Technical documentation

Node class

The Node class holds a board object and then a vector of child nodes. It has two constructors an empty constructer that initializes a new board, and a constructor that takes as argument a board and creates a node containing that board

The NextMove() functions purpose is to return the best child of a node. It works by first populating the list of children of the root node using Expand(). After that, it checks to see if the discard list in the board is empty and if it is it immediately picks the child node that indicates a ‘draw action’ and returns its id. This is done because in those circumstances drawing has no drawbacks and as such it’s the optimal move. After this, a HeuristicValue and position variables are created with the initial value of -1 to hold the best value found and the position of that child. Then the algorithm loops through all the children calling the Search() function at the appropriated depth on each of them. The value returned is then compared to the HeuristicValue, if its higher its value is saved in the HeuristicValue variable and its position in the position variable. If they happen to be equal, then the algorithm call the Heuristic() function on the respective child and on the child currently saved in the position variable and then choses the one with the higher score. After finding the value of the best child, it compares it to the heuristic value of the root node and if its lower it returns -1 to signal that the game is lost. Else the position of the best child is returned.

When generating the child list, the function can come across a child representing a non-deterministic action, these need to be handled differently. During the loop thought all the children, the function will checks the children by calling the isNonDeterministic() function. If it comes across one such node then it will call the predictCard() function, which will generate a list of all the possible outcomes, and then call Search() on each outcome. After that it will use the average value of the scores returned by the Seach() function to compare it to the HeuristicValue variable. The rest of the algorithm behaves the same with the exception being that if a non-deterministic child is selected as the best child then the function revealCard() is called to show which outcome happened out of all the ones possible.

The depth selection works using a variable called AbsoluteDepth saved in the Node class. This variable is initiated as 2 and it can be changed during the subsequent runs of the NextMove() function. After generating the child list, the function saves the current time. After it loops through all the children and it finds the best one, it compares the current time with the time saved to find the run time. If the run time is more than one second it lowers the AbsoluteDepth value by 1. If its lower than the number of currently face down cards multiplied by 0.002 seconds it increments, by 1. The lower bound is set this way to change as the game goes on, at the start of the game there are a lot of face down card so the algorithm needs to take into consideration a lot of uncertainty so the run time can be a bit longer. While at the end of the game, there isn’t a lot to consider so the run time can be shorter.

The current board will be saved in the explored list.

The Search() function works in much the same as the NextMove() function. Given a node, a depth and the explored list, it will save the node in the explored list and then go on to perform the exact same loop that NextMove() performs, by calling Search() on the children nodes at depth -1 , except that this time it doesn’t save the position of the node only the heuristic value found. The only time when this loop isn’t performed is when the depth received is equal with one, at which point it instead returns the heuristic value of the current node as dictated by the Heuristic() function. In both cases the node will be deleted from the explored list right before the return statement.

The Expand() function creates a node list that it populates by calling the several Move function found in the board class and wrapping the results in nodes.

The predictCard() function in the node class only purpose is to call the predictCard() function in the board class, wrap the resulting boards in nodes and return the resulting node list.

Board class

The Board class contains the board object, which is meant to represent a solitaire board at any point in time. It contains the list stock and discard as well as the lists of lists foundations and piles, all of which are meant to represent places on the board where cards can be placed. It also contains a list of cards called undiscovered that holds all the face down cards and a variable named count that represents how many times the algorithm went through the deck without moving any cards drawn of the discard pile and onto one of the other piles or the foundations. The object has two constructors, one that creates an empty board and then proceeds to use the function Deal() to initialize that board, and a second one which is a copy constructor. The equal function is overridden to correctly compare each individual list and lists of lists in the object

The Deal() function first creates a list constating all the cards in the game face down and a list containing the numbers from 0 to 7. The second list is supposed to represent all the piles on the board numbered from 0 to 6 with 7 representing the deck pile. Then the algorithm uses a random number generator to create a random seed, that’s outputted for debugging purposes, which is then used in another random number generator. Using this random number generator, the algorithm picks a card form the first list and a pile from the second. It then moves the card to the chosen pile, deletes it from the first list and then checks to see if the chosen pile has been filled. The piles on the board, when the game starts, should only contain a certain number of cards based on their order, the first pile should have one card the second two cards, as such a pile is considerate filled when they contain that number of cards. The deck pile is considerate filled when it has 24 cards, which is the number of cards left in the deck after the dealing the cards in a normal game. When a certain pile is deemed filled it is removed from the second list and the loop starts again until they run out of cards to be dealt. The undiscovered list is created and then filled out with all the cards in the game. At the end of the game the last card on each pile, excluding the deck pile, is turned face up and then removed from the undiscovered list.

The Win() method is a Boolean method that checks if a given board is a win or not. It achieves this by locking to see if all the foundations have 13 cards.

The Heuristic() function works by first calling the Win function and if the return statement is True it returns the max integer. If then game is not a win then it goes on to create a score variable meant to hold the heuristic value of the board. Because the heuristic value of -1 is often used in the rest of the algorithm to represent a lost or invalid game, the score value cannot go into the negative. As such, the algorithm will give points for good behaviour and no points for bad behaviour. The number of cards in the deck is a good example of that, instead of counting all the cards in the deck and subtracting that amount form the score, the function will first add points to the score equal to the maxim number of cards that can be in the deck, 24, and then take away points for each card that is in the deck. This way, the algorithm still penalizes the A.I. for having to many cards in the deck but the score value cannot go into the negative. To start with, it first goes on to score the cards in the foundation. To achieve this, for each card in the foundation the function will add a predetermine number to the score, the current value is 19, unless the card is higher by two then any of the biggest cards on the two foundations of opposite colour. A black 8 on a foundation will receive 19 points only if the red foundations have cards that are at least as big as a 6, so if the biggest card on either of the red foundation is a 5 the 8 will not receive any score. If only one of the red foundations is so small the 8 will get half the score. This is done to disincentivize the algorithm from prematurely put up cards that could be otherwise useful in pulling cards down form the discard for example. After this the function totals the number of cards on the deck and the discard. It then adds the difference between that number and 24, which is the max value of that number, multiplied by 15. The purpose of this action is to effectively reward the algorithm with 15 points for taking a card of the discard. The number of face down cards is penalized, in much the same way, by adding the difference between the number and 21, the max number of face down cards that can exist in the game, multiplied by 15. For the face down cards the algorithm is also penalized for having all its face down cards on one big pile instead of spreading them across several piles. This is achieved by, for each individual pile, raising the score by 64 minus 2 at the power of the number of face down cards on this pile. Lastly, the function considers the number of empty piles on the board and subtracts them form the score to incentives the A.I. to take advantage of all the space available instead of putting all the cards on a couple piles. To account for the horizon effect, having a king available to be placed on one of the empty piles will cancel out the negative score from having that empty pile.

All the different move functions in the algorithm has the same structure. Using a series of if else statements and for loops to check if a given move is possible and then if it is, generate a copy of that board, make the move in the copy board, and then proceed to return it. For example, the MoveDiscardToPiles() function works by first looking to see if the discard even has cards in it. If it has then, using a for loop to go though all the piles, it checks to see of the top card on each pile is of an opposite colour and has a number bigger by one then the card on top of the discard. If it does a copy board is constructed, the top cards in the discard of said board is moved to the appropriate pile, and then the board is added to the list of newly constructed boards that will be returned at the end of the function. A special clause is made in the if statement, so that if the card on top of the discard is a king, the algorithm will only check if the pile is empty. When the move function deals with a non-deterministic action such as drawing a new card. Because the algorithm can’t know what card happens to be next on the deck, it will leave that card face down to mark that its outcome is unknown.

predictCard() function will first find the face down card. Once it finds it, it will replace it with each card in the undiscovered list, to represent the different possible outcomes of turning the face down card up. For each outcome a now board is created, and the list of boards will be return at the end of the function.

The Boolean function isNonDeterministic() will return True if it find a face down card that marks this as being a non-deterministic board and False otherwise.

The class has two different toString() functions. The first toString() will return a string holding a graphical representation of the board object. The first line will contain the number of cards in the deck and then the top cards on the discard pile and foundation piles respectively. If any of them happen to be empty it will have an empty spot instead. Then, for the next number of rows, equal to the number of cards of in the largest pile, the cards corresponding to the current level on each pile will be shown. On the first row, the cards at the base of each pile will appear next to each other, on the next, the cards one the base of the pile and so on. If one pile does not have a card for a corresponding row in the toString() function the space that it would have occupied will be left empty.

The second toString() function works the same as the first one, but it takes as an argument one more board. This function will then proceed to output the same content as the last one except for a message based on the difference between this boards and the one in the argument. The purpose of the function is being called on a node and being given its parent, to also describe the move that changed the parent to the child. This is done by having several if statements looking for telling differences in between the parent and the child. For example, if the discard pile of the child is bigger then its parent the move was “drawing a card” and the function will out this message along with the graphical representation of the board.

The first toSting() function will only be called on the starting board of any game, due to the lack of parents to compare it to, the second one being preferred everywhere else it can be used

Card class

The card class contains the card object with the variables nr, suit and faceup. The suit variable is an int with different numbers representing different suits, the value of each suit is saved in four final static variable. It has two constructors, a constructor that, being given the values for the number suit and whether it should be face up or not, will create a card containing those values and a copy constructor. It overrides its equal() function to take into account the different values of its variables. The toString() function will return two square brackets with space between them if the card is face down. While if its face up, it will put between the brackets the value of the nr variable and its suit. The Flip() function will change the value of the faceup variable. The class also contains two functions isBlack() and isFaceUp() which will return the colour and the orientation of the card respectively.

Main class

The debugging features of the code are achieved through two different main() functions. They both use a class called Global that only contains a prinstStream to output any information necessary to a text file. The first function will initiate the global printStream, the root node. It will then use a while loop to continually call NextMove() and then replace the current node with the child indicated by the NextMove() function. Using the printStream the main function will output, at each step, the current node, the number of face down cards and the depth at which the search is happening as well as the time taken by each individual NextMove() call. The loop will stop when the NextMove() function return -1

The second main() method will interact with the NextMove() function in much the same way, create a node, loop using the function to find the best children until it tells it to stop. The difference is that this function is doing this several times. Using a while loop tied to a timer the process is done several times until the times runs out. The information chosen to be collected is different as well. After each game played the function, using the printStream, will print in the text file the number of moves made, the time spent and whether it won the game. At the end of the run, the number of total games player, number of wins and the overall win rate will also be displayed.