Høgskolen i Ålesund
Bikram Kawan
October 10, 2015

Assignment 2

Games of Drones

Scenario

The scenario consist zones and drones. Drones can move into the zones and land on the zone. This is the multiplayer game. I will be playing against the AI (default) of codingame. I need to score more than other AI to win the battle. The more zones i own the more points i will receive.

We have some constraints as follows.

- The number of drones are between 3 and 11.
- Drone must fly over radius of 100 units around its center of zone to own that zone.
- When you are only player on that zone you own that zone.
- If the zone is previously occupied by other player you must have number of drones greater than the number of drones belonging to other player.
- The field is 4000 units by 1800 unit.
- The number of players will be 2 to 4.
- The id of yourself will be either 0,1,2, or 3
- The number of drones for each player will be 3 to 11.
- The number of zones on the field will be 4 to 8
- Collision is not necessary to consider
-

Initialization intelligence

We get inputs from codingame in sequential order as follows.

- Number of Players
- Id of yourself
- Number of drones for each player
- Number of zones
- The position of zones in map (field)
- The zone teams

- The position of drones of each players.

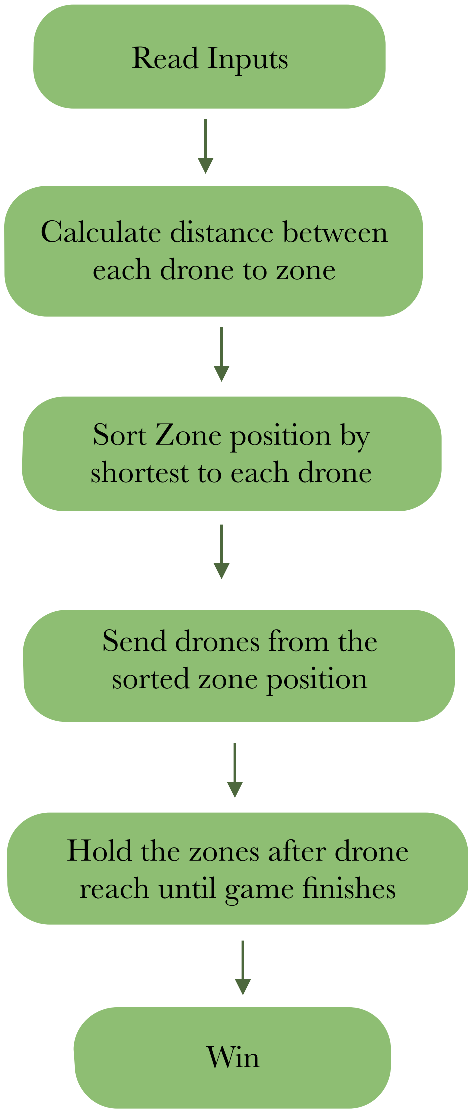
Output

The output should be position of zone.

Example: 1011 2045

Strategy I used for winning

The information I obtained from input about my drones position is sorted by the nearest to the zones position. I let the drones moves towards the nearest zone position.Once my drone reach to the zone my drones just stick on the zone which they have reached.The default AI lost the battle because the drones keep moving from zone to zone and loses the point.


Read Inputs
Calculate distance between each drone to zone
Sort Zone position by shortest to each drone
Send drones from the sorted zone position
Hold the zones after drone reach until game finishes
Win

Haskell Implementation

Step 1:Read Inputs

I define my own data type to read inputs from the coding game and later i stored my states. The

```
data GameState = GameState { playerNum     :: Int
                           , myID          :: Int
                           , droneNum      :: Int
                           , zoneNum       :: Int
                           , zonePositions :: [Position]
                           , zoneTeams     :: [Int]
                           , playerDrones  :: [[Position]]
                           } deriving (Show)
```

data I define looks like

In the main function i read all the inputs from the game and stored in this data type respectively as :

PlayerNum = Number of players

myID = player id for myself

droneNum = Number of drones to each players

zoneNum = number of zones

zonePositions = the coordinate for the zone in map

zoneTeams = zone owned by team

playerDrones = position of drones of each player

Step 2 :Calculate Distance

The second step is to calculate the distance between each drone to each zone positions. We use the distance between two point formula as

```
distance2 :: Position -> Position -> Int
distance2 (x1, y1) (x2, y2) = (x1 - x2)^2 + (y1 - y2)^2
```

We only need to compare between two position so we didn't bother calculating actual distance

using map function we calculate distance from one drone to all zone position and generate list.

Step 3 : Sorting Distance

We sorted the nearest position by comparing distance and the list is sorted according to the sorted distance.

To calculate distance from each drone to each zone and sort according to their position is done by

```
sortAllPositions :: [Position] -> [Position] -> [[Position]]
sortAllPositions  mydronepos zPositions = dr3 4 $  map (nearestPosition zPositions) mydronepos
```

The output looks something like below:

The above function returns like this when number of drones is 3:

```
[[(1864,1693),(2498,1033),(759,791),(3117,1538)]]
```

This is the position of zones which are closest to the drone1 , drone 2 and so on.

I chopped the long list using the following function

```
dr3 m = map snd . filter ((== 1) . fst) . zip (cycle [1..m])
```

Before I use this function the output from sortAllPosition looks like when drone number is 7

```
[[(2898,1197),(1837,1248),(1319,399),(359,1460)],[(359,1460),(1837,1248),(1319,399),
(2898,1197)],[(2898,1197),(1837,1248),(1319,399),(359,1460)],[(2898,1197),(1837,1248),
(1319,399),(359,1460)],[(1837,1248),(2898,1197),(1319,399),(359,1460)],[(1319,399),
(1837,1248),(2898,1197),(359,1460)],[(2898,1197),(1837,1248),(1319,399),(359,1460)]]
```

This is list of list. I am only interested in the first zone position from each list of big list.

So i get this by taking first zone position from first list and first zone position from second list and so on.

This will return seven zone position for seven drones.

Step 4: Preparing for Output

The print should be in string.The content in list are in Int. We convert int to string using show.This need to apply for each zone position so we use map.

The list of position is given by sortAllPosition which is fed to displayPositions function

We fed number of drones and sorted position of zones. We simply use mapM_ function since we do not need return type (results). We generate list of position with the number of drones. To make sufficient enough we use simple mathematics to make sure the zone position is long enough for drones.

```
-- Prints drone target positions
displayPositions :: Int -> [Position] -> IO()
displayPositions d zPositions = mapM_ putStrLn posList
                                where
                                 -- List of positions with length >= number of drones d
                                 posList = take d $ concat $ take n (repeat positionsAsStrings)

                                 -- Calculate n so posList is sufficiently long
                                 n = d `div` (length zPositions) + 1 :: Int
```

For game loop we have created the loop called gamestate. When the drones moves the position gets updated and other are also changed accordingly.

Paranoid Android

Scenario:

It is provided some floors with elevators , exit floor. I need to guide clones to reach the exit to make them escape. If I able to make escape clones then I win the game.

The clones are generated in every three game turns.

If clone tries to move beyond width-1 then it is dead.

Elevators are used to move from one floor to another

The leading clone can make wall indicating no path exists and take next direction.

If leader mistakenly make wall before elevator no one can use elevator hence will loose game.

When clone reaches the exit you save a clone which we want to do in our game.

Constraints	Description
1 <= nbFloors <= 15	Total number of floors
5 <= width <= 100	Width of the grid
10 <= nbRounds <= 200	number of rounds
0 <= exitFloor, elevatorFloor < nbFloors	Exit floor and elevator in floor is less than number of floors
0 <= exitPos , elevatorPos < width	exit position and elevator position is within grid
-1 <= cloneFloor < nbFloors	The floors where the clones are
-1 <= clonePos < width	The positions of the clones
2 <= nbTotalClones <= 50	Total number of clones
0 <= nbElevators <= 100	Total number of elevators
Duration of one game turn: 100 ms	

To win ?

We need to put blocks or wait correctly so that clones can reach the exit in limited time (game rounds).

Initialization

I receive inputs from codingame in following order:

###From Codingame Description###

Line 1: 8 integers:

nbFloors : number of floors in the area. A clone can move between floor 0 and floor (nbFloors - 1)

width : the width of the area. The clone can move without being destroyed between position 0 and position (width - 1)

nbRounds : maximum number of rounds before the end of the game exitFloor : the floor on which the exit is located

exitPos : the position of the exit on its floor

nbTotalClones : the number of clones that will come out of the generator during the game

nbAdditionalElevators : not used for this first question, value is always 0

nbElevators : number of elevators in the area

Line 2:

elevatorFloor - Elevator Floor

elevatorPos - position of an elevator.

Strategy I used for for winning

First I check if the floor of leading clone is same as exit then my target for all clones will be exit position of that floor. Other wise my target will be the elevator of the same floor where leading clone is.

Now to give directions for my clones, I use couple of test criteria as

If target position is greater than the position of leading clone and direction is left then i make wall. Also if target position is less than position of leading clone and direction is right i make wall. If this two conditions are not true clones will wait.

Haskell Implementation:

Step 1:Read Inputs

I define my own data type to read inputs from the coding game and later i stored my states. The data I define looks like

```
data GameState = GameState { nbfloors :: Int
                           , width    :: Int
                           , nbrounds :: Int
                           , exitfloor:: Int
                           , exitpos :: Int
                           , nbtotalclones ::Int
                           , nbadditionalelevators     :: Int
                           , nbelevators  :: Int
                           ,elevators ::[[Int]]
                           ,target::Int
                           } deriving (Show)
```

In the main function i read all the inputs from the game and stored in this data type respectively

nbFloors : number of floors in the area.

width : the width of the area.

nbRounds : maximum number of rounds before the end of the game

exitFloor : the floor on which the exit is located

exitPos : the position of the exit on its floor

nbTotalClones : the number of clones that will come out of the generator during the game

nbAdditionalElevators : not used for this first question, value is always 0

nbElevators : number of elevators in the area

elevators - elevators list with position

target = target

Step 2 : Check target Position

```
let targetpos = if (clonefloor==extf ) then exs else ((ev!!clonefloor)!!1)
```

I compared the floor of leading clone and exit. If this is true then i make target position equal to the exit position Else the target position will be the position of elevator where the clone floor.

I make list of list of list for clone and respective elevator position

```
elevators1<-replicateM ne $ do
        input_line <- getLine
        let input = words input_line
        let elevatorfloor = read (input!!0) :: Int -- floor on which this elevator is found
        let elevatorpos = read (input!!1) :: Int -- position of the elevator on its floor
        let tmp =[]::[Int]
        let dtmp= elevatorfloor:elevatorpos:tmp
        hPutStrLn stderr $show dtmp ++"dtmp"
        return (dtmp)
```

The output of list of list for elevator

```
[[3,10],[4,12],[6,12],[5,10],[1,10],[0,12],[7,10],[2,12]]
```

The position of elevator for floor 3 is 10 and so on.

Step 3: Preparing for Output

The first condition is to check whether the target position is greater or less than position of clone and the direction. We make wall if the target position is greater and direction is left. Similarly wall is build if the target position is less than and direction is right. IF this conditions are not true the clone will just wait.

```
if (clonefloor >= 0 )
      then if (targetpos > clonepos && direction =="LEFT")
            then putStrLn "BLOCK"
            else if (targetpos < clonepos && direction =="RIGHT")
            then putStrLn "BLOCK"
            else putStrLn "WAIT"
            else putStrLn "WAIT"
```