Modelling Project Documentation – Group #34

*Project Formulation*

***Summary***

This project takes in a board configuration of Connect-Four and states whether the player (red circle) has a won the game. The positions of the red circles will correlate with a model where the player needs at least 4 in a row (either with a row, column, or diagonal) to win. If this model does not exist, the player has not won (yet). Additionally, the project will be able to state if the player (red) has an almost winning configuration. This will be checked by seeing if it is reds turn, then checking if any 4 in a row can be created by adding a red disk at the first available empty spot in each column. This may not be possible as there may not be a disk below to support said winning disk, or the position could be held by a black disk. If this model does not exist, the player has not almost won. No winning model exists if a winning model exists for the opponent.

***Propositions/Constraints***

(all coordinates/index positions are 0-based indexing, starting from the top left of the board)

*Background pattern

Description automatically generatedPropositions:*

* *xij*: is true when the position (*i,j*) has a red disk in it. Where *i* is the row of the board and *j* is the column of the board.
  + E.g., *x22, x23, x32, x33, x43, x44, x50, x51, x55* would all be true for the example configuration.
* *yij*: is true when the position (*i,j*) has a black disk in it. Where *i* is the row of the board and *j* is the column of the board.
  + E.g., *y13* (and more) would be true for the example configuration.
* *zij*: is true when position (*i,j*) has not disk in it. Where *i* is the row of the board and *j* is the column of the board. Meaning both *xij* and *yij* are false.
  + *E*.g., *z00* (and more) would be true for the example configuration.
* *cij*: is true when the configuration has a winning column in it. Where column *j* is the column containing the winning column of 4 and *i* is the topmost position of the column.
* *rij*: is true when the configuration has a winning row in it. Where *i* is the row containing the winning row of 4 and *j* is the leftmost position of the row.
* *daij*:is true when the configuration has a winning diagonal in it. Where *a* represents the slope of said diagonal (1 is positive slope, 2 is negative slope) and (*i,j*) represents the leftmost point of the diagonal.
  + *E*.g., *d222* would be true for the example configuration.
* *eij*: is true when there is an almost winning column in the board configuration. Where column *j* has 3 red disks in it and *i* is the topmost position of said column.
* *fij*: is true when there is an almost winning row in the board configuration. Where row *i* has 3 red disks in it and *j* is the leftmost position of said row.
* *gaij*: is true when there is an almost winning diagonal in the board configuration. Where *a* represents the slope of said diagonal (1 is positive, 2 is negative) and (*i, j*) represents the top corner of the diagonal.
* The following represent if black has won. In which case no winning model can exist for red.
  + *mij*: is true when the configuration has a winning black row in it.
  + *nij*: is true when the configuration has a winning black column in it.
  + *o1ij*: is true when the configuration has a winning black diagonal (with a positive slope) in it.
  + *o2ij*: is true when the configuration has a winning black diagonal (with a negative slope) in it.

*Constraints:*

When we started, we thought we would create multiple encodings for each constraint, however we soon realized we can’t return multiple encodings within the same function. To combat this problem, we have created one encoding and created a list of every constraint, then by using the Or import from NNF we can separate each constraint within the list by a disjunct, which we then add to the encoding.

* A winning row only holds when the appropriate *x* marks are there:
  + *rij* → (*xij* ∧ *xi(j+1)* ∧ *xi(j+2)* ∧ *xi(j+3)*)
* A winning column only holds when the appropriate *x* marks are there:
  + *cij* → (*xij* ∧ *x(i+1)j* ∧ *x(i+2)j* ∧ *x(i+3)j*)
* A winning diagonal only holds when the appropriate *x* marks are there:
  + *d1ij* → (*xij* ∧ *x(i-1)(j+1)* ∧ *x(i-2)(j+2)* ∧ *x(i-3)(j+3)*)
  + *d2ij* → (*xij* ∧ *x(i+1)(j+1)* ∧ *x(i+2)(j+2)* ∧ *x(i+3)(j+3)*)
* An almost winning row only holds when the appropriate *x* marks are there:
  + *eij* → ((*xij* ∧ *xi(j+1)* ∧ *xi(j+2)*) ∨ (*xij* ∧ *xi(j+2)* ∧ *xi(j+3)*) ∨ (*xij* ∧ *xi(j+1)* ∧ *xi(j+3)*))
  + Following this, we attempt to add a red disk at the first available spot in each column, each time checking if *rij* is satisfied.
* An almost winning column only holds when the appropriate *x* marks are there:
  + *fij* → ((*xij* ∧ *x(i+1)j* ∧ *x(i+2)j*) ∨ (*xij* ∧ *x(i+2)j* ∧ *x(i+3)j*) ∨ (*xij* ∧ *x(i+1)j* ∧ *x(i+3)j*))
  + Following this, we attempt to add a red disk at the first available spot in each column, each time checking if *cij* is satisfied.
* An almost winning diagonal with positive slope only holds when the appropriate *x* marks are there:
  + *g1ij* → ((*xij* ∧ *x(i-1)(j+1)* ∧ *x(i-2)(j+2)*) ∨ (*xij* ∧ *x(i-2)(j+2)* ∧ *x(i-3)(j+3)*) ∨ (*xij* ∧ *x(i-1)(j+1)* ∧ *x(i-3)(j+3)*))
  + Following this, we attempt to add a red disk at the first available spot in each column, each time checking if *d1ij* is satisfied.
* An almost winning diagonal with negative slope only holds if the appropriate *x* marks are there:
  + *g2ij* → ((*xij* ∧ *x(i+1)(j+1)* ∧ *x(i+2)(j+2)*) ∨ (*xij* ∧ *x(i+2)(j+2)* ∧ *x(i+3)(j+3)*) ∨ (*xij* ∧ *x(i+1)(j+1)* ∧ *x(i+3)(j+3)*))
  + Following this, we attempt to add a red disk at the first available spot in each column, each time checking if *d2ij* is satisfied.
* A winning black row only holds when the appropriate *y* marks are there:
  + *mij* → (*yij* ∧ *yi(j+1)* ∧ *yi(j+2)* ∧ *yi(j+3)*)
* A winning black column only holds when the appropriate *y* marks are there:
  + *nij*: → (*yij* ∧ *y(i+1)j* ∧ *y(i+2)j* ∧ *y(i+3)j*)
* A winning black diagonal only holds when the appropriate *y* marks are there:
  + *o1ij* → (*yij* ∧ *y(i-1)(j+1)* ∧ *y(i-2)(j+2)* ∧ *y(i-3)(j+3)*)
  + *o2ij* → (*yij* ∧ *y(i+1)(j+1)* ∧ *y(i+2)(j+2)* ∧ *y(i+3)(j+3)*)

***Model Exploration:***

The game of Connect-Four is very complex and has several possibilities to be explored. To begin we check if the red player has won the game, this is done with the constraints previously mentioned (*rij, cij, daij*). If the red player has won, then the model is satisfied. Otherwise, we then check if it is the red players turn, if it is, we then check if the player has an almost winning configuration, this is done with the constraints previously mentioned (*eij, fij, gaij*). If it is the red players turn and they have an almost winning configuration, then the player can win the game in the next turn and this is considered a win, therefore the model is satisfied. Finally, if the red player has neither won nor almost won the game, the model is not satisfied.

* We start with a board configuration imported from sample boards (a set of boards we created to test every constraint), with either red, black or empty in every space.
* We then create two new boards, one representing the positions where a red mark is present and the other representing the positions where a black mark is present.
* We then find the first free position of each column of the two boards created in the previous step and create a list of all the positions.
* Using the positions list from the previous step we create a list of new boards with each position filled with a red mark. This will be used to find almost winning configurations as these represent the red player’s next move.
* Next, we have a function which determines if red has the next turn, as this will determine whether the configuration has an almost winning configuration. If it is not reds turn then we will not consider this an almost win as black can fill the winning place.
* We then define our example theory:
  + Starting with the possibility that black could have a winning configuration we add constraints that all possible black victories are negated. Meaning red can only win if all possible black victories are false, all of these are added as conjunction.
    - This also tests for invalid cases, as both red and black cannot have winning configurations at the same time, so this preface checks if black has already won.
  + For every constraint we have a nested for-loop which goes through every possible index position of a winning configuration (row, column, diagonal) these are then added to total constraint list. This is done so we can add all constraints to the same encoding while also being able to separate them all by disjunction.
  + Next, if it is not reds turn or the almost winning list is empty(meaning the board is full) the encoding is returned, and the code will determine if red has won or not.
  + If it is reds turn, we then proceed to the almost winning section:
    - From here we go through every element of the almost winning boards and add it as constraints to the encoding. This acts as adding the next tile in all combinations to see if one of them is a winning configuration.
* Next, we have a print statement which states whether the red player has won, has not won, or if black has won.
* Finally, we have our main function:
  + Firstly, we create the almost winning board list
  + Next, we assign the values to x, y and z as even though they were assigned at the beginning of the code, sometimes when running it wouldn’t catch the assignment of the values.
  + Then we create the example theory using the red disk board created in the second step.
  + Finally, we print if the model is satisfiable.