



GRP_13: The Game of Life

Nicholas Tillo	20njt4
Grace Odunuga	20goo
Al-Barr Ajiboye	20abka
Raksha Rehal	20rsr3

Course Modelling Project

CISC/CMPE 204

Logic for Computing Science

December 2, 2022

Contents

C1) Articulate a Problem

<u>Project Formulation</u>	3
<u>Summary</u>	4

C2) Model the Problem

<u>Propositions</u>	6
<u>Constraints</u>	7
<u>Coding the Model</u>	9
<u>Jape Proofs</u>	11
<u>First-Order Extension</u>	16

C3) Explore the Model

<u>Default Constraints</u>	18
<u>Adjusted Constraints</u>	25
<u>atLeast1 Constraints</u>	28

C1) Articulate a Problem

Corresponds to (D1) Problem Formulation

★ Project Formulation

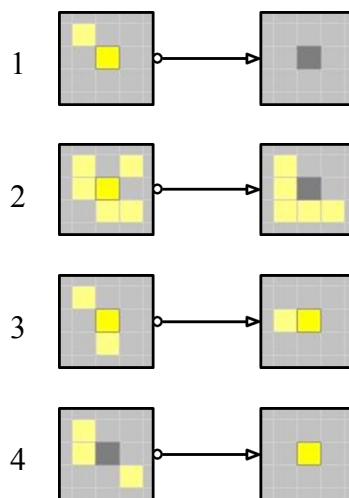
The Game of Life is a discrete model of computation invented by British mathematician John Horton Conway, and was popularized in 1970. It is a cellular automaton that is played on an infinitely-sized grid. The game is self-operating; a “zero player game” where once the initial conditions are set, the game plays itself. Each cell can take on one of two states— it can either be **alive**, or it can be **dead**. Then, in each new stage, the game plays itself and cells either die, revive, continue surviving, or remain dead based on the above conditions.

Provided here is a link to a simulation of the Game of Life that can be played online:

<https://playgameoflife.com/>.

The game advances in stages, where the relationship between stages is defined in 4 rules:

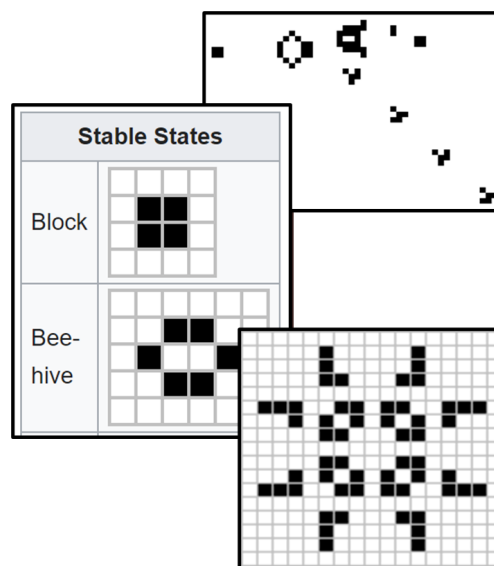
	Current State of Cell <i>A</i>	State of Cell <i>A</i> 's Neighbours	State of Cell <i>A</i> in Next Iteration	Descriptor
1	Alive	1 OR 0 Neighbours	Dead	Cell dies by solitude.
2	Alive	4 OR More Neighbours	Dead	Cell dies by overpopulation.
3	Alive	2 OR 3 Neighbours	Alive	Cell survives.
4	Dead	EXACTLY 3 Neighbours	Alive	Cell comes to life.



The behaviour of a cell is influenced by the state of its 8 neighbours (both in the cardinal and diagonal directions) as outlined by the rules above.

While there are a great number of outcomes that are possible following several different starting configurations, we will be defining two common outcomes of the game for the purpose of this modelling project:

- **Stable state** — The game reaches a point where there are no more changes between interactions... i.e. the scenario where cells that are alive continue to survive, and every cell that is dead remains dead, without the possibility of reviving
- **Infinite state** — The game reaches a point where there are infinite repeating sequences in the same pattern.



Our project aims to determine if any given state is a **stable state**. We will logically determine if any current sequence of cells is a stable state. Specifically, our model will correspond to the positions of cells that are alive and make up a stable state. The outcome will provide us with all the models that are stable states (depending on our configurations), and output a sample solution.

★ Summary

The model of Conway's Game Of Life was done by having each cell be represented by a proposition. The truth value T would mean that the cell is alive, and F would mean that the cell is dead.

The rest of the behaviour was modelled via logical operators within our constraints. These constraints we created mirrored the four rules of the game, and the conditions for the model to be

considered stable. Due to the complexity of the original game, the “neighbours” taken into account when calculating the next iteration of the propositions were lowered from 8 to 4 (only focusing on the cardinal directions).

We explored this model through many techniques. First, we looked through the model we created for changes to aspects notable to the original game, such as stable patterns and the number of infinite sequences that resulted from the changed rules. We also looked at the optimal grid configuration in order to maximize stable states for a given perimeter.

We then explored other variations of the game's rules and found some interesting results. We explored a set of reversed rules that nearly always resulted in infinite loops. Finally, we explored a set of rules that forced the resulting stable states to have at least one proposition that is T or alive, which allowed us to attempt to search for a mathematical relation that fit the data we gathered with this information.

C2) Creating The Model

Corresponds to (D2) Implementation, (D3) Jape Proofs, (D4) First-Order Extension

Our model attempts to answer the following question: “In Conway's *Game Of Life*, how many possible configurations result in a stable state within X amount of turns?”

In order to decrease complexity, when looking for neighbours, we only take into account the cells in the 4 cardinal directions, and not the diagonals. Thus, each cell (represented by propositions) will only have 4 neighbours, instead of the original intended 8 neighbours.

Initially, we did not plan for each cell to be represented by only one proposition. We instead planned for each cell to have two propositions that would represent both its current state (dead or alive) and what its future state action would be (staying alive/staying dead or reviving/dying). It was changed simply to having a current state where the proposition (i.e. the cell) could either be simply alive or not alive. This was done as these propositions were redundant, they are fully defined by the starting propositions, so their behaviour can be modelled by constraints.

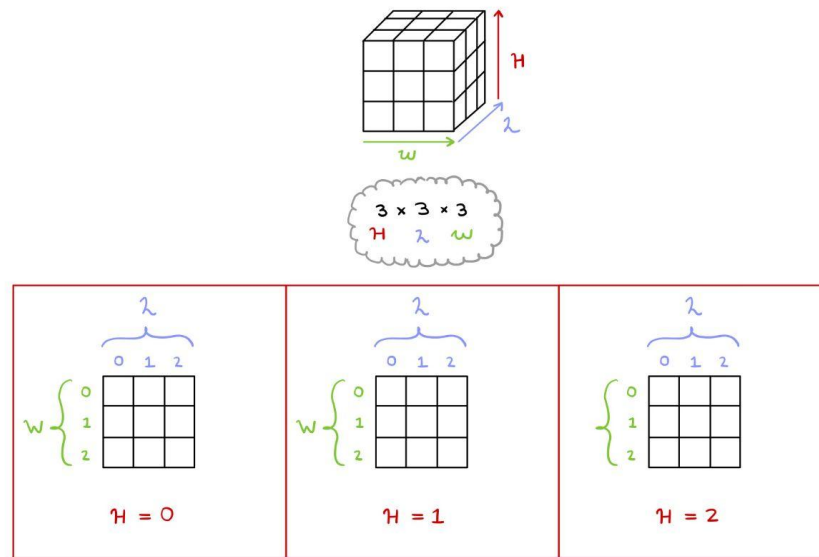
★ Propositions

The Game Of Life can be modeled on a gridded cube, with *specifications* H, L, W:

- H is the time/round number/iteration of the game... i.e. with each new iteration of the game, H is incremented by 1.
- L is the *column* on which any given square is located.
- W is the *row* on which any given square is located.

Thus, the proposition $A(H,L,W)$ is true when the cell in location L,W in turn H is alive. Likewise, this proposition is false when the cell is dead.

- What it means specifically for a cell to be “alive” or “dead” is outlined in the constraints section below.



This can be visualized by placing each interaction of the game on top of the other in order to create a large stacked cube, where each slice represents one grid state of the game.

★ Constraints

- Each cell, apart from the starting state, has constraints that must be followed in order to dictate when it will be alive, and when it will be dead. In each iteration of the game, each cell is reassessed according to these constraints/rules, and changes in state to be alive or dead, accordingly.

Defining the State of a Cell (Alive/Dead)

- There are 4 main constraints, with each corresponding to a rule of the Game of Life...
 - If the cell is **alive** and surrounded by **2 OR 3 neighbours**, it will be **alive** in the next iteration.
 - If the cell is **alive** and surrounded by **NOT 2 OR 3 neighbours**, it will be **dead** in the next iteration.
 - If the cell is **dead** and surrounded by exactly **3 neighbours**, it will be **alive** in the next iteration.
 - If the cell is **dead** and surrounded by **NOT 3 neighbours**, it will be **dead** in the next iteration.
- We have 1 of each of these constraints for every proposition that represents every cell on our grid... for example:

- $A_{(0,0,0)}$ is **alive** and has **2 OR 3** neighbours $\rightarrow A_{(1,0,0)}$ is **alive**.
 - $A_{(0,0,0)}$ is **alive** and has **NOT 2 OR 3** neighbours $\rightarrow A_{(1,0,0)}$ is **dead**.
 - $A_{(0,0,0)}$ is **dead** and has **3** neighbours $\rightarrow A_{(1,0,0)}$ is **alive**.
 - $A_{(0,0,0)}$ is **dead** and has **NOT 3** neighbours $\rightarrow A_{(1,0,0)}$ is **dead**.
- These constraints encapsulate every possible state that A can be in... In other words, there is no other possible state that a square can be in. Thus, we do not have to add a constraint that requires a proposition to adhere to at least one of the constraints.

Defining the State of a Cell's Neighbours and their Implication

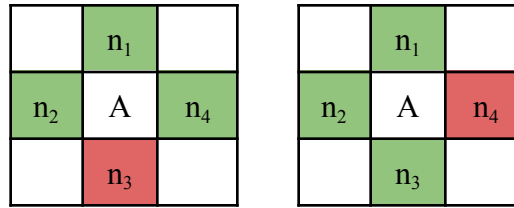
- Within our code, `currentRun.py`, the following can be observed...
 - The 2 OR 3 and EXACTLY 3 neighbour constraints are defined in the functions, `findNeighbours2V3` and `findNeighbours3` respectively.
 - These above functions create logical sequences that depict all the possible ways for any one cell to have 2 or 3 neighbours. These sequences are then disjoined... any by doing this, if *at least one* of these expressions are satisfied, then the entire sequence will be true, indicating that the cell has 2 or 3 neighbours.
- For example, the below defines what it means for a proposition to have exactly three neighbours:
 - The square we were analyzing is A, with its neighbours being n_1 , n_2 , n_3 , and n_4 ...

	n_1	
n_2	A	n_4
	n_3	

Then, all the possible ways that A can have exactly 3 neighbours are:

	n_1	
n_2	A	n_4
	n_3	

	n_1	
n_2	A	n_4
	n_3	



- In propositional logic (i.e. within our code), we can reproduce this idea like so:

```
( (~n1 & n2 & n3 & n4) | (n1 & ~n2 & n3 & n4) | (n1 & n2 & ~n3 & n4)
| (n1 & n2 & n3 & ~n4) )
```

- We applied this same logic to the concept of a cell A having 2 OR 3 neighbours... i.e. we exhausted every possible way that a cell could have either 2 or 3 neighbours, and disjoined all these statements together.
 - This alternatively can be represented by negating all the ways that A could have 0 or 1 neighbours, as these are the cases we don't want, and negating these expressions only validates A if it has 2 or 3 neighbours.

Defining a Stable State

- To define when a state is stable, we must look at the final state (of x iterations), and then ask: "Will there be any changes if we were to run another iteration?"
 - If anything changes (the cells change from alive to dead, or vice-versa) then it is not a stable state.
- We will model this in logic and in our code by giving each proposition in the final stage a separate constraint that says: " **$A_{(x, y, z)}$ is true $\leftrightarrow A_{(x+1, y, z)}$ is true**" where x is the final round for the game.
 - If each proposition does not change between x and $x+1$ iterations, then the state is stable.
 - We can determine $A_{(x+1, y, z)}$ via our other constraints that we used throughout the rest of the game to determine the propositions for each iteration.

★ Coding the Model

The following section documents the major issues we faced whilst coding our model in Python.

There was one specific solution that plagued us... within a 2x2x2 grid, we received the following:

What the starting configuration was:

T	T
T	T

What the code resulted in:

F	T
T	T

What the **correct result** should be:

T	T
T	T

Below are the constraints that refer to $A_{(1,0,0)}$ (which is the cell that is resulting in the faulty value).

```
Custom constraint added:
(A(1,0,0) ∨ (¬A(0,0,0) ∨ ((¬A(0,1,0) ∧ ¬A(0,0,1)) ∨ (¬A(0,1,0) ∧ A(0,0,1) ∧ ¬A(0,0,1)) ∨ (A(0,0,1) ∧ A(0,1,0)) ∨ (¬A(0,1,0) ∧ A(0,1,0) ∧ ¬A(0,0,1)))))

Custom constraint added:
(¬A(1,0,0) ∨ (A(0,0,0) ∨ ((¬A(0,1,0) ∧ A(0,1,0) ∧ A(0,0,1)) ∨ (A(0,0,1) ∧ A(0,1,0) ∧ ¬A(0,0,1)))))

Custom constraint added:
(A(1,0,0) ∨ (((¬A(0,1,0) ∨ A(0,0,1) ∨ ¬A(0,0,1)) ∧ (¬A(0,1,0) ∨ A(0,1,0) ∨ ¬A(0,0,1))) ∨ A(0,0,0)))

Custom constraint added:
(((¬A(0,0,0) ∨ ((A(0,0,1) ∨ A(0,1,0) ∨ ¬A(0,0,1)) ∧ (¬A(0,1,0) ∨ ¬A(0,0,1)) ∧ (A(0,0,1) ∨ A(0,1,0)) ∧ (¬A(0,1,0) ∨ A(0,1,0) ∨ A(0,0,1)))) ∨ ¬A(1,0,0))
```

- All of the constraints just seem to work out for this one. But, we noticed that there was a flaw in the cell pertaining to $A_{(1,0,0)}$
 - The first constraint *should* take care of this. As $A_{(0,0,0)}$ is true, and there are 2 alive neighbours ($A_{(0,1,0)}$ and $A_{(0,0,1)}$). Yet, it does not work.
- During this stage of development (while we have not implemented stable states yet), our output was that there were 16 solutions for the 2x2x2 grid.
 - This is the correct total amount of solutions for a 2x2 grid, which is good.
 - This verifies that the logic are working properly, as fully incorrect logic would result in too many or too little solutions.
- We also further determined that our code broke on multiple occasions, not just on the above instance. Some truth values would output as expected, while others simply would not adhere to the constraints we fed into the SAT solver.

After many grueling hours, we finally determined that this issue was caused from a faulty try-except within the code. We knew that our constraints and logic was correct, however the

code was causing the invalid neighbours (ones adjacent to the grid) to not get assigned the default state of F, which interfered with the rest of the model, failing to trigger the 2or3Neighbour condition.

From here, the model is fully working, now to explore.

★ Jape Proofs

✧ Proof 1

This proof shows us **how** a donut is a stable state.

Using our definition of a stable state, we will show that a donut is a stable state, So starting with the configuration that makes up a donut state, we will show that nothing will change in the next iteration, thereby indicating that it is a stable state.

We will start with the donut square:

T	T	T
T	F	T
T	T	T

And show that for each proposition $A_{(x,y,z)} \leftrightarrow A_{(x+1,y,z)}$

for time = 0

for time = 1

A	B	C
D	$\neg E$	F
G	H	Q0

T	B2	C2
D2	$\neg R$	F2
G2	P	Q

1: A, B, C, D, $\neg E$, F, G, H, Q0, $(A \wedge B \wedge C) \rightarrow T$, $(D \wedge G \wedge H) \rightarrow G2$, $(H \wedge Q0 \wedge F) \rightarrow Q$, $(B \wedge C \wedge F) \rightarrow C2$, $(A \wedge B \wedge C) \rightarrow B2$	premises
2: $(A \wedge D \wedge G) \rightarrow D2$, $(G \wedge H \wedge Q0) \rightarrow P$, $(C \wedge F \wedge Q0) \rightarrow F2$, $(D \wedge B \wedge F \wedge H \wedge \neg E) \rightarrow \neg R$, $(T \wedge B2 \wedge C2 \wedge D2 \wedge \neg R \wedge F2 \wedge G2 \wedge P \wedge Q) \rightarrow S$	premises
3: $A \wedge B$	\wedge intro 1.1,1.2
4: $A \wedge B \wedge C$	\wedge intro 3,1.3
5: T	\rightarrow elim 1.10,4
6: B2	\rightarrow elim 1.14,4
7: $T \wedge B2$	\wedge intro 5,6
8: $B \wedge C$	\wedge intro 1.2,1.3
9: $B \wedge C \wedge F$	\wedge intro 8,1.6
10: C2	\rightarrow elim 1.13,9
11: $T \wedge B2 \wedge C2$	\wedge intro 7,10
12: $A \wedge D$	\wedge intro 1.1,1.4
13: $A \wedge D \wedge G$	\wedge intro 12,1.7
14: D2	\rightarrow elim 2.1,13
15: $T \wedge B2 \wedge C2 \wedge D2$	\wedge intro 11,14
16: $D \wedge B$	\wedge intro 1.4,1.2
17: $D \wedge B \wedge F$	\wedge intro 16,1.6
18: $D \wedge B \wedge F \wedge H$	\wedge intro 17,1.8
19: $D \wedge B \wedge F \wedge H \wedge \neg E$	\wedge intro 18,1.5
20: $\neg R$	\rightarrow elim 2.4,19
21: $T \wedge B2 \wedge C2 \wedge D2 \wedge \neg R$	\wedge intro 15,20
22: $C \wedge F$	\wedge intro 1.3,1.6
23: $C \wedge F \wedge Q0$	\wedge intro 22,1.9
24: F2	\rightarrow elim 2.3,23
25: $T \wedge B2 \wedge C2 \wedge D2 \wedge \neg R \wedge F2$	\wedge intro 21,24
26: $D \wedge G$	\wedge intro 1.4,1.7
27: $D \wedge G \wedge H$	\wedge intro 26,1.8
28: G2	\rightarrow elim 1.11,27
29: $T \wedge B2 \wedge C2 \wedge D2 \wedge \neg R \wedge F2 \wedge G2$	\wedge intro 25,28
30: $G \wedge H$	\wedge intro 1.7,1.8
31: $G \wedge H \wedge Q0$	\wedge intro 30,1.9
32: P	\rightarrow elim 2.2,31
33: $T \wedge B2 \wedge C2 \wedge D2 \wedge \neg R \wedge F2 \wedge G2 \wedge P$	\wedge intro 29,32
34: $H \wedge Q0$	\wedge intro 1.8,1.9
35: $H \wedge Q0 \wedge F$	\wedge intro 34,1.6
36: Q	\rightarrow elim 1.12,35
37: $T \wedge B2 \wedge C2 \wedge D2 \wedge \neg R \wedge F2 \wedge G2 \wedge P \wedge Q$	\wedge intro 33,36
38: S	\rightarrow elim 2.5,37

Explanation of Proof Steps:

- In the proof above, we began with backwards reasoning to perform implication elimination such that a stable state S would be our conclusion.
- We then use conjunction introduction multiple times until we end up with the conjunction $(A \wedge B \wedge C)$ which we can use to get T on its own line by implication elimination. T represents one proposition in the iteration where time = 1.
- We rinse and repeat this process for the rest of the propositions that will make up our grid when time = 1.
- Finally, this fills in the missing steps between our original implication elimination, as we now have all the propositions when time = 1, and we can see that the “sign” of each proposition has not changed. For example A is able to give us T with the premises we used, as A has 2 alive neighbours, B and C. The only F proposition— the middle of the

donut— is $\neg E$, and we are able to get $\neg R$ from it and its neighbours. Thus, as nothing has changed, S is true— and the state is stable.

✧ *Proof 2*

This proof shows us how we get to a donut from a 3x3 grid of 9 T's, showing that the grid will become a stable state within 2 rounds.

We will show that a 3x3 grid where each proposition is T will result in a donut, which is a stable state in 1 step/iteration. We will prove this by focusing on the middle square, as showing the changes for each square of the grid would require manually entering constraints corresponding to each different square, represented by different propositions (which would be even more propositions than Jape would be able to handle)!

In this proof, we use the outcome of the Proof 1 (i.e. that a donut implies a stable state) as our premise, such that we will be able to show that we *can* get to a donut in one iteration of a 3x3 grid of all T's.

We will start with the grid:

T	T	T
T	T	T
T	T	T

Define the variables as:

for time = 0

A	B	C
D	E	F
G	H	P

for time = 1

	$\neg R$	

1: $A, B, C, D, E, F, G, H, P, ((E \wedge \neg(\neg D \wedge \neg B \wedge \neg F \wedge \neg H)) \rightarrow \neg R), ((E \wedge \neg(D \wedge \neg B \wedge \neg F \wedge \neg H)) \rightarrow \neg R), ((E \wedge \neg(\neg D \wedge \neg B \wedge \neg F \wedge \neg H)) \rightarrow \neg R)$	premises
2: $((E \wedge \neg(\neg D \wedge \neg B \wedge \neg F \wedge \neg H)) \rightarrow \neg R), ((E \wedge \neg(D \wedge \neg B \wedge \neg F \wedge \neg H)) \rightarrow \neg R), ((E \wedge \neg(\neg D \wedge \neg B \wedge \neg F \wedge \neg H)) \rightarrow \neg R), (A \wedge B \wedge C \wedge D \wedge \neg R \wedge F \wedge G \wedge H \wedge P) \rightarrow S$	premises
3: $A \wedge B$	\wedge intro 1.1,1.2
4: $A \wedge B \wedge C$	\wedge intro 3,1.3
5: $A \wedge B \wedge C \wedge D$	\wedge intro 4,1.4
6: D	\wedge elim 5
7: $\neg D \wedge \neg B \wedge \neg F \wedge \neg H$	assumption
8: $\neg D \wedge \neg B \wedge F$	\wedge elim 7
9: $\neg D \wedge \neg B$	\wedge elim 8
10: $\neg D$	\wedge elim 9
11: \perp	\neg elim 1.4,10
12: $\neg(\neg D \wedge \neg B \wedge \neg F \wedge \neg H)$	\neg intro 7-11
13: $E \wedge \neg(\neg D \wedge \neg B \wedge \neg F \wedge \neg H)$	\wedge intro 1.5,12
14: $\neg R$	\rightarrow elim 2.1,13
15: $A \wedge B \wedge C \wedge D \wedge \neg R$	\wedge intro 5,14
16: $A \wedge B \wedge C \wedge D \wedge \neg R \wedge F$	\wedge intro 15,1.6
17: $A \wedge B \wedge C \wedge D \wedge \neg R \wedge F \wedge G$	\wedge intro 16,1.7
18: $A \wedge B \wedge C \wedge D \wedge \neg R \wedge F \wedge G \wedge H$	\wedge intro 17,1.8
19: $A \wedge B \wedge C \wedge D \wedge \neg R \wedge F \wedge G \wedge H \wedge P$	\wedge intro 18,1.9
20: S	\rightarrow elim 2.4,19

Explanation of Proof Steps:

- The first important step is the Negation Elimination on lines 7-12, which gives us the truth values of the neighbours for the space we are looking at— the surroundings of E. We want to show how E turns into $\neg R$ using the premises (i.e. the constraints in our model).
 - For simplicity, we are only showing how E turns into $\neg R$ in one iteration; NOT how any of the other propositions remain T within that same iteration.
- We can show that because D, B, F, H are alive, this means that the case of 4 neighbours being alive is true. This proves that the condition of E having 2 or 3 neighbours is F.
- We have that E is T, and the condition of 2 or 3 neighbours is F. Therefore— by the rules of the game— this means that E will die in the next iteration, i.e. $\neg R$ will be T.
- From here, we can conjoin $\neg R$ with the other propositions representing the edges of the donut (A, B, C, D, F, G, H, P) in order to get the full propositions representing the donut as a grid.
- We know that a donut is stable via our Proof 1. Thus, S will hold.

✧ Proof 3

For the third proof we aimed to show how an alive cell with 2 or 3 neighbours will remain alive in the next iteration. To translate this into logic, our proof showed that if we have a proposition that is currently T, given the premises that as long as it does not have four neighbours, one neighbour, or zero neighbours, it will be T in the next iteration.

This above concept is represented by a snippet of our code below, as this is what constitutes an alive, or T proposition:

```
return ~((~n1 & ~n2 & ~n3 & ~n4) | (n1 & ~n2 & ~n3 & ~n4) | (~n1 & n2 & ~n3 & ~n4) | (~n1 & ~n2 & n3 & ~n4) | (~n1 & ~n2 & ~n3 & n4) | (n1 & n2 & n3 & n4))
```

	n ₄	
n ₁	A	n ₃
	n ₂	

1: S, T, B, C, $\neg((\neg S \wedge \neg T \wedge \neg B \wedge \neg C) \rightarrow D) \vee ((S \wedge \neg T \wedge \neg B \wedge \neg C) \rightarrow D) \vee ((\neg S \wedge T \wedge \neg B \wedge \neg C) \rightarrow D) \vee ((\neg S \wedge \neg T \wedge B \wedge \neg C) \rightarrow D) \vee ((\neg S \wedge \neg T \wedge \neg B \wedge C) \rightarrow D) \vee ((S \wedge T \wedge B \wedge C) \rightarrow D)$	premises
2: $\neg S \wedge \neg T \wedge \neg B \wedge \neg C$	assumption
3: $\neg S \wedge \neg T \wedge \neg B$	\wedge elim 2
4: $\neg S \wedge \neg T$	\wedge elim 3
5: $\neg S$	\wedge elim 4
6: \perp	\neg elim 1.1,5
7: D	contra (constructive) 6
8: $(\neg S \wedge \neg T \wedge \neg B \wedge \neg C) \rightarrow D$	\rightarrow intro 2-7
9: $((\neg S \wedge \neg T \wedge \neg B \wedge \neg C) \rightarrow D) \vee ((S \wedge \neg T \wedge \neg B \wedge \neg C) \rightarrow D)$	\vee intro 8
10: $((\neg S \wedge \neg T \wedge \neg B \wedge \neg C) \rightarrow D) \vee ((S \wedge \neg T \wedge \neg B \wedge \neg C) \rightarrow D) \vee ((\neg S \wedge T \wedge \neg B \wedge \neg C) \rightarrow D)$	\vee intro 9
11: $((\neg S \wedge \neg T \wedge \neg B \wedge \neg C) \rightarrow D) \vee ((S \wedge \neg T \wedge \neg B \wedge \neg C) \rightarrow D) \vee ((\neg S \wedge T \wedge \neg B \wedge \neg C) \rightarrow D) \vee ((\neg S \wedge \neg T \wedge B \wedge \neg C) \rightarrow D)$	\vee intro 10
12: $((\neg S \wedge \neg T \wedge \neg B \wedge \neg C) \rightarrow D) \vee ((S \wedge \neg T \wedge \neg B \wedge \neg C) \rightarrow D) \vee ((\neg S \wedge T \wedge \neg B \wedge \neg C) \rightarrow D) \vee ((\neg S \wedge \neg T \wedge B \wedge \neg C) \rightarrow D)$	\vee intro 11
13: $((\neg S \wedge \neg T \wedge \neg B \wedge \neg C) \rightarrow D) \vee ((S \wedge \neg T \wedge \neg B \wedge \neg C) \rightarrow D) \vee ((\neg S \wedge T \wedge \neg B \wedge \neg C) \rightarrow D) \vee ((\neg S \wedge \neg T \wedge B \wedge \neg C) \rightarrow D) \vee ((S \wedge T \wedge B \wedge C) \rightarrow D)$	\vee intro 12
14: \perp	\neg elim 13,1.5
15: D	assumption
16: \perp	hyp 14
17: $\neg D$	\neg intro 15-16

Within our Jape proof;

- Let $n_1 = S$
- Let $n_2 = C$
- Let $n_3 = B$
- Let $n_4 = T$
- Let $\neg D$ = dead cell
- Let D = alive cell

Explanation of Proof Steps:

- We can use the implication introduction to open an assumption on line 2 in order to prove that we have the case of 4 neighbours being alive.
- From here we perform multiple lines of disjunction introduction, in order to get a statement in the form of the negation of the premise.
- Then we can do a negation elimination on the premise in order to obtain a bottom symbol, representing the fact that because there are in fact 4 neighbours, the starting condition of, 0,1 or 4 neighbours is true. Then we can conclude that there are not 2 or 3 neighbours, so therefore the cell will die in the next iteration, i.e $\neg D$
- Shows how an alive cell with two or three neighbours is alive in the next iteration as well (i.e. it will NOT be dead).

★ *First-Order Extension*

The model does not change too much when transferring it into predicate logic, however it does get easier to understand and more concise to model.

Recall that we had a number of propositions $A_i(H,L,W)$ where each proposition A_i represented a different cell on a $L \times W$ grid that iterated through a total of H turns.

Our terms could be defined as $x_{(H,L,W)}$ where these terms stand in for what we have previously defined to be our propositions.

The universe of discourse for the predicates we define below is the set of all the terms in a given starting configuration.

- For example, with a starting configuration of $3 \times 3 \times 3$, our universe of discourse would be the set of $x_{(0-3,0-3,0-3)}$, resulting in a set size of $3 \times 3 \times 3$, which is 27 objects.

With the objects in the universe of discourse defined, we can specify our predicates. We would have the following predicates in order to represent key attributes of the objects:

- $A(x)$ maps to the set of alive cells
 - It would be true if the cell x is alive
 - Would be false if the cell x is dead
- $P(x)$ maps to the set of cells that have either 2 or 3 neighbours
- $Q(x)$ maps to the set of cells that have exactly 3 neighbours

We can define the 4 rules that define the behaviour of the cells from iteration to iteration to be:

- $\forall x ((A(x_{(H,L,W)}) \wedge P(x_{(H,L,W)})) \rightarrow A(x_{(H+1,L,W)}))$

- $\forall x ((A(x_{(H,L,W)}) \wedge \neg P(x_{(H,L,W)})) \rightarrow \neg A(x_{(H+1,L,W)}))$
- $\forall x ((\neg A(x_{(H,L,W)}) \wedge Q(x_{(H,L,W)})) \rightarrow A(x_{(H+1,L,W)}))$
- $\forall x ((\neg A(x_{(H,L,W)}) \wedge \neg Q(x_{(H,L,W)})) \rightarrow \neg A(x_{(H+1,L,W)}))$

This is similar to the current implementation of the constraints. The introduction of predicates does not change this section too much, however it does make it much easier to translate the problem. Predicates allow us to turn our constraints for each proposition into one encapsulating universal quantifier constraint.

We could define a stable state with the following expression:

$$\forall x_{(H,L,W)} ((A(x_{(H,L,W)}) \wedge P(x_{(H,L,W)})) \vee (\neg A(x_{(H,L,W)}) \wedge \neg Q(x_{(H,L,W)})))$$

This means that for all squares, it has to be either of two things:

- It is currently alive, and it will stay alive in the next round, or
- It is dead, and it will stay dead in the next round

This is much easier to implement and understand— rather than having 2 implications for each of the propositions, there is one solid statement that encapsulates the whole condition of a stable state.

C3) Exploring The Model

Corresponds to (D2) Implementation

There were various ideas that we had in mind in order to explore the model we have created for the Game of Life. In order to implement these exploration models in our code, we have created a new parameter called `constraintType`. `constraintType` will refer to four different sets of rules that we will be using to explore our model:

1. **Normal** — “normal” rules feed in our constraints as outlined above for the implementation of the Game of Life in logic as it was originally intended to be... i.e. with the rules such that 2 or 3 neighbours are required to be T for a currently T cell to be T on the next iteration, and that a F cell needs 3 T neighbours to be T on the next iteration.
2. **Reverse** — “reverse” rules switch the output of the normal rules... i.e. 2 or 3 neighbours that are T will result in a T cell being F in the next iteration, and 3 T neighbours to an F cell result in it being F in the next iteration.
3. **At Least 1** — “atLeast1” is a variation of the normal rules, just with an additional added constraint specifying that there must be at least 1 cell alive (a proposition that outputs T) in the last round. This essentially gives us every stable state that is not merely a result of all the propositions being dead, or false. Having at least one proposition be alive adds a layer of interest, as the stable state is likely to be a unique pattern or design that we can further analyze.
4. **Total Solutions** — “totalSolutions” just counts the amount of possible solutions of, this does not count the amount of stable states, just the amount of valid games that can be played on a certain grid.

With these four sets of constraints specified like so, we can use each of them to perform some math and make some observations in order to further explore our model and analyze the results of different constraints in contrast with the normal rules of the game.

★ *Exploring with Default Constraints*

- ☆ The first type of information we will gather pertains to the number of stable states that can be found in different, square-sized $L \times W$ grids when following **normal constraints**. We will be looking at grid sizes of 2x2, 3x3, and 4x4, with the H value ranging from 1-9. In other words, these $L \times W$ sized-grids will iterate H amount of times, and output the number of stable states that are able to form within these certain grid sizes and specified amount of time.

Number of Stable States per Grid Size, Based on Time 😊			
	2x2	3x3	4x4
H = 1	2	10	111
H = 2	12	189	8789
H = 3	16	406	33560
H = 4	16	484	52666
H = 5	16	504	60625
H = 6	16	512	63804
H = 7	16	512	65126
H = 8	16	512	65328
H = 9	16	512	65525
Total Solutions	16 $2^{2 \times 2}$	512 $2^{3 \times 3}$	65536 $2^{4 \times 4}$

With the data we have gathered within this table, we can confirm the following of our previous assumptions:

- With every new iteration of any grid of propositions, a state will have more time to become stable. For example, we may have a state on any given iteration of H that would have resulted in a stable state in just another iteration. Thus, given this time (as H increases), we also see an increase in the number of resulting stable states.
- There are specific stable patterns that we can observe that occur over and over again in different grid sizes.
- We can see that if we allocate more space to any grid in order to make it larger (i.e. increase the $L \times W$ grid sizes), the number of stable states that result increases exponentially.
- There is a cap to the amount of possible stable states given any grid configuration, and it is $2^{L \times W}$
 - If this cap is not the same as all possible combinations, then this is an indication that there are some states that are infinite.

- For 2x2 and 3x3 grids, we found that there are no infinite solutions. With this information, we find that it is likely that there are no infinite solutions for any of the other sizes. Thus, results are inconclusive.

At which iteration do these grids reach total stability?

- The 2x2 grid stabilizes at 3 rounds, and the 3x3 grid stabilizes at 6 rounds.
- Based on the data, it seems as if the 4x4 grid is very close to stabilizing, potentially even on the 10th iteration.
- We can also use the information we gathered to hypothesize that the 4x4 grid will stabilize at $6 \times 2 = 12$ rounds, as this follows the pattern. Unfortunately however, our systems were not powerful enough to compute more than 9 normal iterations of a 4x4 grid. Hence, although there might be a mathematical connection between these values, we lack the data to make sufficient analysis.

☆ Another venture we can embark upon in terms of exploration would be to attempt to determine the optimal configuration of grid size. Optimization within our context would be defined as an algorithm for models to produce the most stable states per area. In other words, what configuration of grid sizes lead to the most number of stable states? For example, we might want to determine if a square has more solutions than a line, or if a rectangle is the best.

Suppose we restrict the grid such that $L + W = 8$, and we have a constant time $H = 3$. We will analyze the resulting models from there.

H x L x W	Stable State Solutions
3x7x1	120
3x6x2	3194
3x5x3	18692
3x4x4	33560

From this table, we can see that the configuration that leads to the most amount of solutions/stable states is the square grid. This is likely due to the fact that the square has the largest area for any given perimeter, thereby maximizing the possible area we have to work with and leading to more propositions.

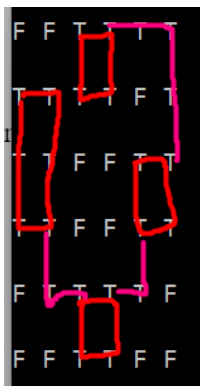
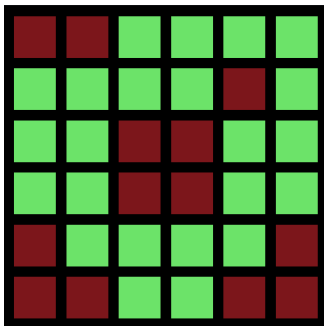
☆ The difference between the amount of solutions over each iteration is what we will refer to as a grid's speed. In other words, the rate at which the amount of solutions increases with respect to time is called speed.

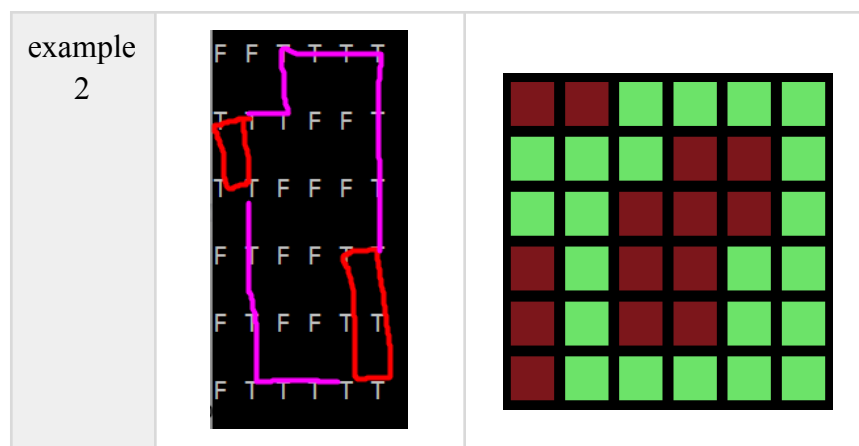
- We have found that the speed of any square grid has a relationship that can be modelled similarly to a normal distribution. The amount of increase in stable states (starting from $H = 0$) tends to start out slow, then grows as `gameLength` increases. It then reaches a maximum from which it starts to slow down as `gameLength` continues to increase.
- This relationship (i.e. the amount of stable states for any given grid) can be modelled by a graph with a curved/S like shape graph, possibly similar to a translated logistic/Sigmoid function.

☆ Something else we decided to focus on during our exploration was to attempt to find any stable patterns, or **still-lives**. These patterns are simply clusters of propositions that will remain stable. For example, a 2x2 square remains stable as each cell always has 2 neighbours that are T.

- This is a point of interest to us, as we modified the rules of the original Game of Life in order to make our model manageable— i.e. we changed each cell from having 8 neighbours to only considering 4 neighbours within the cardinal directions. We wonder if this change gave rise to new still-lives, different from any of the [original ones that have been documented](#).

By using a large grid, 6x6, we set $H = 1$ and rerun the code until we were given randomly generated solution states that had some interesting features to them, which we further analyzed.

	Solution Output	Visualized
example 1		



In both examples displayed above, we noticed that there were recognizable still-lives (those being squares and rectangles of T's), however they were connected with shapes resembling to the shape of an L.

- We have decided to call these L-like shapes **stable connectors**.
- We found that stable states can be connected via a line of T's as long as both ends of these connectors attach to a known still-life. Then, the combination of stable connectors and still-lives will be representative of an overall stable state.

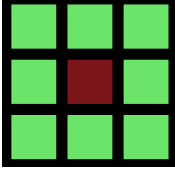
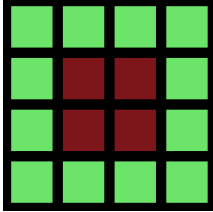
We use this information in conjunction with our other findings in the [following section](#) that outlines a few of the specific stable patterns we managed to observe.

☆ Stable Patterns/Still Lives:

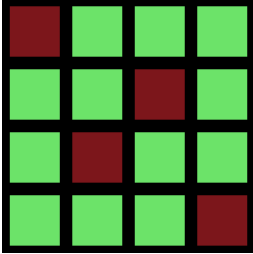
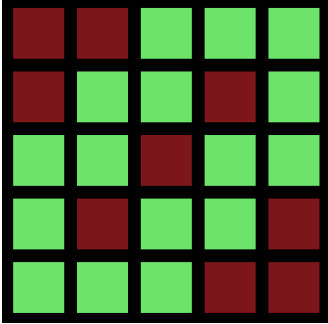
- ✧ **Quadrilaterals** — squares, rectangles, or extensions of these constitute a stable pattern as long as *at least* one of the dimensions (L , W , or both) is exactly 2 propositions.

1. Square	2. Rectangle

- ✧ **Donut** — Any quadrilateral that does not have at least one of L or W being 2 propositions will become a donut, as the cells in the center will die if the dimensions are both greater than 2. Extensions of this pattern also work as donuts.

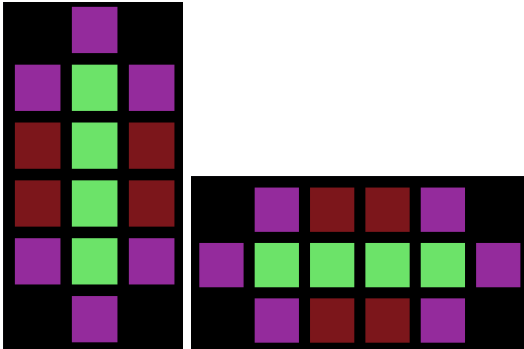
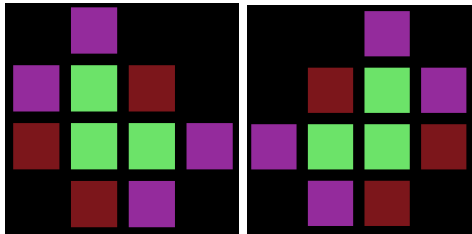
1. Donut	2. Extended Donut
	

✧ **Figure 8** — Another pattern we noticed is what appears to be a diagonal eight-like figure. It is surrounded by stable connectors aside for two corners where the propositions are F, as well as in the “holes” of the eight where the propositions are also F. Extensions of this pattern also work.

1. Figure 8	2. Extended Figure 8
	

☆ Stable Connectors:

Stable connectors are what we have defined to be a subset of stable patterns, but may have some level of variation. This variation is represented by the purple propositions in diagrams below where either 1 or 2 of them can be T, but not 3 and not 0.

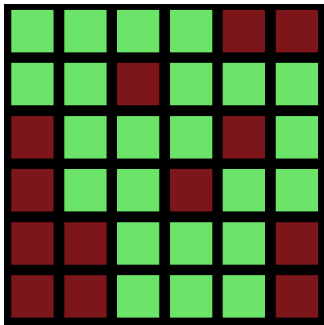
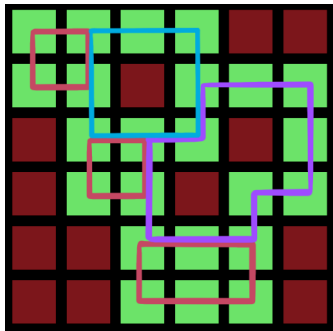
1. Line Connector	2. L Connector
	

☆ From here, we can create a guide that can be followed and used in order to figure out if any given state is stable or not.

1. Classify all stable patterns within the state.
2. Classify all stable connectors within the state.
3. Verify that the *purple propositions* of the stable connectors consist of only 1 or 2 T's.
4. Check if there are any other propositions that are T outside of these.

If yes, the state is not stable. If **not**, then the state is stable.

We will demonstrate the application of this guide using the example illustrated below:

Generated State	Still-Lives Depicted (Step 1)
	

Using the steps indicated above:

1. We will classify all the stable patterns that we can within the grid. We can see that the orange sections indicate quadrilaterals (two squares and one rectangle), the blue section is a donut, and the purple section is a figure 8.
2. We are unable to find any depicted stable connectors, so we will skip this step.
3. Irrelevant as there are no stable connectors.
4. We can see that every T value is accounted for in Step 1, and there are no stray T values that cannot be categorized. Thus, the state depicted is stable.

★ Exploring with Reverse Constraints

In this section, we will explore the model with the truth values of the rules being swapped in order to see what sorts of outputs we receive. Within the code, we use the “reverse” tag that mirrors the following rules in logic:

- $A_{(0,0,0)}$ is **alive** and has **2 OR 3 neighbours** $\rightarrow A_{(1,0,0)}$ is **dead**.
- $A_{(0,0,0)}$ is **alive** and has **NOT 2 OR 3 neighbours** $\rightarrow A_{(1,0,0)}$ is **alive**.
- $A_{(0,0,0)}$ is **dead** and has **3 neighbours** $\rightarrow A_{(1,0,0)}$ is **dead**.
- $A_{(0,0,0)}$ is **dead** and has **NOT 3 neighbours** $\rightarrow A_{(1,0,0)}$ is **alive**.

To begin with, we attempted to test these rules with a 3x3 grid and contrast the solutions over increasing iterations; as H grew. However, we quickly noticed some strange behaviour (documented in the table below). When H was odd, we would have a large number of solutions. However, whenever H was even, these numbers dropped drastically.

	3x3	2x2
H = 1	164	11
H = 2	35	1
H = 3	270	11
H = 4	99	1
H = 5	318	11
H = 6	107	1

Total Solutions	512	16
-----------------	-----	----

While analyzing the values themselves, we could not come up with a pattern or understand why this behaviour was occurring. So, we turned to our code to determine if anything went wrong there. And indeed it did— as we found that we were missing an implication for what constituted a stable state.

As the below truth table demonstrates, the implications in column 5 indicate stability in one proposition. We extend this definition in the actual code to apply to every proposition in a grid. This column essentially demonstrates that propositions that adhere to the $T \rightarrow T$ and $F \rightarrow F$ implications do not change; which is what we want our stable states to be.

- Or at least, this is what the stable state constraint *should* be. However, until this point, we were representing stable states by the expression in column 3. This column does not account for the fact that $F \rightarrow T$ is a change, and thus the “stable state” indicator gets flagged as being T.
- Essentially, whenever the check for a stable state occurred, if a proposition that was currently F will result in it being T in the next iteration (according to the constraints we have in regards to neighbours), then this would be considered stable because $F \rightarrow T$ is T. However, that is not what we wanted in this case, as this does not accurately reflect the nature of a stable state.

English Descriptors	1 Current Cell, A	2 A in $H+1$ Iteration	3 Check for Change if $A = T$	4 Check for Change if $A = F$	5 No Changes Indicate Stability
Logical Expressions	S	P	$S \rightarrow P$	$\neg S \rightarrow \neg P$	$S \rightarrow P \wedge \neg S \rightarrow \neg P$
Truth Values	T	T	T	T	T
	T	F	F	T	F
	F	T	T	F	F
	F	F	T	T	T

Upon further examination, this mistake allowed for the case of

F F T T

F F \rightarrow T T to be considered stable.

Normally, a state of all F's would be considered a stable state. However, since we are using the reversed constraints, this causes each F proposition to be T in the next iteration... meaning that a state of all F's is not stable, as it changes in one iteration.

- Thus, in the solutions listed in [this table](#), states of full F's were considered as stable, though they should not have been.

This ended up triggering a cycle, as a state of full F's would imply a state of full T's which itself is not stable... as— since we have reversed our constraints— this normally would also be a stable state, but with the reversed constraints, a state of full T's results in a state of full F's. We can see that this is a cyclic pattern that will repeat indefinitely.

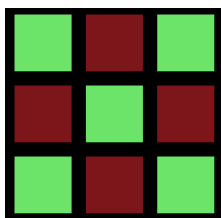
Within this model, this is the only infinitely repeating cycle we were able to find (due to computation limits with our machines):

F F T T F F T T F F T T

F F \rightarrow T T \rightarrow F F \rightarrow T T \rightarrow F F \rightarrow T T and so on.

This helped illustrate the behaviour of the reverse rules. Configurations that aren't reflective of stable patterns within these rules will result in a cycle between a grid of all T's and a grid of all F's. The use of reverse rules tends to end up in this specific infinite sequence very often.

- ☆ The only possible stable state that adhere to the reverse rules is exactly this grid, which we have dubbed the “checkbord”:



- ☆ After fixing our definition of a stable state within logic in our code, we found the proper solutions to a 3x3 grid configuration.

	3x3
H = 1	5
H = 2	13
H = 3	13
H = 4	13
H = 5	13
H = 6	13
Total Solutions	13

Upon testing grids sized 2x2 - 12x12, we find that there are no possible solutions (0) when these reverse constraints are applied. It might be reasonable to conclude that the only stable state in the reverse rules is the checkerboard.

★ *Exploring with atLeast1 Constraints*

Finally, we implemented a condition where at least one proposition must be T, or alive in order to constitute a stable state. We did this using the `Bahaus.atleastone()` function. This applies to only the final stage of each iteration. For example, if $H = 3$ in a 2x2 grid, then the `atLeast1` requirement would apply only to the state on the third iteration; the 3x2x2 grid. This is the grid that is then evaluated to determine whether it is a stable state or not.

- **atLeast1** — states with at least one T, and is stable.
- **Total** — the total number of states (including stable states comprised of all F's) depending on the current iteration, H .
- **Difference** — the difference between (Total - atLeast1)... i.e. the number of stable states within the current iteration that is comprised of all F's.
- **F Percent** — the percent of states resulting in all F's compared to the total number of states for the current iteration.

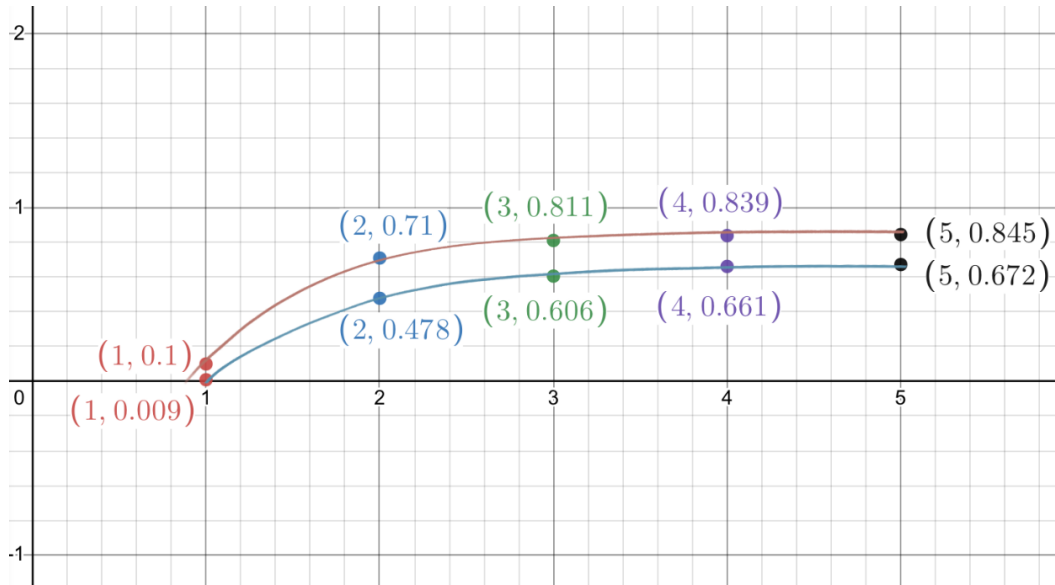
	3x3				4x4			
	atLeast1	Total	Difference	F Percent	atLeast1	Total	Difference	F Percent
H = 1	9	10	1	10%	110	111	1	0.9%
H = 2	59	189	135	71%	4588	8789	4201	47.8%
H = 3	78	406	328	81.1%	13228	33560	20332	60.58%
H = 4	78	484	406	83.9%	17828	52666	34838	66.15%
H = 5	78	504	426	84.5%	19858	60625	40767	67.24%
H = 6	78	512	434	84.8%	Cannot Compute Due To Computer Limitations			
H = 7	78	512	434	84.8%				

We can determine the speed for each iteration by taking the percent of stable states that have at least one proposition being T to the total number of states, and subtracting this from 1 to give us the percentage of stable states that are all F's.

Sample calculation: If we wish to determine what percentage of the total solutions are grids consisting of all F propositions for the 1x3x3 grid, we will do the following arithmetic...

$$1 - \frac{\text{states with at least one T}}{\text{total stable states}} = 1 - \frac{9}{10} = 0.1 = 10\%$$

We could also try to get a visual on the speed for each grid by simply plotting the F percentage against each iteration. Using the gameLength H as the independent variable, and the percentage of F states as the dependent variable, we can plot the values we found to attempt to find a relationship between them.



The red graph indicates data gathered from our work with the 3x3 grid, while the blue graph is for the 4x4 grid. It is interesting to note that the graphs take on the exact same shape, apart from some sort of vertical shift. It appears that the best-fitting relations may be some sort of restricted rational function, potentially in the form of $-M(\frac{1}{x^2} + 1)$, $\{x > 0\}$.

We were able to make some other interesting observations using the data we gathered in the table above...

- A 3x3 grid will always reach a stable pattern with at least one T proposition in 3 rounds or less.
- The F percentage approaches a certain number.
 - In the case of the 3x3 grid, it is approximately 85%.
 - In the case of the 4x4 grid, our computers are unfortunately unable to accurately compute higher grid configurations above $H = 5$. Hence, we are unable to figure which percentage this value is approaching, though it does seem like it is reaching a certain threshold.

Looking at the previous cases, we can extrapolate with the data we gathered for the case of a 5x5 grid...

- Due to the fact that 3x3 approached 85% within 3 turns, and 4x4 approached somewhere around 70% within 5 turns, we are able to notice a slowly decreasing pattern. We can infer that somewhere around 50-60% of all the final stable states in a 5x5 grid will be nothing but F's within somewhere around 7-8 turns.