# Project Summary

The game of Go requires two players of white and black stones, and a board of size i by j. On this grid, the stones are placed on the gridline intersections such that the goal of each individual player is to finish the game with the most points. Points are rewarded by capturing stones, and occupying territory. Stones are captured by surrounding singular or multiple opponent stones. A scoring system works where a point is awarded for each area occupied, and each stone captured. The aim of researching the game of GO is to determine within a set specific board configuration, with a specific number of proceedings moves, what are “winning” moves?

# Propositions

* Bi,j : This is **True** for position (i, j) if there is a black stone in it.
* Wi,j : This is **True** for position (i, j) if there is a white stone in it.
* Si,j : This is **True** for position (i, j) if there is a safe position (i, j) on the board for a stone(s) to be placed
* Ci,j : This is **True** for position (i, j) if there is a captured position (i, j) on board for a stone(s) to be captured.

# Constraints

Stones of two types cannot share the same position. A spot with a white stone cannot also have a black stone on it and vice versa.

Stone identifiers cannot both satisfy the same position. A board placement spot can only be determined as safe or captured.

Adding a constraint for a position to be safe based on its neighbouring adjacent stones.

A stone is capturable if it is not safe.

# Model Exploration

## Automating Tests & Partial Assignments

We began by trying to implement a test method in our Python script. We thought that this would speed up much of the experimentation where we could run and clearly see what edge cases passed or failed. However, in our journey, we encountered many roadblocks and finished with a lot more knowledge of Bauhaus and Python decorators. Here are some brief points of the things we encountered as we built our automated tests.

We first tried to reset the state of our Encoding object, where we searched the Bauhaus documentation- which led us to purge\_propositions() clear\_constraints(). We tried implementing these by putting them in our Test.run() method but ran into issues where the previous state would not be properly reset, such that the result of testcase1 would interfere with testcase2’s result, causing the rest of test cases beyond testcase1 to evaluate to False due to contradictions in our constraints. We deduced this because when we ran each test as the first test in each list, it passed and gave predictable behaviour with the rest of the tests always evaluating to False.

We were hesitant to abandon this approach, so a lot of time was spent debugging and trying to figure out why it wasn’t working. Eventually, we had a breakthrough when we put all of our original code inside our Test.run() method, as fully resetting everything by creating a new Encoding object as well as reassigning new decorators solved the issue of test-to-test interference.

By instantiating brand new variables as well as classes for each test case, we were able to generate the following test cases and their visualizations- this way of testing was also our way of exploring specific partial assignments (board configurations).

The Model exploration was not complex enough and not easily understandable as seen from the provided feedback after the initial draft submission. By changing the aim and scope of the project; determining if all white pieces on a board can be captured, to now what individual white pieces are capturable. This takes into account a present representation of the board, and the individualized capturable pieces. This allows for complexity in the research such that future board moves and instantiations of the board can be analyzed with a newly developed model. This model is built upon the previously developed model submitted from our draft, and changes have been made to the propositions, constraints. Simpler changes were made in aiding the reader to better grasp the game of go, due to its lacking explanation of the game’s rules.

A screen shot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A black screen with white text

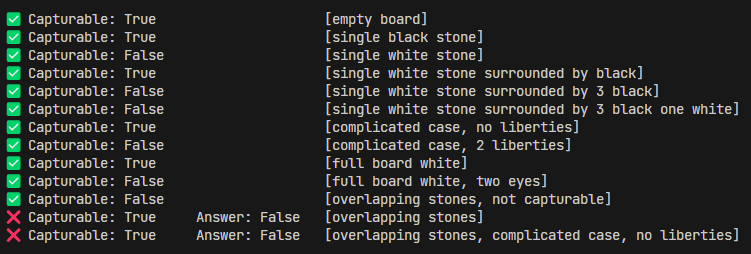
Description automatically generated

## Redundant Constraints

We discovered that none of our constraints were redundant. All are necessary for our model to run properly! However, along the way, we encountered some bugs and fixed them by changing our implementation details. Here are some of the constraints we tried to remove:

### Removal of Overlapping Stones Constraint

Just a detail of an obstacle we encountered on our journey: The first time we ran the tests, despite removing the constraint, it passed all cases. We later deduced that because of the way we implemented our surrounded(i,j) method- specifically, we returned early when finding a black stone, as we assumed that if at a position we found a black stone, there could be no white stone at that position. However, this early return led to us encoding the constraint programmatically. When we removed this to reflect the True nature of our original logical statement- returning True if a white stone is surrounded, and False otherwise, the constraint we implemented in add\_constraints() for overlapping stones became necessary.



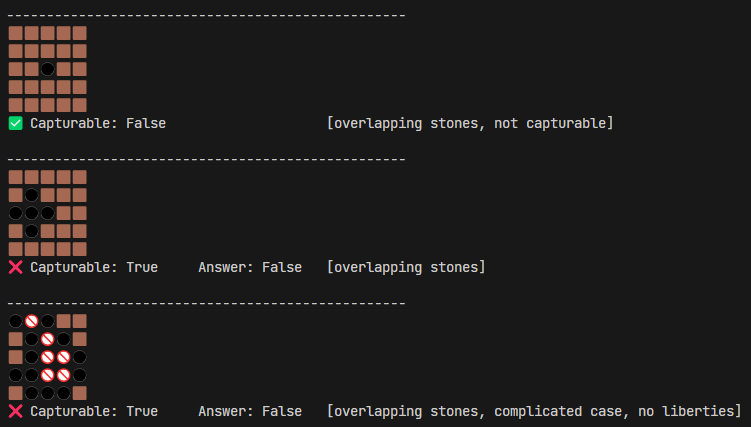
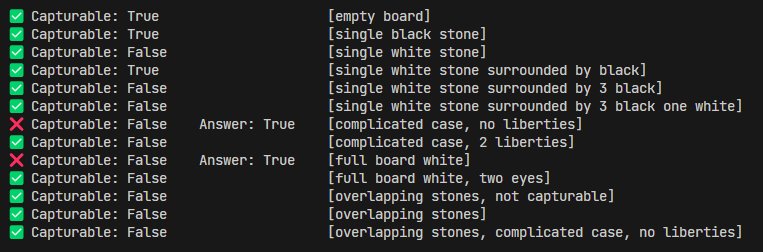


Image 1.2

* Our last two test cases failed because we’d expect any invalid board state- such as one containing overlapping stones to evaluate to be False because it would be a contradiction. However, when the constraint is removed, we do not immediately evaluate to False from contradiction, as there is no constraint to create that contradiction. Instead, we pretend that there is no black stone when stones overlap- causing our last two cases where white stones normally are captured to evaluate to True, meaning our constraint is neccessary.
* Note that in Image 1.2 above, where we show the board state of each test case, the overlapping stone displays as a black stone, and is positioned at (2,2), (1,2), and (1,3) respectively.

### Removal of Out-of-bounds is Black Stone Constraint





* When we removed this constraint, it led to white stones along the edges of the board to assume there were liberties instead of being blocked off.
* (Image 2 Case 2): the stone on (1,0) should be surrounded, but because one of its neighbours is the border, it lacks that proposition needed to deem it a surrounded stone- thereby failing the test case.
* (Image 2 Case 4): all the stones adjacent to the border are not seen as surrounded (notice the difference in emoji, surrounded stones are marked with a red slash and circle outline)

# Jape Proof Ideas

*List the ideas you have to build sequents & proofs that relate to your project.*

* For a single white stone that is surrounded by black stones, prove that it is surrounded.
* If there are no white stones, prove that all white stones are captured.
* If the board is all white stones, prove that all white stones are captured.
* If there exists a position that is out of bounds, then there exists a black stone.
* If a board has all white stone prove there are no black stones.
* For a single white stone surrounded by two black stones, prove it is not captured.
* Some mostly completed proofs can be found adjacent to this file.

# Requested Feedback

*Provide 2-3 questions you’d like the TA’s and other students to comment on.*

* What are some possible extensions that can be implemented to increase the complexity of the model? Do we need to increase the complexity of our model?
* Are our Jape proofs of the right level of complexity?
* Are there any additional constraints which may need to be implemented?
* What other ways of exploration do you see fits our problem/model?

# First-Order Extension

## Propositions

* White(i,j): There is a white stone at position (i,j)
* Black(i,j): There is a black stone at position (i,j)
* Safe(i,j): The position (i,j) is safe and a stone can be placed at that grid position
* Captured(i,j): True if a stones position (i,j) can be taken at that grid position.

## Constraints

* Stones of two types cannot share the same position.
* If there is a stone placement that is not surrounded, then it cannot be captured at that position making it safe.
* Stone placements cannot identify a position to be both Safe and Captured
* A Stone is safe based on its neighbouring stones adjacent to it at its position.
* A stone position is capturable if it is not safe.