
Gaby Haddock Reilly
Haskell final project
Also posted at : https://github.com/gabyhaddock/Submarine

This Haskell program is a pathfinder for the board game Red November. In the Red November board game, each player controls a gnome trapped on a submarine that is sinking, overheating, catching fire, flooding, and in general trying to kill you and your comrades. On your turn, you can move around the ship which has 10 numbered rooms by opening hatches and moving into the attached rooms. Opening hatches and moving rooms will cost you time, and since you hope to achieve your goals in the shortest posssible time, you want to minimize the time you spend moving to your target.

Occasionally, a hatch will become blocked during the game, and you cannot open a hatch that is blocked. In addition, you cannot enter a room that is on fire or flooded to high water. This may mean that you have to take an indirect path to reach your target room. Also, you cannot leave a room that is on fire, so you may not be able to move at all.

In addition, your movement may affect the state of the ship. If you open a hatch between two rooms, and one room is flooded to high water, it will flow into the other room as long as there is no water already there. When the attached room floods to Low Flood levels, the original room that was flooded to high water will also reduce its water level to Low Flood. Then, the you can walk through one of the Low Flooded rooms-- for an additional minute of your time. By thoughtfully flooding rooms, you may be able to create a more direct path to your target room or even put out a fire along the way.

This program will use a representation of the submarine's rooms and hatches and show you paths to each of the rooms of the submarine from your current room. If two paths to the same room have the exact same side effects on the ship, the program will only recommend the shortest path. Since the hatches re-close themselves at the end of your turn, the state of the hatches is ignored, and the state of the rooms is the only side effect that matters for this evaluation. If you are given the opportunity to open a hatch to flood a room, the program will show you that option, even if it takes more time than a path that does not flood anything. This means that the player still has the opportunity to evaluate their options and make an interesting choice about what kind of movement they will make on their turn.

Here is a summary of the basic movement rules from Red November:

* You cannot leave a room that is on Fire
* You cannot enter a room that is on Fire, or flooded to High Flood
* You can enter a room that is flooded to Low Flood, and the movement costs 1 minute
* You can enter a room that is Clear, and the movement costs 0 minutes

* You can open any hatch that is currently Closed, but not a hatch that is Blocked
* Opening a hatch costs 1 minute
* After a hatch is opened, if one room is flooded to High Flood and the other is Clear or on Fire, both rooms end at Low Flood. This movement costs 0 minutes. (If the attached room is at High Flood or Low Flood, nothing happens).
* You can open a hatch in a room without moving through it

```
> import Data.Char
> import Data.List
> import Data.List.Split
```

We start by defining the Room data type, which represents a room and its state. Helpfully, the board game numbers each room from 1 to 10, so we will capture that in the Room data type along with its state - Clear, Low Flood, High Flood, or Fire.

```
> data Room = Room Int RoomState deriving Show

> instance Ord Room where
>     compare (Room x1 s1) (Room x2 s2) | x1 == x2  = compare s1 s2 -- If room nums are
equal, compare states
>                                       | otherwise = compare x1 x2 -- Otherwise order by
room number


> instance Eq Room where
>     r1 == r2 = (compare r1 r2) == EQ

> data RoomState = Clear | LowFlood | HighFlood | Fire  deriving (Show, Eq, Ord)
```

I set Ord and Eq instances on the Room type to facilitate analysis of the submarine, which is described later in the report. When two rooms are compaerd, first the room numbers are compared, and if they are equal, the RoomStates are compared. I use the order of RoomStates as they are written in the definition rather than defining a custom Ord.

Testing:

```
*Main> (Room 1 Fire) < (Room 2 LowFlood)
True
(0.02 secs, 517308 bytes)
*Main> (Room 1 LowFlood) < (Room 1 Fire)
True
(0.00 secs, 517308 bytes)
*Main> (Room 1 LowFlood) == (Room 1 LowFlood)
True
```

We can see that this implementation allows us to compare rooms for Ord and Eq.

Similar to the structure of the Room data type, the Hatch data type will describe its adjacent room numbers with a pair of Ints, along with its state - Open, Closed, or Blocked. For example, a closed hatch that connects rooms 1 and 2 will be Hatch (1,2) Closed.

```
> data Hatch =  Hatch (Int, Int) HatchState deriving (Eq, Show)
> data HatchState = Open | Closed | Blocked  deriving (Show, Eq)
```

Since I am using a pair to represent the adjacent rooms, the order within the pair is relevant. For the specific submarine used in the board game, I use a standard notation of listing the smaller room number as the first element of the pair, and the larger room

number as the second element. However, the logic that follows will work with any hatch format, as long as a single hatch is not duplicated.

To help the player visualize and understand their turn, we will keep a list of actions that the program has completed during each turn. These actions will describe the StartRoom that the player starts in, an OpenHatch action that opens a Closed hatch, a Move action where the player moves through an open hatch into a new Room, and the Flood side effects when an open hatch causes water to flow.

For Move and OpenHatch actions we will also record the cost for each action.

```
> --Represent the various actions that can be taken during the player's move
> data Action = StartRoom Room |
>               OpenHatch Hatch Int |
>               Move Room Int  |
>               Flood Room Room  deriving Show -- Int is Cost
```

The GameState data type will describe the current state of the submarine, in terms of rooms and hatches, after all the listed actions have been completed.

```
> data GameState = GameState {
>                              rooms::[Room],
>                              hatches::[Hatch],
>                              actions:: [Action]
>                            }
```

This custom Show instance increases readability during debugging by adding newlines between the lists. As an added bonus, the GameState can be pasted directly back into the interpreter if needed.

```
> instance Show GameState where
>     show gst  = "GameState { rooms= " ++ show (rooms gst) ++ " , \n" ++
>                 "  hatches = " ++ show (hatches gst) ++ " , \n" ++
>                 "  actions = " ++ show (actions gst) ++ " } \n"
```

Capturing the game state as actions are completed may initially seem like a perfect use case for a monad. However, in this case, I actually intend to generate all possible GameStates from an initial value, then compare and evaluate the GameStates against each other to choose the best actions. Given the final results that I hope to output, I chose to keep the GameState object as a standard data type.

```
--------------------------------
Moving from one room to another
--------------------------------

> --Helper functions for the Room object

> roomNum               :: Room -> Int
> roomNum (Room num _ ) = num

> roomState                :: Room -> RoomState
> roomState (Room _ state) = state
```

```
> -- Get a room by its number from a list of rooms.  Used in adjacentRooms, and
construction of the initial submarine from an input file.
> getRoomByNum           :: Int -> [Room] -> Room
> getRoomByNum num rooms = case matches of
>                            [] -> error ("Could not find room number " ++ show num)
>                            otherwise -> head matches
>            where matches = filter (\r -> roomNum r == num) rooms
```

With these helper functions, we can start to write functions that represent the rules of the game.

Given a Room and a GameState, the adjacentRooms function will return a list of adjacent rooms. It will only list rooms that are connected by an Open hatch.

```
> -- Hatches have a pair of Ints that represent the connected rooms.  The given room's
number could be the first element, or the second.
> adjacentRooms                 :: Room -> GameState -> [Room]
> adjacentRooms (Room n _ ) gst = [ getRoomByNum roomNum allRooms | roomNum <- adjRoomNums
]
>         where adjRoomNums = [ x | (Hatch (x, y) state) <- (hatches gst), y == n, state ==
Open]
>                           ++ [ y | (Hatch (x, y) state) <- (hatches gst), x == n, state ==
Open]
>               allRooms = rooms gst
```

The game rules state that a room cannot be entered if its current state is HighFlood or Fire.

```
> canEnterRoom                    :: Room -> Bool
> canEnterRoom (Room _ HighFlood) = False
> canEnterRoom (Room _ Fire)      = False
> canEnterRoom  _                 = True
```

In the course of all your Actions, you should not be able to move into a room that you already visited. This prevents infinite cycles in our program's graph traversal - and there is no in-game reason to visit a room twice. To enforce this, we need to be able to pull the list of visited rooms from the list of Actions in the game state.

The Actions list may include a variety of action types, but only the StartRoom and Move actions imply that a Room was visited. The getRoomFromAction function gets the Just Room from those relevant actions, and Nothing from the rest.

```
> -- Helper for finding visited rooms based on action list
> getRoomFromAction                 :: Action -> Maybe Room
> getRoomFromAction (StartRoom room) = Just room
> getRoomFromAction (Move room _)    = Just room
> getRoomFromAction (OpenHatch _ _ ) = Nothing
> getRoomFromAction (Flood _ _)      = Nothing
```

To get a list of visited Rooms from a gamestate, we will use the visitedRooms function:

```
> -- Takes in the game state (whose actions may include Moves, Floods, StartStates, and
HatchOpen actions) and returns all the visited Rooms
> -- Note: these are in reverse order of the actual visits
> visitedRooms :: GameState -> [Room]
> visitedRooms gst | null visited =  error "The action list must include at least the
starting room"
>                  | otherwise    = visited
>                  where  maybeVisited  = filter isRoom (map getRoomFromAction  (actions
gst))
>                         visited       = map (\(Just room) -> room)  maybeVisited
>                   -- Get the rooms out of the Move and StartRoom actions, and filter out
the other actions that returned Nothing.
>                         isRoom (Just room) = True
>                         isRoom Nothing     = False
```

And, to determine whether some room can be the target of our next Move, we have a simple isVisited function to describe whether a room's number has been visited or not.

```
> -- This is more than just `elem` visitedRooms -- the game state may have changed since it
was visited, so we need to compare number and ignore state
> isVisited                  :: Room -> GameState -> Bool
> isVisited (Room num _) gst  = num `elem` (map roomNum (visitedRooms gst))
```

The order of actions in the GameState is important. The starting room is the last element of the Actions list, and the final (or current) room is the head of the actions list. When we add a new action to the list, we will add it to the head of the list.

```
> currentRoom       :: GameState -> Room
> currentRoom gst   = head (visitedRooms gst)

> startRoom         :: GameState -> Room
> startRoom gst     = last (visitedRooms gst)
```

Now that we have described the in-game rules, we are ready to ask the program to make a valid move.

The moveRooms function will take in a single GameState and return all possible GameStates that can follow from a single move

The movement obeys the following game rules:
  * The new room must be adjacent and reachable through a single Open hatch.
  * The new room must have not been visited so far in this game state
  * The new room's state must not be HighFlood or Fire
  * The player may not leave a room that is on Fire
  * Moving into a LowFlood room costs 1 minute, and otherwise costs 0 minutes.

```
> moveRooms :: GameState -> [GameState]
> moveRooms gst  =  [ gst{actions = ((Move newRoom  (costToEnter newRoom)) :(actions gst))
} --Add the new room to the head of the actions list
>                                         | newRoom <- adjRooms,
>                                           not (isVisited newRoom gst),
>                                           canEnterRoom newRoom,
```

```
>                                        roomState currRoom /= Fire] -- Special in-game
rule: the player may not leave a room on fire
>        where currRoom    = currentRoom gst
>              adjRooms    = adjacentRooms currRoom gst
>              costToEnter (Room _ LowFlood) = 1
>              costToEnter _                 = 0
```

We can test the moveRooms function on the following sample submarine:

```
> openMini = GameState {
>                   rooms = [
>                           (Room 1 Clear),
>                           (Room 2 LowFlood),
>                           (Room 3 Clear),
>                           (Room 4 Fire)
>                           ],
>                   hatches =  [
>                           (Hatch (1,2) Open),
>                           (Hatch (1,3) Open),
>                           (Hatch (1,4) Open),
>                           (Hatch (2,3) Open)
>                           ],
>                   actions = [StartRoom (Room 1 Clear)]
>                      }
```

The player starts in Room 1, and with one movement, we would expect that they would end up in either Room 2 or Room 3. We should not expect to see a movement into Room 4, since it is currently on fire.

```
*Main> moveRooms openMini
[GameState { rooms= [Room 1 Clear,Room 2 LowFlood,Room 3 Clear,Room 4 Fire] ,
  hatches = [Hatch (1,2) Open,Hatch (1,3) Open,Hatch (1,4) Open,Hatch (2,3) Open] ,
  actions = [Move (Room 2 LowFlood) 1,StartRoom (Room 1 Clear)] }
,GameState { rooms= [Room 1 Clear,Room 2 LowFlood,Room 3 Clear,Room 4 Fire] ,
  hatches = [Hatch (1,2) Open,Hatch (1,3) Open,Hatch (1,4) Open,Hatch (2,3) Open] ,
  actions = [Move (Room 3 Clear) 0,StartRoom (Room 1 Clear)] }
]
```

There are two GameStates returned in the list. The first one shows the movement to Room 2 with a cost of 1, due to the LowFlood, and the second shows a movement to Room 3. As we expected, the moveRooms function does not allow a movement into Room 4.

The moveRooms function shows the possible GameStates with *exactly* one more move. To take several moves, we need to apply moveRooms several times.

```
*Main> concat (map moveRooms (moveRooms openMini))
[GameState { rooms= [Room 1 Clear,Room 2 LowFlood,Room 3 Clear,Room 4 Fire] ,
  hatches = [Hatch (1,2) Open,Hatch (1,3) Open,Hatch (1,4) Open,Hatch (2,3) Open] ,
  actions = [Move (Room 3 Clear) 0,Move (Room 2 LowFlood) 1,StartRoom (Room 1 Clear)] }
,GameState { rooms= [Room 1 Clear,Room 2 LowFlood,Room 3 Clear,Room 4 Fire] ,
```

```
  hatches = [Hatch (1,2) Open,Hatch (1,3) Open,Hatch (1,4) Open,Hatch (2,3) Open] ,
  actions = [Move (Room 2 LowFlood) 1,Move (Room 3 Clear) 0,StartRoom (Room 1 Clear)] }
]
```

Applying moveRooms twice shows all possible GameStates with exactly two moves. We can see that the first GameState moved from Room 1 to Room 2 to Room 3, and the second moved from Room 1 to Room 3 to Room 2.

```
*Main> concat (map moveRooms (concat (map moveRooms (moveRooms openMini))))
[]
```

Applying moveRooms three times shows us that there are no GameStates that make exactly three moves. This is because after two moves, all three of the visitable rooms have already been visited.

By iteratively applying moveRooms, we can perform a breadth-first traversal of the submarine, starting from the initial room and only making valid moves.

Opening hatches

The moveRooms function allows the player to move through Open hatches, but in the game, each turn starts with every hatch Closed. So, in order to leave a room, the player must first open at least one of the hatches attached to that room.

Additionally, opening a hatch may have side effects on the adjoining rooms if one of them is flooded to High Flood. Because of this, opening a hatch will not only add an OpenHatch action to the actions list of the game state, but it will also modify the game state's hatches list, and possibly the rooms list.

```
> --Helper function
> hatchRooms                      :: Hatch -> (Int, Int)
> hatchRooms (Hatch rooms state) = rooms

> -- Given a room, return a list of all connecting hatches that are currently closed
> -- Opened hatches should not be re-opened, and blocked hatches should not be opened.
> adjacentClosedHatches                  :: Room -> GameState -> [Hatch]
> adjacentClosedHatches (Room n _ ) gst  =   [ (Hatch (x, y) state)
>                                            | (Hatch (x, y) state) <- (hatches gst),
>                                              (x == n ||y == n),
>                                              state == Closed]
```

Opening a hatch has side effects on the Hatches list, and may also affect the Rooms list. These helper functions modify the state of Hatches or Rooms in a list.

```
> -- Pass in a hatch to Open and look for it in the existing hatch list.  If found, replace its state with Open
```

```
> -- If Hatch is not found, it is NOT added. Nothing happens to the hatch list.
> -- This preserves the order of the hatch list.
> -- Also note that the order of the room pair (x,y) matters.
> openHatch                          :: Hatch -> [Hatch] -> [Hatch]
> openHatch (Hatch rooms _) hatches  = map (\oldHatch -> if hatchRooms oldHatch == rooms
>                                                        then (Hatch rooms Open)
>                                                        else oldHatch)
>                                          hatches


> -- Pass in a Room to flood and look for its number in the existing room list.  If found,
replace its state with LowFlood.
> -- If the Room is not found, nothing happens to the room list.  This preserves the order
of the room list.
> floodRoom                    :: Room -> [Room] -> [Room]
> floodRoom (Room num _) rooms  = map (\oldRoom -> if roomNum oldRoom == num
>                                                  then (Room num LowFlood)
>                                                  else oldRoom)
>                                     rooms
```

Given a hatch (assumed already opened) and the original list of rooms, the roomsToFlood function will first check the initial states of the two adjacent rooms

* If one is at HighFlood and the other is Clear or Fire, the rooms equalize to LowFlood.
* The roomsToFlood function returns Nothing if we do not need to flow, and Maybe (Room, Room) if the rooms should be flooded

```
> -- Note: The roomsToFlood function will error out if the Room list does not contain both
numbers listed in the Hatch.
> roomsToFlood                    :: Hatch -> [Room] -> Maybe (Room, Room)
> roomsToFlood openedHatch rooms =
>              if (oldFirstState == HighFlood && (oldSecondState == Clear || oldSecondState
== Fire))
>                 || (oldSecondState == HighFlood && (oldFirstState == Clear ||
oldFirstState == Fire))
>              then Just (oldFirstRoom, oldSecondRoom)
>              else Nothing
>              where firstRoomNum = fst (hatchRooms openedHatch)
>                    secondRoomNum = snd (hatchRooms openedHatch)
>                    oldFirstRoom = head (filter (\x -> roomNum x == firstRoomNum) rooms)
>                    oldSecondRoom = head (filter (\x -> roomNum x == secondRoomNum) rooms)
>                    oldFirstState = roomState oldFirstRoom
>                    oldSecondState = roomState oldSecondRoom
```

For a given GameState, openHatches shows all possible GameStates that can follow from opening a single hatch
Obeys the following rules:

* Only Closed hatches can be opened, not Open or Blocked hatches
* The two adjacent rooms can flow water between them
* The rooms and hatches list of the Sub are modified as needed.

```
> openHatches       :: GameState -> [GameState]
> openHatches gst    = [ gst {
>                                  rooms = newRooms,
>                                  hatches = newHatches, -- Replace the closed hatch in
the list with the opened hatch
>                                  actions = newActions
>                                 }
>                        | openedHatch <- adjHatches,
>                         let floodRooms = roomsToFlood openedHatch (rooms gst),
> --If rooms were flooded, update the rooms list to show the side effects. roomsToFlood
returns Maybe (Room, Room) of rooms to flood.
>                         let newRooms = case floodRooms of Nothing -> rooms gst
>                                                           Just (f, s) -> ( floodRoom f .
floodRoom s ) (rooms gst),
>                         let newHatches = openHatch openedHatch (hatches gst),
> --In all cases, add a 1-minute OpenHatch action
>                         let baseActions = (OpenHatch openedHatch 1) : (actions gst),
> --In the case that rooms were flooded, add an additional special Flood action to indicate
which rooms were flooded
>                         let newActions = case floodRooms of Nothing ->  baseActions
>                                                             Just (f, s) -> (Flood f s) :
baseActions
>                       ]
>            where currRoom    = currentRoom gst
>                  adjHatches  = adjacentClosedHatches currRoom gst
```

To test the openHatches result, we will use a sample submarine that has closed hatches and water that can flow from room to room.

```
> miniFlood = GameState  { rooms =
>                [ Room 1 Clear,
>                  Room 2 HighFlood,
>                  Room 3 Fire,
>                  Room 4 LowFlood]
>             , hatches =
>                [ Hatch (1,2) Closed,
>                  Hatch (1,3) Closed,
>                  Hatch (2,3) Closed,
>                  Hatch (2,4) Closed,
>                  Hatch (3,4) Closed]
>             , actions = [StartRoom (Room 2 Clear) ]
>                 }

*Main> openHatches miniFlood
[GameState { rooms= [Room 1 LowFlood,Room 2 LowFlood,Room 3 Fire,Room 4 LowFlood] ,
  hatches = [Hatch (1,2) Open,Hatch (1,3) Closed,Hatch (2,3) Closed,Hatch (2,4)
Closed,Hatch (3,4) Closed] ,
  actions = [Flood (Room 1 Clear) (Room 2 HighFlood),OpenHatch (Hatch (1,2) Closed)
1,StartRoom (Room 2 Clear)] }
,GameState { rooms= [Room 1 Clear,Room 2 LowFlood,Room 3 LowFlood,Room 4 LowFlood] ,
  hatches = [Hatch (1,2) Closed,Hatch (1,3) Closed,Hatch (2,3) Open,Hatch (2,4)
Closed,Hatch (3,4) Closed] ,
  actions = [Flood (Room 2 HighFlood) (Room 3 Fire),OpenHatch (Hatch (2,3) Closed)
```

```
1,StartRoom (Room 2 Clear)] }
,GameState { rooms= [Room 1 Clear,Room 2 HighFlood,Room 3 Fire,Room 4 LowFlood] ,
  hatches = [Hatch (1,2) Closed,Hatch (1,3) Closed,Hatch (2,3) Closed,Hatch (2,4)
Open,Hatch (3,4) Closed] ,
  actions = [OpenHatch (Hatch (2,4) Closed) 1,StartRoom (Room 2 Clear)] }
]
```

Calling openHatches on the miniFlood sub shows each of the hatches that can be opened from room 2, including the side effects on the submarine. When we open a hatch to room 1 or 3, the HighFlood in room 2 flows into the adjacent room and both rooms end at LowFlood. When the hatch to room 4 is opened, the flooding does not happen because, according to game rules, water cannot flow into a room that is already at LowFlood.

Similar to the mechanism for moveRooms, the openHatches function can be called repeatedly on the Game State to incrementally move one action further.

```
*Main> length (concat (map openHatches (openHatches miniFlood)))
6
*Main> length (concat (map openHatches (concat (map openHatches (openHatches miniFlood)))))
6
*Main> length (concat (map openHatches (concat (map openHatches (concat (map openHatches
(openHatches miniFlood)))))))
0
```

Two applications of openHatches gives six results, as does three applications of the function. When we attempt to make the fourth OpenHatch action, we find that there are no more results available. This is because room 2 has three connected hatches, and after three moves, all of them are open.

Coordinating functions for playing the game

By combining the results for moveRooms and openHatches, we can take all possible moves that include one of those valid actions.

```
> -- Take one turn, including all possible Move actions and all possible OpenHatch actions
for each GameState that is passed in.
> takeTurn :: [GameState] -> [GameState]
> takeTurn gameState =  (concat . (map moveRooms)) gameState
>                       ++ (concat . (map openHatches)) gameState


> --Get all moves, regardless of depth.  This terminates when a row in the grid has no
results (no additional moves from the previous row)
> allMoves :: [GameState] -> [GameState]
> allMoves gameState = concat $ takeWhile (not . null) (iterate takeTurn gameState)
```

The allMoves function terminates when the takeTurn function returns no new GameStates. Since we are not allowing cycles, there must be some longest sequence of actions that eventually has no more valid options.

```
> starterSub = GameState {rooms = [Room 1 Clear,
>                                  Room 2 Clear,
>                                  Room 3 Clear,
>                                  Room 4 Clear,
>                                  Room 5 Clear,
>                                  Room 6 Clear,
>                                  Room 7 Clear,
>                                  Room 8 Clear,
>                                  Room 9 Clear,
>                                  Room 10 Clear],
>                          hatches = [Hatch (1,2) Closed,
>                                     Hatch (1,3) Closed,
>                                     Hatch (2,3) Closed,
>                                     Hatch (2,4) Closed,
>                                     Hatch (2,5) Closed,
>                                     Hatch (3,4) Closed,
>                                     Hatch (4,5) Closed,
>                                     Hatch (5,6) Closed,
>                                     Hatch (5,7) Closed,
>                                     Hatch (5,8) Closed,
>                                     Hatch (7,8) Closed,
>                                     Hatch (7,9) Closed,
>                                     Hatch (8,9) Closed,
>                                     Hatch (8,10) Closed,
>                                     Hatch (9,10) Closed],
>                          actions = [StartRoom (Room 1 Clear)]
>                         }
```

The performance of allMoves is great on the 4-room subs like miniOpen and miniFlood, but falls way behind on the 10-room sub starterSub. To explore this, i took the first 10 or 15 steps and observed how long it took GHCI to run the iterations. I also used the length function to see how many GameStates were actually being produced at each step.

```
*Main> length $ (iterate takeTurn [starterSub])!!10
8596
(0.53 secs, 46052252 bytes)
*Main> length $ (iterate takeTurn [starterSub])!!15
357564
(37.17 secs, 3245943772 bytes)
```

On a full-size sub, taking 10 steps resulted in 8,596 possible GameStates in .5 seconds. Taking 5 more steps resulted in 357,000 more GameStates but took a whopping 37 seconds to complete. And since the 15th iteration still returned results, a call to allMoves on a full-size sub would keep going beyond 15 steps.

Game state analysis

We clearly don't want all 300,000 options for a 10-room sub. Some of those options include objectively terrible decisions. There may be some cases where you open a hatch, then walk through a different one, but that only makes sense if the "useless" hatch actually participated in a flood action. In all other cases, you should only open a hatch if you immediately walk through it.

We can say that two GameStates are equivalent if they have an equal list of Rooms (including RoomStates) and the player is in the same currentRoom. If we partition the results into groups where all the GameStates have equivalent rooms, we can choose the GameState that has the lowest cost and discard the rest. This partition will distinguish between two gamestates that end in the same room, but where a Flood action has occurred in one of them, because the Rooms list will not be equal. In this way, we can display both options to the player and allow them to choose whether it is worth taking a longer path to their target room, if it allows them to perform a flood along the way. See Appendix I for a visual example of the choice that the player will make from the result set.

```
> -- Helper functions
> -- Find the total cost of a GameState (used for final output of results)
> actionCost                    :: Action -> Int
> actionCost (Move _ cost)      = cost
> actionCost (OpenHatch _ cost) = cost
> actionCost _                  = 0

> totalCost                      :: GameState -> Int
> totalCost gst = sum (map actionCost (actions gst))

> -- Ordering and equality functions to order & group by various aspects of the GameState
> orderByCost         :: GameState -> GameState -> Ordering
> orderByCost gs1 gs2 = compare (totalCost gs1) (totalCost gs2)

> orderByRooms         :: GameState -> GameState -> Ordering
> orderByRooms gs1 gs2 | currRoom1 < currRoom2  = LT -- Use Ord for rooms
>                      | currRoom1 > currRoom2  = GT
>                      | otherwise              = compare (rooms gs1) (rooms gs2)
>     where currRoom1 = currentRoom gs1
>           currRoom2 = currentRoom gs2

> equalByRooms         :: GameState -> GameState -> Bool
> equalByRooms gs1 gs2 = (orderByRooms gs1 gs2) == EQ
```

The prune function Given a list of possible game states, group into "equivalent" game states (same final room, same state of all rooms) and then chooses the lowest cost game state from each equivalent group by sorting and keeping the head.

```
> -- Note that haskell groupBy only groups adjacent elements, so I need to sortBy
orderByRooms first
> prune    :: [GameState] -> [GameState]
> prune gst = map (head . sortBy orderByCost)
>             (groupBy equalByRooms (sortBy orderByRooms   gst))
```

For the openMini sub, the allMoves function returns 5 possible states, but we can see that

only 3 of them are optimal after pruning out the inefficient results.

```
*Main> openMini
GameState { rooms= [Room 1 Clear,Room 2 LowFlood,Room 3 Clear,Room 4 Fire] ,
  hatches = [Hatch (1,2) Open,Hatch (1,3) Open,Hatch (1,4) Open,Hatch (2,3) Open] ,
  actions = [StartRoom (Room 1 Clear)] }

*Main> length (allMoves [openMini])
5

[GameState { rooms= [Room 1 Clear,Room 2 LowFlood,Room 3 Clear,Room 4 Fire] ,
  hatches = [Hatch (1,2) Open,Hatch (1,3) Open,Hatch (1,4) Open,Hatch (2,3) Open] ,
  actions = [StartRoom (Room 1 Clear)] }
,GameState { rooms= [Room 1 Clear,Room 2 LowFlood,Room 3 Clear,Room 4 Fire] ,
  hatches = [Hatch (1,2) Open,Hatch (1,3) Open,Hatch (1,4) Open,Hatch (2,3) Open] ,
  actions = [Move (Room 2 LowFlood) 1,StartRoom (Room 1 Clear)] }
,GameState { rooms= [Room 1 Clear,Room 2 LowFlood,Room 3 Clear,Room 4 Fire] ,
  hatches = [Hatch (1,2) Open,Hatch (1,3) Open,Hatch (1,4) Open,Hatch (2,3) Open] ,
  actions = [Move (Room 3 Clear) 0,StartRoom (Room 1 Clear)] }
,GameState { rooms= [Room 1 Clear,Room 2 LowFlood,Room 3 Clear,Room 4 Fire] ,
  hatches = [Hatch (1,2) Open,Hatch (1,3) Open,Hatch (1,4) Open,Hatch (2,3) Open] ,
  actions = [Move (Room 3 Clear) 0,Move (Room 2 LowFlood) 1,StartRoom (Room 1 Clear)] }
,GameState { rooms= [Room 1 Clear,Room 2 LowFlood,Room 3 Clear,Room 4 Fire] ,
  hatches = [Hatch (1,2) Open,Hatch (1,3) Open,Hatch (1,4) Open,Hatch (2,3) Open] ,
  actions = [Move (Room 2 LowFlood) 1,Move (Room 3 Clear) 0,StartRoom (Room 1 Clear)] }
]
```

Without pruning, we have five results, but the third and fourth game states are equivalent because they both end in Room 3, and the second and fifth results both end in Room 2. However, in each pair, one of the paths went through an irrelevant room, so they should be removed from the result set.

```
*Main> length (prune (allMoves ([openMini])))
3

*Main> prune (allMoves [openMini])
[GameState { rooms= [Room 1 Clear,Room 2 LowFlood,Room 3 Clear,Room 4 Fire] ,
  hatches = [Hatch (1,2) Open,Hatch (1,3) Open,Hatch (1,4) Open,Hatch (2,3) Open] ,
  actions = [StartRoom (Room 1 Clear)] }
,GameState { rooms= [Room 1 Clear,Room 2 LowFlood,Room 3 Clear,Room 4 Fire] ,
  hatches = [Hatch (1,2) Open,Hatch (1,3) Open,Hatch (1,4) Open,Hatch (2,3) Open] ,
  actions = [Move (Room 2 LowFlood) 1,StartRoom (Room 1 Clear)] }
,GameState { rooms= [Room 1 Clear,Room 2 LowFlood,Room 3 Clear,Room 4 Fire] ,
  hatches = [Hatch (1,2) Open,Hatch (1,3) Open,Hatch (1,4) Open,Hatch (2,3) Open] ,
  actions = [Move (Room 3 Clear) 0,StartRoom (Room 1 Clear)] }
]
```

After pruning, there are only three results. One result ends in each of the enterable rooms.

For the miniFlood sub, which still only has four rooms, there are more relevant results because our initial hatch opening causes a flood. We also are considering OpenHatch actions here, and as discussed, it is easy to perform a "useless" OpenHatch if it is not moved through, and does not cause a flood.

```
*Main> length (allMoves [miniFlood])
182
*Main> length (prune (allMoves [miniFlood]))
9
```

In this case, we get a massive improvement in the result set from 182 to 9 results while still preserving all the relevant player choices.

The prune function initially makes the performance of our 10-room sub worse, as we would expect, since it must group and order all the results in the set.

```
*Main> length $ (iterate takeTurn [starterSub])!!10
8596
(0.53 secs, 46052252 bytes)
*Main> length $ (iterate takeTurn [starterSub])!!15
357564
(37.17 secs, 3245943772 bytes)

*Main> length $ prune ((iterate takeTurn [starterSub])!!10)
9
(2.11 secs, 185105528 bytes)
*Main> length $ prune ((iterate takeTurn [starterSub])!!15)
5
(137.45 secs, 11684597024 bytes)
```

Note that of the 357,000 15-step GameStates, there were only 5 unique configurations remaining after the prune function call. By keeping track of all the possible states and pruning at the end, I was holding on to a lot of junk data.

I then explored the possibility of pruning at regular intervals during the generation of the result sets. Since pruning itself has some overhead, I wasn't sure how this tradeoff would affect the overall run time, so I experimented with different intervals for pruning.

Since we are pruning early, there is a risk that we would never get a completely empty iteration of takeTurn like we relied on in allMoves. Why? Suppose there are two GameStates that the prune function is comparing. State 1 has the player staying in a room for cost 0. State 2 has the player opening three hatches but never moving out of the room for cost 3. The pruning function will observe that State 1 and State 2 are equivalent and keep State 1 since it has a lower cost, removing State 2 from the result set. However, if we iteratively call takeTurn on the pruned result set, we will again generate State 2 as a valid game state.

To safely capture all moves, we can set an upper bound on the number of takeTurns that should be taken during the player's movement. The worst case is a linear sub where you need to do (numRooms -1) HatchOpen actions and (numRooms - 1) Moves. (The actual board game has a much shorter maximum, but I am trying to keep the code general at this point). So, we can set the upper bound as numRooms * 2. There is no problem with taking too many turns; allTurns may just return empty results at that point.

```
> --Take pruneInterval turns and prune afterwards
> takeOptimalTurns             :: Int -> [GameState] -> [GameState]
> takeOptimalTurns pruneInterval gameState = prune (concat (take pruneInterval (iterate
```

```
takeTurn gameState)))


> -- Choose how frequently to prune the result set. (aka how many turns to take between
pruning phases)
> startGameTemp ::  Int -> GameState -> [GameState]
> startGameTemp pruneInterval gst = (iterate (takeOptimalTurns pruneInterval)
gameState)!!numIterations
>           where numRooms = length (rooms (gst))
>                 numIterations = (2 * numRooms) `div` pruneInterval + 1 -- Round up to be
sure to capture all options
>                 gameState = [gst] -- we need to start takeOptimalTurns with a [GameState]
```

Pruning the result set early shows a major improvement in the performance!!

```
*Main> length (startGameTemp 10 starterSub)
10
(25.53 secs, 2209836972 bytes)
*Main> length (startGameTemp 6 starterSub)
10
(0.86 secs, 79642608 bytes)
*Main> length (startGameTemp 5 starterSub)
10
(0.48 secs, 41402792 bytes)
*Main> length (startGameTemp 4 starterSub)
10
(0.14 secs, 13483624 bytes)
*Main> length (startGameTemp 3 starterSub)
10
(0.06 secs, 5201360 bytes)
```

Pruning more frequently clearly improves the run time.

Below a pruning frequency of 3, we start getting incorrect values. On the sub with no rooms flooded, we expect 10 possible states (1 for each final room). However, pruning after 2 or 1 steps means we are removing valid values.

```
*Main> length (startGameTemp 2 starterSub)
1
(0.00 secs, 520088 bytes)
*Main> length (startGameTemp 1 starterSub)
1
(0.00 secs, 517076 bytes)
```

These runs produce incorrect results is because take 2 from the takeTurn iteration means we get the 0th move (start state) and the 1st move (opening a hatch). Pruning after these two steps will never allow us to move out of a room, since we always need to open a hatch before a move.

Why don't we lose any results when we prune every 3 steps?

In general, you can only perform one "useless" hatch opening before moving. If your first hatch open causes your room to move to LowFlood (either decreasing or increasing your

room's flood level), then you are standing in a room at LowFlood and you can't participate in any more floods per the rules of the game. So at the very most, you would perform two hatch opens & one room move that would all be relevant to the game state, and any additional hatch opens will eventually pruned out. Thus it is safe to prune every 3 moves without a loss of any relevant actions.

```
> startGame :: GameState -> [GameState]
> startGame gst = startGameTemp 3 gst

*Main> length (startGame starterSub)
10
(0.06 secs, 5721652 bytes)

*Main> startGame starterSub
...
(0.08 secs, 8818820 bytes)
```

We can now evaluate the 10-room sub in a much shorter time than the allMoves followed by a single prune at the end.

Reading from a file and exporting to JSON

To simplify testing, I created a very basic IO input that configures the initial GameState from a flat file. This is the first part of the program that will only specifically work for the description of the specific board in Red November. The ten room states are listed in order on one line, then the 15 hatch states, then the number of the start room.

```
> inputFromFile          :: String -> IO GameState
> inputFromFile filePath = do
>                            f <- readFile filePath
>                            return (parseInput f)


> parseInput             :: String -> GameState
> parseInput fileString  = GameState {
>                                     rooms = newRooms,
>                                     hatches = newHatches,
>                                     actions = newActions
>                                    }
>            where fileLines = lines fileString
>                  roomsString = fileLines!!0
>                  newRooms = parseRoomsString roomsString
>                  hatchesString = fileLines!!1
>                  newHatches = parseHatchesString hatchesString
>                  currentRoomNum  = read (fileLines!!2) :: Int
>                  startRoom = getRoomByNum currentRoomNum newRooms
>                  newActions = [StartRoom startRoom ]
>
```

```
> roomNumbers = [1..10]

> parseRoomsString             :: String -> [Room]
> parseRoomsString roomsString = zipWith createRoom roomNumbers (splitOn "," roomsString)

> createRoom          :: Int -> String -> Room
> createRoom num state = case toLower (head state) of
>                            'c'  -> Room num Clear
>                            'f'  -> Room num Fire
>                            'l'  -> Room num LowFlood
>                            'h'  -> Room num HighFlood
>                           otherwise -> error ("Could not understand room state " ++
state)

> hatchNumbers = [
>   (1, 2) ,
>   (1, 3) ,
>   (2, 3) ,
>   (2, 4) ,
>   (2, 5) ,
>   (3, 4) ,
>   (4, 5) ,
>   (5, 6) ,
>   (5, 7) ,
>   (5, 8) ,
>   (7, 8) ,
>   (7, 9) ,
>   (8, 9) ,
>   (8, 10),
>   (9, 10)
>   ]

> parseHatchesString               :: String -> [Hatch]
> parseHatchesString hatchesString = zipWith createHatch hatchNumbers (splitOn ","
hatchesString)

> createHatch              :: (Int, Int) -> String -> Hatch
> createHatch num state    = case toLower (head state) of
>                                'o'  -> Hatch num Open
>                                'c'  -> Hatch num Closed
>                                'b'  -> Hatch num Blocked
>                               otherwise -> error ("Could not understand hatch state "
++ state)
```

The output for the result set is more interesting. I convert the GameState for each result to JSON, a portable and javascript-friendly format. This could be sent over HTTP if the program were running on a web server, but in this case, I am just outputting it directly to a second flat javascript file.

I use a Json type class to define the JSON string for each of the classes used in GameState.

```
> class Json a where
```

```
>  json :: a -> String

> instance Json a => Json [a] where
>    json a = "[\n" ++
>                intercalate ",\n" (map json a) ++
>               "\n]"

> instance Json Room where
>    json (Room number state)   = "{ \"number\": " ++ (show number) ++
>                                 " , \"state\": \"" ++ (show state) ++ "\" }"

> --Represent the numbers from the hatch as an string Hatch (1,2) _ -> "number" : "1-2"
> instance Json Hatch where
>    json (Hatch (x,y) state)  = "{ \"number\": \"" ++ (show x) ++ "-" ++ (show y) ++
>                                "\" , \"state\": \"" ++ (show state) ++ "\" }"


> instance Json Action where
>    json (Move room cost)         = "{ \"type\": \"Move\", \"room\": " ++ json room ++ ",
\"cost\": " ++ show cost ++ " }"
>    json (OpenHatch hatch cost)   = "{ \"type\": \"OpenHatch\", \"hatch\": " ++ json hatch
++ ", \"cost\": " ++ show cost ++ " }"
>    json (Flood room1 room2) = "{ \"type\": \"Flood\", \"room1\": " ++ json room1 ++ ",
\"room2\": " ++ json room2 ++ " }"
>    json (StartRoom room)         = "{ \"type\": \"StartRoom\", \"room\": " ++ json room
++ " }"

> instance Json GameState where
>    json gst                     = "{ \"rooms\": " ++ json (rooms gst) ++
>                                   ", \n \"hatches\": " ++ json (hatches gst) ++
>                                   ", \n \"actions\": " ++ json (tail (reverse (actions
gst))) ++
> -- We store the actions in reverse order, flip them to print. Also, remove the starting
room (which is the head after reversal)
>                                   ", \n \"finalRoom\": " ++ json (currentRoom gst) ++
>                                   ", \n \"startRoom\": " ++ json (startRoom gst) ++
>                                   ", \n \"totalCost\": " ++ show (totalCost gst) ++
>                                   "}"
```

To test this, we can print one of our sample gamestates in JSON format:

```
*Main> putStrLn (json miniFlood)
{ "rooms": [
{ "number": 1 , "state": "Clear" },
{ "number": 2 , "state": "HighFlood" },
{ "number": 3 , "state": "Fire" },
{ "number": 4 , "state": "LowFlood" }
],
 "hatches": [
{ "number": "1-2" , "state": "Closed" },
{ "number": "1-3" , "state": "Closed" },
{ "number": "2-3" , "state": "Closed" },
{ "number": "2-4" , "state": "Closed" },
```

```
{ "number": "3-4" , "state": "Closed" }
],
 "actions": [

],
 "finalRoom": { "number": 2 , "state": "Clear" },
 "startRoom": { "number": 2 , "state": "Clear" },
 "totalCost": 0}
```

Note that we remove the starting room from the action list. We kept it in the action list in the GameState to make currentRoom calculations easier, but we separate it out into a separate attribute in JSON.

The results viewer always requires the same hard-coded output .js file with the JSON result, but we can feed different input file paths to the program.

```
> runFileToFile          :: String -> IO ()
> runFileToFile filePath = do inputState <- inputFromFile filePath
>                             playToFile inputState
```

Our output file includes a description of the initial GameState and all the possible GameStates that are generated from startGame.

```
> playToFile :: GameState -> IO ()
> playToFile gst = do
>          writeFile "html/sub.js" "var starterSub = "
>          appendFile "html/sub.js" (json gst)
>          appendFile "html/sub.js" "; \n\n var results = "
>          appendFile "html/sub.js" (json (startGame gst))
>          appendFile "html/sub.js" ";"
>
```

The results viewer is written in HTML5 canvas, javascript, and some basic CSS. The code for the viewer is given in Appendix II.

Conclusions

This program generates all the possible move actions in the Red November board game. The allowable actions depend on the state of the hatches and the rooms of the submarine, and they have a certain cost to the player. In addition, some actions cause side effects on the submarine when water flows from one room to the other. The program represents the state of the submarine as well as the actions that the player has taken.

The original strategy of this program was to generate all possible sequences of actions, and then evaluate which of those are optimal at the end of the calculation. However, the performance of this strategy was unacceptable on a full-sized submarine. Instead, I chose to prune out the inefficient states every 3 turns, and the performance massively improved.

To make the results easily visible, I wrote a HTML and Javascript based web application to render the data onto a webpage. The program outputs the result as JSON, which is a natural

input for javascript, using a Json typeclass and instances for all the relevant data types.

Future directions for this program would be to take JSON as an input as well, and have the Haskell program read and write over HTTP on a web server. In addition, it currently only handles the move action of a single player's turn -- there are many other moving parts in the board game left to model and analyze.