CISC 204 - Project Draft (D2-D5)

# REQUESTED FEEDBACK

**Question 1**: Is there anything our group can do to make the code runnable for grid sizes greater than 4x4x4? We want to find patterns and extend the model above this, however larger grid sizes surpassing this are far too big to be able to be run.

**Question 2**: Is there any insight you can provide on why this error continues to show up, as detailed in the “[During Development](#2hdx4gpjvt4n)” section of this document?

# 

# C2) Creating The Model

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 will only take into account 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.

## 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.

|  |
| --- |
| 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

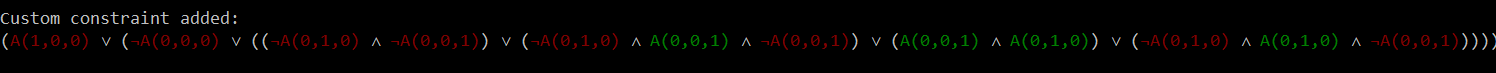
* Each cell, apart from the starting state, has constraints that must be followed in order to dictate when it will be alive, and when it will be dead. In each iteration of the game, each cell is reassessed according to these constraints/rules, and changes in state to be alive or dead, accordingly.
* 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 byexactly **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** → A(1, 0, 0) is **alive**.
  + A(0, 0, 0) is **alive** and has **NOT 2 OR 3 neighbours** → A(1, 0, 0) is **dead**.
  + A(0, 0, 0) is **dead** and has **3 neighbours** → A(1, 0, 0) is **alive**.
  + A(0, 0, 0) is **dead** and has **NOT 3 neighbours** → A(1, 0, 0) is **dead**.
* 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.
* 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 ↔ A(x+1, y, z) is true” where *x* is the final round for the game.
  + 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.
  + *We are yet to implement this feature in our code*, however, this is the plan that we have decided on in order to model stable states in logic.

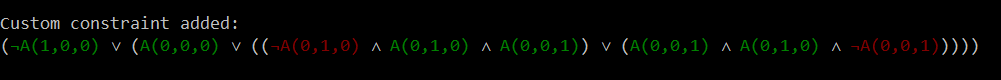
## During Development

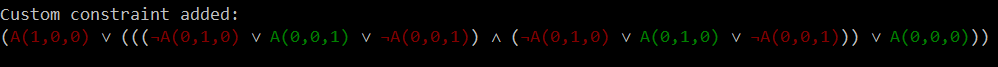
There was one specific solution that plagued us… within a 2x2x2 grid, we received the following:

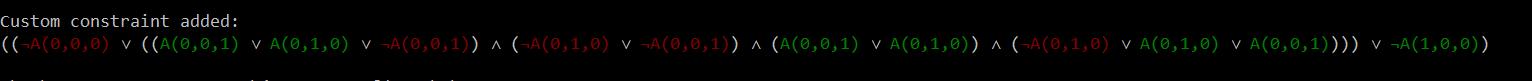
| What the starting configuration was: | What the code resulted in: | What the **correct result** should be: |
| --- | --- | --- |
| | T | T | | --- | --- | | T | T | | | F | T | | --- | --- | | T | T | | | T | T | | --- | --- | | T | T | |

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









* 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!!!
  + *We must figure out what is causing this issue.*
* 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.
  + We should come back to verify that we receive the same number after implementing stable states.

# C3) Exploring The Model

Below, we detail various ways in which we can explore the model we have created in logic…

* We can change the size of the grid, and the number of iterations in order to spot interesting finds.
  + Currently, turning each of the proposition sizes (H, W, L) to 5 crashes our computer systems, as the computation is much too large… :(
  + We could try experimenting with non-square/non-cubical grid sizes, as this would lower the amount of computations required.
  + A game length (H) of 1 will give all the stable states for a certain grid size… looking at all of those allows us to observe every possible stable pattern.
* We are looking for unique patterns of stable states, due to our reduction of each square's scope (i.e. from each cell having 8 neighbours to 4 neighbours).
  + Perhaps look into repeating patterns, recognizing and defining them in logic (the same way we defined a stable state in logic).
  + Currently, we have found donuts, squares and rectangles as stable states that exist within our model.

# D5) Jape Proofs

## Proof 1

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 making it 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) ↔ A(x+1, y, z)

## Proof 2

We will show that a 3x3 grid where each proposition is T, this will result in a stable state in 1 step/iteration. We will most likely 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.

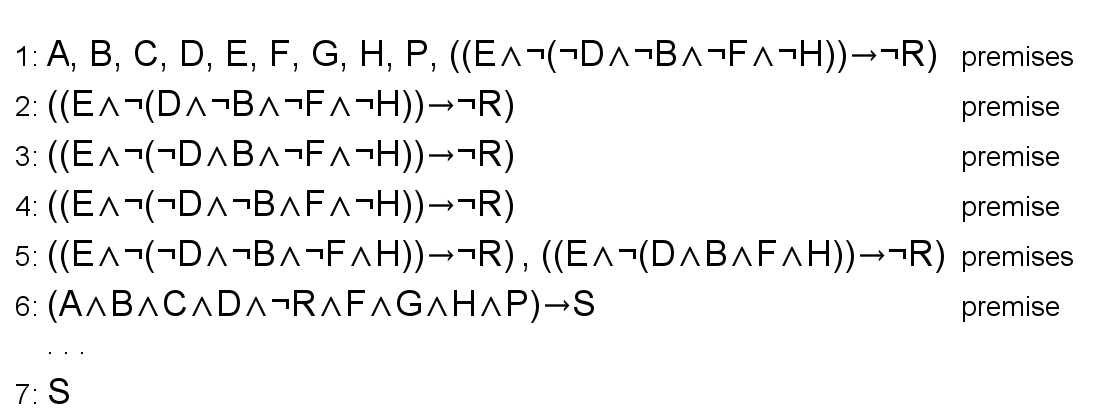
In this proof, we use the outcome of the Proof 1… i.e. that a donut implies a stable state, as our starting configuration will change to be a donut in 1 iteration.

We will start with the grid:

| T | T | T |
| --- | --- | --- |
| T | T | T |
| T | T | T |

Define the variables as:

| *for time = 0* | *for time = 1* |
| --- | --- |
| | A | B | C | | --- | --- | --- | | D | E | F | | G | H | I | | |  |  |  | | --- | --- | --- | |  | ¬ R |  | |  |  |  | |



## **Proof 3** *To Be Determined.*