**Subject：**COMP1039 Coursework 2

**Student Name：**Youyao Gao

**Student ID：**20516639

**Email：**scyyg6@nottingham.edu.cn

**Section A: Solving the Farmer Crosses River Puzzle using State Space Search**

Each state in the Farmer Crosses River puzzle is represented as a combination of locations for the farmer, wolf, goat, and cabbage. The position of each item (farmer, wolf, goat, and cabbage) is denoted as either being on the west bank ('w') or the east bank ('e').

The Farmer Crosses River puzzle can be effectively solved using a Depth-First Search (DFS) algorithm. This approach is particularly useful for exploring the entire depth of potential solutions before backtracking, which is helpful in puzzles like this where all possible state transitions need to be considered to find a valid solution.

Here is an example on how DFS works supposing that the initial state is [‘w’’w’’w’’w’] and the goal state is [‘e’,’e’,’e’,’e’]

A diagram of a family tree

Description automatically generated

**Section B：Haskell Source Code**

import Data.List (intercalate)

type State = [Char]

type Path = [State]

type Solution = [Path]

solutionPath :: Path -> IO ()

solutionPath path

| not (isValidInput0 (head path)) || not (isValidInput0 (last path)) = putStrLn "Invalid input, please change your states."

| isValidInput1 (head path) || isValidInput1 (last path) = putStrLn "On the west bank, the wolf will eat the goat, please change your states."

| isValidInput2 (head path) || isValidInput2 (last path) = putStrLn "On the west bank, the goat will eat the cabbage, please change your states."

| isValidInput3 (head path) || isValidInput3 (last path) = putStrLn "On the east bank, the wolf will eat the goat, please change your states."

| isValidInput4 (head path) || isValidInput4 (last path) = putStrLn "On the east bank, the goat will eat the cabbage, please change your states."

| isValidInput5 (head path) || isValidInput5 (last path) = putStrLn "On the west bank, the wolf will eat the goat and the goat will eat the cabbage, please change your states."

| isValidInput6 (head path) || isValidInput6 (last path) = putStrLn "On the east bank, the wolf will eat the goat and the goat will eat the cabbage, please change your states."

| otherwise = do

let finalState = last path

initialPaths = [init path]

allPaths = exploreAllPaths finalState initialPaths

shortestPaths = findShortestPaths allPaths

formattedShortestPaths = intercalate ", " $ map formatPath shortestPaths

stepsCount = calculateSteps (head shortestPaths)

pathsOutput = "Shortest paths are: " ++ formattedShortestPaths

tripsOutput = "Least number of trips is: " ++ show stepsCount

putStrLn "All possible paths are:"

mapM\_ (putStrLn . formatPath) allPaths

putStrLn pathsOutput

putStrLn tripsOutput

exploreAllPaths :: State -> Solution -> Solution

exploreAllPaths \_ [] = []

exploreAllPaths targetState paths = concatMap (explorePath targetState) paths

where

explorePath target path

| last path == target = [path]

| otherwise =

let possibleMoves = move (last path)

validMoves = filter (`notElem` path) possibleMoves

newPaths = map (\nextMove -> path ++ [nextMove]) validMoves

in concatMap (explorePath target) newPaths

findShortestPaths :: Solution -> Solution

findShortestPaths paths =

let minLength = minimum $ map length paths

in filter ((== minLength) . length) paths

formatPath :: Path -> String

formatPath = concatMap (\state -> formatState state )

formatState :: State -> String

formatState state = "[" ++ intercalate ", " (map (\c -> "'" ++ [c] ++ "'") state) ++ "]"

calculateSteps :: Path -> Int

calculateSteps path = length path - 1

isValidInput0 :: [Char] -> Bool

isValidInput0 xs = xs `elem` validInputs

where

validInputs = [['w','w','w','w'],['w','w','w','e'],['w','w','e','w'],['w','e','w','w'],['e','w','w','w'],['w','w','e','e'],['w','e','w','e'],['e','w','w','e'],['w','e','e','w'],['e','w','e','w'],['e','e','w','w'],['w','e','e','e'],['e','w','e','e'],['e','e','w','e'],['e','e','e','w'],['e','e','e','e']]

isValidInput1 :: [Char] -> Bool

isValidInput1 xs = xs `elem` validInputs

where

validInputs = [['e','w','w','e']]

isValidInput2 :: [Char] -> Bool

isValidInput2 xs = xs `elem` validInputs

where

validInputs = [['e','e','w','w']]

isValidInput3 :: [Char] -> Bool

isValidInput3 xs = xs `elem` validInputs

where

validInputs = [['w','e','e','w']]

isValidInput4 :: [Char] -> Bool

isValidInput4 xs = xs `elem` validInputs

where

validInputs = [['w','w','e','e']]

isValidInput5 :: [Char] -> Bool

isValidInput5 xs = xs `elem` validInputs

where

validInputs = [['e','w','w','w']]

isValidInput6 :: [Char] -> Bool

isValidInput6 xs = xs `elem` validInputs

where

validInputs = [['w','e','e','e']]

showPath :: Path -> String

showPath = concatMap (\state -> "[" ++ state ++ "] ")

move:: State -> Path

move [f, w, g, c] = filter isPossible[crossFarmer[f, w, g, c], crossWolf[f, w, g, c],crossGoat[f, w, g, c], crossCabbage[f, w, g, c]]

isPossible :: State -> Bool

isPossible [f, w, g, c] =

not (f /= w && f /= g && w == g) && not (f /= g && f /= c && g == c)

crossFarmer:: State -> State

crossFarmer [f, w, g, c] = [cross f, w, g, c]

crossWolf:: State -> State

crossWolf [f, w, g, c]

| f == w = [cross f, cross w, g, c]

| otherwise = [f, w, g, c]

crossGoat:: State -> State

crossGoat [f, w, g, c]

| f == g = [cross f, w, cross g, c]

| otherwise = [f, w, g, c]

crossCabbage:: State -> State

crossCabbage [f, w, g, c]

| f == c = [cross f, w, g, cross c]

| otherwise = [f, w, g, c]

cross :: Char -> Char

cross xs

| xs == 'e' = 'w'

| xs == 'w' = 'e'

**Section C: Experimentation with the Program (Input-Output Sessions)**

ValidInput:

A black screen with white dots

Description automatically generated

A black and white screen with white dots

Description automatically generated

InValidInput:



A black screen with white text

Description automatically generated

**Section D: Discussion on Solving the Same Problem in OOP Way**

Inheritance

Inheritance is an OOP feature allowing new classes to receive, or inherit, properties and methods from existing classes. In Java, this can be employed to create a hierarchical relationship between superclass and subclasses. For example, we could have a general class called RiverBankEntity from which specific entities like Farmer, Wolf, Goat, and Cabbage inherit. This setup promotes reusability and a clearer organizational structure.

Polymorphism

Polymorphism allows methods to do different things based on the object it is acting upon, which means the same method name can have different behaviors when used with objects of different classes. In the context of this coursework, this could mean having a move method in each entity class (Farmer, Wolf, etc.) that behaves slightly differently per entity, such as different rules about what can be moved simultaneously.

A screenshot of a computer screen

Description automatically generated

Function Overloading

Function overloading involves having multiple functions with the same name but different parameters within the same class. This allows functions to handle different types and numbers of inputs. In Java, we could overload the move() function to handle different scenarios, such as moving the farmer alone, or the farmer with the wolf, goat, or cabbage.

Imperative Programming Paradigms:

This style, typical in OOP languages like Java, involves writing code that describes the mechanics of how to perform operations. It focuses on describing the steps that change the program's state. In the Farmer Crosses River puzzle, this could manifest in methods that explicitly modify the positions of the entities.

Declarative Programming Paradigms:

In contrast, declarative programming focuses on what should be accomplished without necessarily specifying how. This style is common in functional programming, where we might describe a series of transformations on data rather than direct data manipulation.