Adam Willis

COM2108 Report

Design

scoreBoard

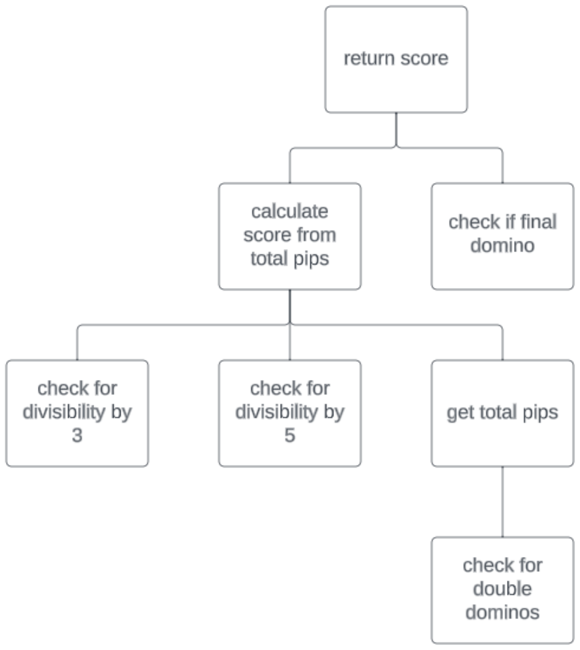
The scoreBoard function takes a Board state and a Boolean that describes whether the domino placed was the last in hand and returns the score of that current Board state.

The score is calculated following the rules of five-and-threes dominos and if it was the final domino, an extra point is added.

To calculate the score, the total pips need to be checked for divisibility by 3 and 5.

The total pips are the pips of the leftmost and rightmost open ends added together.

However, if one of the end dominos is a double, then the pips from the entire domino are added to the total, so this needs to be checked.

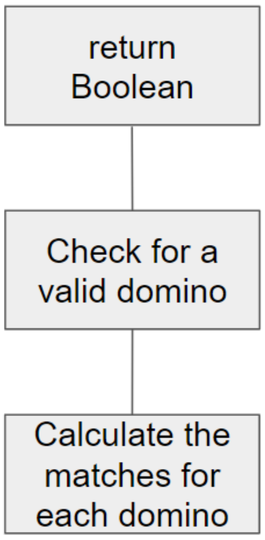

return score
calculate score from total pips
check if final domino
check for divisibility by 3
check for divisibility by 5
get total pips
check for double dominos

blocked

The blocked function takes a Hand of dominos and a Board state and returns a Boolean that describes whether the player with that Hand has any playable dominos or not.

The Boolean is found by checking if any dominos in the Hand match the current state of the Board.

A domino matches the state of the Board if there is at least one match between either side of the domino and leftmost or the rightmost side of the Board.



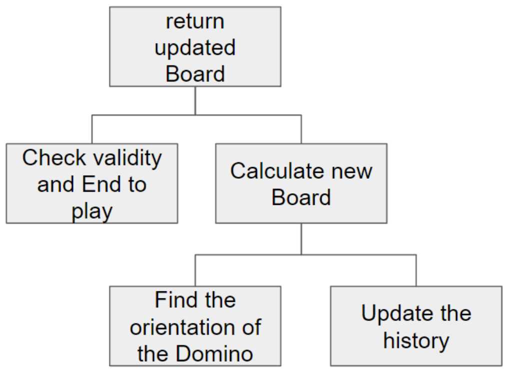
Adam Willis

playDom

The playDom function takes a Player, a Domino to play, the current state of the Board and an End to play the Domino. It returns the new Board if it is a valid play, or Nothing.

It returns the new Board according to the End and whether the Domino is valid.

The new Board is calculated by finding the orientation of the Domino and updating the History.

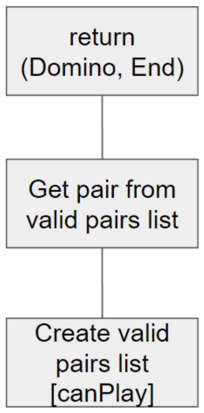

return updated Board
Check validity and End to play
Calculate new Board
Find the orientation of the Domino
Update the history

simplePlayer

The simplePlayer function takes a Hand of dominos, the current Board, the Player and the current Scores. It returns the Domino to play and the End to play it on.

The Domino and End pair is the first pair from the list of valid pairs.

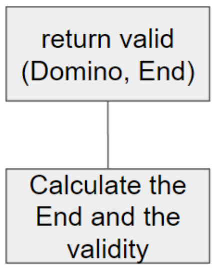
The list of valid pairs is found by applying the canPlay function to the Hand of dominos.


return (Domino, End)
Get pair from valid pairs list
Create valid pairs list [canPlay]

canPlay

The canPlay function takes a Domino and a state of the Board. It returns the Domino and the End to play it on if it is a valid move.

The validity and the End is calculated by checking which side of the Domino matches to which end of the Board.


return valid (Domino, End)
Calculate the End and the validity

smartPlayer

The smartPlayer function takes a Hand of dominos, the current Board, the Player and the current Scores. It returns the Domino to play and the End to play it on.

The strategy for picking the Domino and End pair changes depending on the state of the game.

Adam Willis

The strategy for the first drop checks whether the (5,4) domino is in the Hand as it is the best first move, and picks a domino from the Hand's biggest suit if not to keep all options open.

The biggest suit list is calculated by the biggestSuit function.

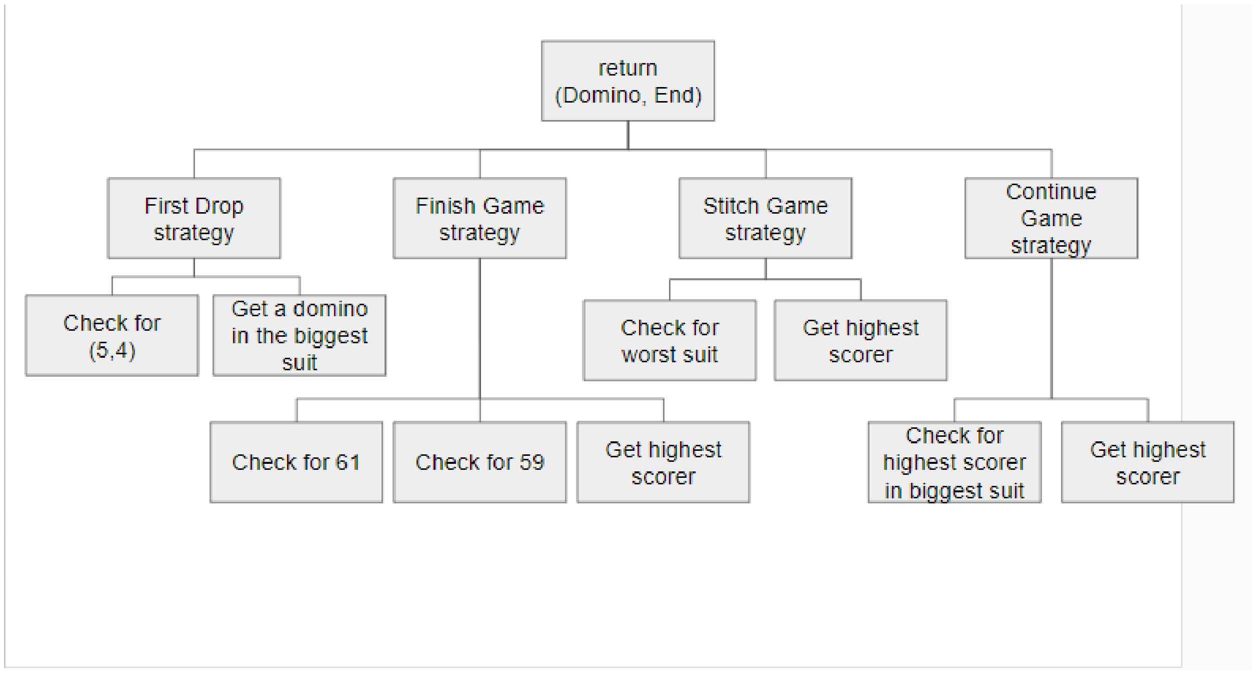
The strategy for finishing the game (when the Player is close to the goal score) checks whether there is a Domino that can get to 61, whether there is a Domino that can get to 59 or simply the highest scoring domino if not.

The correct scoring domino is calculated in the doms2ScoreN function and the highest scoring domino is calculated in the highestScorer function.

The strategy for stitching the game (when the opponent is close to the goal score) checks whether there is a domino in the Hand that is part of the opponents worst suit, or the highest scorer if not.

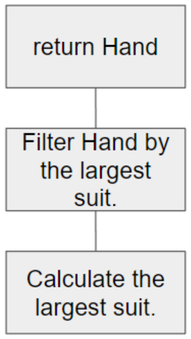
The domino in the worst suit is calculated by the worstSuit function.

The strategy for continuing the game (when neither the player or the opponent are close to the goal score) checks whether there is a high scoring domino in the Hand's biggest suit or simply the highest scorer if not.



Adam Willis

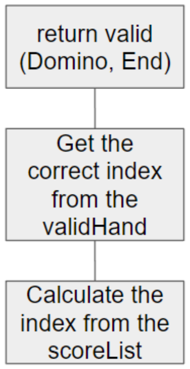
biggestSuit



The biggestSuit function takes a Hand of dominos. It returns a subset of that Hand that contains dominos that are part of the biggest suit in the Hand.

The biggest suit is found by calculating the largest suit and then filtering the original Hand.

doms2ScoreN

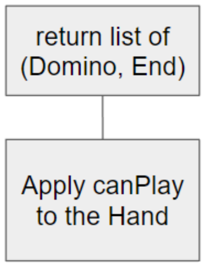

return valid (Domino, End)
Get the correct index from the validHand
Calculate the index from the scoreList

The doms2ScoreN function takes a Hand of dominos, the state of the Board, the goal value. It returns a Domino and End pair if there is a valid domino.

The Domino and End pair is found by checking if the list of scores contains the goal value and taking the correct domino from the list of valid dominos.

The list of scores is calculated by the scoreList function and the list of valid dominos is calculated by the validHand function.

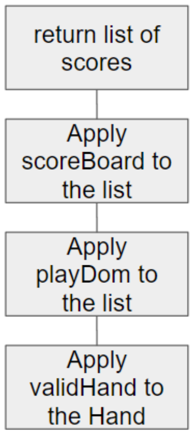
validHand



The validHand function takes a Hand of dominos and a state of the Board. It returns a list of the valid Domino, End pairs for that Board.

It calculates the list by applying the canPlay function to each domino in the Hand.

scoreList

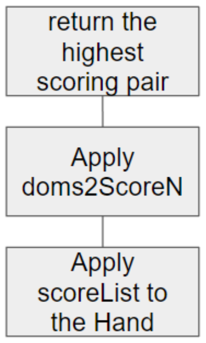

return list of scores
Apply scoreBoard to the list
Apply playDom to the list
Apply validHand to the Hand

The scoreList function takes a Hand of dominos and a state of the Board. It returns a list of scores that each valid domino in the Hand would create on the Board.

It calculates the list by applying the validHand function to the Hand, applying the playDom function to the validHand list and applying the scoreBoard function to the playDom list.

Adam Willis

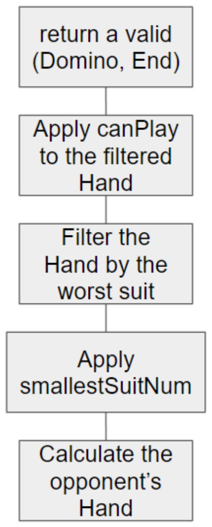
highestScorer



The highestScorer function takes a Hand of dominos, a state of the Board. It returns a Domino and End pair if there is a valid domino.

It finds the highest scoring domino by finding the highest score in the Hand by applying the scoreList function and then applies the doms2ScoreN function with the maximum score to find the highest scoring pair.

worstSuit


return a valid (Domino, End)
Apply canPlay to the filtered Hand
Filter the Hand by the worst suit
Apply smallestSuitNum
Calculate the opponent's Hand

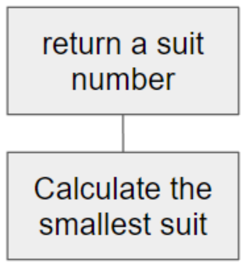
The worstSuit function takes a Hand of dominos, the state of the Board and the current Player. It returns a Domino and End pair that is part of the opposing player's worst suit if there is a valid one.

To find the Domino, End pair, the canPlay function is applied to a subset of the Hand that contains only the dominos that are part of the opposing player's worst suit.

The filtered Hand is created by filtering by the opponent's worst suit, which is calculated by applying the smallestSuitNum function with the opponent's Hand.

The opponent's Hand is formed from the history of the Board.

smallestSuitNum



The smallestSuitNum function takes a Hand of dominos. It returns the number of the smallest suit in the Hand.

The smallest suit is calculated by filtering the Hand by every suit and finding the smallest subset.

Testing

The scoreBoard, blocked and playDom functions were implemented and tested first to complete the stub implementations.

scoreBoard

Input	Output	Reasoning
scoreBoard (State (3,4) (6,6) []) True	9	Shows that the dividing by 3 and 5, checking for doubles and final domino logic all work correctly.
scoreBoard InitState False	0	Shows that it works with no dominos on the board.
scoreBoard (State (5,4) (5,4) []) False	3	Shows that it works with only one domino on the board.

blocked

Input	Output	Reasoning
blocked [(2,6),(4,3)] (State (0,4) (6,5) [])	True	Shows the checking for pip matches logic works.
blocked [(2,6), (4,3)] InitState	False	Shows that it works when there are no dominos on the board.

playDom

Input	Output	Reasoning
playDom P1 (6,1) (State (2,3) (4,1) [((4,1),P2,1)]) R	Just (State (2,3) (1,6) [((4,1),P2,1),((1,6),P1,2)])	Shows the domino orientation and update history logic work.
playDom P2 (3,4) (State (1,2) (5,6) []) L	Nothing	Shows the domino validation logic works.
playDom P1 (5,4) InitState R	Just (State (5,4) (5,4) [((5,4),P1,1)])	Shows that it works with no dominos on the board.

Secondly, the simplePlayer function was implemented and tested by creating the lowest level functions first and testing each before moving up the function design.

Adam Willis

<u>canPlay</u>

Input	Output	Reasoning
canPlay (3,4) (State (3,2) (2,5) [])	Just ((3,4),L)	Shows the validation logic works.
canPlay (3,4) InitState	Just ((3,4),R)	Shows that it works with no dominos on the board.

<u>simplePlayer</u>

Input	Output	Reasoning
simplePlayer [(5,4),(2,1)] (State (1,3) (5,6) []) P1 (30,30)	((2,1),L)	Shows that the valid domino filtering logic works.
domsMatch 100 7 61 simplePlayer simplePlayer 2	(52,48)	Shows that it works in the context of domsMatch and gives accurate scores.

Finally, the smartPlayer function was implemented and tested using the same method as the simplePlayer.

<u>biggestSuit</u>

Input	Output	Reasoning
biggestSuit [(1,2),(3,3),(2,1),(1,4)]	[(1,2),(2,1),(1,4)]	Shows that the suit ranking and filtering logic works.
biggestSuit [(1,1),(2,2)]	[(1,1)]	Shows that it can handle ties in the suit rankings.

<u>validHand</u>

Input	Output	Reasoning
validHand [(3,4),(5,5),(4,1),(2,3)] (State (2,6) (6,4) [])	[((3,4),R),((4,1),R), ((2,3),L)]	Shows that the domino filtering logic works.

<u>scoreList</u>

Input	Output	Reasoning
validHand [(3,4),(5,5),(4,1),(2,3)] (State (2,6) (6,4) [])	[1,1,0]	Shows that the score calculation logic works.

Adam Willis

doms2ScoreN

Input	Output	Reasoning
doms2ScoreN [(3,4),(2,2)] (State (4,2) (6,6) []) 8	Just ((3,4),L)	Shows that the score matching logic works.
doms2ScoreN [(3,4),(2,2)] (State (4,2) (6,6) []) 4	Nothing	Shows that it can handle having no score matches in the given hand.

highestScorer

Input	Output	Reasoning
highestScorer [(3,4),(2,2),(5,4),(6,6)] InitState	Just ((6,6),R)	Shows that the score ranking logic works.

smallestSuitNum

Input	Output	Reasoning
smallestSuitNum [(4,5),(2,3),(6,0)]	1	Shows that the suit ranking works.
smallestSuitNum [(4,5),(2,3),(6,6)]	0	Shows that it can handle ties in the ranking.

worstSuit

Input	Output	Reasoning
worstSuit [(2,1),(3,4),(5,6)] (State (3,4) (0,2) [((0,1),P1,1),((3,4),P1,2),((5,6),P1,3)]) P2	Just ((2,1),R)	Shows that the history parsing and filtering logic work.
worstSuit [(1,1),(3,4),(5,6)] (State (3,4) (0,1) [((0,1),P1,1),((3,4),P1,2),((5,6),P1,3)]) P2	Nothing	Shows that it can handle having no valid dominos.

smartPlayer

Input	Output	Reasoning
smartPlayer [(4,3),(2,5),(3,6)] InitState P1 (0,0)	((4,3),R)	Shows that the first drop logic works.
smartPlayer [(5,4),(2,1),(6,6)] (State (4,3) (3,4) []) P1 (58,50)	((5,4),R)	Shows that the finish game logic works.
smartPlayer [(2,1),(0,4),(6,6)] (State (4,3) (3,4) [((1,2),P2,1),((3,4),P2,2),((5,6),P2,3)]) P1 (43,55)	((0,4),R)	Shows that the stitch game logic works.

smartPlayer [(2,1),(0,4),(3,5),(6,4)] (State (0,3) (3,5) []) P1 (23,23)	((0,4),L)	Shows that the continue game logic works.
domsMatch 100 7 61 smartPlayer simplePlayer 3	(91,9)	Shows that it is more effective than the simplePlayer implementation.

Critical Reflection

Before this module, I only had experience in object-oriented programming and going into the assignment, I knew functional programming was fundamentally different. I felt that trying to design the first functions (scoreBoard, blocked and playDom) for the assignment really encapsulated those differences in programming style. While designing these functions I realized that I was creating a solution with OOP in mind, and going forward I had to make sure that I was working with a functional programming mindset where I would have to approach problems differently.

I had mostly implemented Haskell code without considering the design of the function and therefore, I felt that translating OOP solutions was the best solution. However, while I was designing those first functions I realized that it was important to think about how I could use the benefits of functional programming to reach a better solution that I wouldn't have been able to achieve using only my previous knowledge and techniques.

It was good that I realized this early into the project as it allowed me to create a solution that really embraced the functional programming principles. However, it was definitely a learning curve, and I wish that I had taken the time to actually put the new knowledge into practice because it would have made the project flow more efficiently.

Going forward, I think it is important to take a lesson from this experience and always approach new topics with the mindset that there is something new to learn that could help to improve my work, as in this module I have seen that sometimes I try to apply my knowledge to a new area without attempting to learn new methods or find better solutions.