CS 4613
Spring 2019
Project 1
Graeme Ferguson – 03/03/19

RUNNING INSTRUCTIONS

It was impossible for me to include one runnable text file with my program as I used separate compilation and a dependency manager in my development. As such, I've tried to alleviate this with including a zip file of my working directory wherein there is located a built binary file capable of being run on Linux systems (and hopefully Mac systems, I haven't tested it).

With the extracted zip file, there should appear a *puzzle-problem* directory. Navigate into it and the following command format should work:

./app/Main filePath heuristicFlag

filePath represents the input file path. For example: ./input/Input1.txt
heuristicFlag represents the heuristic choice and can be either 0 or 1 with 0 representing a heuristic of only the sum of the manhattan distances and 1 representing the sum of manhattan distances + 2x linear conflicts.

An example running of the program with input *./input/Input1.txt* and heuristic *0* would be:

./app/Main ./input/Input1.txt 0

The output of which should be writing to the *output.txt* file in the *output* directory.

Another possible way to run this program is to install my dependency manager, stack, for which the instructions can be found on this page: https://docs.haskellstack.org/en/stable/install_and_upgrade/

Once installed, at least on a Linux system, one should be able to run my program, once again in the *puzzle-program* directory, you can type first:

slack build

To build the project and then you can type:

stack exec puzzle-problem-exe filePath heuristicFlag

Where filePath and heuristicFlag correspond to the previous usage of the command line arguments as described above for the running of the file with the binary

If this does not work, please contact me (ggf221@nyu.edu) and I can arrange a time to meet in person to show you the running of my program with any text file provided.

OUTPUT

Output1_A

7 1 6
8 3 5
2 0 4

8 7 6
1 0 5
2 3 4

5
12
U U L D R
5 5 5 5 5 5

Output1_B

7 1 6
8 3 5
2 0 4

8 7 6
1 0 5
2 3 4

5
12
U U L D R
5 5 5 5 5 5

Output2_A

2 6 0
1 3 4
7 5 8

1 2 3
4 5 6
7 8 0

10
27
L D R U L L D R D R
10 10 10 10 10 10 10 10 10 10 10

Output2_B

2 6 0
1 3 4
7 5 8

1 2 3
4 5 6
7 8 0

10
24
L D R U L L D R D R
10 10 10 10 10 10 10 10 10 10 10

Output3_A
5 4 3
2 6 7
1 8 0

1 2 3
4 5 6
7 8 0

22
1921
U L U L D D R U U L D D R R U L L D R U R D
12 12 12 12 12 12 12 14 16 16 16 16 18 18 18 20 22 22 22 22 22 22 22

Output3_B
5 4 3
2 6 7
1 8 0

1 2 3
4 5 6
7 8 0

22
1131
U L U L D D R U U L D D R R U L L D R U R D
12 14 14 14 14 14 14 16 18 18 18 18 18 18 18 20 22 22 22 22 22 22 22

Output4_A
8 7 3
0 4 5
6 2 1

1 2 3
4 5 6
7 8 0

23
1164
U R D D R U L D L U U R D R D L L U U R D R D

17 17 17 19 19 19 21 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23

Output4_B
8 7 3
0 4 5
6 2 1

1 2 3
4 5 6
7 8 0

23
696
U R D D R U L D L U U R D R D L L U U R D R D
17 17 17 19 19 19 21 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23

SOURCE CODE

Board.hs

```
module Board where

type Board = [Int]

boardWidth = 3 :: Int
```

Index.hs

```
module Index where

import Data.List (elemIndex)
import Data.Tuple (swap)

import Board

toIndex :: (Int, Int) -> Int
toIndex (x,y) = x + (y * boardWidth)

fromIndex :: Int -> (Int, Int)
fromIndex i = swap $ divMod i boardWidth

-- validSingle checks if x or y val is on game board
validSingle :: Int -> Bool
validSingle x
    | x < 0 = False
    | x > boardLimit = False
    | otherwise = True
    where
        boardLimit = boardWidth - 1

-- validCoord checks if co-ord given is on game board
```

```
validCoord :: (Int, Int) -> Bool
validCoord (x,y)
  | not $ validSingle x = False
  | not $ validSingle y = False
  | otherwise = True

-- finds the starting index and the goal index for a value
findStartGoalIndexes :: Board -> Board -> Int -> Maybe (Int, Int)
findStartGoalIndexes current goal element =
  case elemIndex element current of
    Just currentIndex ->
      case elemIndex element goal of
        Just goalIndex -> (Just (currentIndex, goalIndex))
        Nothing -> Nothing
    Nothing -> Nothing

-- returns the sum of a list of Maybe Int
sumMaybeInt :: [Maybe Int] -> Maybe Int
sumMaybeInt = fmap sum . Sequence
```

Manhattan.hs

```
module Manhattan where

import Data.List (elemIndex)
import Data.Functor (fmap)
import Control.Monad (sequence)

import Index
import Board

-- Get the distance between two x,y coords
coordDistance :: (Int, Int) -> (Int, Int) -> Int
coordDistance (x1,y1) (x2,y2) =
  xDistance + yDistance
  where
    xDistance = abs $ x1 - x2
    yDistance = abs $ y1 - y2

-- Get the manhattan distance given an element on the board
manhattanDistance :: Board -> Board -> Int -> Maybe Int
manhattanDistance current goal element =
  case findStartGoalIndexes current goal element of
    Just (currentIndex, goalIndex) ->
      (Just distance)
      where
        distance = coordDistance (fromIndex currentIndex) (fromIndex goalIndex)
    Nothing -> Nothing

-- Get the sum of all calculated manhattan distances on the board
```

```
manhattanSum :: Board -> Board -> Maybe Int
manhattanSum current goal =
    sumMaybeInt distanceList
    where
        distanceList = map (manhattanDistance current goal) [1..8]
```

LinearConflict.hs

```
module LinearConflict where

import Index
import Board

-- listEquals checks if all elements in a list are the same
listEquals :: [Int] -> Bool
listEquals xs = and $ map (== head xs) (tail xs)

-- coordLinConflict returns if two vals (with start ang goal coords known) are
-- in linear conflict. This function could use improvement. The nested ifs are
-- ugly.
coordLinConflict ::
       (Int, Int) -- startLeftCoord
    -> (Int, Int) -- goalLeftCoord
    -> (Int, Int) -- startRightCoord
    -> (Int, Int) -- goalLeftCoord
    -> Bool -- if linear conflict
coordLinConflict
    (leftStartX, leftStartY)
    (leftGoalX, leftGoalY)
    (rightStartX, rightStartY)
    (rightGoalX, rightGoalY) =
        -- are the coords in the same column?
        if listEquals $ leftStartX:leftGoalX:rightStartX:rightGoalX:[]
        then
            -- check whether they're in a linear conflict within the column
            if leftStartY < rightStartY
            then
                if leftGoalY > rightGoalY
                then True
                else False
            else if leftStartY > rightStartY
            then
                if leftGoalY < rightGoalY
                then True
                else False
            else False
        -- are the coords in the same row?
        else if listEquals $ leftStartY:leftGoalY:rightStartY:rightGoalY:[]
        then
            -- check if they're in a linear conflict within the row
```

```
      if leftStartX < rightStartX
      then
        if leftGoalX > rightGoalX
        then True
        else False
      else if leftStartX > rightStartX
      then
        if leftGoalX < rightGoalX
        then True
        else False
      else False
    else False

-- linConflict returns whether two squares on a start board are in linear conflict
-- with reference to the goal board
linConflict :: Board -> Board -> Int -> Int -> Maybe Bool
linConflict current goal left right =
  case findStartGoalIndexes current goal left of
    Just (startLeft, goalLeft) ->
      case findStartGoalIndexes current goal right of
        Just (startRight, goalRight) ->
          (Just conflict)
          where
            conflict = coordLinConflict (fromIndex startLeft) (fromIndex goalLeft) (fromIndex
startRight) (fromIndex goalRight)
        Nothing -> Nothing
    Nothing -> Nothing

-- conv maybe bool to maybe int
mBoolToInt :: Maybe Bool -> Maybe Int
mBoolToInt mbool =
  case mbool of
    Just val ->
      if val then (Just 1)
      else (Just 0)
    Nothing -> Nothing

mBoolToIntList :: [Maybe Bool] -> [Maybe Int]
mBoolToIntList = map mBoolToInt

-- returns the sum of a list of bools as an int
sumMaybeBool :: [Maybe Bool] -> Maybe Int
sumMaybeBool = sumMaybeInt . mBoolToIntList

-- for one value on the board, compute all its linear conflicts
linConflictSumVal :: Board -> Board -> Int -> Maybe Int
linConflictSumVal current goal val =
  sumMaybeBool conflicts
  where
```

```
        conflicts = map (linConflict current goal val) [1..8]

-- for all values on the board, compute the total linear conflicts
linConflictSum :: Board -> Board -> Maybe Int
linConflictSum current goal =
    sumMaybeInt conflictSums
    where
        conflictSums = map (linConflictSumVal current goal) [1..8]
```

Heuristics.hs

```
module Heuristics where

import Board
import Index
import Manhattan
import LinearConflict

-- heuristic 1: only the manhattan distance
manhattanHeuristic :: Board -> Board -> Maybe Int
manhattanHeuristic current goal =
    manhattanSum current goal

-- heuristic 2: the sum of manhattan distance + 2x * linear conflicts
manhattanLinConflictHeuristic :: Board -> Board -> Maybe Int
manhattanLinConflictHeuristic current goal =
    sumMaybeInt $ manhattanResult:linConflictResult:[]
    where
        manhattanResult = manhattanSum current goal
        linConflictResult = linConflictSum current goal
```

AStar.hs

```
module AStar where

import Board
import Index
import Data.List (elemIndex, minimumBy)
import Heuristics

data Move = L | R | U | D
    deriving (Eq)

moveSet = [L, R, U, D] :: [Move]

instance Show Move where
    show move =
        case move of
            L -> "L"
            R -> "R"
            U -> "U"
```

```
        D -> "D"

-- type NodeSet = Map.Map NodeState Bool
type NodeSet = [NodeState]

data SearchState = SearchState {
    startBoard :: Board, -- holds the start board
    goalBoard :: Board, -- holds the goal board
    heuristic :: (Board -> Board -> Maybe Int), -- holds the heuristic function
    solutionDepth :: Int, -- holds the depth of the solution
    nodesGenerated :: NodeSet, -- holds the nodes that have been generated previously
    nodeQueue :: [NodeState], -- holds unexpanded nodes
    numNodesGenerated :: Int, -- holds number of generated nodes
    solutionMoves :: [Move], -- holds the moves to the solution
    solutionVals :: [Int] -- holds the f(n) values to the solution
}

data NodeState = NodeState {
    currentBoard :: Board, -- holds the current board
    fval :: Int, -- holds the f(n) value of the node
    depth :: Int, -- holds the depth of the node
    moves :: [Move], -- holds the moves from the ancestor nodes
    fvals :: [Int] -- holds the f(n) value of the ancestor nodes
}

instance Show NodeState where
    show (NodeState current fval depth moves fvals)
        = "\nCurrent Board: " ++ show current ++
          "\nF(n) value: " ++ show fval ++
          "\nDepth: " ++ show depth ++
          "\nMoves (newest first): " ++ show moves ++
          "\nF(n) values (newest first): " ++ show fvals ++
          "\n----------------------------------------"

instance Eq NodeState where
    (==) (NodeState currentLeft _ _ _ _) (NodeState currentRight _ _ _ _) =
        currentLeft == currentRight
    (/=) x y =
        not ((==) x y)

compareNode :: NodeState -> NodeState -> Ordering
compareNode (NodeState _ leftFVal _ _ _) (NodeState _ rightFVal _ _ _)
    | leftFVal > rightFVal = GT
    | leftFVal < rightFVal = LT
    | otherwise = EQ

instance Ord NodeState where
    compare x y = compareNode x y
```

```
instance Show SearchState where
   show (SearchState startBoard goalBoard _ solutionDepth nodesGenerated nodeQueue
numNodesGenerated solutionMoves solutionVals)
      = "\nStart Board: " ++ show startBoard ++
        "\nGoal Board: " ++ show goalBoard ++
        "\nDepth: " ++ show solutionDepth ++
      --   "\nNodes Generated: " ++ show nodesGenerated ++
      --   "\nNode queue: " ++ show nodeQueue ++
        "\n# Nodes Generated: " ++ show (fromIntegral (length nodesGenerated)) ++
        "\nSolution Moves (newest first): " ++ show solutionMoves ++
        "\nSolution f(n)s (newest first): " ++ show solutionVals ++
        "\n----------------------------------------------"

-- swaps two elements in a list
swapTwo :: Int -> Int -> [a] -> [a]
swapTwo first second xs = zipWith (\x y ->
   if x == first then xs !! second
   else if x == second then xs !! first
   else y) [0..] xs

-- generates new coord based on move
moveCoord :: (Int, Int) -> Move -> (Int, Int)
moveCoord (x, y) move =
   case move of
      L -> (x-1, y)
      R -> (x+1, y)
      U -> (x, y-1)
      D -> (x, y+1)

-- generates all possible moves for coord
validMovesCoord :: (Int, Int) -> [Move]
validMovesCoord coord = [ move | move <- moveSet, validCoord $ moveCoord coord move ]

-- generates all possible moves for current board
allMoves :: Board -> Maybe [Move]
allMoves current =
   -- find the blank position
   case elemIndex 0 current of
      Just blankIndex -> (Just moves)
         where
            -- find the valid moves for the blank position
            moves = validMovesCoord $ fromIndex blankIndex
      Nothing -> Nothing

-- generates new board given current board and a move
moveBoard :: Board -> Move -> Maybe Board
moveBoard current move =
   -- find the blank position
   case elemIndex 0 current of
```

```
        Just blankIndex -> (Just newBoard)
          where
            -- find the index of the blank tile once you move it accordinly
            moveIndex = toIndex $ moveCoord (fromIndex blankIndex) move
            -- generate the new board by swapping the blank with the tile its moving to
            newBoard = swapTwo blankIndex moveIndex current
        Nothing -> Nothing

-- bad code but running out of time
unsafeUnmaybe :: Maybe a -> a
unsafeUnmaybe maybe =
    case maybe of
        Just a -> a
        Nothing -> error "unsafeUnmaybe ran into some trouble"

nextNodeState :: Board -- Goal board
              -> (Board -> Board -> Int) -- f(n)
              -> NodeState -- Expanding node
              -> Move -- Current move
              -> NodeState -- New node
nextNodeState goal fn (NodeState current fval depth moves fvals) move =
    -- return newly generate node
    NodeState current' fval' depth' moves' fvals'
    where
        -- get the new current board
        current' = unsafeUnmaybe $ moveBoard current move
        -- get the new f(n) value
        fval' = fn current' goal
        -- increase depth counter by 1
        depth' = depth + 1
        -- add move to move list
        moves' = moves ++ move:[]
        -- add f(n) value to f(n) value list
        fvals' = fvals ++ fval':[]


possibleNodes :: Board -- Goal board
              -> (Board -> Board -> Maybe Int) -- Heuristic
              -> NodeState -- Expanding node
              -> [NodeState] -- List of new nodes
possibleNodes goal heuristic (NodeState current fval depth moves fvals) =
    -- generate all node states given new f(n) and valid moves
    map (nextNodeState goal fn (NodeState current fval depth moves fvals)) validMoves
    where
        -- generating new level, increasing depth (gn)
        gn = depth + 1
        -- generate new f(n)
        fn = constructFn gn heuristic
        -- get all valid moves given current board
```

```
        validMoves = unsafeUnmaybe $ allMoves current

constructFn :: Int -> (Board -> Board -> Maybe Int) -> (Board -> Board -> Int)
constructFn gn hn =
    -- return a function that adds a given g(n) value with a heuristic function
    (\x y -> (+) gn (unsafeUnmaybe $ hn x y))

-- finds the node with the lowest f(n) value in a list of nodes
nextBestNode :: [NodeState] -> NodeState
nextBestNode = minimumBy compareNode

-- given a list of already generated nodes and a list of possible nodes, return
-- only the nodes that haven't already been generated
generateNewNodes :: NodeSet -> [NodeState] -> [NodeState]
generateNewNodes _ [] = []
generateNewNodes nodeset nodes =
    [node | node <- nodes, not $ elem node nodeset]

-- check if a node is a goal node
checkGoalNode :: Board -> NodeState -> Bool
checkGoalNode goal (NodeState current _ _ _ _) =
    goal == current

-- add a list of nodes to a list of already generated nodes
recordNewNodes :: NodeSet -> [NodeState] -> NodeSet
recordNewNodes nodeset nodes =
    nodeset ++ nodes

-- test data
testStartBoard = [2, 8, 3, 1, 6, 4, 7, 0, 5]
testGoalBoard = [1, 2, 3, 0, 8, 4, 7, 6, 5]
testFn = constructFn 0 manhattanHeuristic
testFval = testFn testStartBoard testGoalBoard
testNode = NodeState testStartBoard testFval 0 [] $ testFval:[]
testStartState = SearchState testStartBoard testGoalBoard manhattanHeuristic (-1) [] [] 0 [] []

-- run a recurisve astar algorithm
runAStar :: SearchState -> SearchState
runAStar (SearchState startBoard goalBoard heuristic solutionDepth nodesGenerated nodeQueue
numNodesGenerated solutionMoves solutionVals) =
    -- check if the the algorithm has started
    case null nodesGenerated of
        -- if it hasn't, generate the first node and call algorithm again
        True ->
            runAStar $ SearchState startBoard goalBoard heuristic solutionDepth nodesGenerated'
nodeQueue' numNodesGenerated' solutionMoves solutionVals
            where
                fn = constructFn 0 heuristic
                fval = fn startBoard goalBoard
```

```haskell
          fvals = fval:[]
          node = NodeState startBoard fval 0 [] fvals
          nodesGenerated' = nodesGenerated ++ node:[]
          nodeQueue' = node:[]
          numNodesGenerated' = numNodesGenerated' + 1
    -- if it has, find the next best move
    False ->
      -- check if the next best move is the goal state
      case checkGoalNode goalBoard $ NodeState current fval depth moves fvals of
        -- if it is, return the goal state
        True ->
          SearchState startBoard goalBoard heuristic depth nodesGenerated nodeQueue' numNodesGenerated moves fvals
        -- if it isn't, generate new nodes and record the data, before recursing again
        False ->
          runAStar newState
          where
            newNodes = generateNewNodes nodesGenerated $ possibleNodes goalBoard heuristic $ NodeState current fval depth moves fvals
            nodesGenerated' = recordNewNodes nodesGenerated newNodes
            nodeQueue'' = nodeQueue' ++ newNodes
            numNodesGenerated' =  numNodesGenerated + length newNodes
            newState = SearchState startBoard goalBoard heuristic solutionDepth nodesGenerated' nodeQueue'' numNodesGenerated' solutionMoves solutionVals
      where
        -- get next best node
        (NodeState current fval depth moves fvals) = nextBestNode nodeQueue
        -- remove best node from unexpanded node queue
        nodeQueue' = filter ((/=) $ NodeState current fval depth moves fvals) nodeQueue
```

Lib.hs

```haskell
module Lib where

import Data.List (lines)
import Data.List.Split (splitOn, chunksOf)

import Board
import Heuristics
import AStar

-- readBoards takes the data received from reading a test file and cleans it
-- into manageable data (a tuple of two Int arrays)
readBoards :: String -> (Board, Board)
readBoards content =
  -- construct a tuple from the generated array
  (flattenedBoards !! 0, flattenedBoards !! 1) :: (Board, Board)
  where
    -- split data into cleaned (no \r\n) lines
    fLines = map (filter (\x -> ((/=) '\r' x) && ((/=) '\n' x))) $ lines content
```

```haskell
        -- remove new line and split integer characters
        splitFLines = map (splitOn " ") $ filter (/= "") fLines
        -- convert strings to ints
        cleanedContent = map (map (\x -> read x :: Int)) splitFLines
        -- separate the big array of data into two boards
        twoBoards = chunksOf 3 cleanedContent
        -- flatten from two 2d array to two 1d arrays
        flattenedBoards = map concat twoBoards

-- to2d takes a puzzle board and chunks it into a 2d list with elements of
-- length boardWidth for easier printing
to2d :: Board -> [[Int]]
to2d = chunksOf boardWidth

-- stringBoard "stringifies" a 2d-ified puzzle board
stringBoard :: [[Int]] -> String
stringBoard [] = ""
stringBoard (x:xs) = (show x) ++ "\n" ++ stringBoard xs

stringList :: Show a => [a] -> String
stringList [] = ""
stringList (x:xs) =
    (show x) ++ " " ++ stringList xs


string2dBoard :: [[Int]] -> String
string2dBoard [] = ""
string2dBoard (x:xs) =
    stringList x ++ "\n" ++ string2dBoard xs

-- ppBoard prints the a puzzle board in a pretty fashion
ppBoard :: Board -> String
ppBoard = string2dBoard . to2d

printSolution :: SearchState -> String
printSolution (SearchState startBoard goalBoard _ solutionDepth nodesGenerated _ _ solutionMoves
solutionVals)
    = ppBoard startBoard ++ "\n" ++
      ppBoard goalBoard ++ "\n" ++
      show solutionDepth ++ "\n" ++
      show (fromIntegral (length nodesGenerated)) ++ "\n" ++
      stringList solutionMoves ++ "\n" ++
      stringList solutionVals ++ "\n"


getSolution :: String -> Board -> Board -> String
getSolution hnChoice startBoard goalBoard =
    case hnChoice of
        "0" ->
```

```
            printSolution solution
            where
                solution = runAStar $ SearchState startBoard goalBoard manhattanHeuristic (-1) [] [] 0 [] []
        "1" ->
            printSolution solution
            where
                solution = runAStar $ SearchState startBoard goalBoard manhattanLinConflictHeuristic (-1)
[] [] 0 [] []
```

Main.hs

```
module Main where

import System.Environment (getArgs)

import Lib

main :: IO ()
main = do
    -- Get filepath and heuristic choice from command line args
    args <- getArgs
    case args of
        [] -> putStrLn "Missing filepath and heuristic choice"
        [inputFilePath] -> putStrLn "Missing heuristic choice - input 0 for manhattan distance heuristic or
1 for manhattan & 2x linear conflict heuristic"
        [inputFilePath, hnChoice] -> do
            -- Read the file into a string
            content <- readFile (inputFilePath)
            -- Process that string into two workable arrays
            let (startBoard, goalBoard) = readBoards content
                solution = getSolution hnChoice startBoard goalBoard

            -- Write the solution to an outfile
            writeFile "./output/output.txt" solution
```