# **Confirming Hexapawn**

Group 49

Yutong Jiang

Kevin Tran

Colin Wasik

Andrew Wilker

# Project Summary

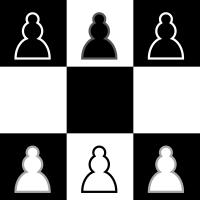
For our project, we are confirming Hexapawn.

Hexapawn is a game played on a 3x3 board. Both players get three pawns on their row that move using the same rules as normal pawns in chess, except for the initial double space move.

* White makes the first move.
* Pawns can move 1 space forward if the space in front of them is empty.
* They can attack the opposing pawn with a diagonal move forward.
* The goal of the game is to either advance a pawn to the other side of the board, or to prevent the opposing player from moving (this includes white not being able to move if by chance they lose all their pawns).

In our project we will be checking to see if white is in a winning state on a given valid board, and if they are not then we will check how many possible move they are able to make on there turn.

A valid board state is a board that is can occur during a game of Hexapawn. Examples of valid board states are: 



# Propositions

Let x, y be within {0, 1, 2}

w[y][x] is defined as being a white pawn on coordinates (x, y)

b[y][x] is defined as being a black pawn on coordinates (x, y)

l[y][x] is defined as a white pawn being able to capture forward-left (x, y)

f[y][x] is defined as a white pawn being able to move forward (x, y)

r[y][x] is defined as a white pawn being able to capture forward-right (x, y)

We found that a few of our original constraints (H, S, & T) did not fit for our project, we write more on this in the “Model Exploration” section.

# Constraints

1. White has a winning pawn by moving to the other side of the board:
2. White has a winning pawn by attacking to get to the other side of the board:
3. White blocks a black pawn:

(b[0][j] & w[2][j] & ~b[1][j] & ~w[1][j])

1. White can only make 1 move:

xor(f[1][0], …, f[2][2], r[1][0], …, r[2][1], l[1][1], …, l[2][2])

1. White pawn not being on a square means that it can’t make a move

(~w[i][j] >> ~f[i][j]) & (~w[i][j] >> ~l[i][j]) & (~w[i][j] >> ~r[i][j])

1. Move constraining conditions (must capture, must be an empty square) means that pawn on (j, i) can’t move

((b[i-1][j] | w[i-1][j]) >> ~f[i][j]) & (~b[i-1][j-1] >> ~l[i][j]) & (~b[i-1][j+1] >> ~r[i][j])

We found that a majority of our original constraints didn’t work, mainly because our original propositions did not fit the project.

# Model Exploration

We have looked at winning configurations with valid board configurations where a white pawn reaches the other end of the board, where white can move to the end, or it can attack to get to the end. Or we found whites winning configuration by finding when black has no available moves with 1, 2 or 3 pawns on the board.

We have also looked at the configurations where the number of possible moves is counted. Calculating for each board state, how many possible moves that white can play on their turn.

To begin our coding section of the project, we contemplated hard coding all possible positions of the board because we could not find a way to add the constraints in an efficient way. Eventually we built loops to run through each position on a board and check if white is in a winning position. We built off this code, fixing it and extending it to get to our final product that runs through the board and checks if white is or is not in a winning position on their turn.

After all the code was pretty much done, we added an error logger that checked the board if the board had more than 3 or each pawn, or if the pawns were at the opposing side of the board which would automatically result in a win.

We have made 3 jape proofs for the project. One where if 1, 2 or 3 black pawns cannot move, then white wins. Another shows that a white pawn cannot move if there is not a pawn it can capture and no empty space for it to move to. And the final one proves if there is no pawn on a square then it is an empty square.

While exploring the model we found out very early that the T literal (true when its white’s turn) was completely redundant, because we based our project on it being white’s turn. This constraint simply increased the complexity of the project without giving any benefit, so we removed it. We also found that H (true when white has won) was another useless literal very quickly while exploring as well; by the time we need to use H, we have already found out if white has won. This led us to remove H. Finally (for the literals), we found that S (true when black has an immobile pawn) did not yield any useful results either, so we scrapped this literal as well. Finding that all these literals were not need was an essential first step to the project, because it helped us understand the project specific functions which were previously more nebulous to us. It also led to us finding the extraordinarily useful literals, like l, f, & r

For the counting theory section, we explored using bi-implications and found that we could get a single solution that contained all the possible moves, this was prior to us adding the xor function. This actually led to the eureka of finding the solution to the counting theory, because it taught us how to reliably include all the l, f, & r literals in their proper true/false form

We also explored, for the find win theory, what the model might look like if we were taking black’s turn instead and found that it was mostly the same. We found that looking at a model that included what black’s next turn would be gave us insight into how to check to see if we can force black not to move on white’s turn.

# First-Order Extension

For the constraint where black pawns cannot move, you could use a universal to show that all black pawns whether there are 1, 2 or 3 black pawns that are on the board do not have the ability to move on their turn. This would be a more apt description of constraint (3)

∀j.(b[0][j] & w[2][j] & ~b[1][j] & ~w[1][j])

Another constraint is For All j if a white can move forward to the opposing side next turn, then white wins (this is the equivalent to constraint (1)): ∀j.(w[1][j] & ¬b[0][j])

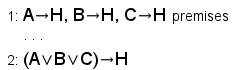
There exists an i and there exists a j such that a pawn can attack left or right (This is a much more generalised version of constraints (2) and (6)):

∃i.∃j.(w[i][j] ∧ b[i –1][j –1])

∃i.∃j.(w[i][j] ∧ b[i-1][j+1])

There exists an i and there exists a j such that a pawn that can move one space forward (part of constraint (6)): ∃i.∃j.(w[i][j] ∧ ¬b[i-1][j]∧ ¬w[i-1][j])

# Jape Proofs



* A, B, and C are simplified versions of the positions of the black pawn cannot move, A as 3 black pawns, B as 2, and C as 1 black pawn on the board.
* H is when white has won the game.

# 

* P represents a white pawn.
* A represents a black pawn that can be diagonally taken leftwards
* B represents a black pawn that can be diagonally taken rightwards
* C represents an empty space in front of the white pawn
* This proves that if there is some place that a pawn can move to, then there is a pawn on the board.



* B represents a black pawn on a given square
* P is a white pawn
* E is an empty square
* This proves that if there is no pawn on a square then it’s an empty square