange) has(3,cherry)
has(3,tomato)
Answer: 3374
has(1,banana) has(1,orange) has(1,cherry) has(1,tomato) has(2,banana)
has(2,orange) has(2,tomato) has(3,banana) has(3,orange) has(3,cherry)
has(3,tomato)
Answer: 3375
has(1,banana) has(1,orange) has(1,cherry) has(1,tomato) has(2,banana)
has(2,orange) has(2,cherry) has(2,tomato) has(3,banana) has(3,orange)
has(3,cherry) has(3,tomato)
SATISFIABLE
```

A deeper explanation of Choice Rules can be found in the Supplementary.

Definition 8: An **Integrity Constraint** or **Integrity Rule** is a rule that outlines illegal conditions and removes them from the design space; denoted as "headless" rules. If the conditions of an Integrity Constraint are satisfied, the Answer containing the illegal conditions is removed from the Solution Set.
EXAMPLE: Considering the following AnsProlog code,

```
p(1..2).
q(1..2).
{s(P,Q): q(Q)} = 1 :- p(P).
#show s/2.
```

We can expect our Solution Set to be

```
Solving...
Answer 1:    s(1,1) s(1,2)
Answer 2:    s(1,1) s(2,1)
Answer 3:    s(1,2) s(2,1)
Answer 4:    s(1,1) s(2,2)
SATISFIABLE
```

Say that we wanted to exclude some of these Answers from our Solution Set. Imagine we consider all Answers that contain *s(X,X)* illegal, or nonvalid. There are many ways we could enforce this constraint, but one way is to subtract the two fields of the *s(..)* predicate and compare the difference to zero. If the difference is zero, we know that the entire Answer that contains the invalid *s(X,X)* predicate is invalid and is to be excluded from our Solution Set.

Integrity Constraints are how we enforce these kinds of exclusions. Below is an example of a rule that excludes all Answers with a *s(X,X)* predicate.

```
p(1..2).
q(1..2).
{s(P,Q): q(Q)} = 1 :- p(P).
:- s(P,Q), P-Q == 0.        % Integrity Constraint
#show s/2.
```

The rule can be read as, "If there is some predicate *s(..)* with fields *P* and *Q*, and *P - Q* = 0, that implies FALSE, or invalid."

Applying this constraint, we can expect our Solution Set to be

```
Solving...
Answer: 1    s(1,3) s(2,1)
SATISFIABLE
```

If there is still confusion on the process of ASP, how clingo works, or the function of complex AnsProlog rules, we encourage the reader to revisit those sections and any related section within the *Supplementary*.

IV. Experiments

Although there are many variations of the Sudoku puzzle, most adhere to a theme of unique number combinations, which are enforced as constraints on possible solutions (and therefore on the solver or player). Classical Sudoku has the least amount of constraints that must be obeyed and is perhaps then the simplest to solve. Logically then, although complex Sudoku game types may share constraints with simpler game types, they will surely have new constraints that increase difficulty by decreasing the number of possible solutions. In this section, we explore the multiple Sudoku variations for which logical constraints were defined and encoded. The encoding of these constraints are completed in the syntax of AnsProlog.

A. Universal Sudoku Constraints

There are three constraints that are universal to all Sudoku puzzles. The first two are that a Sudoku puzzle has nine columns and nine rows. This implies then that there are eighty-one spaces on any and all Sudoku boards. The third constraint is that each space on this board holds one character, out of nine possible options. By an overwhelming majority, these nine characters are often the integers 1 through 9; therefore, all Sudoku game types we encode in this project solve with these values.

To encode these constraints and model our Sudoku board in AnsProlog, we begin with declaring the constant n.

```
#const n = 9.
```

Next, we use three primitive facts to encode the nine rows, columns, and values:

```
row(1..n).
col(1..n).
num(1..n).
```

As explained in *Supplementary*, we can interpret these statements as instantiating the following predicates:

```
row(1). row(2). row(3). row(4). row(5). row(6). row(7). row(8).  row(9).
col(1). col(2). col(3). col(4). col(5). col(6). col(7). col(8). col(9).
num(1). num(2). num(3). num(4). num(5). num(6). num(7). num(8). num(9).
```

These universal constraints are necessary to encode as they are used to describe our eighty-one spaces on the Sudoku board. As you will see in the next section, we will use a valid row(R), col(C), and num(V) to create an instance of the predicate *sudoku(R,C,V),* which represents one space on the Sudoku board with a valid row, column, and number.

B. Classical Sudoku

We modelled the rules and logic of a Classical Sudoku puzzle in multiple steps. First, we populated a very large number of boards to create a collection of all possible Sudoku boards. Next, we enforced unique rows, unique columns, and unique subsquares, meaning any boards from our collection that did not adhere to our constraints were removed from our collection to leave us with a final Solution Set of boards that satisfied all the rules of Classical Sudoku.

1. Populating the Board

To populate the board with possible candidates we implement a Choice Rule, seen below.

```
{sudoku(R,C,V): num(V)} == 1  :- row(R), col(C).
```

This rule defines the candidates as "sudokus" which all have three values: a row, a column, and a number. This rule reads as follows, "for a given row R and a given column C, create a predicate called sudoku that has three properties, a row, a column and a number. Set the values of that predicate row and column to R and C respectively, then assign one value to the number predicate." It is important to identify the number 1 in this choice rule. This ensures that each position from (1, 1) to (9, 9) has one and only one value. After execution, this choice rule generates a VLN (very large number) of answer sets with 81 *sudoku(..)* predicates each. The following rules operate on this very large solution set and begin to prune out answer sets with predicates that conflict with the rules of the classical sudoku game.

2. Unique Rows

```
:- sudoku(R,C,V), sudoku(R',C,V),  R != R'.
```

In this constraint, we have four different variables, we have R (represents a row value), C (a column value), V(a number value) and R'(a row value that cannot be the same as R). So, in further understanding this constraint: "if we have a valid instance of *sudoku(R, C, V)*, where we have a value from one to nine for R, C and V (it does not matter if the values repeat), then we move on to the next part of the constraint. Now, if we have a valid instance of *sudoku(R', C, V)* where the values of C, V and R' fall within the range of one through nine, then we will move onto the next part. Finally, we will then check if R and R' (most important part), are values that are not equal in value, if they are not equal, then the whole constraint is true, meaning that it will not be included into the final Sudoku solution". In essence, what this translates to is that if there is a situation where a value is in two different rows, but that value is shared in the same column, then it does not satisfy the rules of Sudoku, so we do not include it into our final solution.

3. Unique Columns

Likewise, there is also another constraint used to get rid of repeating numbers in each column of the Sudoku board. The constraint used for this operation is

```
:- sudoku(R,C,V), sudoku(R,C',V),  C != C' .
```

The variables used in this constraint are R (represents a row value), C (a column value), V (a number value) and C'(another column value). The way this reads is very similar to the previous constraint mentioned: "if we have a valid instance of Sudoku, where we have a value from one to nine for R,C and V, then we move on. Now, we will be checking if we have a valid instance of Sudoku where the values for R, C' and V are from one to nine; if this holds true, then we will continue and verify that C is not equal to C'. If C and C' are not equal in value, then the whole constraint is valid meaning it will not be included into the solution." Take for example a situation where the values for R, V and C are the same for both instances of Sudoku; the only difference is C and C'. This means that there are two numbers (V) that are the same that are in the same row (R), but different columns(C and C'). So we have the same value repeating twice, in a row; this breaks the rule of Sudoku, therefore it is not included in the solution.

4. Unique Subsquares

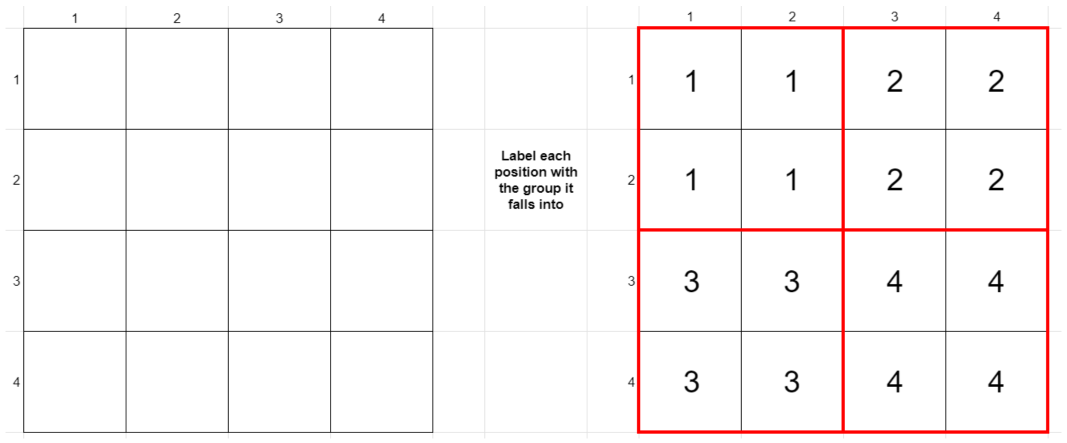
There is one more final step in solving Sudoku: the board must not contain repeating values in each subsquare. There are a couple of constraints that have to be implemented to solve this task; first we must declare the constant *g*, which is equal to $\sqrt{n}$ and represents the width/height of subsquares.

```
#const g = 3.
```

Next, we use a rule to instantiate 81 instances of the *group(..)* predicate. We know there will be exactly 81, because that is the number of unique combinations of row and column. Given a position, we are able to use some simple algebra to calculate which subsquare, or group, that position falls into.

```
group(R,C,G) :- row(R), col(C),
                D = n / g,
                X = (R - 1) / D,
                Y = (C - 1) / D,
                G = (Y *g) + X + 1.
```

Below is a diagram to display the function of this rule.



After this rule to create 81 *group(..)* predicates, we utilize an Integrity Constraint to enforce unique subsquares. Here is the Integrity Constraint we used:

```
:- sudoku(R,C,V), sudoku(R',C',V), group(R,C,G), group(R',C',G), (R,C) != (R',C').
```

The unique subsquares rule can be understood as violated when 3 conditions are met: first, we must have two spaces on our board with the same value, or number; second, we must have two *group(..)* predicates, with the rows and columns of the two spaces, that share the same G; finally, if these two spaces are unique from each other, we can verify the Answer they are in breaks the rule of unique subsquares. If any two *sudoku(..)* predicates in an Answer satisfy the conditions of this Integrity Constraint, that Answer is removed from the Solution Set.

The reader should note that the second condition to be met, that mentions *group(R,C,G)* and *group(R',C',G)*, does not instantiate any new predicates. Rather, that part references a predefined *group(..)* predicate. If the first predicate exists (which it always will) it assigns a value to the variable G. If the next predicate does not exist, because it exists in a different group, therefore has a different G value, the condition is not met and so the rule is ignored and the Answer in the Solution Set.

C. Diagonal Sudoku

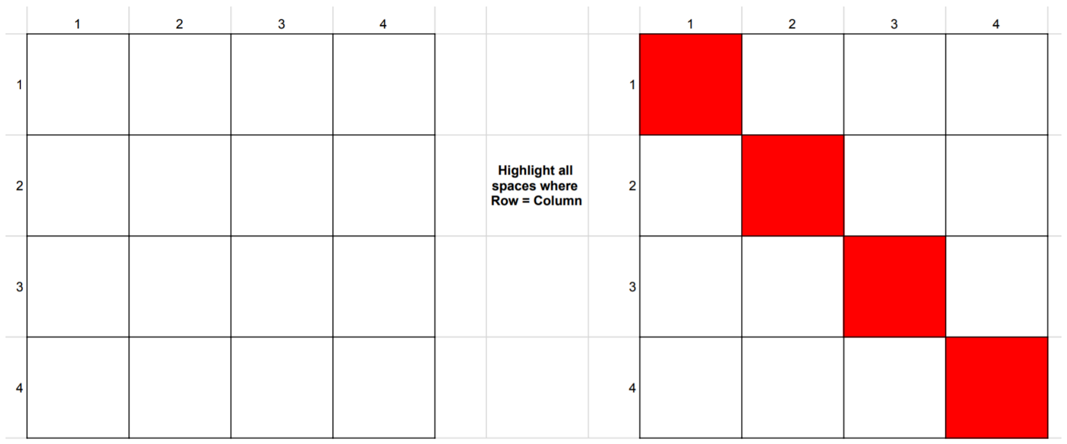
The logical constraints that encode the rules of Classic Sudoku are shared by Diagonal Sudoku: we use a Choice Rule to instantiate eighty-one instances of the *sudoku(R,C,V)* predicate and three Integrity Constraints to enforce unique rows, columns, and sub-squares. The additional constraints of Diagonal Sudoku are that both main diagonals of the board must also be unique; the values 1 through 9 must each have exactly one occurrence for both main diagonals. Similar to before, we utilize two Integrity Constraints to remove Answers from our Solution Set that do not adhere to unique main diagonals.

1. Unique A-Diagonal

To provide a clear definition: the A-diagonal is the collection of spaces on our board reaching from the top left corner to the bottom right corner. Here is the Integrity Constraint we use to enforce a unique A-diagonal:

```
:- sudoku(R,R,V), sudoku(C,C,V), R != C.
```

Understanding the intuition behind this constraint is quite simple. First, draw out a Sudoku puzzle and label rows and columns. Now clearly identify the A-diagonal, do you notice any relationship shared among all spaces in the A-diagonal? Hopefully it is clear that for each of these spaces, its row is equal to its column.



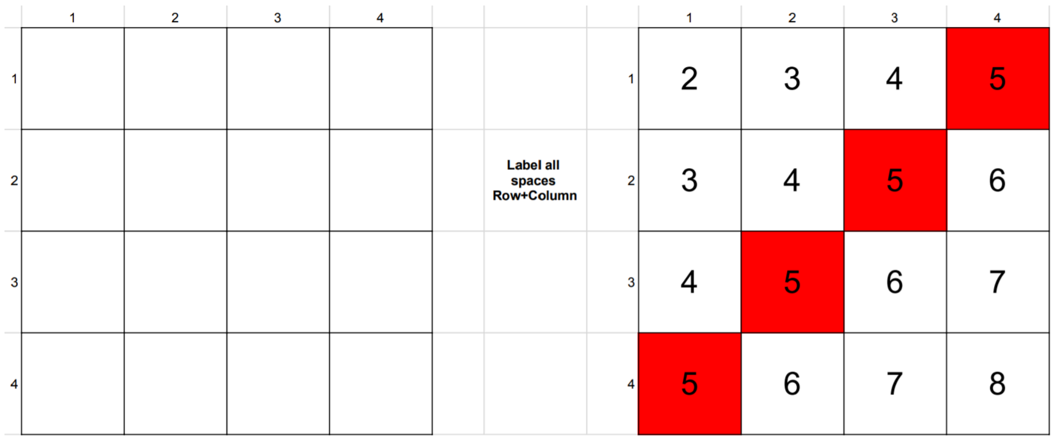
So how do we construct an Integrity Constraint that removes Answers with a non-unique A-diagonal from our Solution Set? We can understand this violation as occurring when three conditions are met: first, we must have one space on our board in the A-diagonal with the number V; second, we need another another space in the A-diagonal with the same number V; lastly, if these two spaces are at different positions, we can verify the Answer they are in breaks the constraint of a unique A-diagonal. If any two *sudoku(..)* predicates in an Answer satisfy the conditions of this Integrity Constraint, that Answer is removed from the Solution Set.

2. Unique B-Diagonal

To provide a clear definition: the B-diagonal is the collection of spaces on our board reaching from the bottom left corner to the top right corner. Here is the Integrity Constraint we use to enforce a unique B-diagonal:

```
:- sudoku(R,C,V), sudoku(R',C',V), R+C == n+1, R'+C' == n+1, (R,C) != (R',C').
```

Understanding the intuition behind this constraint is a bit more difficult First, draw out a Sudoku puzzle and label rows and columns. Now clearly identify the B-diagonal, do you notice any relationship shared among all spaces in the B-diagonal? If you label each space as the sum of its row and column, you will see that all spaces in the B-diagonal share the same value, equal to n+1.



	1	2	3	4
1				
2				
3				
4				

Label all spaces Row+Column

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

So how do we construct an Integrity Constraint that removes Answers with a non-unique B-diagonal from our Solution Set? Similar to before, we can understand this violation as occurring when three conditions are met: first, we must have two spaces on our board with the number V; second, we need confirm that both of these spaces occur on the B-diagonal, done with a simple comparison; lastly, if these two spaces are at different positions, we can verify the Answer they are in breaks the constraint of a unique B-diagonal. If any two *sudoku(..)* predicates in an Answer satisfy the conditions of this Integrity Constraint, that Answer is removed from the Solution Set.

D. Reflection on Solution Methods

The reader should keep in mind that the above rules demonstrate our *final* methods of encoding Sudoku. For example, our first attempted Choice Rule was the following:

```
{sudoku(R,C,V): row(R), col(C), num(V)} == 81  :- row(R), col(C).
```

Hopefully it is obvious that this rule and similar variations of it failed miserably.

Truthfully, the most complex task of this encoding process was understanding the particular features of AnsProlog and the general procedure of Answer Set Programming. Once a solid grasp on these two was achieved, the next challenge was attempting encoding through trial and error. Once the encoding produced valid solutions, it was in our interest to remove redundant constraints and format our program to maximize readability. The intuition of our final logic program, executed with the Choice Rule and Integrity Constraints, proves more than satisfiable. As you will see in the next section, our program successfully outputs valid solutions to multiple Sudoku puzzle-types.

V. Results

To visualize the results of these experiments with Classical and Diagonal Sudoku, we authored Python programs to parse the Solutions Set as a text file and organize Answers so as to visualize each as an individual board solution. These Python programs were then executed on a Jupyter Notebook for ease of use. The example demonstrated below assumes the Jupyter notebook cell was run.

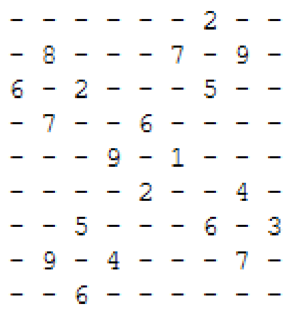
A. Classical Sudoku

The classic Sudoku rules as previously mentioned:

1. All rows and columns must have unique values ranging from one to nine.
2. Each three by three subsquare (nine in total) need to have unique values as well.

▸ Run [1]

```
# Set Example
!python ./tools/sudoku_problem_viz.py "./Sudoku Game Types/Classic
Sudoku/classic-sudoku-problem.lp"
```

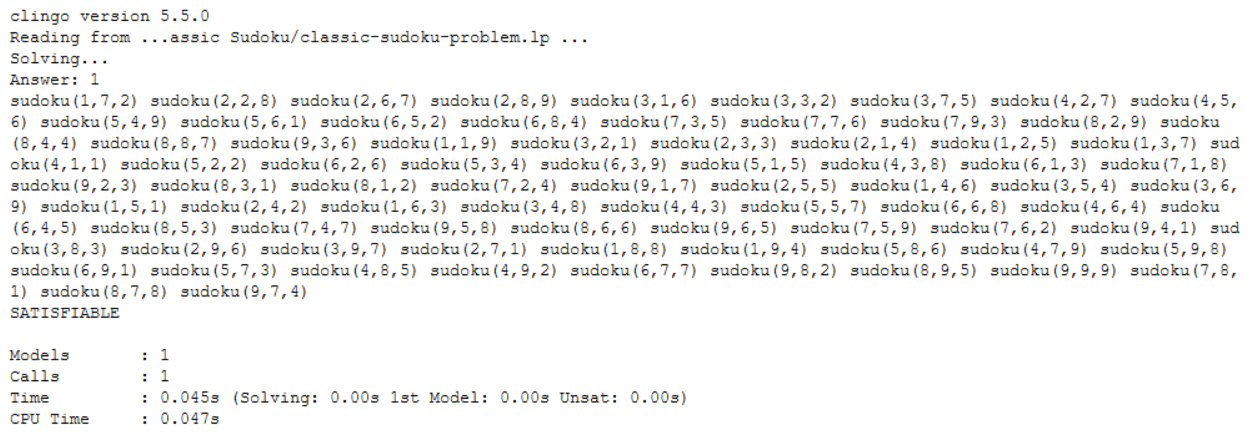


```
- - - - - - 2 - -
- 8 - - - 7 - 9 -
6 - 2 - - - 5 - -
- 7 - - 6 - - - -
- - - 9 - 1 - - -
- - - - 2 - - 4 -
- - 5 - - - 6 - 3
- 9 - 4 - - - 7 -
- - 6 - - - - - -
```

A pre-made Sudoku board is going to be solved by the clingo answer set solver. There is an example of a generated Sudoku board solution on the attached Jupyter notebook. In this example, the Jupyter notebook cell outputs a visual of the Sudoku problem at hand.

▸ Run [2]

```
# Run Solver
!clingo "./Sudoku Game Types/Classic Sudoku/classic-sudoku-problem.lp" "./Sudoku Game
Types/Classic Sudoku/classic-sudoku.lp" -n 0 > "./Sudoku Game Types/Classic
Sudoku/classic-solution.txt"
```

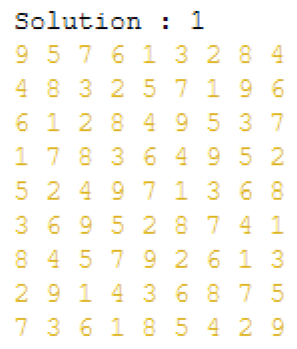


```
clingo version 5.5.0
Reading from ...assic Sudoku/classic-sudoku-problem.lp ...
Solving...
Answer: 1
sudoku(1,7,2) sudoku(2,2,8) sudoku(2,6,7) sudoku(2,8,9) sudoku(3,1,6) sudoku(3,3,2) sudoku(3,7,5) sudoku(4,2,7) sudoku(4,5,
6) sudoku(5,4,9) sudoku(5,6,1) sudoku(6,5,2) sudoku(6,8,4) sudoku(7,3,5) sudoku(7,7,6) sudoku(7,9,3) sudoku(8,2,9) sudoku
(8,4,4) sudoku(8,8,7) sudoku(9,3,6) sudoku(1,1,9) sudoku(3,2,1) sudoku(2,3,3) sudoku(2,1,4) sudoku(1,2,5) sudoku(1,3,7) sud
oku(4,1,1) sudoku(5,2,2) sudoku(6,2,6) sudoku(5,3,4) sudoku(6,3,9) sudoku(5,1,5) sudoku(4,3,8) sudoku(6,1,3) sudoku(7,1,8)
sudoku(9,2,3) sudoku(8,3,1) sudoku(8,1,2) sudoku(7,2,4) sudoku(9,1,7) sudoku(2,5,5) sudoku(1,4,6) sudoku(3,5,4) sudoku(3,6,
9) sudoku(1,5,1) sudoku(2,4,2) sudoku(1,6,3) sudoku(3,4,8) sudoku(4,4,3) sudoku(5,5,7) sudoku(6,6,8) sudoku(4,6,4) sudoku
(6,4,5) sudoku(8,5,3) sudoku(7,4,7) sudoku(9,5,8) sudoku(8,6,6) sudoku(9,6,5) sudoku(7,5,9) sudoku(7,6,2) sudoku(9,4,1) sud
oku(3,8,3) sudoku(2,9,6) sudoku(3,9,7) sudoku(2,7,1) sudoku(1,8,8) sudoku(1,9,4) sudoku(5,8,6) sudoku(4,7,9) sudoku(5,9,8)
sudoku(6,9,1) sudoku(5,7,3) sudoku(4,8,5) sudoku(4,9,2) sudoku(6,7,7) sudoku(9,8,2) sudoku(8,9,5) sudoku(9,9,9) sudoku(7,8,
1) sudoku(8,7,8) sudoku(9,7,4)
SATISFIABLE

Models       : 1
Calls        : 1
Time         : 0.045s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.047s
```

The clingo solver is called to find all of the solutions to the Sudoku board. In this case, only one solution was found and the values in the Sudoku board are outputted.

▸ Run [3]

```
# Visualize Solution
!python ./tools/sudoku_solution_viz.py "./Sudoku Game Types/Classic
Sudoku/classic-solution.txt"
```



```
Solution : 1
9 5 7 6 1 3 2 8 4
4 8 3 2 5 7 1 9 6
6 1 2 8 4 9 5 3 7
1 7 8 3 6 4 9 5 2
5 2 4 9 7 1 3 6 8
3 6 9 5 2 8 7 4 1
8 4 5 7 9 2 6 1 3
2 9 1 4 3 6 8 7 5
7 3 6 1 8 5 4 2 9
```

The values from the outputted solution are parsed through the Python visualizer. Since there is only one solution, only one board is seen. It can be seen that the classical Sudoku rules are followed. On the first row and column, there are only unique values ranging from one to nine.

B. Diagonal Sudoku

In addition to the classic Sudoku rules, Diagonal Sudoku has one additional rule.

1. The two main diagonals must have unique values from one to nine.

▸ Run [1]

```
# For seed
!echo %RANDOM%
```

▸ Run [2]

```
# Generate a problem and display example
!clingo -n 1 --rand-freq=1 --seed=18248 "./Sudoku Game Types/Diagonal
Sudoku/diagonal-sudoku.lp" > "./generated/diagonal_board_gen.txt"
# Display Full Board
!python ./tools/sudoku_solution_viz.py "./generated/diagonal_board_gen.txt"
```

```
Solution : 1
8 1 9 5 4 3 7 6 2
2 6 4 7 9 8 5 1 3
5 7 3 2 6 1 4 9 8
6 2 5 9 3 7 1 8 4
7 4 8 1 5 6 3 2 9
9 3 1 8 2 4 6 5 7
1 8 6 4 7 9 2 3 5
4 9 2 3 1 5 8 7 6
3 5 7 6 8 2 9 4 1
```

For this section, we will generate a random example problem and solve it with the clingo answer set solver. First, we will generate a complete, valid board before we remove parts of the board.

▸ Run [3]

```
# 70% to remove an entry
!python ./tools/sudoku_remove.py "./generated/diagonal_board_gen.txt" 0.7
"./generated/diagonal_problem_gen.lp"
# Display Problem
!python ./tools/sudoku_problem_viz.py "./generated/diagonal_problem_gen.lp"
```

```
- 1 - - 4 3 - - 2
- - 4 - - - 5 - 3
5 - 3 - - - - - -
- - 5 - - - 1 - -
- - - 1 - 6 - 2 -
- - - - 2 - - - -
- - - 4 - - - - -
- - - - - 5 - - -
3 - - - - - - 4 -
```

The Python script file is called to remove parts of the board to “generate” a Sudoku puzzle. Now, we will call the clingo answer set solver to find a solution.

▸ Run [4]

```
# Run solver
!clingo "./generated/diagonal_problem_gen.lp" "./Sudoku Game Types/Diagonal
Sudoku/diagonal-sudoku.lp" -n 1 > "./generated/diagonal_solution_gen.txt"
# Visualize Solution
```

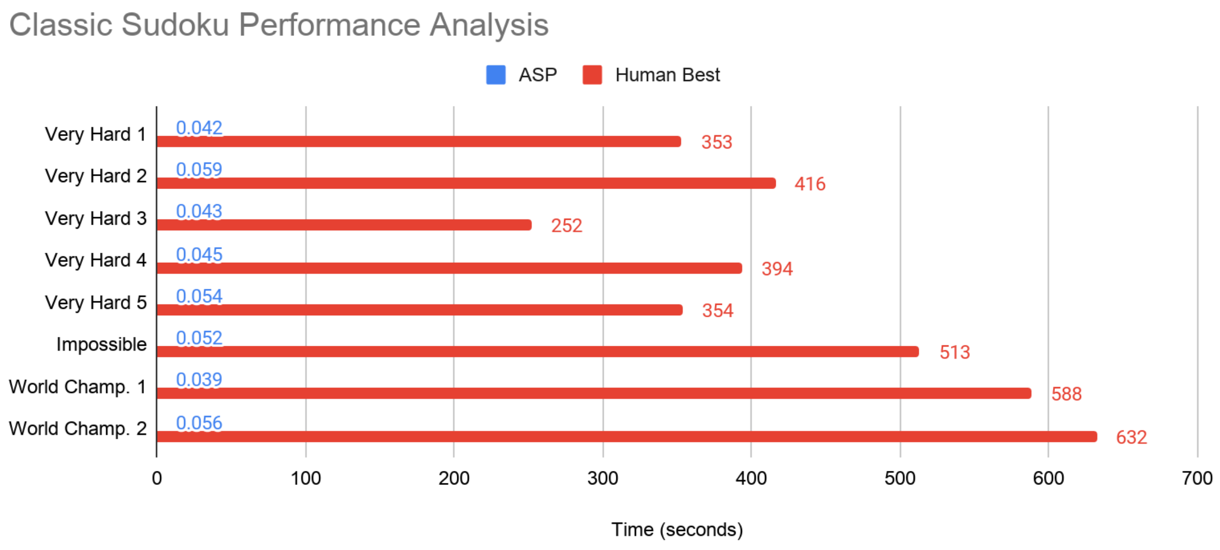
```
!python ./tools/sudoku_solution_viz.py "./generated/diagonal_solution_gen.txt"
```

```
Solution : 1
8 1 6 5 4 3 9 7 2
2 9 4 8 6 7 5 1 3
5 7 3 2 1 9 6 8 4
4 2 5 7 3 8 1 9 6
7 3 9 1 5 6 4 2 8
6 8 1 9 2 4 3 5 7
9 6 7 4 8 1 2 3 5
1 4 2 3 7 5 8 6 9
3 5 8 6 9 2 7 4 1
```

The clingo answer set solver is called to find a single solution. Notice that the solution presented here is different from the generated board we had. There may be multiple answers to the problem if entries are randomly removed.

C. Performance Analysis & Extended Results

Wsudoku.com is a public site where users can play classical Sudoku from a database of partially-filled boards. Registered users may record their time into a leaderboard, where the quickest player's time is displayed; first place times are always verified as valid. Because the puzzles and fastest times are public, we are able to compare them against the performance of our implemented ASP Sudoku solver. Below is a diagram of performance times for the ASP solver versus the best human time, playing puzzles of various difficulties. As expected, the ASP method had a much greater performance time, averaging at around 1/20th of a second for the 9x9 board.



Despite these results, we were not so impressed with a 9x9 board. How would the ASP method perform with a board fifteen times that size? Solving a 36x36 board, of nearly three thousand spaces, took 157.4 seconds- just over 2 ½ minutes. Below is a condensed version of the output, as the entire solution takes six and a half pages.

```
Solving...
Answer: 1
sudoku(5,6,3) sudoku(2,3,4) sudoku(3,4,6) sudoku(6,4,7)...
...
... sudoku(34,31,32) sudoku(35,31,3) sudoku(36,31,15)
SATISFIABLE

Models      : 1+
Calls       : 1
Time        : 157.400s (Solving: 145.15s 1st Model: 145.14s Unsat: 0.00s)
% 157.4 seconds is 2.62 minutes
CPU Time    : 152.219s
```

Dividing 157.4 by 0.05 (roughly the average ASP time), one may mistakenly conclude that in the time it took the ASP method to solve the single 36x36 puzzle, it could have solved 3,148 of the 9x9 puzzles. However, note that solving a 36x36 puzzle is not equivalent to solving a 9x9 puzzle sixteen times. Rather, the complexity of any Sudoku puzzles increases exponentially with size, not linearly. In this light, the solving times of our implemented ASP method are quite impressive, especially compared to human performance.

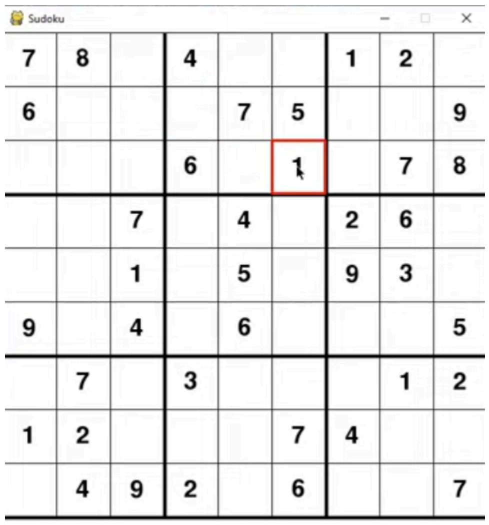
For more information on the specific puzzles referenced in this section, visit the Performance Analysis directory in the GitHub repository.

VI. Future Work

A. GUI Visualization

To visualize the output, we use a Python script file to output a sudoku board line by line on the CLI from an input text file. There are obvious flaws to this approach. For one, it is quite difficult to customize the Sudoku board to be targeted for variant problems. It is also difficult to make it more aesthetically pleasing. In addition, there are no additional lines added to separate the nine subsquares on the sudoku board.

To solve this, we can use the PyGame library or other graphic libraries to have a more pleasing sudoku board. There are many tutorials on YouTube that discuss easy steps to make a game board including a Sudoku board. Below is what we would have imagined how our Sudoku GUI would look like.



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

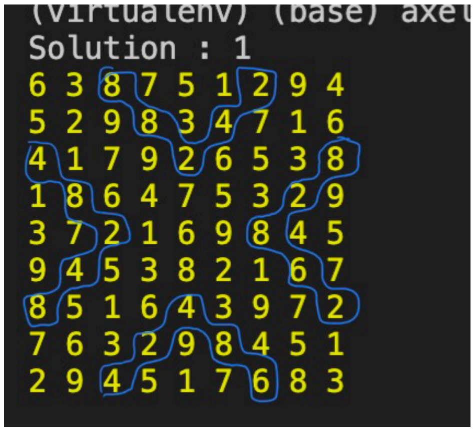
Thanks to *Tech with Tim* from YouTube for the graphic.

B. Other Sudoku Variants

As you may know or not know, there are many other variations of the Sudoku rules. There are many variants we have attempted. One of the incomplete variants that was worked on was the Even/Odd Diagonal.

Even Diagonal

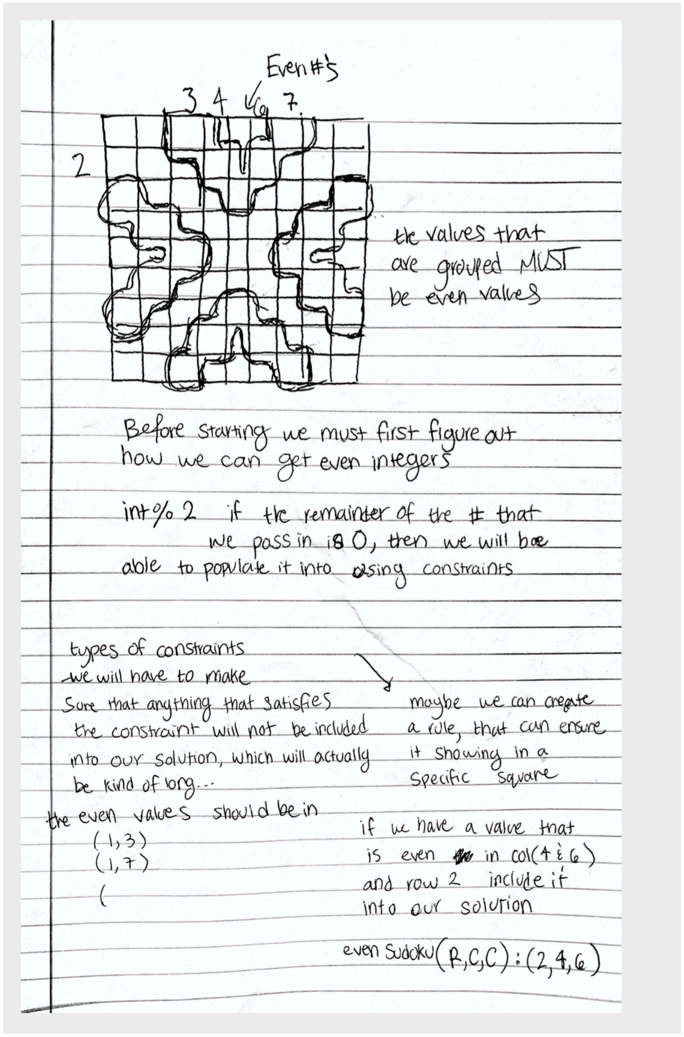
In addition to the Classical Sudoku rules, this variant has one additional rule. The values inside the blue outline must contain even/odd values.



```
(virtualenv) (base) axe
Solution : 1
6 3 8 7 5 1 2 9 4
5 2 9 8 3 4 7 1 6
4 1 7 9 2 6 5 3 8
1 8 6 4 7 5 3 2 9
3 7 2 1 6 9 8 4 5
9 4 5 3 8 2 1 6 7
8 5 1 6 4 3 9 7 2
7 6 3 2 9 8 4 5 1
2 9 4 5 1 7 6 8 3
```

However, we had issues implementing a constraint that can only allow even numbers. In addition, it was difficult to tell the solver where the blue outline is located at. We could not find/author a formula that can be done to place the blue outline.

Below is the work we have done on the blue outline problem.


Even #'s
3 4 6 7
2

the values that are grouped MUST be even values

Before starting we must first figure out how we can get even integers

int % 2 if the remainder of the # that we pass in is 0, then we will be able to populate it into using constraints

types of constraints we will have to make sure that anything that satisfies the constraint will not be included into our solution, which will actually be kind of long...

maybe we can create a rule, that can ensure it showing in a specific square

the even values should be in
(1,3)
(1,7)
(

if we have a value that is even in col(4 & 6) and row 2 include it into our solution

even Sudoku(R,C,C) : (2,4,6)

C. Combinational Games: Rubik’s Cube

In addition to solving a sudoku puzzle, Answer set programming can most likely be used to solve at least a normal three by three Rubik’s Cube. The challenge for solving Rubik’s Cubes would be abstracting the idea of steps to solve a solution. The steps would need to be in order because you would find a different outcome if you did the steps in a different order.

To implement steps, the *#include <incmode>* to reflect new changes on the Rubik’s Cube. The include statement contains a time component and performs one of the available actions until a solution is found or a maximum of steps has been succeeded. Each predicate that would perform a step would require the new time component.

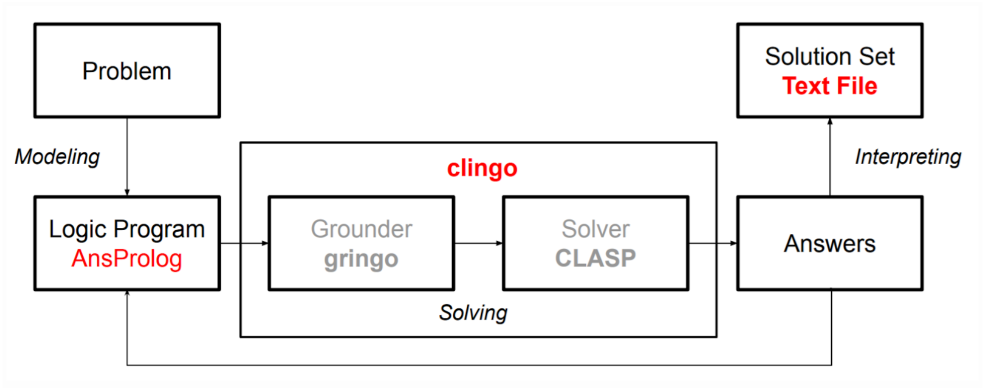
Here’s a bare program of how it would look like:

```
#include <incmode>
#const imax = 50. # maximum # of steps

#program base.
% initialize rubix cube

#program step(t).
% predicates that require steps. t is the time (step number) variable
rotate(..., ..., t).

#program check(t).
# query is true as long as steps < imax
# goal(t) is found, then don’t throw the answer set
:- query(t), not goal(t).
```

VII. Conclusion

The main objective of this project was to use Answer Set programming to solve a 9x9 sudoku board. Throughout the entire process of problem solving, we found out there was always more to learn. It all began with learning the fundamentals of Answer Set programming; this meant understanding the motivation in using Answer Set Programming (ASP), the use of ASP in practice, the semantics and syntax of ASP, the language of AnsProlog, the understanding of concepts through logical reasoning in Answer Set Programming and the process of how a problem is solved.

The diagram below reviews the Answer Set Programming process.



Learning to apply new concepts such as modeling problems, understanding choice rules and formulating constraints, was significantly difficult as it was entirely new to the group. These skills, however were first put into practice, through Potassco which is an organization that focuses on providings bundles of tools for the purpose of Answer Set Programming. In learning the fundamentals of Answer Set Programming at a high level, the first implementation was not actually sudoku. Resources provided by Potassco allowed us to learn and apply the concepts of Answer Set Programming, modeling problems, creating choice rules, and creating constraints, through an N-Queens problem that Potassco provided. Utilizing the concepts provided in the N-Queens problem, helped set a foundation and helped prepare to approach solving a 9x9 sudoku board.

As learned through Potassco, when initializing a problem it must first be modeled. Similarly to the N-Queens example we had to populate the board. This was done by using a Choice Rule that allowed in representing a 9x9 sudoku board populated with values from 1-9. Once having 81 spaces of sudoku populated, we then had to create constraints in regard to the rules of sudoku; meaning rules were created to not allow repeating values for rows, columns, and subsquares.

Throughout the process of creating choice rules and constraints, there was difficulty in understanding how to create them; this was due to not knowing how to logically formulate and visualize constraints that can be written in AnsProlog. Not knowing how to approach the sudoku problem and define that values should not repeat in the same column, row, and subsquare. However, with practice and familiarizing ourselves with the implementation of formulating constraints and understanding of its logical reasoning, constraints were able to be established which prevented repeating values from showing in the same columns, rows, subsquares, and main diagonals of any of the sudoku board variants.

VIII. Supplementary

A. Other AnsProlog Syntax

1. Using *<predicate>(<#>..<#>)*

The two periods are an AnsProlog shorthand that allow you to assign multiple numerical fields to a single predicate in a clean form. The statement

```
integer(1..10).
```

is equivalent to the following fact declarations:

```
integer(1). integer(2). integer(3). integer(4). integer(5). integer(6). integer(7). integer(8).
integer(9). integer(10).
```

2. Comments

Comments are denoted with the *%* symbol and are ignored during the solving process.

```
integer(1..10).
even(X) :- integer(X), X/2 == 0.                    %Only even X's will create even(..) predicates
```

3. Constants

Constants are denoted by *#const <name> = <value>*. The reader should note that they may be set in the command line calling Clingo, but will always default to the value assigned in the program.

```
#const min = 1.
#const max = 99.
integer(min..max).
```

4. Using *#show*

The statement *#show <predicate>/<number of fields>* is used to configure our output. It in no way affects the solving process- it only specifies which predicates to output in each Answer.

```
integer(1..10). sophomore(alexis). married(chris, jason). family(max, ann, diego).
#show married/2.
#show integer/1.
```

For this example, the output should only contain predicates of married and integer.

```
Solving...
Answer: 1
integer(1) integer(2) integer(3) integer(4) integer(5) integer(6) integer(7) integer(8)
integer(9) integer(10) married(chris,jason)
SATISFIABLE
```

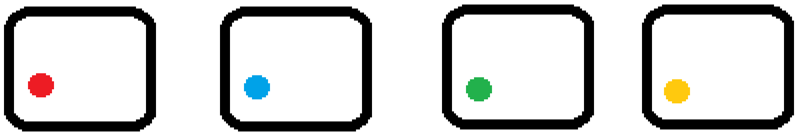
Notice that the family predicate or the sophomore predicate do not appear.

B. More on Choice Rules

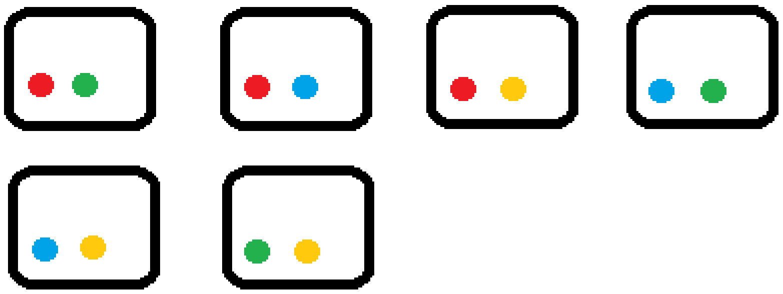
To demonstrate the Choice Rule more clearly, we will use a simple example of colored marbles in a bag. This bag can hold any number of marbles in any combination; one green, a red and a green, a red and a blue, etc. To list all combinations of marbles in this bag mathematicians would normally use the nCr equation. nCr is defined as a selection of items from a collection, such that the order of selection does not matter, where the result is a combination of n objects taken r at a time

$$ {}_{n}C_{r} = \frac{n!}{r!(n-r)!} $$

For our bag example, given 4 colored marbles, if we only take one marble, we would have 4 ways of representing our bag i.e. 4C1.



If we increase the amount we take, to two marbles, we would have 6 combinations i.e. 4C2.



The choice rule models this equation and also allows us to define bounds for r.

$$ \sum_{1}^{r} nCr $$

So with the power of the choice rule we can solve for all possible combinations of marbles in the bag given n marbles and taking r amount. Note in this equation the lower bound is 1, but the lower bound can be set to any positive integer greater than 1 as well. To define all combinations for a single r value, both bounds are the same.

$$\sum_{1}^{r} nCr = 4C1 + 4C2 = 10$$

For our example we can see that if we add the totals of 4C1 and 4C2 we get 10, which matches our equation.
Now that we have a background for the inner workings of the choice rule, we will model the equation with real code. We will first start with defining our set of colors, and creating marbles of each color.

```
color(red).
color(blue).
color(green).
color(yellow).

marble(C) :- color(C).  %Read as, for every color C, there exists a marble of that same color.
```

We can see a sample output of our environment thus far.

```
Solving...
Answer: 1
color(red) color(blue) color(green) color(yellow) marble(red) marble(blue) marble(green)
marble(yellow)
SATISFIABLE
```

Now, using the Choice Rule, we can describe all possible combinations of marbles that can be in a bag. For our bag predicate, we define an owner of the bag, and the marble it contains. Every answer set will then have a set of bags with all colored marbles in that bag. This output shows $\sum_{1}^{2} 4Cr$

```
owner(alec).

1{ bag(O, M): marble(M) }2:- owner(O).

#show bag/2.
```

```
Solving...
Answer: 1
bag(alec,yellow)
```

```
Answer: 2
bag(alec,green)
Answer: 3
bag(alec,green) bag(alec,yellow)
Answer: 4
bag(alec,blue)
Answer: 5
bag(alec,blue) bag(alec,yellow)
Answer: 6
bag(alec,blue) bag(alec,green)
Answer: 7
bag(alec,red)
Answer: 8
bag(alec,red) bag(alec,green)
Answer: 9
bag(alec,red) bag(alec,yellow)
Answer: 10
bag(alec,red) bag(alec,blue)
SATISFIABLE
```

IX. Reference

Crick, Tom. *Superoptimisation: Provably Optimal Code Generation using Answer Set Programming*. August 2009, University of Bath, Bath, Somerset, United Kingdom. *Superoptimisation: provably optimal code generation using answer set programming*, https://researchportal.bath.ac.uk/en/studentTheses/superoptimisation-provably-optimal-code-generation-using-answer-s. Accessed 24 April 2021.

Lifschitz, Vladimir. "What Is Answer Set Programming?" *Proceedings of the AAAI Conference on Artificial Intelligence*, 2008, pp. 1594-1597. *wiasp.dvi - wiasp.pdf*, https://www.cs.utexas.edu/users/vl/papers/wiasp.pdf. Accessed 24 April 2021.

Lifschitz, Vladmir. *Programming with CLINGO*. 2019. *pwc.pdf*, https://www.cs.utexas.edu/~vl/teaching/378/pwc.pdf. Accessed 24 April 2021.

Martens, Chris. *Notes on Answer Set Programming*. CSC 791 Generative Methods for Game Design. 20 September 2017. *asp-notes.pdf*, http://www.cs.cmu.edu/~cmartens/asp-notes.pdf. Accessed 24 April 2021.

Nelson, Mark J., and Adam M.. Smith. "ASP with applications to mazes and levels." *Procedural Content Generation in Games*, Springer, 2016, p. 15. *Procedural Content Generation in Games*, http://pcgbook.com/. Accessed 24 April 2021.

SudokuCup. "Jakub Ondroušek has beaten the world record." *Jakub Ondroušek has beaten the world record*, 16 June 2010, http://sudokucup.com/node/754. Accessed 24 April 2021.

University of Potsdam. "clingo and gringo." *clingo and gringo*, 5 May 2016, https://potassco.org/clingo/. Accessed 24 April 2021.

University of Potsdam. *Potassco*. This channel provides illustrative material explaining the technology underlying Answer Set Programming along with the suite of tools developed within the Potassco project. 27 February 2014. *Potassco*, Potsdam, YouTube, https://www.youtube.com/c/Potassco-live/. Accessed 24 April 2021. Video.