4995 Final Report - Freecell

Joe Huang jch2220

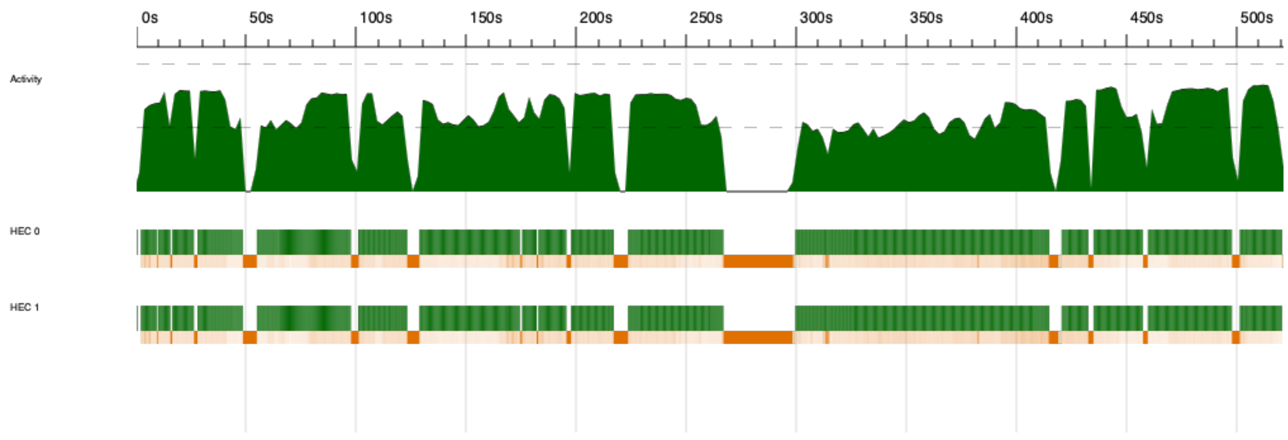
December 2019

1 Implementation

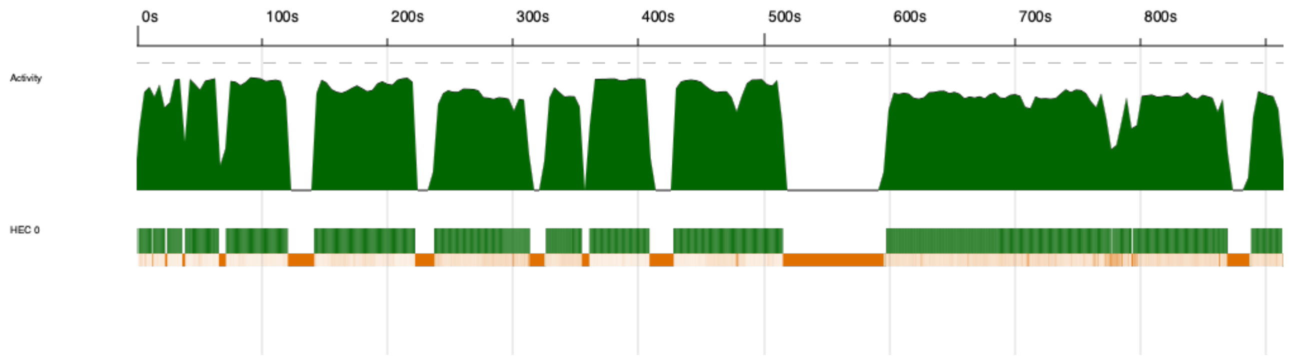
In this project, I implemented a Freecell game and a solver. The project includes an algorithm to deal cards for the game, the Freecell game, and the solver. For dealing the cards, I followed the algorithm proposed by Jim Horne with deals numbered from 1 to 32000. Then I implemented the game structures/rules and the game controller. The game file has the data structure for the game and functions that allow the controller to play the game. The controller acts as the interface for the user to interact with the game. Finally, the solver is implemented following the Heineman's Staged Deepening Heusirtic. All the details of each file/algorithms and examples of running the game and the solver can be found in the README.md as well as in the file descriptions.

2 Parallel Performance

For the solver to search in parallel, the program is implemented such that given a game status and all its possible moves, all the future moves are found in parallel and then combine together for scoring. In the game, all different moves are calculated in parallel and then concatenate together before returning to the user. However, there are parts of the program that are not ran in parallel, such as applying the move to the game and sorting from Data.List. The program was ran with -N1 and -N2 tags to see the difference when the programs are running in



1. Solver running with -N2



2. Solver running with -N1

parallel. From the -N2 figures, we can see that the load are balancing well between the two threads. However, because Freecell solver needs to keep and dispose a lot of previous states, we can see that the GC takes a lot of time in between. Comparing the time with -N1, -N2 time finished solving a game in around 500s and -N2 solved in around 900s, which shows that running in parallel really speed up the process.

3 Future Work

While the solver was able to solve game 164, other games actually takes a long time and I didn't have time to find any more game that it could solve. I believe that more complicated heuristic are needed to guide the solver, for example: considering more statistic of the game and using different weights for each features. From the paper 'Evolutionary Design of FreeCell Solvers' by Elyasaf et al., they suggested to use genetic algorithm to evolves heuristic that guides the searching, which are good directions to continue for the solver. Finally, the current Freecell game is only text-based and requires the user to keep the game state. Another future work could be to provide a graphic-based game.

4 Code - README

```
# 4995 Final Project

### by Joe Huang (jch2220)


This is the final project for COMS 4995 attempting to solve the

freecell games. The project includes an algorithm to deal cards

for the game, the Freecell game, and the solver.


In this readme, we will first introduce the game of freecell,

one example of running the game controller and of running the solver,

the details the purpose and key functions in each files, and finally
```

a full demo of the game and the solver.

## What is Freecell?

Freecell is a card game consists of piles of cards, four freecells, and four foundations. The goal of the game is to place all the cards in the foundations. Each foundations represent one suit of the card and to place a card in a foundation, that card must be in the same suit and one rank higher (except for Ace which can be placed when the foundation is empty). Players can move the top-most card from one pile to another or from freecell to a pile as long as placing card is one rank lower and in opposite color. Players can leverage the freecells to put any card they want from the piles. Once a card is placed in the foundation, it cannot be removed. In addition, multiple cards can be moved together from pile to pile if they are in desending order from the top and in alternating color. The number of cards they can move at once is one more than all the current free spaces on the board (freecell or empty pile column).

## Example

### Running the Game Controller

To run the game, the following APIs are used:

```
start <gameNumber>: this command takes a game number and output a
                    game status. The user will need to keep the game
                    status to make the next move.

makeMove <gameStatus> <move>: this command takes a game status and a
                              move. It returns two values: a boolean
                              value represent whether the move was valid
                              and a new game status if the move was
                              applied.
```

An example to run would be the following:

```
‘‘‘

> stack ghci

Prelude>:l gameController

*GameController>g1 = start 1

*GameController>(valid, g2) = makeMove g1 (MoveCasToFC (Card Six Spade) 0)

*GameController>g2


FC: 6S

FD: [0,0,0,0]

0  1  2  3  4  5  6  7

JD 2D 9H JC 5D 7H 7C 5H

KD KC 9S 5S AD QC KH 3H

2S KS 9D QD JS AS AH 3C

4C 5C TS QH 4H AC 4D 7S

3S TD 4S TH 8H 2C JH 7D

6D 8S 8D QS 6C 3D 8C TC

   9C 2H 6H


All Possible Moves:

[MoveCasToFC (Card Six Diamond) 0,MoveCasToFC (Card Nine Club) 1,

MoveCasToFC (Card Two Heart) 2,MoveCasToFC (Card Six Heart) 3,

MoveCasToFC (Card Six Club) 4, MoveCasToFC (Card Three Diamond) 5,

MoveCasToFC (Card Eight Club) 6,MoveCasToFC (Card Ten Club) 7]
‘‘‘
```

The user can print the game status to get all the possible move they can take.

### Running the solver

The solver can be run directly as executable. Once it is ran, a
message will ask the user to input a game number. If the number
is valid, the solver will return all the steps from start to
finish once a solution is found.

To run the solver, enter the following command:

```
>stack ghc -- -O2 -Wall solver.hs -threaded -rtsopts -eventlog
>./solver +RTS -N2 -s
Please enter a game number:
164
[MoveCasToCas (Card Six Spade) 0 6,MoveCasToFC (Card Queen Heart) 3
...(All the actions lead to winning)
```

## File Description

### deal.hs

This file handles the card dealing in the free cell. The deal allows input number from 1 to 32000 and follows the algorithm developed by Jim Horne.

The algorithm are as followed:

1. Seed the RNG with the number of the deal.
2. Create an array of 52 cards: Ace of Clubs, Ace of Diamonds, Ace of Hearts, Ace of Spades, 2 of Clubs, 2 of Diamonds, and so on through the ranks: Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King. The array indexes are 0 to 51, with Ace of Clubs at 0, and King of Spades at 51.
3. Until the array is empty:
    - Choose a random card at index next random number (mod array length).
    - Swap this random card with the last card of the array.
    - Remove this random card from the array. (Array length goes down by 1.)
    - Deal this random card.
4. Deal all 52 cards, face up, across 8 columns. The first 8 cards go in 8 columns, the next 8 cards go on the first 8 cards, and so on.

To deal a certain game number, run the following:

‘‘‘

```
Prelude> :l deal
*Deal> startAndShowDeal 1
["JD","2D","9H","JC","5D","7H","7C","5H"]
["KD","KC","9S","5S","AD","QC","KH","3H"]
["2S","KS","9D","QD","JS","AS","AH","3C"]
["4C","5C","TS","QH","4H","AC","4D","7S"]
["3S","TD","4S","TH","8H","2C","JH","7D"]
["6D","8S","8D","QS","6C","3D","8C","TC"]
["6S","9C","2H","6H"]
```

‘‘‘

### game.hs

This file describes the freecell game structure. Here it defines the data structure for the game component, show (print) functions, loading functions to load the deal from deal.hs, game functions that provide all possible move given a game status, apply functions that apply move to a game status, and some helper functions.

The Game consists of several components:
Rank, Suit, Card, Cascade, Cascades, Freecell, Foundation, Stack,Game, and Move. Each card is made of a Rank and a Suit. Each Cascade and Freecell is made of a list of Card. A Foundation is made of list of integer representing the current highest foundation for each suit.
A Cascades is made of a list of Cascades. A Stack is made of a list of cards that can be moved together. A Move specifies information about that move, including the type of move, the destination of the move the card to be moved, etc. Finally, a Game consists of the cascades, the foundation, and the freecell.

An example of a game:

‘‘‘

```
FC:
FD: [0,0,0,0]
```

```
0  1  2  3  4  5  6  7
JD 2D 9H JC 5D 7H 7C 5H
KD KC 9S 5S AD QC KH 3H
2S KS 9D QD JS AS AH 3C
4C 5C TS QH 4H AC 4D 7S
3S TD 4S TH 8H 2C JH 7D
6D 8S 8D QS 6C 3D 8C TC
6S 9C 2H 6H
‘‘‘
```

Given a game, the getAllMove function provides all the move that can
be made. A list of moves are as followed:

- MoveCasToFC card from: Move a card from a cascade to the freecell.
- MoveCasToFoun card from: Move a card from a cascade to the foundation.
- MoveCasToCas card from to: Move a card from a cascade to another.
- MoveFCToCas card to: Move a card from freecell to a cascade.
- MoveFCToFoun card: Move a card from freecell to foundation
- MoveMult card stack from to: Move a stack of cards from a cascade to
                                another.

```
‘‘‘
*Game> game = initializeGame 1
*Game> getAllMove game
[MoveCasToFC (Card Six Spade) 0,MoveCasToFC (Card Nine Club) 1,
MoveCasToFC (Card Two Heart) 2,MoveCasToFC (Card Six Heart) 3,
MoveCasToFC (Card Six Club) 4,MoveCasToFC (Card Three Diamond) 5,
MoveCasToFC (Card Eight Club) 6,MoveCasToFC (Card Ten Club) 7]
‘‘‘
```

Once we have a valid move, we can apply the move to the game. The apply
function takes the game and the move, then returns the game after the move
has been applied.

```
‘‘‘
*Game> game = initializeGame 1
```

```
*Game> getAllMove game
[MoveCasToFC (Card Six Spade) 0,MoveCasToFC (Card Nine Club) 1,MoveCasToFC (Card Two Heart) 2,
MoveCasToFC (Card Six Heart) 3,MoveCasToFC (Card Six Club) 4,MoveCasToFC (Card Three Diamond) 5,
MoveCasToFC (Card Eight Club) 6,MoveCasToFC (Card Ten Club) 7]
*Game> applyMove game (MoveCasToFC (Card Six Spade) 0)
applyMove game (MoveCasToFC (Card Six Spade) 0)
FC: 6S
FD: [0,0,0,0]
0  1  2  3  4  5  6  7
JD 2D 9H JC 5D 7H 7C 5H
KD KC 9S 5S AD QC KH 3H
2S KS 9D QD JS AS AH 3C
4C 5C TS QH 4H AC 4D 7S
3S TD 4S TH 8H 2C JH 7D
6D 8S 8D QS 6C 3D 8C TC
   9C 2H 6H
```

### gameController.hs

The game controller connects the user and the game as well as keeps track of the game history. This file simplifies the API to play the game as well as make it easier for the solver to make moves. The two command are as followed:

start <gameNumber>: return a gamestatus with the given game number deal

makeMove <gameStatus> <move>: return a move status and tbe game status given a move. If the move is successful, move status would be set to Success and the game status would reflect the change. If the move is not valid, then the move stauts would be set to Fail with the original game

status. Finally, if the game is finished
the move status would return Win.

```
*GameController> game = start 2
*GameController> makeMove game (MoveCasToFC (Card Three Heart) 0)
(Success,FC: 3H
FD: [0,0,1,1]
0  1  2  3  4  5  6  7
QD QC KC 3C 4C 2C KD 5C
4D JD JS 6H QS 6D 2D 9C
TD JC 8C 6C 8S 4S 5D QH
7S 9D KS 7C 6S 4H AC 8H
   9S TC 2S 3S TS 9H 2H
   AD 7H 3D 5H 8D KH 7D
   5S TH JH



All Possible Moves:
[MoveCasToCas (Card Seven Spade) 0 5,MoveCasToFC (Card Seven Spade) 0,
MoveCasToFC (Card Five Spade) 1,MoveCasToFC (Card Ten Heart) 2,
MoveCasToFC (Card Jack Heart) 3,MoveCasToFC (Card Five Heart) 4,
MoveCasToFC (Card Eight Diamond) 5,MoveCasToFC (Card King Heart) 6,
MoveCasToFC (Card Seven Diamond) 7]

)
```

### solver.hs

The solver implements the Heineman's Staged Deepening Heusirtic (HSDH) to find a solution for a freecell game. HSDH first find all the states k steps away from the current one and rank them based on the heusirtic. The heusirtic checks the foundation's current cards and finds all the card that

are supposed to be placed next. For those next cards, calculate how many
cards are on top of them. The heusirtic mulplies that score by two if
all freecell are used or any foundation cell is 0. Then, the best state are
used for the next iteration until a solution is found.

## Demo

### Run a entire game from start to finish

```
*GameController> g1 = start 15140
*GameController> (_, g2)= makeMove g1 (MoveCasToCas (Card Queen Spade) 7 5)
*GameController> (_, g3) = makeMove g2 (MoveCasToCas (Card Eight Diamond) 3 6)
*GameController> (_, g4) = makeMove g3 (MoveCasToCas (Card Seven Spade) 1 6)
*GameController> (_, g5) = makeMove g4 (MoveCasToCas (Card Eight Spade) 2 3)
*GameController> (_, g6) = makeMove g5 (MoveCasToCas (Card Seven Heart) 4 3)
*GameController> (_, g7) = makeMove g6 (MoveCasToCas (Card Six Diamond) 4 6)
*GameController> (_, g8) = makeMove g7 (MoveCasToCas (Card Jack Club) 2 1)
*GameController> (_, g9) = makeMove g8 (MoveCasToCas (Card Four Spade) 7 2)
*GameController> (_, g10) = makeMove g9 (MoveMult (Card Nine Spade)
    (Stack [Card Nine Spade,Card Eight Diamond,Card Seven Spade,Card Six Diamond]) 6 7)
*GameController> (_, g11) = makeMove g10 (MoveCasToCas (Card Ten Spade) 6 4)
*GameController> (_, g12)= makeMove g11 (MoveMult (Card Nine Heart)
    (Stack [Card Nine Heart,Card Eight Spade,Card Seven Heart]) 3 4)
*GameController> (_, g13) = makeMove g12 (MoveMult (Card King Diamond)
    (Stack [Card King Diamond,Card Queen Spade]) 5 6)
*GameController> (_, g14) = makeMove g13 (MoveMult (Card Jack Heart)
    (Stack [Card Jack Heart,Card Ten Spade,Card Nine Heart,Card Eight Spade,Card Seven Heart]) 4 6)
*GameController> (_, g15) = makeMove g14 (MoveCasToCas (Card Six Spade) 4 6)
*GameController> (_, g16) = makeMove g15 (MoveMult (Card Queen Diamond)
    (Stack [Card Queen Diamond,Card Jack Club]) 1 4)
*GameController> (_, g17) = makeMove g16 (MoveMult (Card Ten Diamond)
    (Stack [Card Ten Diamond,Card Nine Spade,Card Eight Diamond,Card Seven Spade,Card Six Diamond])
    7 4)
*GameController> (_, g18) = makeMove g17 (MoveCasToCas (Card Seven Diamond) 1 3)
```

```
*GameController> (_, g19) = makeMove g18 (MoveCasToFC (Card Jack Diamond) 5)
*GameController> (_, g20) = makeMove g19 (MoveCasToCas (Card Five Heart) 0 6)
*GameController> (_, g21) = makeMove g20 (MoveCasToCas (Card Queen Heart) 1 0)
*GameController> (_, g22) = makeMove g21 (MoveCasToCas (Card Jack Spade) 1 0)
*GameController> (_, g23) = makeMove g22 (MoveMult (Card King Club)
    (Stack [Card King Club,Card Queen Heart,Card Jack Spade]) 0 1)
*GameController> (_, g24) = makeMove g23 (MoveCasToFC (Card King Heart) 0)
*GameController> (_, g25) = makeMove g24 (MoveCasToFC (Card Nine Club) 0)
*GameController> (_, g26) = makeMove g25 (MoveMult (Card Eight Club)
    (Stack [Card Eight Club,Card Seven Diamond]) 3 0)
*GameController> (_, g27) = makeMove g26 (MoveCasToCas (Card Ten Heart) 3 1)
*GameController> (stat, g28) = makeMove g27 (MoveCasToFC (Card Nine Diamond) 3)
*GameController> stat
Win
*GameController> g28
FC:
FD: [13,13,13,13]
0  1  2  3  4  5  6  7




All Possible Moves: []
```

‘‘‘

### Run a solver and print the solutions

The example of running game number 164. It takes about 13 minutes on
a dual-core machine.

‘‘‘

```
>stack ghc -- -O2 -Wall solver.hs -threaded -rtsopts -eventlog
>./solver +RTS -N2 -s
Please enter a game number:
164
```

```
[MoveCasToCas (Card Six Spade) 0 6,MoveCasToFC (Card Queen Heart) 3,
MoveCasToFC (Card Eight Club) 3,MoveCasToFC (Card Nine Club) 3,
MoveCasToFC (Card Two Spade) 0,MoveCasToCas (Card Five Heart) 0 6,
MoveCasToFoun (Card Ace Diamond) 0,MoveCasToCas (Card Five Spade) 0 1,
MoveCasToFoun (Card Ace Spade) 0,
MoveCasToFoun (Card Ace Heart) 0,MoveFCToFoun (Card Two Spade),
MoveCasToFoun (Card Three Spade) 5,
MoveCasToFoun (Card Four Spade) 5,MoveCasToFoun (Card Five Spade) 1,
MoveFCToCas (Card Eight Club) 5,
MoveMult (Card Seven Diamond) (Stack [Card Seven Diamond,Card Six Spade,Card Five Heart]) 6 5,
MoveCasToFC (Card King Spade) 3,MoveCasToCas (Card Eight Heart) 3 0,
MoveCasToCas (Card Four Heart) 3 7,MoveCasToFoun (Card Ace Club) 3,
MoveMult (Card Five Club) (Stack [Card Five Club,Card Four Heart]) 7 2,
MoveFCToCas (Card Nine Club) 3,MoveCasToCas (Card Eight Heart) 0 3,
MoveCasToCas (Card Seven Heart) 6 0,
MoveCasToFC (Card Six Heart) 1,MoveCasToCas (Card Six Club) 1 0,
MoveCasToFC (Card Four Diamond) 7,
MoveCasToCas (Card Ten Club) 1 7,MoveCasToFoun (Card Two Heart) 1,
MoveCasToFoun (Card Three Heart) 4,
MoveCasToFoun (Card Four Heart) 2,MoveCasToFoun (Card Five Heart) 5,
MoveCasToFoun (Card Six Spade) 5,
MoveFCToFoun (Card Six Heart),
MoveMult (Card Six Diamond) (Stack [Card Six Diamond,Card Five Club]) 2 4,
MoveFCToCas (Card Four Diamond) 4,MoveCasToFC (Card King Diamond) 2,
MoveCasToCas (Card Three Club) 2 4,
MoveCasToCas (Card Six Club) 0 5,MoveCasToFoun (Card Seven Heart) 0,
MoveCasToFoun (Card Eight Heart) 3,
MoveMult (Card Seven Diamond) (Stack [Card Seven Diamond,Card Six Club]) 5 0,
MoveCasToFC (Card Nine Club) 3,
MoveFCToCas (Card King Diamond) 3,MoveCasToCas (Card Queen Club) 6 3,
MoveMult (Card Jack Diamond) (Stack [Card Jack Diamond,Card Ten Club]) 7 3,
MoveCasToCas (Card Nine Diamond) 6 3,
MoveCasToFoun (Card Two Club) 6,MoveCasToFoun (Card Three Club) 4,
MoveCasToFC (Card Jack Club) 7,
MoveCasToFoun (Card Two Diamond) 7,MoveCasToFoun (Card Three Diamond) 6,
MoveCasToFoun (Card Four Diamond) 4,MoveCasToCas (Card Jack Spade) 1 6,
MoveCasToCas (Card Ten Diamond) 1 6,MoveCasToFoun (Card Five Diamond) 1,
MoveCasToCas (Card Eight Club) 5 1,MoveCasToFoun (Card Nine Heart) 5,MoveCasToFoun (Card Four Club) 5,
MoveCasToFoun (Card Five Club) 4,MoveCasToFoun (Card Six Club) 0,
MoveCasToFoun (Card Six Diamond) 4,
MoveCasToFoun (Card Seven Diamond) 0,MoveCasToFoun (Card Seven Club) 4,
MoveCasToFoun (Card Eight Club) 1,MoveCasToFoun (Card Eight Diamond) 7,
MoveCasToFoun (Card Nine Diamond) 3,MoveCasToFoun (Card Ten Diamond) 6,
MoveFCToFoun (Card Nine Club),MoveCasToFoun (Card Ten Club) 3,
MoveCasToFoun (Card Jack Diamond) 3,MoveFCToFoun (Card Jack Club),
MoveCasToFoun (Card Queen Club) 3,MoveCasToCas (Card King Heart) 2 0,
MoveCasToFoun (Card Seven Spade) 2,MoveCasToFoun (Card King Club) 2,
MoveCasToFoun (Card Eight Spade) 5,
MoveCasToFoun (Card Nine Spade) 4,MoveCasToFoun (Card Ten Spade) 4,
MoveCasToFoun (Card Ten Heart) 4,MoveCasToFoun (Card Jack Heart) 2,
MoveCasToFoun (Card Queen Diamond) 4,
MoveCasToFoun (Card King Diamond) 3,MoveCasToFoun (Card Jack Spade) 6,
MoveCasToFoun (Card Queen Spade) 5,
MoveFCToFoun (Card King Spade),MoveFCToFoun (Card Queen Heart),
MoveCasToFoun (Card King Heart) 0]
```
‘‘‘

5 Codes

5.1 deal.hs

```
module Deal where
import Data.List


{-


deal.hs
by Joe Huang (jch2220)


4995 Final Project
Description:
This file handles the card dealing in the free cell. The deal allows
input number from 1 to 32000 and follows the algorithm developed by
Jim Horne.

The algorithm are as followed:


1. Seed the RNG with the number of the deal.
2. Create an array of 52 cards: Ace of Clubs, Ace of Diamonds, Ace of
    Hearts, Ace of Spades, 2 of Clubs, 2 of Diamonds, and so on through
    the ranks: Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King. The
    array indexes are 0 to 51, with Ace of Clubs at 0, and King of
    Spades at 51.
3. Until the array is empty:
    - Choose a random card at index  next random number (mod array length).
    - Swap this random card with the last card of the array.
    - Remove this random card from the array. (Array length goes down by 1.)
    - Deal this random card.
4. Deal all 52 cards, face up, across 8 columns. The first 8 cards go
    in 8 columns, the next 8 cards go on the first 8 cards, and so on.

Reference:
Deal Cards for FreeCell: https://tfetimes.com/c-deal-cards-for-freecell/


-}


suits :: [String]
suits = ["C","D","H","S"]


ranks :: [String]
ranks = ["A","2","3","4","5","6","7","8","9","T","J","Q","K"]


initialDeck :: [String]
initialDeck = [a++b | a <- ranks, b <- suits]
```

```
rng51 :: Int -> [Int] -> [Int]
rng51 seed rngList
    | length rngList == 51 = reverse $ map (`div` (2 ^ 16)) rngList
    | otherwise = rng51 rnd (rnd:rngList)
        where rnd = ((214013 * seed + 2531011) `mod` ((2 ^ 31)))

rng51ModLength :: Int -> [Int]
rng51ModLength seed = zipWith (mod) (rng51 seed []) [52,51..0]

findAndReplaceWithLast :: Int -> [String] -> (String, [String])
findAndReplaceWithLast index currentDeck
    | index >= length currentDeck =
        error "DealCards: find and replacing index exceed the deck length."
    | otherwise = (head sndDeck,
                  fstDeck ++ (if length removedDeck > 0
                              then (last removedDeck):(init removedDeck)
                                    else []))
        where (fstDeck, sndDeck) = splitAt index currentDeck
              currentCard = head sndDeck
              removedDeck = delete currentCard sndDeck

deal :: [Int] -> [String] -> [String]
deal indexList currentDeck
    | length currentDeck == 1 = [head currentDeck]
    | otherwise = (newDeal):(deal (tail indexList) newDeck)
        where (newDeal, newDeck) = findAndReplaceWithLast (head indexList) currentDeck

startDeal :: Int -> [String]
startDeal i
    | i < 1 || i > 32000 = error "DealCards: exceed input range [1,32000]."
    | otherwise = deal (rng51ModLength i) initialDeck

separateDealRow :: [String] -> [[String]]
separateDealRow dealedCard =
    unfoldr (\x -> if null x then Nothing else Just $ splitAt 8 x) dealedCard
```

```
showDeal :: [[String]] -> IO ()
showDeal dealedCard = mapM_ print dealedCard

startAndShowDeal :: Int -> IO ()
startAndShowDeal i = showDeal $ separateDealRow $ startDeal i
```

5.2 game.hs

```
module Game where
import Deal
import Data.List (elemIndex, inits)
import Control.Monad (guard)
import Control.Parallel.Strategies (parMap, rpar)

{-

game.hs
by Joe Huang (jch2220)

4995 Final Project

Descriptions:
This file describes the freecell game structure. Here it defines the
data structure for the game component, show (print) functions, loading
functions to load the deal from deal.hs, game functions that provide
all possible move given a game status, apply functions that apply move
to a game status, and some helper functions.

The basic rule of the Freecell is as followed (from the reference):

    In a game of FreeCell solitaire, there are 8 columns, 4 foundations
    spaces, and 4 free cells. At the beginning of each game, columns 1
    through 4 are dealt 7 cards each; columns 5 through 8 are dealt 6 cards each.
    The four foundation spaces are used to store the cards from each
    suit (Hearts, Diamonds, Spades and Clubs). Cards must be placed in
```

```haskell
    the foundation of their suit in ascending order from the ace to the
    king, but once a card has been placed in the foundation it cannot be
    removed. The free cells start empty and can hold a single card of
    any suit as long as the cell is empty. Cards can be placed in the
    columns onto cards that are of the opposite suit color and one value higher.

    Stacks of cards can be moved as long as the stack is alternating suit
    colors and is stacked in ascending order. To move a stack of cards
    of size n, the number of open free cells and the number of empty
    columns must add up to n-1. The objective of the game is to move
    all 52 cards into the foundation spaces. To complete the objective
    the cards must be organized in a way that allows the cards to be placed
    into the foundations, thus solving the game.

Reference:
Freecell Solitaire Optimization:
http://people.uncw.edu/tagliarinig/Courses/380/S2015%20papers%20and%20presentations/Freecell%20Opt-Beas

-}

-- ---------------------------------------------------------------------------
--                                Data Structures
-- ---------------------------------------------------------------------------

data Rank = Ace | Two | Three | Four | Five | Six | Seven | Eight | Nine |
            Ten | Jack | Queen | King
    deriving (Eq,Ord,Enum,Bounded,Show,Read)

data Suit = Club | Diamond | Heart | Spade
    deriving (Eq,Ord,Enum,Bounded,Show,Read)

data Card = Card Rank Suit
    deriving (Eq,Ord,Bounded,Show,Read)

data Cascade = Cascade [Card]
```

```
    deriving (Show,Read,Eq,Ord)

data Cascades = Cascades [Cascade]
    deriving (Eq, Ord)

instance Show Cascades where
   show (Cascades ccs) = unlines $ (parMap rpar) unwords $ showCascades (Cascades ccs)

data Freecell = Freecell [Card]
    deriving (Eq, Ord)

instance Show Freecell where
    show (Freecell cards) = unwords $ (parMap rpar) showCard cards

data Foundation = Foundation [Int]
    deriving (Show, Read, Eq, Ord)

data Game = Game {cascades :: Cascades, freecell :: Freecell,
                  foundation :: Foundation }
    deriving (Eq, Ord)

instance Show Game where
    show (Game ccs fc (Foundation fd)) = unlines ["FC: " ++ show fc,
                                     "FD: " ++ show fd, show ccs]

data Stack = Stack [Card]
    deriving (Show, Read, Eq, Ord)

data Move = MoveCasToFC Card Int
          | MoveCasToFoun Card Int
          | MoveCasToCas Card Int Int
          | MoveFCToCas Card Int
          | MoveFCToFoun Card
          | MoveMult Card Stack Int Int
    deriving (Show, Read, Eq, Ord)
```

```
-- ----------------------------------------------------------------------------
--                              Load functions
-- ----------------------------------------------------------------------------

loadRank :: Char -> Rank
loadRank rank =
    case rank of
        'A' -> Ace
        '2' -> Two
        '3' -> Three
        '4' -> Four
        '5' -> Five
        '6' -> Six
        '7' -> Seven
        '8' -> Eight
        '9' -> Nine
        'T' -> Ten
        'J' -> Jack
        'Q' -> Queen
        'K' -> King
        _ -> error "Invalid input for suit"

loadSuit :: Char -> Suit
loadSuit suit =
    case suit of
        'C' -> Club
        'D' -> Diamond
        'H' -> Heart
        'S' -> Spade
        _ -> error "Invalid input for suit"

loadCard :: String -> Card
loadCard card
    | length card /= 2 = error "Invalid input for card"
    | otherwise = Card (loadRank (head card)) (loadSuit (last card))


loadCascade :: [String] -> Cascade

loadCascade cascade = Cascade ((parMap rpar) loadCard cascade)


loadCascades :: [[String]] -> Cascades

loadCascades ccs = Cascades ((parMap rpar) loadCascade $ foldl combineListofLists []

                                             $ (parMap rpar) breakListofList ccs)


-- ------------------------------------------------------------------------------

--                              Show functions

-- ------------------------------------------------------------------------------


showRank :: Rank -> Char

showRank rank =

    case rank of

        Ace -> 'A'

        Two -> '2'

        Three -> '3'

        Four -> '4'

        Five -> '5'

        Six -> '6'

        Seven -> '7'

        Eight -> '8'

        Nine -> '9'

        Ten -> 'T'

        Jack -> 'J'

        Queen -> 'Q'

        King -> 'K'


showSuit :: Suit -> Char

showSuit suit =

    case suit of

        Club -> 'C'

        Heart -> 'H'
        Spade -> 'S'
        Diamond -> 'D'

showCard :: Card -> String
showCard (Card rank suit) = [showRank rank, showSuit suit]

-- function to make cascade from column into row so showCascades can combine them
showCascade :: Cascade -> Int -> Int -> [[String]]
showCascade (Cascade cards) l maxSpace
    | l == maxSpace = [[]]
    | length cards == 0 && l < maxSpace = ["  "]:(showCascade (Cascade []) (l + 1) maxSpace)
    | otherwise = [showCard (head cards)]:(showCascade (Cascade (tail cards)) (l+1) maxSpace)

-- function to show cascades by combining cascade as row
showCascades :: Cascades -> [[String]]
showCascades (Cascades ccs) = [(show i ++ " ") | i <- [1..8::Int]]:
    (foldl combineListofLists []
           $ (parMap rpar) (\x->showCascade x 0 (getMaxLengthCascasde (Cascades ccs) 0 )) ccs )

-- ---------------------------------------------------------------------------
--                              Game functions
-- ---------------------------------------------------------------------------

-- initialize a game given a deal number
initializeGame :: Int -> Game
initializeGame gameNumber = Game (loadCascades $ separateDealRow $ startDeal gameNumber)
                                 (Freecell [])
                                 (Foundation [0,0,0,0])

-- check if the game is finished
-- ex: isFinished (Game (Cascades []) (Freecell []) (Foundation [13,13,13]))
isFinished :: Game -> Bool
isFinished (Game (Cascades ccs) (Freecell fc) (Foundation fd)) =
    (checkAllCascadeEmpty (Cascades ccs)) &&
    (length fc == 0) && (and $ (parMap rpar) (\x -> x == 13) fd)
```

```haskell
-- given a game, get all Move
getAllMove :: Game -> [Move]
getAllMove game = do
    (getAllCasToCas game) ++ (getAllCasToFC game) ++ (getAllCasToFoun game)
    ++ (getAllFCToCas game) ++ (getAllMoveMult game) ++ (getAllFCToFoun game)

-- given a game, return all possible MoveMult moves
getAllMoveMult :: Game -> [Move]
getAllMoveMult (Game (Cascades ccs) fc fd) =
    concat $
        (parMap rpar) (\x-> getMoveMult (Game (Cascades ccs) fc fd) x)
            (getAllValidStack (Game (Cascades ccs) fc fd))

-- given a game and a stack, return all possible MoveMult
-- ex: getAllMoveMult (Game (Cascades
-- [Cascade [Card Two Spade, Card Ace Heart], Cascade [Card Three Diamond]]) (Freecell [])
-- (Foundation []))
getMoveMult :: Game -> (Int, Stack) -> [Move]
getMoveMult (Game (Cascades _) _ _) (_, Stack []) = error "getMoveMult: empty stack"
getMoveMult (Game (Cascades ccs) _ _) (from,(Stack (card:cards))) =
    do
        (Cascade cc) <- ccs
        let Just index = elemIndex (Cascade cc) ccs
        guard ((length cc > 0 && checkValidNextCard card (last cc)) || (length cc == 0))
        return (MoveMult card (Stack (card:cards)) from index)

-- given a game, return all valid stacks whose size is one less than the free space available
-- ex: getAllValidStack (Game (Cascades
-- [(Cascade [Card Two Spade, Card Ace Diamond])])
-- (Freecell [Card Ace Spade, Card Queen Spade, Card Two Club]) (Foundation []))
getAllValidStack :: Game -> [(Int, Stack)]
getAllValidStack (Game (Cascades ccs) fc fd) =
    if freespace >= 1 then concat $ (parMap rpar) (\(Cascade x) ->
                                    if (length x > 1)
```

```
                                                       then (parMap rpar)
                                                            (\s -> ((getCascadeIndex
                                                            (Cascade x) (Cascades ccs)), s))
                                                            $ (getValidStack (Cascade x) freespace [])
                                                   else [])
                                               ccs
    else []
    where freespace = ((checkFreeSpace (Game (Cascades ccs) fc fd)))


-- given a game and a cascade, return all valid stack assume that limit is
-- greater than 1 and length of cascade is greather than 1
-- ex: getValidStack
-- (Cascade [Card Two Spade, Card Three Diamond, Card King Heart,
-- Card Queen Spade, Card Jack Diamond]) 3 []
getValidStack :: Cascade -> Int -> [Card] -> [Stack]
getValidStack (Cascade cc) limit cards
    | length cc > 1 && length cards - 1 <= limit && checkValidStack ((last cc):cards) =
        getValidStack (Cascade (init cc)) limit ((last cc):cards)
    | length cc == 1 && length cards - 1 <= limit && checkValidStack ((last cc):cards) =
        getValidStack (Cascade []) limit ((last cc):cards)
    | otherwise = if length cards > 1
                    then (parMap rpar) Stack
                                    $ drop 2
                                    $ (parMap rpar) reverse
                                    $ inits (reverse cards)
                  else []


-- given a list of cards, check if they are alternating color and in decending order
checkValidStack :: [Card] -> Bool
checkValidStack [] = error "checkValidStack: should never feed empty"
checkValidStack [_] = True
checkValidStack [c1,c2] = checkValidNextCard c2 c1
checkValidStack cs = (checkValidNextCard (last cs) (last (init cs)))
                     && checkValidStack (init cs)
```

```haskell
-- given a game, calculate number of free spaces to move
checkFreeSpace :: Game -> Int
checkFreeSpace (Game ccs fc _) = checkFreeCascade ccs + checkFreecell fc

-- check if cascades have free cascade
checkFreeCascade :: Cascades -> Int
checkFreeCascade (Cascades []) = 0
checkFreeCascade (Cascades ((Cascade cc):ccs)) =
    if cc == [] then (1 + checkFreeCascade (Cascades ccs))
    else checkFreeCascade (Cascades ccs)

-- check if freecell has free space
checkFreecell :: Freecell -> Int
checkFreecell (Freecell fc)
    | length fc >= 4 = 0
    | otherwise = (4 - length fc)

-- given a game, return all possible MoveFCToFoun
-- ex: getAllFCToFoun (Game (Cascades []) (Freecell [Card Ace Spade]) (Foundation [0,0,0,0])
getAllFCToFoun :: Game -> [Move]
getAllFCToFoun (Game ccs (Freecell fc) fd) =
    concat $ (parMap rpar) (\x -> getFCToFoun (Game ccs (Freecell fc) fd) x) fc

-- given a card (from FC) and a game, return all possible MoveFCToFoun move
getFCToFoun :: Game -> Card -> [Move]
getFCToFoun (Game _ _ (Foundation fd)) (Card rank suit) =
    if ((fd !! index) == (rankIndex)) then [MoveFCToFoun (Card rank suit)] else []
    where Just index = elemIndex suit [minBound..maxBound::Suit]
          Just rankIndex = elemIndex rank [minBound..maxBound::Rank]

-- given a game, return all possible MoveFCToCas moves
-- ex: getAllFCToCas (Game (Cascades [Cascade [Card Two Spade]])
-- (Freecell [Card Ace Diamond]) (Foundation [0,0,0,0]))
getAllFCToCas :: Game -> [Move]
getAllFCToCas (Game ccs (Freecell fc) fd) =
    concat [getFCToCas c (Game ccs (Freecell fc) fd) | c <- fc]


-- given a card (from FC) and a game, return all possible MoveFCToCas move
getFCToCas :: Card -> Game -> [Move]
getFCToCas card (Game (Cascades ccs) _ _) =
    do
        (Cascade cc) <- ccs
        let Just index = elemIndex (Cascade cc) ccs
        guard ((length cc > 0 && checkValidNextCard card (last cc)) || (length cc == 0))
        return (MoveFCToCas card index)


-- given a game, return all possible MoveCasToFC moves
-- ex: getAllCasToFC (Game (Cascades [Cascade [Card Ace Spade],
-- Cascade [Card Two Diamond]]) (Freecell []) (Foundation []))
getAllCasToFC :: Game -> [Move]
getAllCasToFC (Game (Cascades ccs) fc fd) =
    concat $ (parMap rpar) (\(Cascade x) -> if length x > 0 then
                              getCasToFC (last x)
                                          (getCascadeIndex (Cascade x) (Cascades ccs))
                                          (Game (Cascades ccs) fc fd)
                              else [])
              ccs


-- given a card and a game, return all possible MoveCasToFC move
getCasToFC :: Card -> Int -> Game -> [Move]
getCasToFC card casNum (Game _ fc _) =
    if checkFreecell fc > 0 then [MoveCasToFC card casNum] else []


-- given a game, return all possible MoveCasToFoun moves
-- ex: getAllCasToFoun (Game (Cascades [Cascade [Card Ace Spade],
-- Cascade [Card Two Diamond]]) (Freecell []) (Foundation [0,0,0,0]))
getAllCasToFoun :: Game -> [Move]
getAllCasToFoun (Game (Cascades ccs) fc fd) =
    concat $ (parMap rpar) (\(Cascade x) -> if length x > 0 then
                              getCasToFoun (last x)
```

```
                                                (getCascadeIndex (Cascade x) (Cascades ccs))
                                                (Game (Cascades ccs) fc fd)
                                  else [])
                ccs

-- given a card and a game, return all possible CasToFound move
getCasToFoun :: Card -> Int -> Game -> [Move]
getCasToFoun (Card rank suit) casNum (Game _ _ (Foundation fd)) =
    if ((fd !! index) == (rankIndex)) then [(MoveCasToFoun (Card rank suit) casNum)] else []
    where Just index = elemIndex suit [minBound..maxBound::Suit]
          Just rankIndex = elemIndex rank [minBound..maxBound::Rank]

-- given a game, get all possible MoveCasToCas moves
-- ex: getAllCasToCas (Game (Cascades [Cascade [Card Ace Spade],
-- Cascade [Card Two Diamond]]) (Freecell []) (Foundation [0,0,0,0]))
getAllCasToCas :: Game -> [Move]
getAllCasToCas (Game (Cascades ccs) fc fd) =
    concat $ (parMap rpar) (\(Cascade x) -> if length x > 0 then
                                  getCasToCas (last x)
                                              (getCascadeIndex (Cascade x) (Cascades ccs))
                                              (Game (Cascades ccs) fc fd)
                                  else [])
                ccs

-- given a card and a game, return all possible CasToCas move
getCasToCas :: Card -> Int -> Game -> [Move]
getCasToCas card casNum (Game (Cascades ccs) _ _) =
    do
        (Cascade cc) <- ccs
        let Just index = elemIndex (Cascade cc) ccs
        guard (length cc > 0 && checkValidNextCard card (last cc) || (length cc == 0))
        return (MoveCasToCas card casNum index)

-- check if the first card can be placed under the second card
checkValidNextCard :: Card -> Card -> Bool
```

```haskell
checkValidNextCard c1 c2 = (not $ checkSameColor c1 c2) && (checkRightRank c2 c1)

-- check if two cards have the same color
checkSameColor :: Card -> Card -> Bool
checkSameColor (Card _ s1) (Card _ s2)
    =  (s1 `elem` redSuits && s2 `elem` redSuits)
    || (s1 `notElem` redSuits && s2 `notElem` redSuits)
        where redSuits = [Heart, Diamond]

-- check if the first card is one higher than the second
checkRightRank :: Card -> Card -> Bool
checkRightRank (Card r1 _) (Card r2 _) = ri1 == (ri2 + 1)
                                         where rankList = [minBound..maxBound::Rank]
                                               Just ri1 = elemIndex r1 rankList
                                               Just ri2 = elemIndex r2 rankList

-- ----------------------------------------------------------------------------
--                             Applying functions
-- ----------------------------------------------------------------------------

-- given a game and a move, apply the move
applyMove :: Game -> Move -> Game
applyMove game move =
    case move of
        MoveCasToCas _ _ _ -> applyMoveCasToCas game move
        MoveCasToFC _ _ -> applyMoveCasToFC game move
        MoveCasToFoun _ _ -> applyMoveCasToFoun game move
        MoveMult _ _ _ _-> applyMoveMult game move
        MoveFCToFoun _ -> applyMoveFCToFoun game move
        MoveFCToCas _ _ -> applyMoveFCToCas game move

-- given a game and a MoveMult, apply the move
-- ex: applyMoveMult (Game (Cascades [Cascade [Card Two Spade, Card Ace Diamond],
-- Cascade []]) (Freecell []) (Foundation [])) (MoveMult (Card Two Spade)
-- (Stack [Card Two Spade, Card Ace Diamond]) 0 1)
```

```
applyMoveMult :: Game -> Move -> Game
applyMoveMult (Game (Cascades ccs) fc fd) move =
    case move of
        (MoveMult card stack from to) ->
            (Game (Cascades (addStackToCascades
                (removeStackFromCascades ccs 0 from card) 0 to stack)) fc fd)
        _ -> error "Wrong move type"

-- given a game adn a MoveCasToCas, apply the move
applyMoveCasToCas :: Game -> Move -> Game
applyMoveCasToCas (Game (Cascades ccs) fc fd) move =
    case move of
        (MoveCasToCas card from to) -> (Game (Cascades
            (addCardToCascade (removeCardFromCascade ccs 0 from) 0 to card)) fc fd)
        _ -> error "Wrong move type"

-- given a game and a MoveCasToFC, apply the move
applyMoveCasToFC :: Game -> Move -> Game
applyMoveCasToFC (Game (Cascades ccs) (Freecell fc) fd) move =
    case move of
        (MoveCasToFC card from) -> (Game (Cascades
            (removeCardFromCascade ccs 0 from)) (Freecell (card:fc)) fd)
        _ -> error "Wrong move type"

-- given a game and a MoveCasToFoun, apply the move
applyMoveCasToFoun :: Game -> Move -> Game
applyMoveCasToFoun (Game (Cascades ccs) fc (Foundation fd)) (MoveCasToFoun (Card _ suit) from) =
    (Game (Cascades (removeCardFromCascade ccs 0 from)) fc (Foundation newFd))
    where Just suitIndex = elemIndex suit [minBound..maxBound::Suit]
          newFd = (take suitIndex fd) ++ [(fd !! suitIndex) + 1] ++ (drop (suitIndex + 1) fd)

-- given a game and a MoveFCToFoun, apply the move
applyMoveFCToFoun :: Game -> Move -> Game
applyMoveFCToFoun (Game (Cascades ccs) fc (Foundation fd)) (MoveFCToFoun (Card rank suit)) =
    (Game (Cascades ccs) (removeCardFromFC fc (Card rank suit)) (Foundation newFd) )
    where Just suitIndex = elemIndex suit [minBound..maxBound::Suit]
          newFd = (take suitIndex fd) ++ [(fd !! suitIndex) + 1] ++ (drop (suitIndex + 1) fd)

-- given a game and a MoveFCToCas, apply the move
applyMoveFCToCas :: Game -> Move -> Game
applyMoveFCToCas (Game (Cascades ccs) fc fd) move =
    case move of
        (MoveFCToCas card to) -> (Game (Cascades (addCardToCascade ccs 0 to card))
        (removeCardFromFC fc card) fd)
        _ -> error "Wrong move type"

-- ----------------------------------------------------------------------------
--                            Apply Helper functions
-- ----------------------------------------------------------------------------

-- given a cascades, a current index (which should always start with 0),
-- a from index, remove the first card from the cascade
-- ex: removeCardFromCascade [Cascade [Card Ace Spade], Cascade [Card Two Diamond]] 0 1
removeCardFromCascade :: [Cascade] -> Int -> Int -> [Cascade]
removeCardFromCascade [] _ _ = error "removeCardFromCascade: empty cascade"
removeCardFromCascade ((Cascade cc):ccs) cur from
    | cur < from = (Cascade cc):(removeCardFromCascade ccs (cur + 1) from)
    | cur == from = (Cascade (init cc)):ccs
    | otherwise = error "RemoveCardFromCascade error"

-- given a cascades, a current index (which should always start with 0),
-- a to index, and a card, add the card to that cascade
-- ex: addCardToCascade [Cascade [Card Ace Spade], Cascade [Card Two Diamond]] 0 1 (Card Three Heart)
addCardToCascade :: [Cascade] -> Int -> Int -> Card -> [Cascade]
addCardToCascade [] _ _ _ = error "addCardToCascade: empty cascade"
addCardToCascade ((Cascade cc):ccs) cur to card
    | cur < to = (Cascade cc):(addCardToCascade ccs (cur + 1) to card)
    | cur == to = (Cascade (cc ++ [card])):ccs
    | otherwise = error "AddCardToCascade error"
```

```haskell
-- given a freecell, a card, remove the card from the freecell
-- ex: removeCardFromFC (Freecell [Card Ace Spade, Card Two Spade]) (Card Ace Spade)
removeCardFromFC :: Freecell -> Card -> Freecell
removeCardFromFC (Freecell fc) card =
    case index of
        Just i -> (Freecell ((take i fc) ++ (drop (i + 1) fc)))
        Nothing -> error "removeCardFromFC: card was not found"
    where index = elemIndex card fc

-- given a cascades, a current index, a from index, and card
-- remove all the cards from that stack
removeStackFromCascades :: [Cascade] -> Int -> Int -> Card -> [Cascade]
removeStackFromCascades [] _ _ _ = error "removeStackFromCascades: empty cascade"
removeStackFromCascades ((Cascade cc):ccs) cur from card
    | cur < from = (Cascade cc):(removeStackFromCascades ccs (cur + 1) from card)
    | cur == from = (Cascade $ removeStackFromCascade cc card):ccs
    | otherwise = error "removeStackFromCascades: error"

-- given a cascade, a card, return cascade with all cards removed
-- after that card
removeStackFromCascade :: [Card] -> Card -> [Card]
removeStackFromCascade cc card =
    case index of
        Just i -> (take i cc)
        Nothing -> error "removeStackFromCascade: error"
    where index = elemIndex card cc

addStackToCascades :: [Cascade] -> Int -> Int -> Stack -> [Cascade]
addStackToCascades [] _ _ _ = error "addStackToCascades: empty cascade"
addStackToCascades ((Cascade cc):ccs) cur to (Stack cards)
    | cur < to = (Cascade cc):(addStackToCascades ccs (cur + 1) to (Stack cards))
    | cur == to = (Cascade (cc ++ cards)):ccs
    | otherwise = error "addStackToCascades: error"

-- ---------------------------------------------------------------------------
--                              Helper functions
-- ------------------------------------------------------------------------------


-- break a list into a list of lists with individual element in a list
breakListofList :: [a] -> [[a]]
breakListofList a
    | length a == 0 = []
    | otherwise = [head a] : breakListofList (tail a)

-- for printing: combine two list of lists into one list of lists by merging at each index
combineListofListsForPrint :: [[String]] -> [[String]] -> [[String]]
combineListofListsForPrint a b
    | length a == 0 && length b == 0 = []
    | length a == 0 = (["  "] ++ head b):(combineListofListsForPrint [] (tail b))
    | length b == 0 = (head a ++ ["  "]):(combineListofListsForPrint (tail a) [])
    | otherwise = (head a ++ head b):(combineListofListsForPrint (tail a) (tail b))

-- combine two list of lists into one
combineListofLists :: [[String]] -> [[String]] -> [[String]]
combineListofLists a b
    | length a == 0 = b
    | length b == 0 = a
    | otherwise = (head a ++ head b):(combineListofLists (tail a) (tail b))

-- get the index of a cascade
getCascadeIndex :: Cascade -> Cascades -> Int
getCascadeIndex cc (Cascades ccs) =
    case index of
        Just i -> i
        Nothing -> error "Should never put in non-existing cascade"
    where index = elemIndex cc ccs

-- get the max length of current cascades
getMaxLengthCascasde :: Cascades -> Int -> Int
getMaxLengthCascasde (Cascades []) maxLength = maxLength
getMaxLengthCascasde (Cascades ((Cascade cc):ccs)) maxLength
    | length cc > maxLength = getMaxLengthCascasde (Cascades ccs) (length cc)
    | otherwise = getMaxLengthCascasde (Cascades ccs) maxLength

-- check if all the cascades are empty
checkAllCascadeEmpty :: Cascades -> Bool
checkAllCascadeEmpty (Cascades []) = True
checkAllCascadeEmpty (Cascades ((Cascade cc):ccs))
    | length cc == 0 = checkAllCascadeEmpty (Cascades ccs)
    | otherwise = False
```

5.3 gameController.hs

```
module GameController where
import Game
import Data.List (elem)

{-

gameController.hs
by Joe Huang

4995 Final Project

Descriptions:
The Game Controller keeps track of the current state of the game,
all the possible move a player can do, and the previous game states.
When there are any card that could be move to the foundation, it
will be moved automatically.

The purpose of this controller is to ensure that no invalid move
can be applied to the game and only the given ones can be used.

-}

data MoveStatus = Success | Fail | Win
```

```
    deriving (Show, Eq, Ord)


data GameStatus = GameStatus {curGame :: Game, curMoves :: [Move], madeMove :: [Move]}
    deriving (Eq, Ord)


instance Show GameStatus where
    show (GameStatus game moves _) = unlines [show game,
            "All Possible Moves: " ++ show moves, " " ] --, "Previous Moves: " ++ show state]


-- given a game number, return the initial game status
start :: Int -> GameStatus
start gameNumber =
    (checkFounMove ((GameStatus newGame newMoves []), newMoves))
    where newGame = initializeGame gameNumber
          newMoves = getAllMove newGame


-- given a game status and a move, apply the move and return the move status
-- and the game status after the move.
makeMove :: GameStatus -> Move -> (MoveStatus, GameStatus)
makeMove (GameStatus game moves state) move =
    if (not (elem move moves)) then (Fail, (GameStatus game moves state))
    else
         if (isFinished nng) then (Win, (GameStatus nng nnm nns))
         else (Success, (GameStatus nng nnm nns))
    where newGame = applyMove game move
          (GameStatus ng nm ns) = (GameStatus newGame (getAllMove newGame) (move:state))
          (GameStatus nng nnm nns) = (checkFounMove ((GameStatus ng nm ns), nm))


-- given a game status and its moves, recurse until all possible foundation moves
-- are applied
checkFounMove :: (GameStatus, [Move]) -> GameStatus
checkFounMove ((GameStatus game moves state), movesList)
    | length movesList == 0 = (GameStatus game moves state)
    | otherwise =
        case (head movesList) of
```

```
            MoveCasToFoun _ _->

                checkFounMove (makeFounMove (GameStatus game moves state) (head movesList))

            MoveFCToFoun _ ->

                checkFounMove (makeFounMove (GameStatus game moves state) (head movesList))

            _ -> checkFounMove ((GameStatus game moves state), (tail movesList))


-- given a game status, a foundation move, return the new game status

-- and its respective moves

makeFounMove :: GameStatus -> Move -> (GameStatus, [Move])

makeFounMove (GameStatus game moves state) move =

    if valid == Success || valid == Win then (newGame, newMoves)

        else error "makeFounMove: invalid move"

    where (valid, newGame) = makeMove (GameStatus game moves state) move

          (GameStatus _ newMoves _) = newGame
```

5.4 solver.hs

```
import GameController

import Game

import Data.Maybe (fromJust)

import Data.Map (Map)

import Data.List (elem, elemIndex, sort)

import qualified Data.Map as Map

import Control.Parallel.Strategies (parMap, rpar)


{-


solver.hs

by Joe Huang


4995 Final Project


Descriptions:

The solver implements the Heineman's Staged Deepening Heusirtic (HSDH) to

find a solution for a freecell game. HSDH first find all the states k steps

away from the current one and rank them based on the heusirtic. The
heusirtic checks the foundation's current cards and finds all the card that
are supposed to be placed next. For those next cards, calculate how many
cards are on top of them. The heusirtic mulplies that score by two if
all freecell are used or any foundation cell is 0. Then, the best state are
used for the next iteration until a solution is found.

References:
FreeCell Solitaire Optimization:
http://people.uncw.edu/tagliarinig/Courses/380/S2015%20papers%20and%20presentations/Freecell%20Opt-Beas

-}

boundedSearch :: GameStatus -> Int -> Map GameStatus Int ->[GameStatus]
boundedSearch (GameStatus game moves states) bound seenGS
    | bound == 0 = newGS
    | otherwise = concat $ (parMap rpar) (\x-> boundedSearch x (bound -1) newSeenGS) newGS
    where newGS = (parMap rpar) (\(_, status) -> status) $
                   filter (\(_, status)-> Map.notMember status seenGS) $
                   (parMap rpar) (\x -> makeMove (GameStatus game moves states) x) moves
          newSeenGS = Map.union seenGS $ Map.fromList ((parMap rpar) (\x->(x,1)) newGS)

-- Heineman's Staged Deepening Heusirtic (HSDH), get the foundation score and multiple
-- if no available freecell or empty
hSDH :: [GameStatus] -> GameStatus
hSDH [] = error "Empty GameStatus List"
hSDH gss = snd
           $ head
           $ sort
           $ map (\(GameStatus (Game ccs fc fd) moves gs) ->
                   ((calculateAllFounScore fd ccs) * (penaltyScore fc fd),
                    (GameStatus (Game ccs fc fd) moves gs)))
             gss

-- given a freecell and a foundation, calculate the penalty score
penaltyScore :: Freecell -> Foundation -> Int
penaltyScore (Freecell fc) (Foundation fd) =
    if (length fc >= 4 || elem 0 fd) then 2 else 1

-- gievn a game, calculate the sum of all four foundation next card score
-- ex: calculateAllFounScore (Foundation [0,0,0,0])
-- (Cascades [Cascade [Card Ace Spade, Card Two Diamond], Cascade [Card Ace Diamond],
-- Cascade [Card Ace Club], Cascade [Card Ace Heart]])
calculateAllFounScore :: Foundation -> Cascades -> Int
calculateAllFounScore (Foundation fd) (Cascades ccs) =
    sum $ map (\(Card r s) -> if (r == King) then 0
                              else calculateFounScore (Card r s) (Cascades ccs))
          nextCards
    where nextCards = getFoundationNextCards (Foundation fd)

-- given a foundation and a cascades, find the cascade that holds the card and return
-- the number of card on top of it
-- ex: calculateFounScore (Card Three Spade) (Cascades [Cascade [Card Ace Spade, Card Three Spade]])
calculateFounScore :: Card -> Cascades -> Int
calculateFounScore _ (Cascades []) = error "Shouldn't happen given a valid game"
calculateFounScore card (Cascades ((Cascade cc):ccs)) =
    if (elem card cc) then (length cc - fromJust (elemIndex card cc) - 1)
    else calculateFounScore card (Cascades ccs)

-- given a foundation, return a list of card one higher than the current one
getFoundationNextCards :: Foundation -> [Card]
getFoundationNextCards (Foundation fd) =
    joinCards suits ranks
    where suits = [minBound..maxBound::Suit]
          ranks = map (\x-> if x >= 13 then ([minBound..maxBound::Rank]) !! 12
                            else ([minBound..maxBound::Rank]) !! x) fd

-- join a list of suits and ranks together as a list of cards
joinCards :: [Suit] -> [Rank] -> [Card]
joinCards [] [] = []
joinCards [] (_:_) = error "joinCards: length not equal"
joinCards (_:_) [] = error "joinCards: length not equal"

joinCards (s:suits) (r:ranks) = (Card r s):(joinCards suits ranks)


-- the main funciton to run the solver

main :: IO ()

main = do putStrLn "Please enter a game number: "

          input <- getLine

          let gameNum = (read input :: Int)

          let game = start gameNum

          let (GameStatus _ _ states) = loop game

          print $ reverse states


-- the loop to keep running the solver until it is solved

loop :: GameStatus -> GameStatus

loop (GameStatus oldGame oldMove oldState) =

    if game == oldGame then error "Didn't improve...Something is wrong"

    else

        if (isFinished game) then (GameStatus game move state)

            else loop (GameStatus game move state)

    where (GameStatus game move state) = hSDH

        $ boundedSearch (GameStatus oldGame oldMove oldState) 5 (Map.empty)
```