**Dominoes Player Design**

**Introduction**

My dominoes player was designed for maximum flexibility, both when adding new players and creating new strategies for these players to use.

Each strategy has been programmed as a Tactic:

**Tactic = DomBoard -> Player -> Hand -> Scores -> [PossMove]**

Which takes the current board, player hand and scores and returns a list of PossMove entities. A PossMove is an algebraic type defined as:

**data PossMove = PossMove {**

**dom :: Dom,**

**end :: End,**

**joinPip :: Int**

**}**

As above, a PossMove stores the domino and the end it should be played at, as well as the joining pip on the domino. Storing the pip for a play which is “joined” to the board is useful as it lets us infer the orientation of a domino on the board. This is especially useful in knocking tactics which require information of which pip on an end is “open”. This data type is much neater to work with within the player than tuples of type (Dom, End) seen throughout the program. A player can take the decided upon PossMove to make and convert this back to a tuple of (Dom, End) leaving behind the extra information used internally within the player from the PossMove data type.

**Tactic Evaluation Strategy**

Tactics have been designed to be of two types **critical** and **advisory**. **Critical** tactics have a logical order and either give the player a definite upper hand on a turn (winning the game, preventing a loss, playing (5,4) on the initial board). The PossMoves returned from these critical tactics have a logical order and are checked one by one until one of them returns a domino to play. The order of consideration for critical tactics can be seen in **Figure 1** :



Figure 1

On the other hand, **advisory** tactics return a list of PossMoves which meet a condition which could, but do not guarantee the player an upper hand on the current game state. Should the player exhaust its critical tactics, it can fall back to using these tactics to evaluate the best possible move. For this assignment, I will be experimenting with two different strategies for evaluating these tactics:

* Most Common Occurrence – Assessing PossMoves based on which is most commonly suggested by the set of tactics.
* Logical Ordering – Trying to order the tactics in terms of effectivity, and taking a play from the most useful tactic to return a suggested play.

The tactic framework works well because it allows tactics which prevent losses or incur wins to be considered first, before then falling back to more ambiguous tactics. New tactics can be easily plugged into either the advisory function or the list of critical tactics with ease, allowing for flexible implementation of new strategies. The tactic type also allows for tactics to be easily plugged into new players as each is coded independently without reliance on a specific Player.

**Player Structure**

**Figure 2** outlines the structure explained above, where advisory tactics have been “plugged into” a getAdvice function, which executes an evaluation strategy to choose the best play, and critical tactics are plugged straight into the main player function.



**Tactic Structure**

For the sake of readability, the individual design of each tactic has been included below, Each tactic was designed top down to a level where evaluation seemed trivial.

**DropFiveFour**

This tactic is already trivial without the need for decomposition. If the player has (5,4) and the board has no dominoes on it, the domino is played. Otherwise the tactic returns nothing.



**DomWins**

This tactic returns dominoes which win the game, by trying every possible play on the board and checking whether the augmented board after each of these plays gives a score, which when added to the players current score, equals 61.



**StopOpponentWin**

This tactic returns dominoes which stop the opponent from winning the game on their next turn. For efficiency, it first checks whether the opponent’s score is less than 51 and if so considers the opponent ineligible for a win. Otherwise, the tactic tries making each play the current player can make, and returns those where the domWins tactic (this time given the augmented board, and the opponent’s possible hand) returns no possible plays for the opponent.



**MakeFiftyNine**

This tactic is similar in structure to DomWins, except instead of trying to make the player’s score up to 61, we instead try and make it up to 59, as there are more ways for the player to win from a score of 59 than any other value greater than 51.



**HighScoreDom**

This tactic returns the highest scoring play from a player’s hand, where the opponent’s subsequent play is not better.



**RecklessHighScoreDom**

This is a more brazen version of the strategy above and is advisory rather than critical, as it may not result in a good outcome for the player. It simply returns the highest scoring domino in a pack for the player, with disregard for whether it enables the opponent to have a better next turn.



**PlayMajority**

This function plays a domino where the open-pip for the play is one where the player has the majority hence meaning the opponent may be stitched at one end, and the player can knock off the play on their next turn.



**OpponentStitch**

This tactic returns dominoes which stitch a player’s opponent at either end. It is designed to narrow down the number of plays which could be made by the opponent on their next turn.



**KnockOff**

Returns dominoes which the player can “Knock Off” on their next turn. In essence, it’s a less safe version of playMajority.



**Updated Design**

This piece of design work was completed when implementing the final player. The diagram below details the logical structure of an advisory function. Each of the advisory functions uses the same functions and information structure, just the top-level function uses this information in different ways to decide what domino it should return. As a result I have included below a design for the advisory function used by the final version of my player.



In addition to the above. Strategies which tried to logically order advisory functions were immediately discarded as informal tests showed that they did not provide an optimal result.