Project Summary

The game of Go requires two players of white and black stones playing on a goban (a 19 by 19 grid, where the stones are placed on the grid line intersections). The goal of each individual player is to finish the game with the most points. Points are rewarded by capturing stones. Stones are captured if there are no “liberties”, or gaps, in the area around a group of same coloured stones. If a group of white stones is surrounded by black stones without liberties, then they are taken. Capturability is prioritized based on which colour was placed down last. If black went last, then even if there are no liberties around the stone that they placed, if it is able to capture a white stone it will get a “liberty” and then cannot be captured. A point is awarded for each stone captured in a given move. The aim of our research into the game of Go is to determine within a set specific board configuration, what are the best moves to play? For our model exploration, the goban was constrained to a grid size of 5 by 5, and it is assumed that white had just played their turn and it is now the black side's turn to play. We will consider the best move as the move that captures the most white stones relative to how many black stones are captured. We will also only be considering one move for each side when looking ahead to determine the best move.

In logic, we will see if a stone on the board is captured by first seeing if it is safe. We accomplish this by checking that it is connected to a “liberty”. We ensure that these basic rules of the game are included in our model by using the necessary constraints. We will explore the model to see the best possible moves for black given what moves white can play in response. We will also explore how it handles unusual stone placement such as liberties and snapbacks, as well as self-capturing moves.

Propositions

- $B_{i,j}$: This is **True** for position (i, j) if there is a black stone in it.
- $W_{i,j}$: This is **True** for position (i, j) if there is a white stone in it.
- $S_{i,j}$: This is **True** for position (i, j) if the stone at this position is safe.
- $C_{i,j}$: This is **True** for position (i, j) if the stone at this position is captured.

Constraints

$\forall i. \forall j. \left(\neg\left(B_{i,j} \wedge W_{i,j}\right)\right)$ 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.

$\forall i. \forall j. \left(\neg\left(S_{i,j} \wedge C_{i,j}\right)\right)$ Stone identifiers cannot both satisfy the same position. A board placement spot can only be determined as safe or captured or neither.

$\forall i. \forall j. \forall di. \forall dj. \left(\neg(W_{i,j} \vee B_{i,j}) \wedge (W_{i+di,j+dj} \vee B_{i+di,j+dj}) \rightarrow S_{i+di,j+dj}\right)$ (Where di, dj are the offsets to represent positions orthogonally adjacent to (i,j)) If there is no stone in a given position, then all stones orthogonally adjacent to it are considered safe.

$\forall i. \forall j. \forall di. \forall dj. \left(W_{i,j} \wedge \left(W_{i+di,j+dj} \wedge S_{i+di,j+dj}\right) \vee B_{i,j} \wedge \left(B_{i+di,j+dj} \wedge S_{i+di,j+dj}\right) \rightarrow S_{i,j}\right)$ (Where di, dj are the offsets to represent positions orthogonally adjacent to (i,j)) If a stone at (i,j) is orthogonally adjacent to another stone of the same colour that is safe then the stone at this position is safe as well.

$\forall i. \forall j. \left(\neg S_{i,j} \wedge (W_{i,j} \vee B_{i,j}) \leftrightarrow C_{i,j}\right)$ If a stone exists at position (i,j) and is not safe then it is captured.

Model Exploration

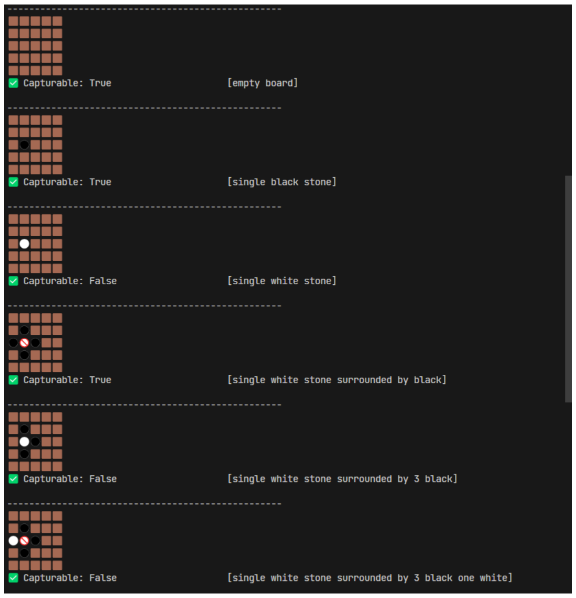
Our initial method worked to detect if **all** white pieces can be captured by seeing if they were surrounded and nothing else. This model utilized a constraint called surrounded and used another constraint called out of bounds to treat the bounds as a wall.

Automating Tests & Partial Assignments

As we were beginning to explore the simple model that we constructed based on the surrounded proposition system, we began trying to implement a testing routine in our Python script. We thought that this would speed up much of the experimentation where we could run our tests and see what test cases passed or failed. However, we encountered many roadblocks and finished with a lot more knowledge of Bauhaus and Python decorators as a result. Here are some brief points of the things we encountered as we built our automated tests.

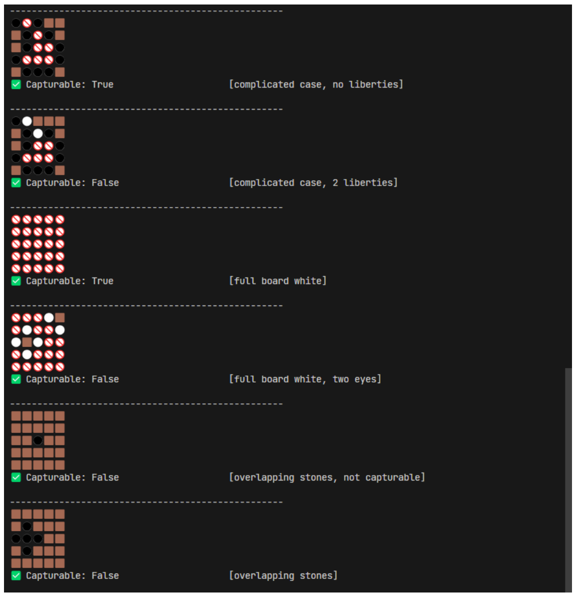
We initially tried to reset the state of our Encoding object. After searching through the Bauhaus documentation for a way to do this, we ran into issues where the previous state would not be properly reset, such that the result of *testcase1* would interfere with *testcase2*'s result, causing the rest 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 our testing, it passed and gave predictable behaviour with the rest of the tests always evaluating to False.

By instantiating brand new variables and classes for each test case, essentially resetting our code every iteration, we were able to generate the good test cases and visualizations- this way of testing was also our way of exploring specific partial assignments (board configurations). At this point, our model had the following output:



```
Capturable: True                  [empty board]
Capturable: True                  [single black stone]
Capturable: False                 [single white stone]
Capturable: True                  [single white stone surrounded by black]
Capturable: False                 [single white stone surrounded by 3 black]
Capturable: False                 [single white stone surrounded by 3 black one white]
```

Result 1



```
Capturable: True                  [complicated case, no liberties]
Capturable: False                 [complicated case, 2 liberties]
Capturable: True                  [full board white]
Capturable: False                 [full board white, two eyes]
Capturable: False                 [overlapping stones, not capturable]
Capturable: False                 [overlapping stones]
```

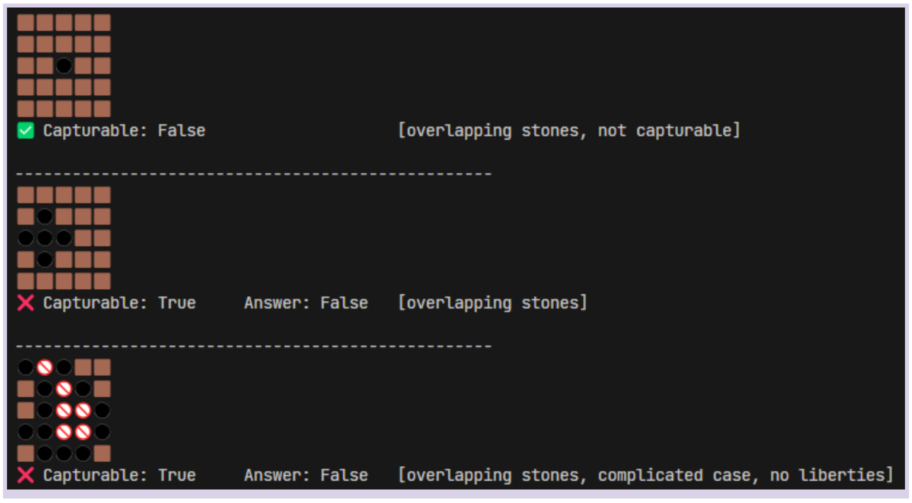
Result 2

Removal of Constraints

After completing the initial model we tried to discover what constraints were necessary and which were not. After the other constraints like “surrounded” and “out-of-bounds” were removed, they completely broke our model. But, when we removed the following constraint and found there was no significant difference, we were surprised:

$$\forall i. \forall j. \left(\neg\left(B_{i,j} \wedge W_{i,j}\right)\right)$$

One small thing we noticed later on after removing our overlapping stones constraint was that our last two test cases in *Result 3* failed. We expected an invalid board state- such as one containing overlapping stones- to evaluate to “not capturable” because the board itself would be a contradiction to the rules of Go. However, when the constraint was removed, the board was evaluated as “capturable”. Instead, we found that the model pretended 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 was necessary for ensuring overlapping stones do not affect our model’s results.



```
🟫🟫🟫🟫🟫
🟫🟫🟫🟫🟫
🟫🟫⚫🟫🟫
🟫🟫🟫🟫🟫
🟫🟫🟫🟫🟫
✅ Capturable: False                 [overlapping stones, not capturable]

-------------------------------------------------
🟫🟫🟫🟫🟫
🟫⚫🟫🟫🟫
⚫⚫⚫🟫🟫
🟫⚫🟫🟫🟫
🟫🟫🟫🟫🟫
❌ Capturable: True      Answer: False    [overlapping stones]

-------------------------------------------------
⚫🚫⚫🟫🟫
🟫⚫🚫⚫🟫
🟫⚫🚫🚫⚫
⚫⚫🚫🚫⚫
🟫⚫⚫⚫🟫
❌ Capturable: True      Answer: False    [overlapping stones, complicated case, no liberties]
```

Result 3

Although we were successful in implementing the basic model, the model exploration was not complex enough and not easily understandable, as seen from the provided feedback from the initial draft submission. By changing the aim and scope of the project; moving from determining if all white pieces on a board can be captured to which individual pieces are captured within 2 moves (initial black move and one additional white move), we added more complexity to our model. This complexity allowed our model to explore future board moves and different instantiations of the board. Although we tried to build the new model upon the previously developed model that we submitted as our draft, the initial method proved too restrictive for the scope of our project.

Post-Feedback Changes

The major shift within our model was to pivot from checking if all the white pieces on the board could be taken to checking individual pieces. Instead of searching for each white piece and checking if they were capturable, we decided to look at the "liberties" on the board. In the game of Go, liberties are a vacant spot adjacent to any stone (Figure 1). By identifying liberty positions we can ensure that any white piece that is not in contact with at least one liberty is taken. By searching for liberties we can set adjacent white stones to be "safe" stones, for which we created a new proposition. Any white stone adjacent to a safe white stone was also considered safe. In the end, any stone which was not safe was considered to be captured. This new method allowed us to identify each white stone as individually capturable. Since the nature of how we detected safety did not rely on checking if stones were surrounded, it was not necessary to have "out-of-bounds" and "surrounded" propositions/constraints, meaning they could be safely removed.

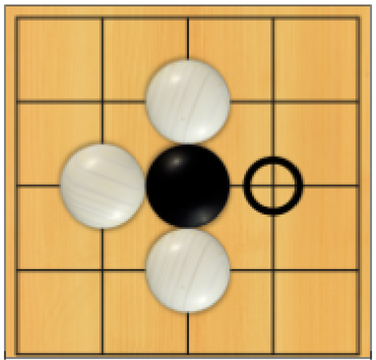


Figure 1, Black liberty position

At this stage, our model only evaluated whether white pieces were captured, which raised an issue when looking ahead as we need to account for black pieces which can be taken by the white's next move. Our initial method for implementing the one move lookahead function was to flip the board state, make all white pieces black and all black pieces white. This allowed us to run our original model with minimal changes. However, this method turned out to be unintuitive, more performance-intensive and oftentimes led to a variety of errors and bugs. Instead, we simply extended the propositions of captured and safe to black stones as well, and this showed to be another advantage of looking at liberties instead of specific stones.

Suicide Moves

In our model we explored a variety of positions including self-captured positions (example: Figure 2, Figure 3, Figure 4). In the game of Go, under most rule sets, moves which lead to self-capture are not allowed. Since our model works through a brute force method, our initial implementation led to cases where the model would return the positions of invalid moves. In order to fix this, after every iteration we check if the newly placed stone is "captured", which would mean the move is invalid. However, we ran into another issue with suicide moves. In the case seen in *Figure 4* the "self-capture" move is actually valid since it leads to the capture of its surrounding pieces, making it safe. Since the model correctly assumes said position would have no liberties it was classified as an invalid move. In order to resolve this we create a function named check_valid_move which runs after every iteration to see if the surrounding stones of the position are also captured after running our model, if so it returns true and overrides the original invalid move statement.

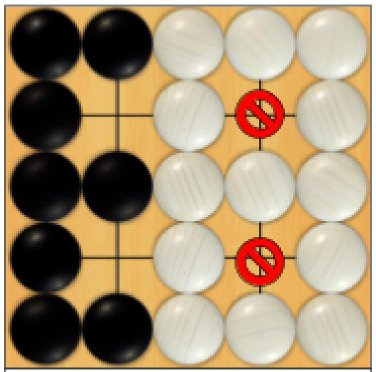


Figure 2, Self-Capture position 1

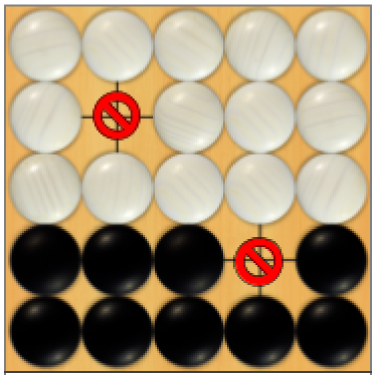


Figure 3, Self-Capture position 2

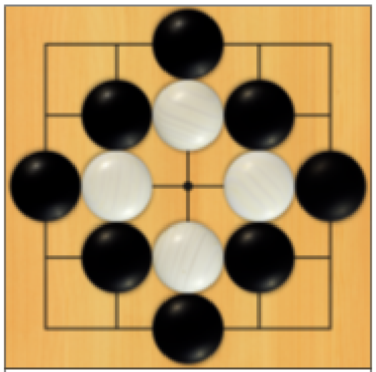
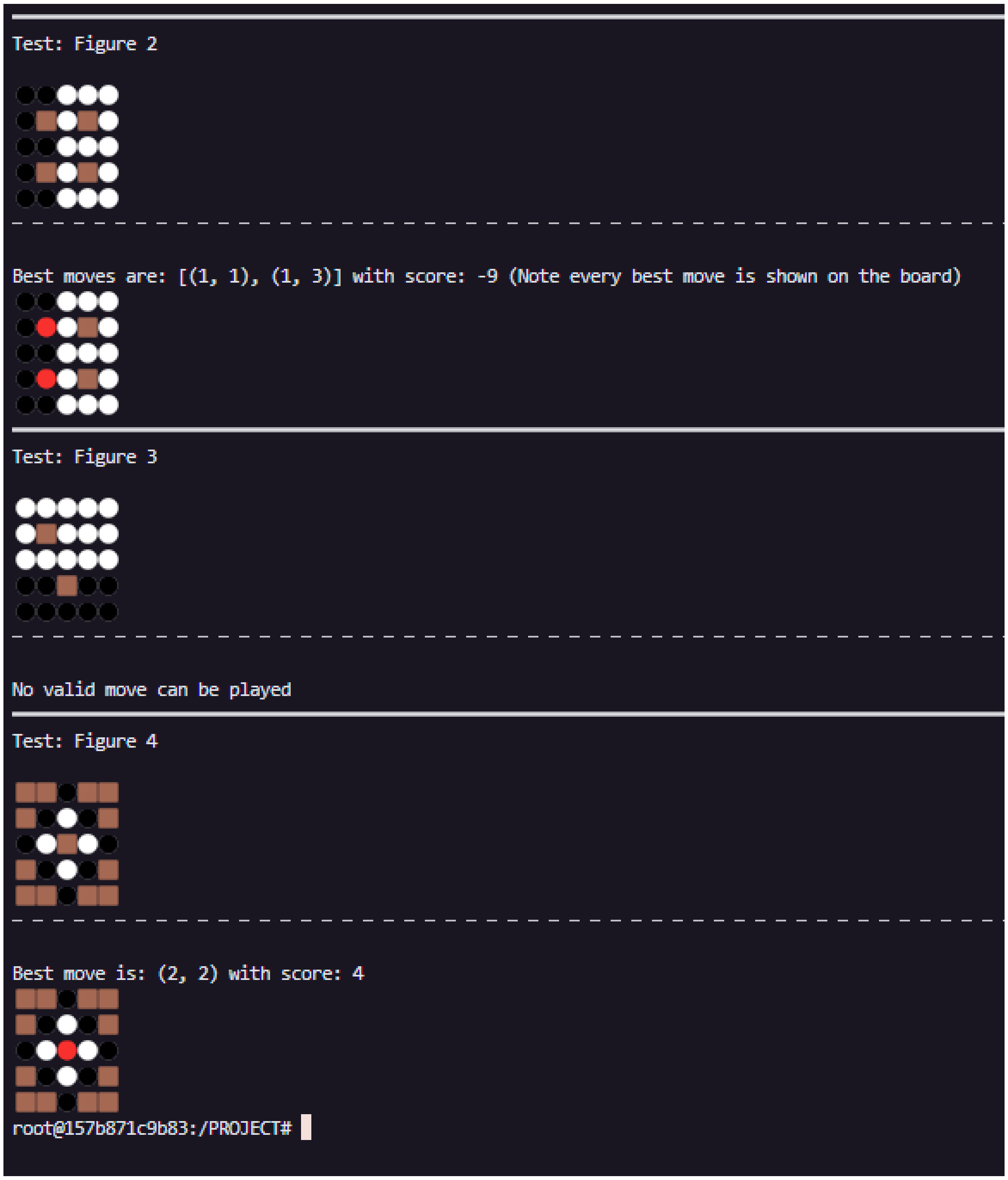


Figure 4, Self-Capture position 3

Results of Figures 2-4:



```
Test: Figure 2

⚫⚫⚪⚪⚪
⚫🟫⚪🟫⚪
⚫⚫⚪⚪⚪
⚫🟫⚪🟫⚪
⚫⚫⚪⚪⚪
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

Best moves are: [(1, 1), (1, 3)] with score: -9 (Note every best move is shown on the board)
⚫⚫⚪⚪⚪
⚫🔴⚪🟫⚪
⚫⚫⚪⚪⚪
⚫🔴⚪🟫⚪
⚫⚫⚪⚪⚪

Test: Figure 3

⚪⚪⚪⚪⚪
⚪🟫⚪⚪⚪
⚪⚪⚪⚪⚪
⚫⚫🟫⚫⚫
⚫⚫⚫⚫⚫
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

No valid move can be played

Test: Figure 4

🟫🟫⚫🟫🟫
🟫⚫⚪⚫🟫
⚫⚪🟫⚪⚫
🟫⚫⚪⚫🟫
🟫🟫⚫🟫🟫
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

Best move is: (2, 2) with score: 4
🟫🟫⚫🟫🟫
🟫⚫⚪⚫🟫
⚫⚪🔴⚪⚫
🟫⚫⚪⚫🟫
🟫🟫⚫🟫🟫
root@157b871c9b83:/PROJECT#
```

Result 4

Snapbacks

A Snapback is a position in Go where a move to capture opponent pieces will lead to a larger capture of the player's own pieces as a result. An example of this can be seen in *Figure 5* where if black plays the move (4,3) and takes the white piece, white can "snapback" and take all the black pieces (-10). In this situation, the best first move may not always be to capture the most stones immediately, but rather prevent a possible snapback and minimize your losses. To account for this, our model runs all the possible moves black can play, and then all moves white can play in response, then their score is tallied. If white is able to take more pieces than black by replacing one of their stones, this is labelled as a snapback and the function will return the next best move. Another example can be seen in *Figure 6*, where a move by black in position (2,2), will capture the white stone at (1,2). If white places a stone back in the same location, it can take all of the black stones in positions (1,1),(2,1),(2,2). The model will also display that this board is a snapback configuration.

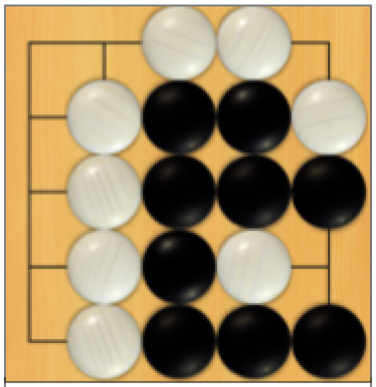


Figure 5, Snap back position 1

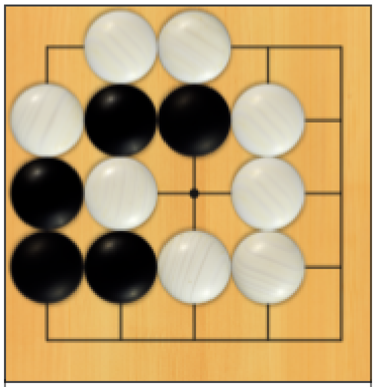
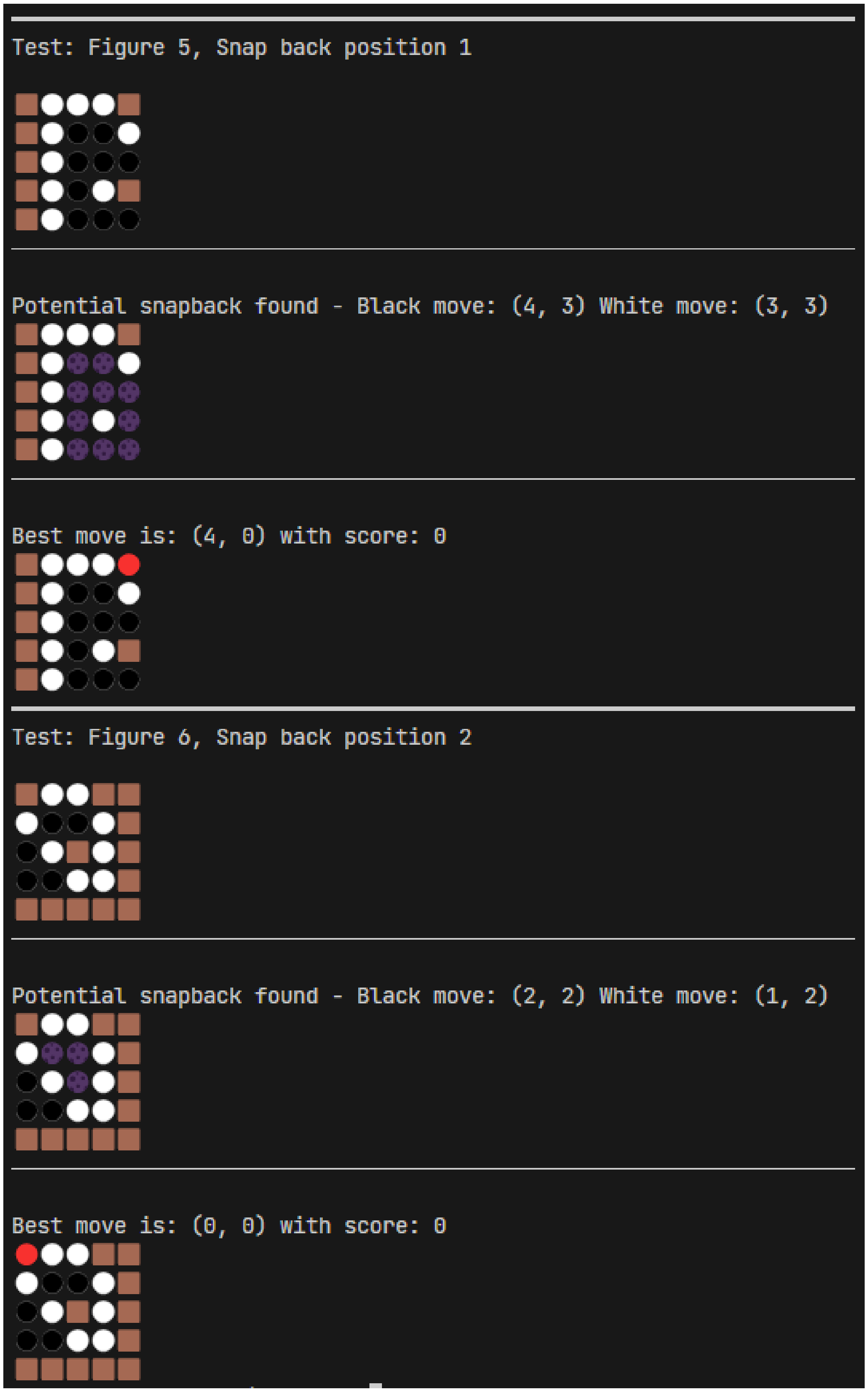


Figure 6, Snap back position 2

Results of Figure 5 and 6 are on the next page:



```
Test: Figure 5, Snap back position 1

🟫⚪⚪⚪🟫
🟫⚪⚫⚫⚪
🟫⚪⚫⚫⚫
🟫⚪⚫⚪🟫
🟫⚪⚫⚫⚫

Potential snapback found - Black move: (4, 3) White move: (3, 3)
🟫⚪⚪⚪🟫
🟫⚪🟣🟣⚪
🟫⚪🟣🟣🟣
🟫⚪🟣⚪🟣
🟫⚪🟣🟣🟣

Best move is: (4, 0) with score: 0
🟫⚪⚪⚪🔴
🟫⚪⚫⚫⚪
🟫⚪⚫⚫⚫
🟫⚪⚫⚪🟫
🟫⚪⚫⚫⚫

Test: Figure 6, Snap back position 2

🟫⚪⚪🟫🟫
⚪⚫⚫⚪🟫
⚫⚪🟫⚪🟫
⚫⚫⚪⚪🟫
🟫🟫🟫🟫🟫

Potential snapback found - Black move: (2, 2) White move: (1, 2)
🟫⚪⚪🟫🟫
⚪🟣🟣⚪🟫
⚫⚪🟣⚪🟫
⚫⚫⚪⚪🟫
🟫🟫🟫🟫🟫

Best move is: (0, 0) with score: 0
🔴⚪⚪🟫🟫
⚪⚫⚫⚪🟫
⚫⚪🟫⚪🟫
⚫⚫⚪⚪🟫
🟫🟫🟫🟫🟫
```

Result 6

Edge Cases

We also tested a variety of invalid board states with our model. These included boards with overlapping pieces, stones placed out of bounds, full boards, and larger board sizes. The results of these tests are seen below:

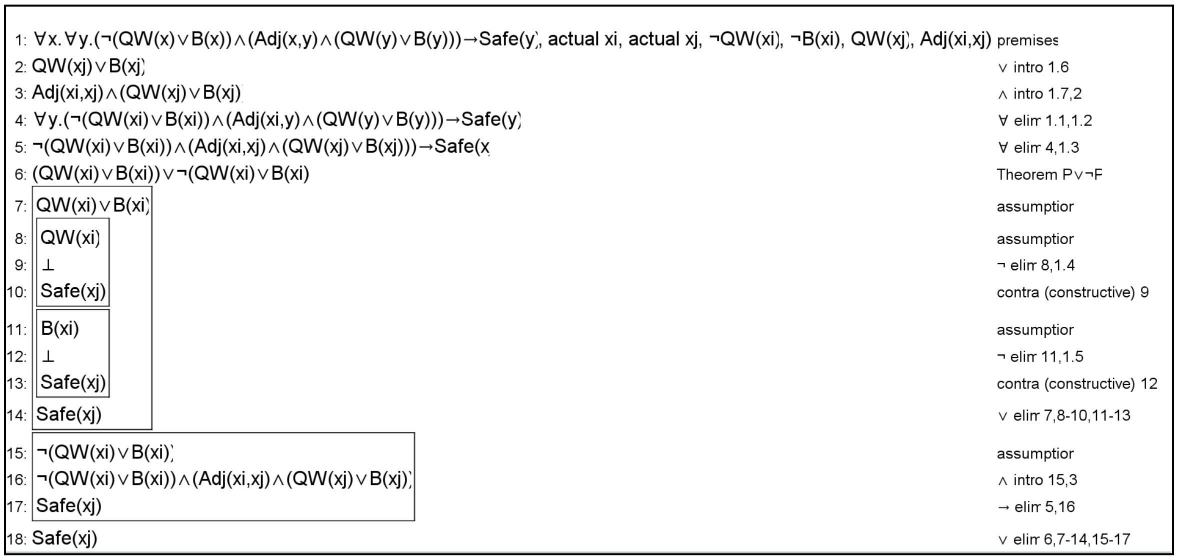
- Overlapping Stones: Model returns "No valid move can be played"
- Out-of-bounds: Model ignores out-of-bounds stones and returns the best move within the constraints of the board.
- Full Board: Will remove all pieces that have no liberties, If there is a combination of white and black pieces only black pieces will be removed as it is assumed that white had just played.
- Larger board sizes: Due to how our model works by checking every single board position it is fairly performance-intensive, thus, once reaching a board size of ~9 the maximum runtime (where the board is empty) becomes very slow. This trend will increase exponentially with board size.

In conclusion, our exploration of our model has been marked by iterative refinement and productive adjustments. During this process, we have had to shift our approach to bring more flexibility and efficiency to our model to address self-capture moves and snapbacks. Finally, testing against various edge cases further solidified the robustness of our solution. Through critical feedback and rigorous testing, we were able to adjust our scope and focus to provide an accurate model for a small aspect of the ancient game of Go.

Jape Proofs

Note that the propositions used are similar to those used in the first order extension. In all examples, *xi* and *xj* are positions on the board. Different proposition names were used as indicated.

If there is a white stone adjacent to a liberty (no stones in this location) then it is safe.

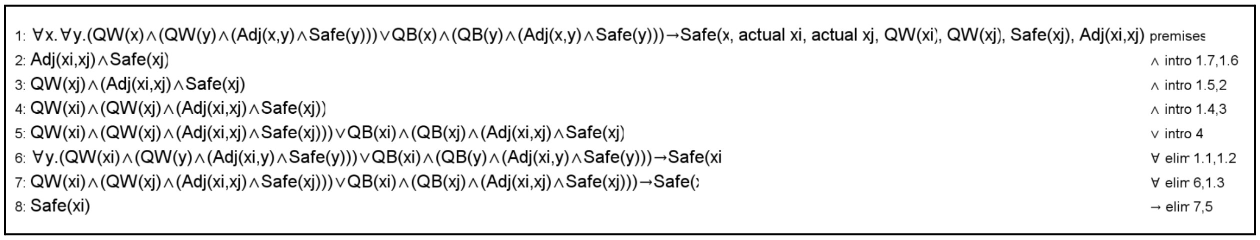


```
1: ∀x.∀y.(¬(QW(x)∨B(x))∧(Adj(x,y)∧(QW(y)∨B(y)))→Safe(y), actual xi, actual xj, ¬QW(xi), ¬B(xi), QW(xj), Adj(xi,xj)   premises
2: QW(xj)∨B(xj)                                          ∨ intro 1.6
3: Adj(xi,xj)∧(QW(xj)∨B(xj)                              ∧ intro 1.7,2
4: ∀y.(¬(QW(xi)∨B(xi))∧(Adj(xi,y)∧(QW(y)∨B(y)))→Safe(y)  ∀ elim 1.1,1.2
5: ¬(QW(xi)∨B(xi))∧(Adj(xi,xj)∧(QW(xj)∨B(xj)))→Safe(x    ∀ elim 4,1.3
6: (QW(xi)∨B(xi))∨¬(QW(xi)∨B(xi)                         Theorem P∨¬P
7:  QW(xi)∨B(xi)                                         assumption
8:   QW(xi)                                              assumption
9:   ⊥                                                   ¬ elim 8,1.4
10:  Safe(xj)                                            contra (constructive) 9
11:  B(xi)                                               assumption
12:  ⊥                                                   ¬ elim 11,1.5
13:  Safe(xj)                                            contra (constructive) 12
14: Safe(xj)                                             ∨ elim 7,8-10,11-13
15: ¬(QW(xi)∨B(xi))                                      assumption
16: ¬(QW(xi)∨B(xi))∧(Adj(xi,xj)∧(QW(xj)∨B(xj))           ∧ intro 15,3
17: Safe(xj)                                             → elim 5,16
18: Safe(xj)                                             ∨ elim 6,7-14,15-17
```

Figure 7

QW represents White, *B* represents Black, and *Adj* represents adjacent positions on the board.

If a white stone is next to another white stone that is also safe, then it is safe.

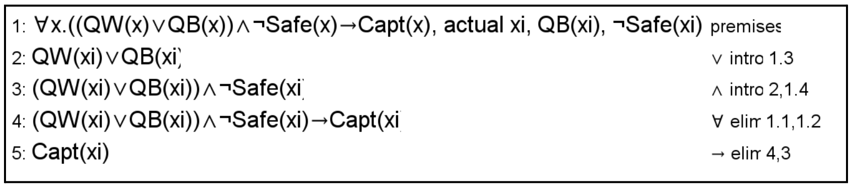


```
1: ∀x.∀y.(QW(x)∧(QW(y)∧(Adj(x,y)∧Safe(y)))∨QB(x)∧(QB(y)∧(Adj(x,y)∧Safe(y)))→Safe(x, actual xi, actual xj, QW(xi), QW(xj), Safe(xj), Adj(xi,xj)   premises
2: Adj(xi,xj)∧Safe(xj)                                                        ∧ intro 1.7,1.6
3: QW(xj)∧(Adj(xi,xj)∧Safe(xj)                                                ∧ intro 1.5,2
4: QW(xi)∧(QW(xj)∧(Adj(xi,xj)∧Safe(xj))                                       ∧ intro 1.4,3
5: QW(xi)∧(QW(xj)∧(Adj(xi,xj)∧Safe(xj)))∨QB(xi)∧(QB(xj)∧(Adj(xi,xj)∧Safe(xj)   ∨ intro 4
6: ∀y.(QW(xi)∧(QW(y)∧(Adj(xi,y)∧Safe(y)))∨QB(xi)∧(QB(y)∧(Adj(xi,y)∧Safe(y)))→Safe(xi   ∀ elim 1.1,1.2
7: QW(xi)∧(QW(xj)∧(Adj(xi,xj)∧Safe(xj)))∨QB(xi)∧(QB(xj)∧(Adj(xi,xj)∧Safe(xj)))→Safe(   ∀ elim 6,1.3
8: Safe(xi)                                                                   → elim 7,5
```

Figure 8

QW represents White, *QB* represents Black, and *Adj* represents adjacent positions on the board.

If a black stone is not safe then it is captured.



```
1: ∀x.((QW(x)∨QB(x))∧¬Safe(x)→Capt(x), actual xi, QB(xi), ¬Safe(xi)   premises
2: QW(xi)∨QB(xi)                                                     ∨ intro 1.3
3: (QW(xi)∨QB(xi))∧¬Safe(xi                                          ∧ intro 2,1.4
4: (QW(xi)∨QB(xi))∧¬Safe(xi)→Capt(xi                                 ∀ elim 1.1,1.2
5: Capt(xi)                                                          → elim 4,3
```

Figure 9

QW represents White, *QB* represents Black, and *Capt* represents Captured.

First-Order Extension

To extend our problem using predicate logic, we can begin by defining the predicates implicitly. Note that we modified the constraints from what was listed above in our propositional logic model. New predicates were also added that made implementing said constraints easier.

Predicates

- **Pos(x)**: True when *x* is a valid position tuple within the *N* by *N* grid (in our case 5 by 5).
- **Stone(x)**: True when there is a stone of either colour at a position *x*.
- **White(x)/Black(x)**: True when there is a white/black stone at position *x*.
- **Adj(x,y)**: True when position *y* is orthogonally adjacent to position *x*.
- **Safe(x)**: True when the stone at position *x* is adjacent to a safe stone of the same colour or is adjacent to a liberty.
- **Capt(x)**: True when the stone at position *x* is going to be captured in the current board position.
- **SameColour(x,y)**: True when the stone at position *x* and the stone at position *y* are both black or both white.

Constraints

- Stones of two types cannot share the same position.
    - $\forall x.\,(Pos(x) \rightarrow \neg(Black(x) \wedge White(x)))$
- Stone placements cannot identify a position to be both Safe and Captured
    - $\forall x.\,(Pos(x) \rightarrow \neg(Safe(x) \wedge Capt(x)))$
- A Stone is safe if it is adjacent to a liberty.
    - $\forall x.\,\forall y.\,(Pos(x) \wedge \neg Stone(x) \wedge Pos(y) \wedge Adj(x,y) \wedge Stone(y) \rightarrow Safe(y))$
- A Stone is safe based if its neighbouring stones adjacent to it at its position.
    - $\forall x.\,\forall y.\,(Pos(x) \wedge Stone(x) \wedge (Pos(y) \wedge Adj(x,y) \wedge SameColour(x,y) \wedge Safe(y)) \rightarrow Safe(x))$
- A stone position is capturable if it is not safe.
    - $\forall x.\,(Pos(x) \wedge Stone(x) \wedge \neg Safe(x) \leftrightarrow Capt(x))$