1 Introduction

In 1988, the game Connect 4 was proven to be a solved game meaning that with perfect play the first player will always win. With only one non-optimal move by the first player, the second player can force a draw, and with two non-optimal moves, the second player can win. One common algorithm to solving Connect 4 is Minimax. A minimax algorithm computes a decision tree of all of the possible moves and countermoves by each player. The first player can simply make decisions that lead to a leaf node of a victory. My motivation for this project was to take this idea and do a similar approach with a verification tool. Like Minimax, a model checker does a similar approach by traversing the state space non-deterministically according to some rules and invariants. My goal was to use a model checker to traverse the space of Connect 4 games for optimal play.

2 Implementation

My model was made with Murphi and run on Rumur. Each possible move is accomplished by specifying some rule.

```
--------------------------------------------------------
Rule "First player drops token in first column"
turn = 1 --first players turn to go
==>
DropToken(0, 1) -- player 1 drops token in col 0
EndRule;

--------------------------------------------------------
Rule "First player drops token in second column"
turn = 1
==>
DropToken(1, 1) -- player 1 drops token in col 1
EndRule;

--------------------------------------------------------
Rule "First player drops token in third column"
turn = 1
==>
DropToken(2, 1) -- player 1 drops token in col 2
EndRule;

--------------------------------------------------------
Rule "First player drops token in fourth column"
turn = 1
==>
DropToken(3, 1) -- player 1 drops token in col 3
EndRule;

--------------------------------------------------------
Rule "First player drops token in fifth column"
turn = 1
==>
DropToken(4, 1) -- player 1 drops token in col 4
EndRule;
```

These rules get executed for both player 1 and 2. After a rule is executed through the DropToken function, the turn variable is changed to the other player. These rules help to enumerate all of the possible Connect 4 games. With this approach the first player can obviously make some non-optimal moves. There are 2 approaches to accounting for this. Either make the first player's move deterministically or throw out the situations that lead to a first player's defeat. With the first option, you have to enumerate the space of all of the Connect 4 games anyway which is what we are currently trying to do right now.

It also adds a fair bit of complexity. I decided to let Rumur handle this by making sure the first player doesn't lose:

```
invariant !isFull() --&& !checkWinner(1)

--discard situations in which first player loses
assume !checkWinner(2)
```

If I were to do the aforementioned first option the invariant would be that the second player never wins. The assume command is what actually discards situations in which the first player would lose, since it makes sure the second player never wins. The program ends when the board is full. There is an edge case here in a situation where a tie has occurred and the board is full. The simple fix is to augment the assume command with the implication that if the board is full, the first player must be in a winning configuration. This will throw out all of the ties. The reason that I didn't have that is because I was trying to cut down the execution time. The commented out part of the invariant is implementing a counter-invariant to show situations in which the first player is winning (similar to the river crossing example). This is useful because currently there is a huge state explosion problem:

```
        thread 13: 1560660000 states explored in 1673s, with 291541047 rules fired and 55354592 states in the queue.
        thread 0: 1560670000 states explored in 1673s, with 299131840 rules fired and 54970811 states in the queue.
        thread 8: 1560680000 states explored in 1673s, with 289661946 rules fired and 54937505 states in the queue.
        thread 7: 1560690000 states explored in 1673s, with 292538428 rules fired and 52931699 states in the queue.
        thread 14: 1560700000 states explored in 1673s, with 293621790 rules fired and 56732060 states in the queue.
        thread 14: 1560710000 states explored in 1673s, with 293623741 rules fired and 56732598 states in the queue.
        thread 1: 1560720000 states explored in 1673s, with 293014948 rules fired and 45686560 states in the queue.
        thread 12: 1560730000 states explored in 1673s, with 291637001 rules fired and 54092757 states in the queue.
        thread 0: 1560740000 states explored in 1673s, with 299147211 rules fired and 54972694 states in the queue.
        thread 9: 1560750000 states explored in 1673s, with 291477889 rules fired and 59443719 states in the queue.
        thread 9: 1560760000 states explored in 1673s, with 291479997 rules fired and 59444416 states in the queue.
        thread 12: 1560770000 states explored in 1673s, with 291645676 rules fired and 54094108 states in the queue.
        thread 4: 1560780000 states explored in 1673s, with 293999818 rules fired and 62614815 states in the queue.
        thread 12: 1560790000 states explored in 1673s, with 291650009 rules fired and 54095119 states in the queue.
        thread 15: 1560800000 states explored in 1673s, with 289574297 rules fired and 55758939 states in the queue.
        thread 12: 1560810000 states explored in 1673s, with 291654563 rules fired and 54096021 states in the queue.
        thread 10: 1560820000 states explored in 1673s, with 291678408 rules fired and 56790965 states in the queue.
        thread 12: 1560830000 states explored in 1673s, with 291659680 rules fired and 54097113 states in the queue.
        thread 14: 1560840000 states explored in 1673s, with 293651323 rules fired and 56738536 states in the queue.
        thread 8: 1560850000 states explored in 1673s, with 289697798 rules fired and 54942335 states in the queue.
        thread 9: 1560860000 states explored in 1673s, with 291500642 rules fired and 59449718 states in the queue.
make: *** [makefile:7: connect4] Killed
```

3 Conclusion and Evaluation

Enumerating the space of Connect 4 games with my parameters was very challenging for the model checker. Certain reductions may lead to better possible optimizations. For example, a lot of Connect 4 games may start differently but end up in the same configuration. These situations should be considered the same. A more mature exploration of the state space may be able to only pick out the unique situations. Even though it also enumerates the entire state space, it's also possible to precompute all of the correct moves for player 1 and simply query these during the execution of the program. This shows one of the challenges of model checking because not only do you have to correctly encode something in a model checker, but you also have to make sure that the execution will eventually fully verify.