# Protein to SAT Explanation

Chase Maguire

December 2018

## 1 Introduction

This document is meant to serve as an explanation to the conversion of the two-dimensional ammino acid layout to the SAT logic that then gets plugged into Lingeling. [lin, 2018]. The sections will be broken up based on the rules that they cover. Throughout the explanations I will refer to each digit as an *acid* and the whole string as the *protein*.

## 2 Installation and usage

After installing the Lingeling SAT solver, download the conversion program. [Maguire, 2018]. Using the terminal, use Python 3 to run the script *protein2sat.py* with the string wanted as the first argument. A file of the same string will be output as a .cnf file. Use this file with Lingling, calling it as *./lingeling filename.cnf*

## 3 A representation

A basic concept that I used to represent the proteins is as such:
For a string $n$ with length $l$ I represent each acid, or each single digit within the string, as an $l$x$l$ matrix. Each position in the each matrix, starting at the top left, is numbered, starting at 1, going all the way to $l^3$. This matrix also represents where the acids are placed, and will represent in the end, how the entire sequence is laid out. It should be more clear why this way was chosen as I explain each placement rule.

## 4 Each acid must be placed

The first, and most simple rule. To make the SAT solver at least place the acids, (not worrying about overlapping or sequentially) we must force them to be placed somewhere within its matrix.
This is fairly simple, we take each variable for each protein in the matrix, and *or* each one with the other variables with within the same matrix.

At the end of the process, we should have $l$ different clauses, one for each acid, with $l^2$ variables in each.

## 5 One acid per position

This is meant to disallow the overlapping placements of acids. The SAT logic is a bit more complex, but the representation is simple. For each matrix, for each possible position, or each variable in each matrix, we go through each other matrix with the same position and negate both of those variables within a clause. Doing this for each matrix, and each position within each matrix.
At the end of this process, we have $(l-1)! * l^2$ clauses, with no new variables, just reusing the same from the previous rule.

## 6 Each acid must be placed sequentially

Aside from the very first acid in the sequence, each acid must be placed adjacent to the last. We do this be restricting where the placement of the next acid. Besides on the corners and edges, the acids can be placed above, below, or on either side of the previously placed one. again, we do not do this for the first acid. Another special case also exists, in which we don't need to write any rules for anything following the last acid, since nothing follows it.
The SAT logic is as follows:
For each possible acid placement, we imply that the next acid must go in a valid place next to it. Where $A$ is a acid in some spot, for this example, not on the edge, and $B, C, D, E$ are the adjacent positions

$$A \implies B \wedge C \wedge D \wedge E$$

Then, we can break this down into something that fits neatly into our CNF form by breaking the implication down, which ends up as

$$-A \vee B \vee C \vee D \vee E$$

This is the logic, and the amount of clauses created is equal to $3n^2$. Not all variables above are used, if something is on the edge. No new variables are added, still using the same ones from the very first rule.

## 7 Defining matches

The previous rules have just defined how the acids are to be placed. Defining a match, where two non-sequential 1's are adjacent to each other, leads us to to our goal. There are two sides of the logic that need to be done, that the placement implies the matching variable being "turned on", and that the variable being "turned on" implies the placement.

### 7.1 Placement implies matching

This is the part of the rule we care about. It allows us to create new variables that get turned 'on' when a matching is detected. We do this by looking at the matrices that are 'owned' by a 1. Then we take this set and break it down further into what 1's can possibly become adjacent. So we get

$$\text{Adjacent position} \vee \text{Adjacent Position} \implies \text{Matching Var}$$

The edges also count on what could define a valid matching. Placing a 1 in the top right corner only allows for a matching if a 1 is placed below or to the left of it. We again will break down the implication into a CNF as we did with one of our previous rule.

### 7.2 Matching implies Placement

This part of the rule disallows the SAT solver to just turn on all the variables created in the last.

$$\text{Matching Var} \implies \text{Adjacent position} \vee \text{Adjacent position}\cdot$$

Since breaking down the implication will turn the ORs into ANDs, it won't fit neatly as the others with CNF. We do a trick explained in one of Knuth's books to allow us to do such, at the expense of 3 extra clauses. After this, the matching variables are done, we just need a way to count them.

## 8 Counting the matching variables

The way we will count is by using a a binary tree style of counting. First, we need a perfectly balanced tree. So, we create more matching variables until we get enough leafs to do so. We simply force these to be false, by putting them in their own clause and forcing them to be false. The basic idea is that we will "trickle up" with our variables for counting.
At the bottom most level, we have our concrete variables. The matching ones we created in the past two rules. They are either on, or off. With the iff we can be sure of this. Now, we create more variables that represent the possible combination, this represents a level above the leafs. We create more variables doing this, but less clauses as more different combinations are created. Since there are going to be dummy variables, the ones we used to make the perfect amount of leafs, there will be combinations that are impossible to take care of. The logic is simple.

$$\text{Counting variable} \implies \text{Counting Variable from a lower level} \wedge \text{Counting Variable from a lower level}$$

Again, we break down the implication, and it fits neatly into the CNF form that constrains the solver. I, am unsure if we need to do the other side of the implication, but I think it must be done.
After that process finishes, add a target, by adding a single clause with which top most counting variable we want on, and force it to be on by putting in its own clause. Running it through the SAT solver will either give us an emplacement that gives us our goal, or fails, letting us know it can't be.

# References


Lingeling website, 2018. URL `http://fmv.jku.at/lingeling`.

Chase Maguire. Protein to sat program, 2018. URL `https://github.com/Wirdal/Protein2SAT`.