Final Report: Mancala Modeling

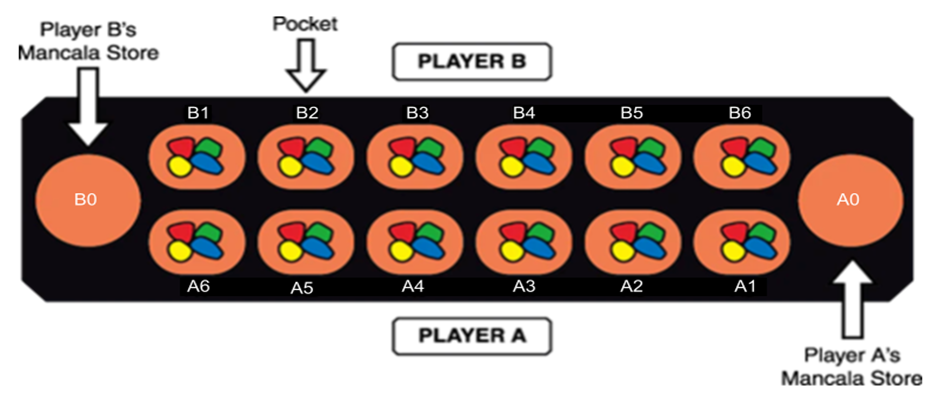
Amy Brons, Alan Teng, Dan Munteanu, Kate Edgar

Group 14 - CISC 204

# Project Summary

Mancala is a two-person ancient board game in which players strategically ‘sow’ and ‘capture game pieces called gems. The objective of the game is to get a higher score by strategically manipulating gems around the board, which is made up of 12 ‘pits’ (six per side) and two ‘stores’ (larger pockets at the end of each board, one per player). At the start of the game, each pit has four gems. Once it has been decided who goes first, that player picks up all the pieces from one of the pockets on their side of the board, and deposits one gem in each of the preceding pockets to the right, until they have no more pieces left in their hand. If the player’s last deposit lands in their store, they get another turn; This is called ‘banking’. Once a gem has been banked, it cannot be moved from the store and gems cannot be deposited in the opponent’s store. Similarly, if the player’s last gem lands in an empty pit on their side of the board, they claim all the pieces in the pit directly opposite theirs. The game ends when either of the players has six empty pits on their side. The player with the most gems in their respective store wins.

*Visualisation of a Mancala board:*

**

For our project, we will be modelling the optimal next move for player A, given a randomised board state. Move optimization is based on the four following strategies listed in order of precedence:

1. Putting player A’s final gem in their store
2. Preventing player B from depositing their final gem in their store
3. Collecting from player B’s pit
4. Move the right most non-zero pit

Our program will look at the number of gems in each pit and test if a strategy is applicable. The proposition with the highest truth value will be selected as the optimal next move.

# Propositions

Each pit has the following properties:

* GEMS = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24}
  + Each pit can contain anywhere between 0 and 24 gems
* ROW = {0, 1}
  + Where 0 represents player B’s side of the board and 1 represents player A’s side of the board
* COLUMN = {0, 1, 2, 3, 4, 5, 6}
  + Where 0 represents the pit closest to player A’s store and 6 represents the left most pit on player A’s side

We model a plausible game state using the following propositions:

* **Finalgem(r, c)**: The final gem landed in the pit located at row **r**, column **c**
* **SelectPit(r, c)**: The current pit is located at row **r**, column **c**
* **PitProposition(r, c, g)**: The current pit is located at row **r**, column **c** and contains **g** gems
* **PlayerTurnNext()**: True if player A gets another turn
* **PlayerCollects()**: True if player A will collect gems

# Constraints

The constraints are grouped into two categories (1) board specific constraints and (2) game constraints.

### Board Specific Constraints

* *A pit can only have a fixed number of gems* 
  + For every row **r** and column **c**, exactly one pit proposition is selected:

exactly-one(PitProposition(r, c, 0), PitProposition(r, c, 1))

* *The final gem can only be dropped in a single pit*
  + exactly\_one(Finalgem(1, 0) ∧ ¬ Finalgem(1, 1) ∧ ¬ Finalgem(1, 2) ∧¬ Finalgem(1, 3) ∧¬ Finalgem(1, 4) ∧ ¬ Finalgem(1, 5) ...
* *A selected pocket must have gems gems in it* 
  + For every column **c**, gems must be present:

constraint(SelectPit(0, c) ∧ ¬ PitProposition(0, c, 0))

### Game constraints

* *Selecting a pit will lead to a certain fixed board state* 
  + For every row **r** and column **c**:

constraint(SelectPit(r, c) → PitProposition(r, c+1, g+1) ∧ PitProposition(r, c+2, g+1) ∧ PitProposition(r, c+3, g+1) ∧ Finalgem(r, c)

* *If the final gem is dropped in player A’s store, then player A gets another turn* 
  + constraint(Finalgem(1, 6) → PlayerTurnNext())
* *Player A collects gems from two opposite pit if Player A’s last gem is dropped in an empty pit that is not their bank* 
  + For every row **r** and column **c**:

constraint(Finalgem(r, c) ∧ PitProposition(r, c, 1) → PlayerCollects())

# Model Exploration

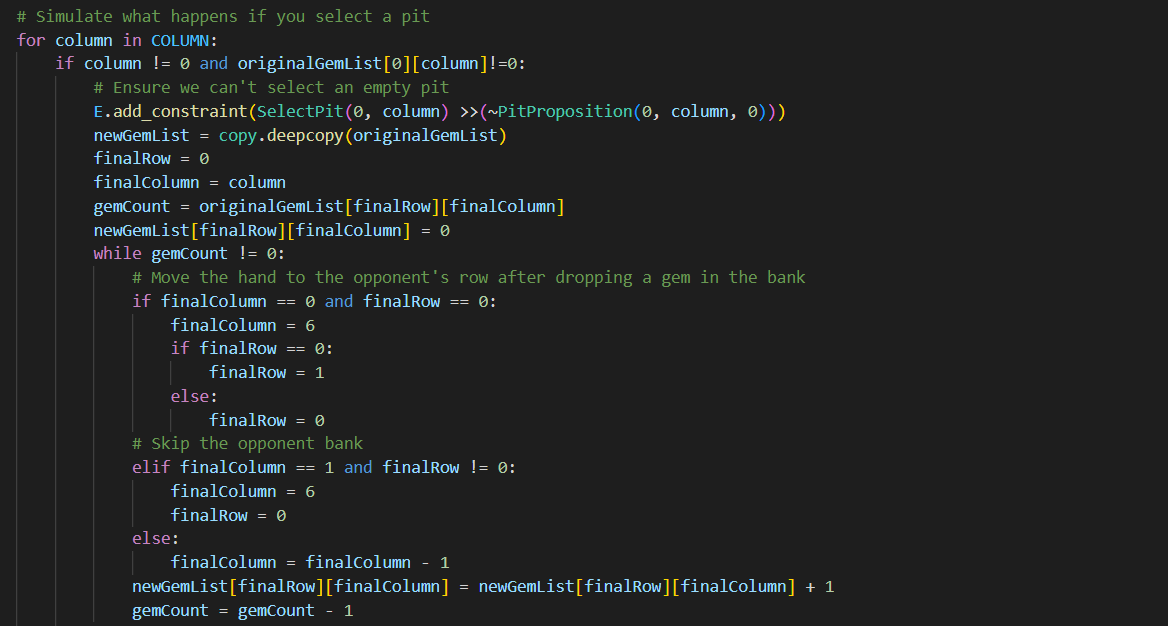
### Our Method

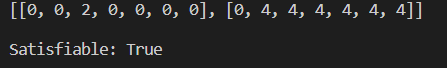
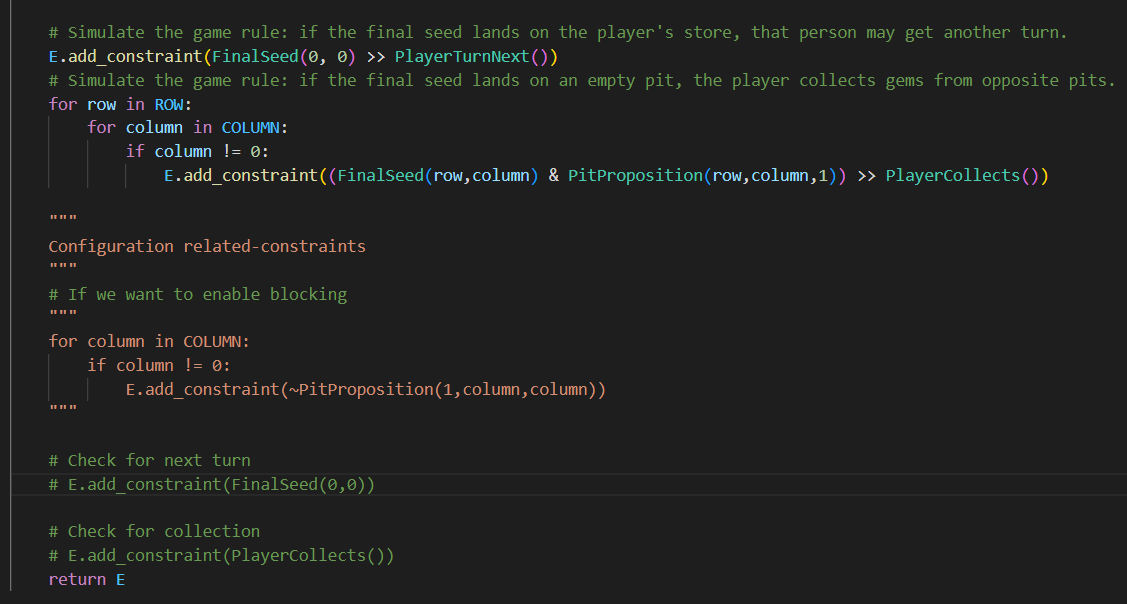
For this model we decided to take two parallel and connected approaches. After creating the logic file, we began to create two additional important files: *resolver.py* and *board.py*. These files work together to output a randomised board, and resolve it into small, logically readable functions. In an ideal world, (currently commented out in *board.py*) user input would build the board so the user could choose where to drop and disperse their gems. However, for the scope of this assignment, we suspended this ability and relied on a randomised placement of the 24 gems. Next, our *resolver.py* and *run.py* files both read and analyse the output as follows:

1. The *run.py* file reads the board (generated in *board.py*) as a global variable and feeds into our constraint function, which feeds into the propositions built.
2. Using Bauhaus, we were able to construct a logically sound *run.py* file that returns this board, as well as the satisfiability of the random board.
3. The *resolver.py* file uses this same logic to build and output the satisfiability per pit.

### *Run.py* - Logic File

To begin with our project we first constructed our logic using Bauhaus. In this file, we initially came across some difficulty in determining how to translate our constructed logic into the python file. After much deliberation, we were able to construct a solid logic run file.

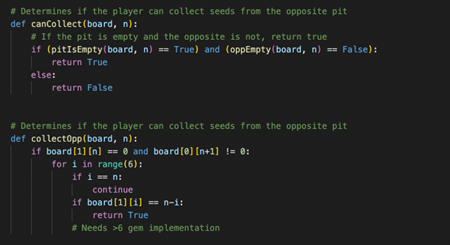
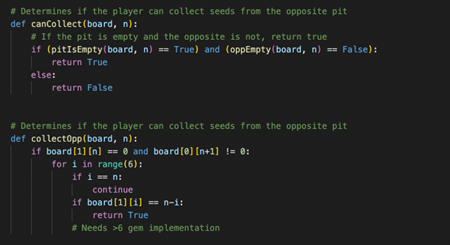
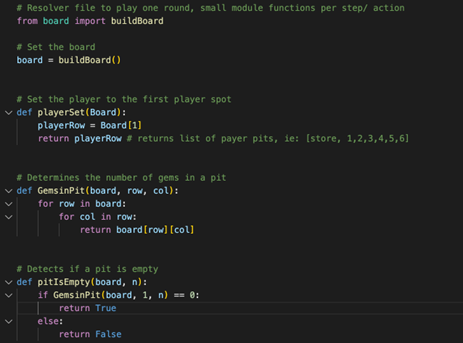
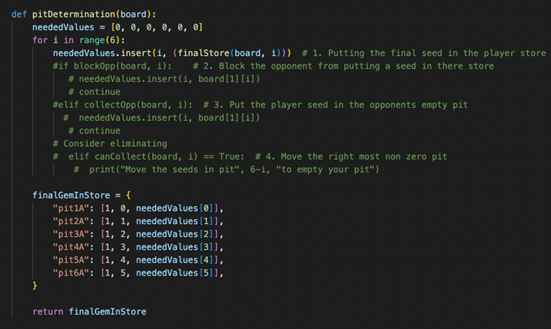




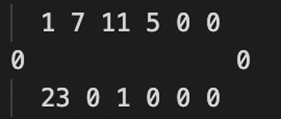
We also attempted to run an empty board (or at the very least an empty board on the player’s side). This returns an error, which makes sense given no pits can be selected.

### *Resolver.py* - Resolving and Testing

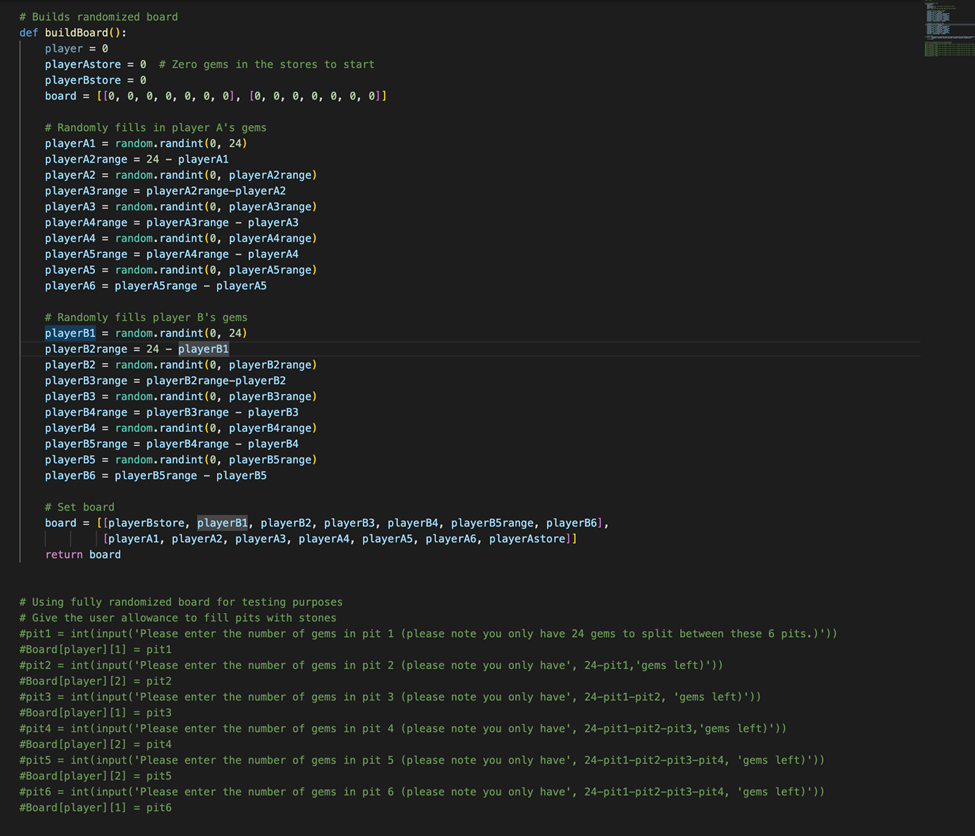
The second part of our modelling was creating a resolver file that modelled the board on the same logic, which we could use for output, visualisation, testing and debugging. The integration of these two files created many issues, but created a more stable and well rounded model.

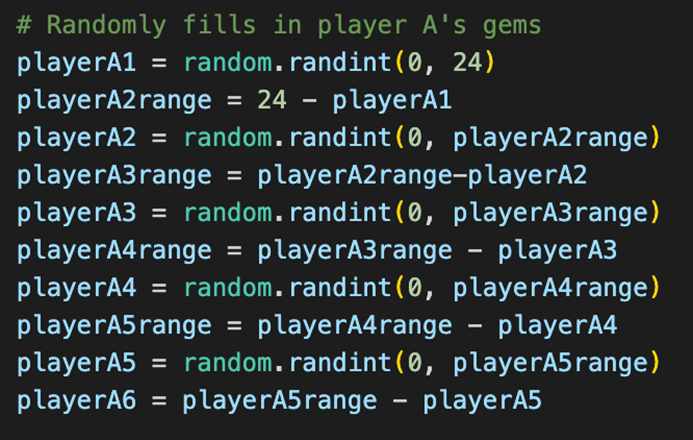
Here is some of the code that we created as functions, equal to the logic in *run.py:*

As mentioned before, a small visualisation of the board is produced in this file, as seen here for example:



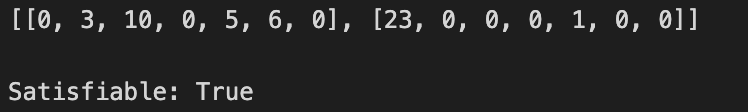
This visualisation is created in-part due to a file called *board.py* which randomises the gems in each pit, off of the initial 24 gems per player. Here is the code, and a close up of the randomization of the pits, which occurs for both players:

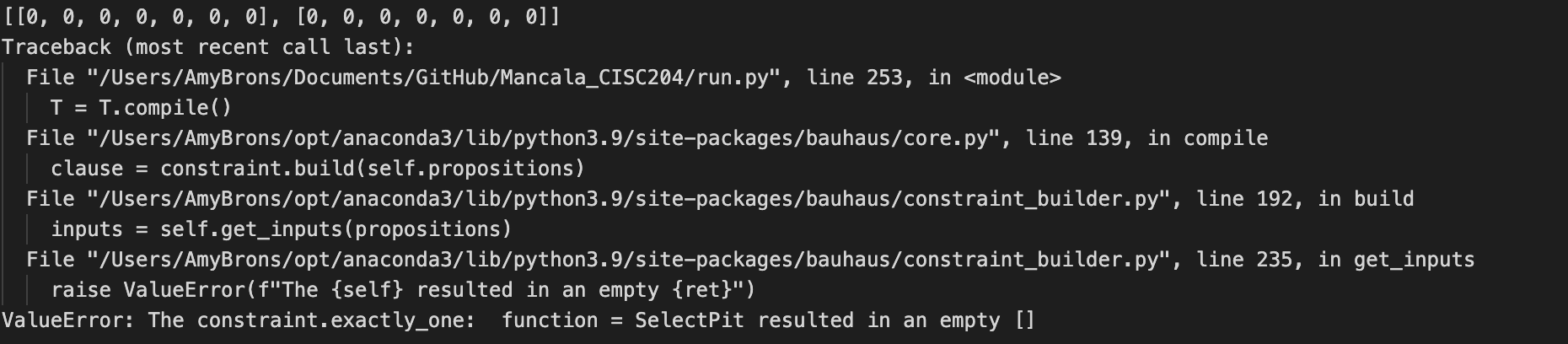




### Outputs - *Run.py* and *Resolver.py*

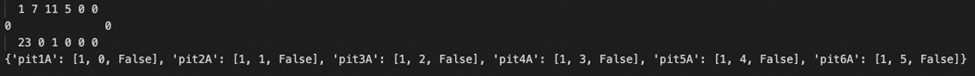
The *run.py* file output consists of the board print, with the satisfiability report. See here for context:



See the final section of this report labelled *Incomplete Items* for more context on this output. This output shows us that the board can be solved for this randomised output. This is correct, as theoretically that is true. To test this further, the only situation where the output would not be true is where there are zero gems. This is is the output for that test:  
  


This is correct, as there are no gems, so no pit can be selected. This helps us realise that our logic is working correctly.

Finally, here is an example of our full resolver output.

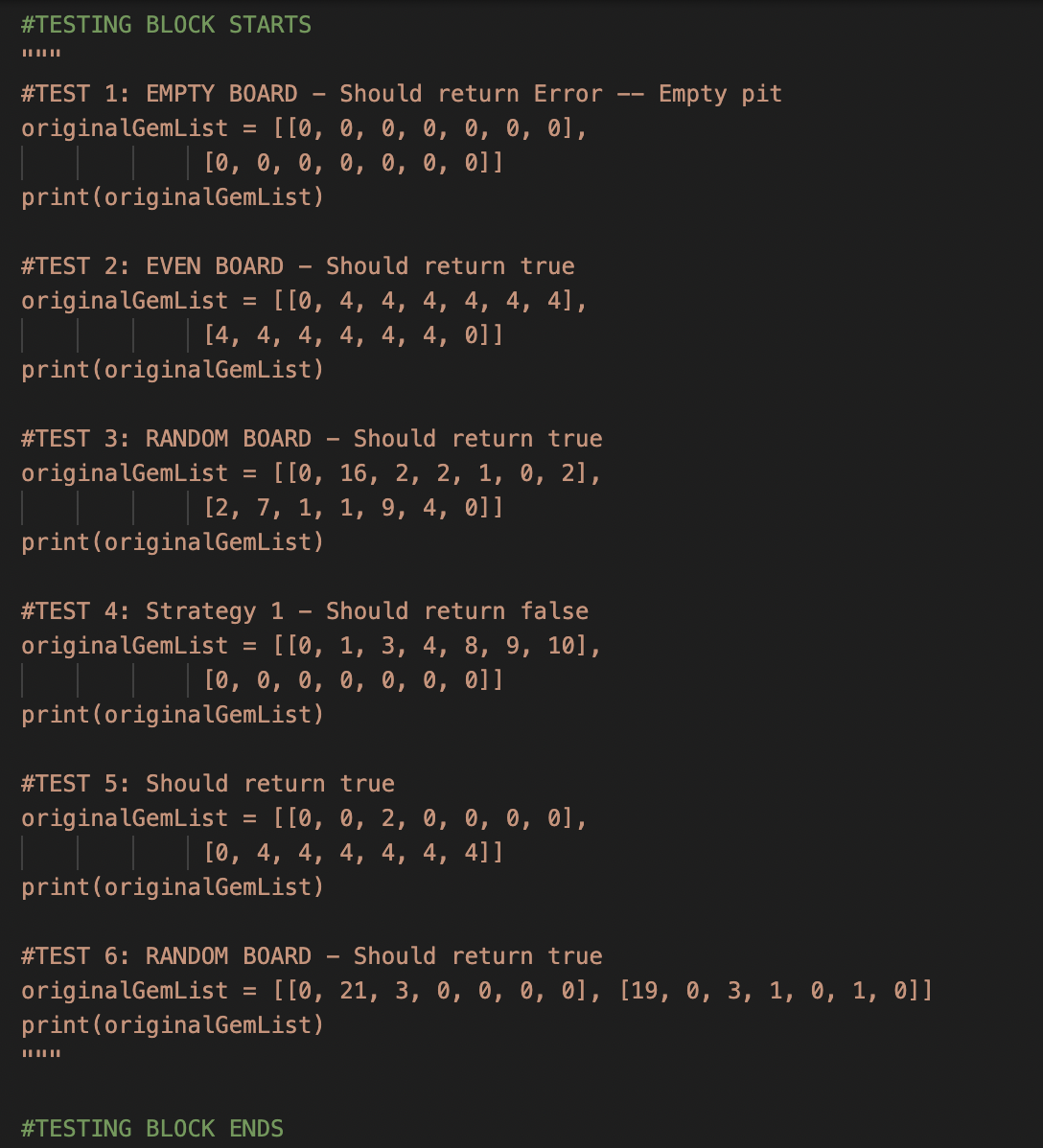


Here is the final logic check and test on the resolver. Returned is the initial board. Under the board is a list with the following: Pit name: [*pit row, pit column, boolean: Can this pit return one of our objectives?*].

If True is returned, the pit can be chosen to solve one of the strategies. If all is false, strategy 5 can be chosen. For example, in the return above, all are false. Therefore, the rightmost pit will be chosen. In the lower return, pit1 will solve for a strategy, therefore pit1 will be chosen.   
*Figure reads: {‘pit1A’: [1,0,True], ‘pit2A’: [1,1,False], ‘pit3A’: [1,2,False], ‘pit4A’: [1,3,False], ‘pit5A’: [1,4,False], ‘pit6A’: [1,5,False]}*

Ideally, the logic file would be able to import this, and return the ideal pit in the case that two pits return true. In the case of this project, that will not be solved for, as this would land outside of our determined project scope.

For testing methodologies, we created 6 tests in the run file (these are in a comment block labelled TESTING BLOCK). These test the satisfiability for 6 different boards. See tests here:



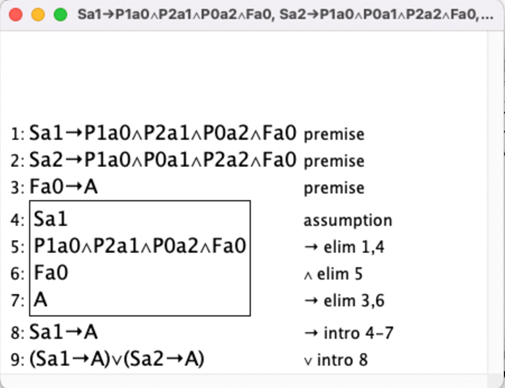
### Results

For the run.py file using logic predicates, we were able to test the satisfiability of each criteria of the four criteria to get an optimal turn. As well, for the resolver.py, we were able to find the satisfiability for each of the criteria at the same time. This showed us which was the optimal move based on the dictionaries of pits given. The model’s results were very useful in interpreting the different ways the mancala board can be optimised. For example in the test data, an empty mancala table cannot be optimised as well there was some ambiguity with empty pits. For example if the player can land a seed in an empty pit but the opposing player’s pit is also empty, then that move is not optimal. The results helped us refine the model and it provided us with interesting results.

# Jape Proofs

### Player A gets another Turn

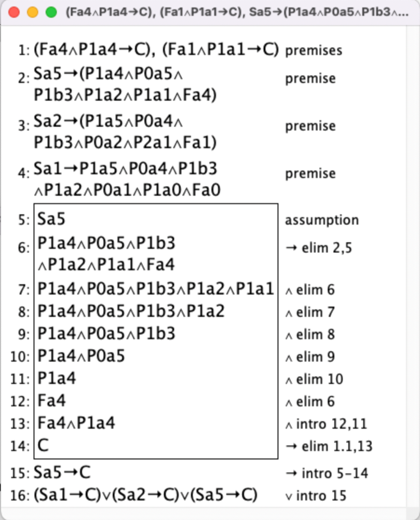
Our first proof checks if player A can get an extra turn by checking the conditions of all the other pits. In this scenario, we have 2 gems on A2 and 1 gem on A1 and every other pit is empty. We model using this formula using the following premises.



The first two premises are what happens if the pit is selected while the last premise is what happens if the final gem is in a0. Our conclusion we are trying to make is that if one of the two pits are selected, then it will lead to another turn. The proof is used to show that the first strategy can be done in the model.

### Player A Collects Gems from Opposite Pit

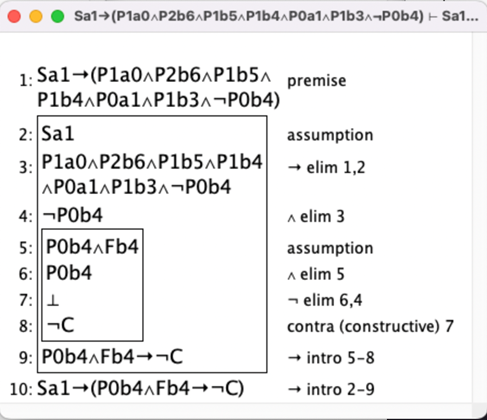
Our second proof is the same as before, but we check to see if there are any pits that will result in us collecting gems directly from two opposing pits. In this scenario, there is a gem in A5, B3, A2, A1 and none in the other pits. We model it using this formula:



Similar to our first proof, the first three premises are used to represent what happens if we select the pit whereas the final two are used to represent what happens if the following conditions are met. It aligns with the rules of the game. Like before, the proof is used to show that the third strategy can be done in the model.

### Player A Blocks Player B

Our final proof is proving that selecting pit a1 (which would have 5 gems) would block the collection of a3 and b4. We model it using this formula.



The premise is to represent what happens if pit A1 is selected and the conclusion is to determine if selecting the pit would lead to the opponent not being able to collect A3 and B4. The conclusion is represented by making the pit b4 not have any gems since it allows for the opponents to collect. This proof has similar logic towards the second strategy where instead of blocking the opponent from collecting, the player blocks the opponent from getting another turn.

# First-Order Extension

If we were to apply predicate logic to this problem, we would use it to test if there exists a model that satisfies the same strategies as previously listed. For instance, we could rework our propositions and constraints to satisfy our optimization strategies in the following manner:

* Let r represent the row of a pit
* Let c represent the column of a pit
* Let g represent the number of gems in pit x
* Let P(r, c, g) represent the number of gems **g** in a pit located at row **r** column **c**
* Let S(g) represent the number of gems **g** needed to land in the store

Then, say we want to add a constraint to test if there exists a pit which contains enough gems for the final gem to land in player A’s store. This can be represented as:

∃r. ∃c. ∃g. (P(r, c, g) ∧ S(r, g))

A model that would satisfy this constraint is:

domain = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24}

P = {(1, 2, 2)}

S = {(1, 2)}

Because Mancala is a very number oriented game, utilising predicate logic would eliminate the counting issues we had with propositional logic. Using predicate logic would help to resolve the issues that were found in the final section of this paper labelled *Incomplete Items.*

# Incomplete Items

For both the resolver and run files we were able to achieve the task of checking satisfiability. Unfortunately, we were unable to return the ideal pit, given a board. To solve this, next steps would be to test for specific strategies. For example, given a board with a set amount of gems, we could determine if there exists a pit where we would get the next turn. By default, if there is a gem in the player’s side, the model would return true as it tests for playable pits. When attempting to add in custom constraints for specific strategies, the custom constraints don't seem to affect the outcome of the model. If we have more time or more ways to debug, we would attempt to resolve that issue.