KAUNO TECHNOLOGIJOS UNIVERSITETAS

FAKULTETAS

Programavimo kalbų teorija (P175B124)

Laboratorinių darbų ataskaita

Atliko:

IFF-4/1 gr. studentas

Mangirdas Kazlauskas

2017 m. gegužės 31 d.

Priėmė:

Lekt. Guogis Evaldas

KAUNAS 2017

TURINYS

1. Python (L1) 3

1.1. Darbo užduotis 3

1.2. Programos tekstas 4

1.3. Pradiniai duomenys ir rezultatai 6

2. Scala (L2) 7

2.1. Darbo užduotis 7

2.2. Sukurto Scalatron žaidimo roboto kodas 7

2.3. Kai kurių sukurto roboto programos kodo vietų paaiškinimai 19

3. Haskell (L3) 21

3.1. Darbo užduotis 21

3.2. Programos tekstas 22

3.3. Pradiniai duomenys ir rezultatai 22

4. Prolog (L4) 23

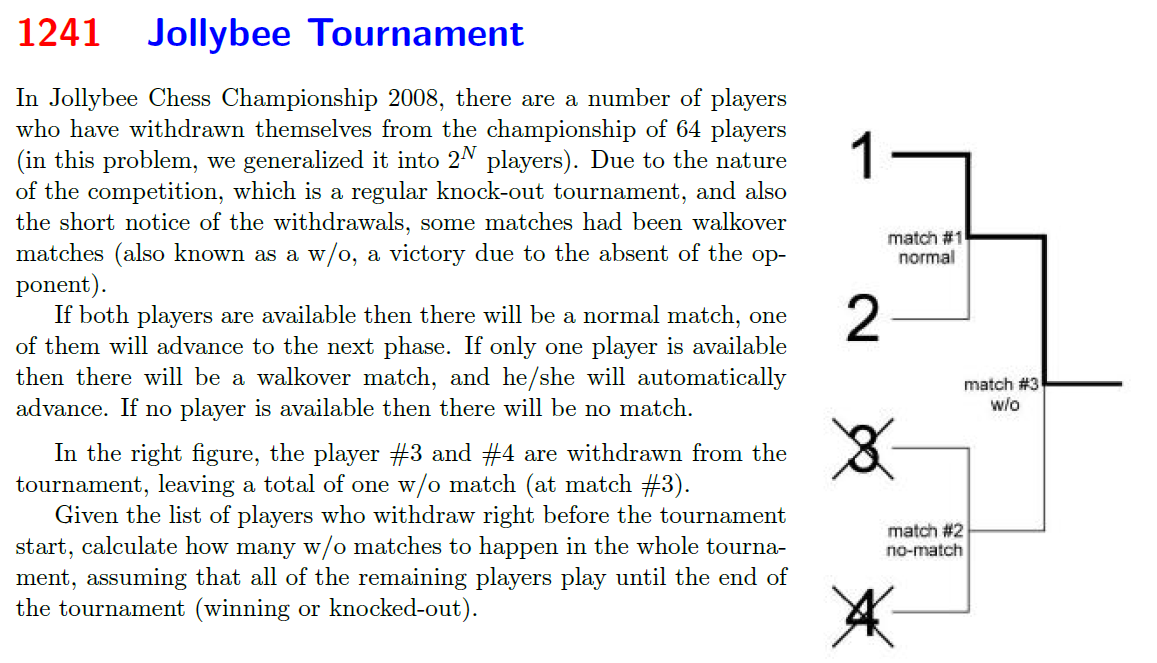
4.1. Darbo užduotis 23

4.2. Programos tekstas 23

# Python (L1)

## Darbo užduotis

Laboratorinio darbo atlikimui pasirinkau Python programavimo kalbą (3.5 versiją) bei iš puslapio <https://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8> kategorijos „Problem Set Volumes (100…1999) pasirinkau išspręsti 1241 užduotį, kurios angliška formuluotė tokia:



pav. 1241 “Jollybee Tournament” užduoties sąlyga

**Trumpas užduoties paaiškinimas:** šachmatų turnyre, kuriame dalyvių skaičius yra 2N, dalyviai rungiasi su kitais dalyviais vienas prieš vieną vieno minuso turnyre (jei žaidėjas pralaimi – jis iškrenta iš turnyro). Jei abu žaidėjai gali rungtis tarpusavyje, laikoma, kad tai yra normalios rungtynės, o jei abu žaidėjai žaisti negali – rungtynės neįvyksta. Tačiau turnyre galimas toks atvejis, kai žaidėjui užskaitoma techninė pergalė, jei jo priešininkas atsisako ar dėl kitos priežasties negali rungtis.

**Užduoties tikslas:** suskaičiuoti, kiek turnyro metu žaidėjams bus skirta techninių pergalių.

**Užduoties sprendimo idėja:** pirmiausia reikia sudaryti turnyro žaidėjų sąrašą. Kadangi turnyro žaidėjai numeruojami skaičiais nuo 1 iki 2N, tai i-tąjį turnyro žaidėją atitiks (i-1)-asis sąrašo elementas. Pradinis dalyvių sąrašas suformuojamas tokiu principu: jeigu žaidėjas dalyvauja turnyre, jį atitinkančioje sąrašo vietoje įrašomas 1, priešingu atveju (jeigu žaidėjas rungtyniauti atsisakė) – 0. Techninės pergalės skaičiuojamos taip:

1. Formuojamas naujas dalyvių sąrašas, kuris bus dvigubai trumpesnis nei prieš tai buvęs (nes po kiekvieno varžybų turo pusė dalyvių iškrenta iš turnyro);
2. Einama per dalyvių sąrašą nuo pradžios, imant po du elementus, kurie atitinka turnyro rungtynių poras;
3. Kiekvienai rungtynių porai taikoma XOR operacija, t.y. jeigu dalyvius atitinkančių sąrašo elementų reikšmės skiriasi, reiškia, kad vienas dalyvis negali dalyvauti, todėl vienam iš dalyvių skiriama techninė pergalė. Į naujai sudaromą dalyvių sąrašą įrašoma reikšmė, atitinkamai pagal reikšmes iš 1 lentelės:

lentelė. Naujai formuojamo sąrašo reikšmė pagal poros dalyvių reikšmes

|  |  |  |
| --- | --- | --- |
| 1-ąjį poros dalyvį atitinkanti reikšmė sąraše | 2-ąjį poros dalyvį atitinkanti reikšmė sąraše | Į naujai formuojamą sąrašą įrašoma reikšmė |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

1. 2-3 veiksmai kartojami tol, kol sąrašo ilgis nelygus 1 (kai sąrašo ilgis lygus 1, tai atitinka, jog turnyras baigėsi ir jau aiškus turnyro nugalėtojas).

## Programos tekstas

"""

Problem: 1241 - Jollybee Tournament

Link: https://uva.onlinejudge.org/index.php?option=com\_onlinejudge&Itemid=8&category=247&page=show\_problem&problem=3682

Mangirdas Kazlauskas IFF-4/1

"""

from math import pow

class Tournament:

def \_\_init\_\_(self, n, m\_list):

self.\_n = n

self.\_m\_list = m\_list

def n(self):

return self.\_n

def m\_list(self):

return self.\_m\_list

def read\_input():

t = int(input())

n\_list = list()

m\_lists = list()

for i in range(t):

n, \_ = [int(x) for x in input().split()]

m\_list = [int(x) for x in input().split()]

n\_list.append(n)

m\_lists.append(m\_list)

return n\_list, m\_lists

def print\_output(result):

for i in result:

print(i)

def read\_from\_file(filename):

with open(filename, mode='rt') as f:

t = int(next(f))

n\_list = list()

m\_lists = list()

for i in range(t):

n, \_ = [int(x) for x in next(f).split()]

m\_list = [int(x) for x in next(f).split()]

n\_list.append(n)

m\_lists.append(m\_list)

return n\_list, m\_lists

def print\_to\_file(filename, results):

with open(filename, mode='wt') as f:

f.writelines("{}\n".format(i) for i in results)

def init\_tournaments(n\_list, m\_lists):

tournaments = [Tournament(n\_list[x], m\_lists[x]) for x in range(len(n\_list))]

return tournaments

def calculate\_walkover\_matches(tournament):

walkover\_matches\_counter = 0

participants\_list = [0 if i+1 in tournament.m\_list() else 1 for i in range(int(pow(2, tournament.n())))]

while len(participants\_list) > 1:

iterator = iter(participants\_list)

new\_participants\_list = list()

for i in iterator:

status, increment = get\_match\_status(i, next(iterator))

new\_participants\_list.append(status)

walkover\_matches\_counter += increment

participants\_list = new\_participants\_list

return walkover\_matches\_counter

def get\_match\_status(x, y):

if x != y:

return 1, 1

elif x == y and x == 1:

return 1, 0

else:

return 0, 0

def main(io\_file=True, input\_file='data.txt', output\_file='results.txt'):

input\_output\_file = io\_file

if input\_output\_file:

n\_list, m\_lists = read\_from\_file(input\_file)

else:

n\_list, m\_lists = read\_input()

tournaments = init\_tournaments(n\_list, m\_lists)

walkover\_matches\_list = [calculate\_walkover\_matches(tournament) for tournament in tournaments]

if input\_output\_file:

print\_to\_file(output\_file, walkover\_matches\_list)

else:

print\_output(walkover\_matches\_list)

if \_\_name\_\_ == '\_\_main\_\_':

main()

## Pradiniai duomenys ir rezultatai

**Pradiniai duomenys:**

3

2 2

3 4

3 5

1 2 3 4 5

2 1

2

**Rezultatai:**

1

2

1

**Paaiškinimas:** pirmoje duomenų failų eilutėje duotas sveikasis skaičius *t*, nurodantis, kiek bus nagrinėjama testavimo atvejų (kiek bus nagrinėjama turnyrų). Kiekvienas nagrinėjamas atvejis prasideda dviem sveikaisiais skaičiais *N* ir *M .* Kitoje nagrinėjamo atvejo (turnyro) eilutėje pateikta *M* sveikųjų skaičių, atitinkančių numerius žaidėjų, kurie atsisakė ar dėl kitų priežasčių negali dalyvauti turnyre.

# Scala (L2)

## Darbo užduotis

Naudojantis žaidimo kūrimo imitatoriumi „Scalatron“, suprogramuoti žaidimo robotą, išsiaiškinti Scala kalbos sintaksę bei plačiau panagrinėti šią programavimo kalbą.

## Sukurto Scalatron žaidimo roboto kodas

// Mangirdas Kazlauskas IFF-4/1

/\*\* This bot builds a 'direction value map' that assigns an attractiveness score to

\* each of the eight available 45-degree directions. Additional behaviors:

\* - aggressive missiles: approach an enemy master, then explode

\* - defensive missiles: approach an enemy slave and annihilate it

\*

\* The master bot uses the following state parameters:

\* - dontFireAggressiveMissileUntil

\* - dontFireDefensiveMissileUntil

\* - lastDirection

\* The mini-bots use the following state parameters:

\* - mood = Aggressive | Defensive | Lurking

\* - target = remaining offset to target location

\*/

import util.Random;

import scala.collection.mutable.ListBuffer

import scala.collection.mutable.ArrayBuffer

import scala.collection.mutable.Queue

import scala.collection.mutable.Stack

import scala.util.control.\_

object ControlFunction

{

val rand = new Random();

def forMaster(bot: Bot) {

val (directionValue, nearestEnemyMaster, nearestEnemySlave, nearestPlant, nearestEnemyMasterIndex) = analyzeViewAsMaster(bot.view)

val dontFireAggressiveMissileUntil = bot.inputAsIntOrElse("dontFireAggressiveMissileUntil", -1)

val dontFireDefensiveMissileUntil = bot.inputAsIntOrElse("dontFireDefensiveMissileUntil", -1)

val dontSpawnGatheringBotUntil = bot.inputAsIntOrElse("dontSpawnGatheringBotUntil", -1)

val dontThrowMinesUntil = bot.inputAsIntOrElse("dontThrowMinesUntil", -1)

val lastDirection = bot.inputAsIntOrElse("lastDirection", 0)

// determine movement direction

directionValue(lastDirection) += 10 // try to break ties by favoring the last direction

val bestDirection45 = directionValue.zipWithIndex.maxBy(\_.\_1).\_2

val direction = XY.fromDirection45(bestDirection45)

bot.set("lastDirection" -> bestDirection45)

// BFS based movement

if (nearestPlant != -1){

val bfs = BreadthFirstSearch(bot, bot.viewString)

bfs.stringViewToMatrix()

bfs.findAllAdjacentNodes()

bfs.findPath(480, nearestPlant)

val bfsPath = bfs.path.toString

bot.log("bfsPath:\n" + bfsPath)

val directions = bfsPath.split(";")

var moves = new ArrayBuffer[XY]()

for (i <- 0 to directions.length-1) moves += XY(directions(i))

bot.move(moves(0))

}

// Agressive missile

if(dontFireAggressiveMissileUntil < bot.time && bot.energy > 100) { // fire attack missile?

nearestEnemyMaster match {

case None => // no-on nearby

case Some(relPos) => // a master is nearby

val unitDelta = relPos.signum

val remainder = relPos - unitDelta // we place slave nearer target, so subtract that from overall delta

bot.spawn(unitDelta, "mood" -> "Aggressive", "target" -> remainder)

bot.set("dontFireAggressiveMissileUntil" -> (bot.time + relPos.stepCount + 1))

}

}

// Deffensive missile

else

if(dontFireDefensiveMissileUntil < bot.time && bot.energy > 100) { // fire defensive missile?

nearestEnemySlave match {

case None => // no-on nearby

case Some(relPos) => // an enemy slave is nearby

if(relPos.stepCount < 8) {

// this one's getting too close!

val unitDelta = relPos.signum

val remainder = relPos - unitDelta // we place slave nearer target, so subtract that from overall delta

bot.spawn(unitDelta, "mood" -> "Defensive", "target" -> remainder)

bot.set("dontFireDefensiveMissileUntil" -> (bot.time + relPos.stepCount + 1))

}

}

}

// Food gatherer

if(dontSpawnGatheringBotUntil < bot.time && bot.energy > 100){

bot.spawn(bot.view.center, "mood" -> "Gathering", "target" -> "", "limit" -> 500 \* (rand.nextInt(4) + 1))

bot.set("dontSpawnGatheringBotUntil" -> (bot.time + rand.nextInt(2)))

}

// Mines

if(dontThrowMinesUntil < bot.time && bot.energy > 100 && nearestEnemyMasterIndex != -1){

val bfsForMines = BreadthFirstSearch(bot, bot.viewString)

bfsForMines.stringViewToMatrix()

bfsForMines.findAllAdjacentNodes()

bfsForMines.findPath(480, nearestEnemyMasterIndex)

val bfsMinesPath = bfsForMines.path.toString

val places = bfsMinesPath.split(";")

var mines = new ArrayBuffer[XY]()

for (i <- 0 to places.length-1) mines += XY(places(i))

nearestEnemyMaster match{

case None => // do nothing (don't throw)

case Some(relPos) =>

bot.spawn(mines(0), "mood" -> "Mine")

bot.set("dontThrowMinesUntil" -> (bot.time + relPos.stepCount + 1))

}

}

bot.move(direction)

}

def forSlave(bot: MiniBot) {

bot.inputOrElse("mood", "Gathering") match {

case "Aggressive" => reactAsAggressiveMissile(bot)

case "Defensive" => reactAsDefensiveMissile(bot)

case "Gathering" => reactAsGatheringBot(bot)

case "Returning" => reactAsReturningBot(bot)

case "Mine" => reactAsMine(bot)

case s: String => bot.log("unknown mood: " + s)

}

}

def reactAsMine(bot: MiniBot) {

bot.view.offsetToNearest('m') match {

case Some(delta: XY) =>

// another master is visible at the given relative position (i.e. position delta)

// close enough to blow it up?

if(delta.length <= 1) {

// yes -- blow it up!

bot.explode(2)

}

case None =>

// no target visible -- follow our targeting strategy

// yes -- but we did not detonate yet, and are not pursuing anything?!? => switch purpose

bot.set("mood" -> "Returning", "target" -> "")

bot.say("Returning")

}

}

def reactAsAggressiveMissile(bot: MiniBot) {

bot.view.offsetToNearest('m') match {

case Some(delta: XY) =>

// another master is visible at the given relative position (i.e. position delta)

// close enough to blow it up?

if(delta.length <= 2) {

// yes -- blow it up!

bot.explode(4)

} else {

// no -- move closer!

bot.move(delta.signum)

bot.set("rx" -> delta.x, "ry" -> delta.y)

}

case None =>

// no target visible -- follow our targeting strategy

val target = bot.inputAsXYOrElse("target", XY.Zero)

// did we arrive at the target?

if(target.isNonZero) {

// no -- keep going

val unitDelta = target.signum // e.g. CellPos(-8,6) => CellPos(-1,1)

bot.move(unitDelta)

// compute the remaining delta and encode it into a new 'target' property

val remainder = target - unitDelta // e.g. = CellPos(-7,5)

bot.set("target" -> remainder)

} else {

// yes -- but we did not detonate yet, and are not pursuing anything?!? => switch purpose

bot.set("mood" -> "Returning", "target" -> "")

bot.say("Returning")

}

}

}

def reactAsDefensiveMissile(bot: MiniBot) {

bot.view.offsetToNearest('s') match {

case Some(delta: XY) =>

// another slave is visible at the given relative position (i.e. position delta)

// move closer!

bot.move(delta.signum)

bot.set("rx" -> delta.x, "ry" -> delta.y)

case None =>

// no target visible -- follow our targeting strategy

val target = bot.inputAsXYOrElse("target", XY.Zero)

// did we arrive at the target?

if(target.isNonZero) {

// no -- keep going

val unitDelta = target.signum // e.g. CellPos(-8,6) => CellPos(-1,1)

bot.move(unitDelta)

// compute the remaining delta and encode it into a new 'target' property

val remainder = target - unitDelta // e.g. = CellPos(-7,5)

bot.set("target" -> remainder)

} else {

// yes -- but we did not annihilate yet, and are not pursuing anything?!? => switch purpose

bot.set("mood" -> "Returning", "target" -> "")

bot.say("Returning")

}

}

}

def reactAsGatheringBot(bot: MiniBot){

val (directionValue, nearestEnemyMaster, \_, master) = analyzeViewAsGatheringBot(bot.view)

if (nearestEnemyMaster.isDefined && nearestEnemyMaster.get.stepCount <= 2){

bot.explode(4)

return

}

val energyLimit = bot.inputAsIntOrElse("limit", 0)

if (bot.energy > energyLimit && !master.isEmpty){

bot.set("mood" -> "Returning", "target" -> "")

reactAsReturningBot(bot)

}

else{

val lastDirection = bot.inputAsIntOrElse("lastDirection", 0)

directionValue(lastDirection) += 10

val bestDirection45 = directionValue.zipWithIndex.maxBy(\_.\_1).\_2

val direction = XY.fromDirection45(bestDirection45)

bot.move(direction)

bot.set("lastDirection" -> bestDirection45)

}

}

def reactAsReturningBot(bot: MiniBot){

val (directionValue, nearestEnemyMaster) = analyzeViewAsReturningBot(bot, bot.view)

if (nearestEnemyMaster.isDefined && nearestEnemyMaster.get.stepCount <= 2){

bot.explode(4)

return

}

val lastDirection = bot.inputAsIntOrElse("lastDirection", 0)

directionValue(lastDirection) += 10

val bestDirection45 = directionValue.zipWithIndex.maxBy(\_.\_1).\_2

val direction = XY.fromDirection45(bestDirection45)

bot.move(direction)

bot.set("lastDirection" -> bestDirection45)

}

/\*\* Analyze the view, building a map of attractiveness for the 45-degree directions and

\* recording other relevant data, such as the nearest elements of various kinds.

\*/

def analyzeViewAsMaster(view: View) = {

val directionValue = Array.ofDim[Double](8)

var nearestEnemyMaster: Option[XY] = None

var nearestEnemyMasterIndex: Int = -1

var nearestEnemySlave: Option[XY] = None

var nearestPlant: Int = -1

val cells = view.cells

val cellCount = cells.length

for(i <- 0 until cellCount) {

val cellRelPos = view.relPosFromIndex(i)

if(cellRelPos.isNonZero) {

val stepDistance = cellRelPos.stepCount

val value: Double = cells(i) match {

case 'm' => // another master: not dangerous, but an obstacle

nearestEnemyMasterIndex = i

nearestEnemyMaster = Some(cellRelPos)

if(stepDistance < 2) -1000 else 0

case 's' => // another slave: potentially dangerous?

nearestEnemySlave = Some(cellRelPos)

-100 / stepDistance

case 'S' => // out own slave

0.0

case 'B' => // good beast: valuable, but runs away

if(stepDistance == 1) 600

else if(stepDistance == 2) 300

else (150 - stepDistance \* 15).max(10)

case 'P' => // good plant: less valuable, but does not run

nearestPlant = i

if(stepDistance == 1) 500

else if(stepDistance == 2) 300

else (150 - stepDistance \* 10).max(10)

case 'b' => // bad beast: dangerous, but only if very close

if(stepDistance < 4) -400 / stepDistance else -50 / stepDistance

case 'p' => // bad plant: bad, but only if I step on it

if(stepDistance < 2) -1000 else 0

case 'W' => // wall: harmless, just don't walk into it

if(stepDistance < 2) -1000 else 0

case \_ => 0.0

}

val direction45 = cellRelPos.toDirection45

directionValue(direction45) += value

}

}

(directionValue, nearestEnemyMaster, nearestEnemySlave, nearestPlant, nearestEnemyMasterIndex)

}

def analyzeViewAsGatheringBot(view: View) = {

val directionValue = Array.ofDim[Double](8)

var nearestEnemyMaster: Option[XY] = None

var nearestEnemyBot: Option[XY] = None

var master: Option[XY] = None

view.cells.zipWithIndex foreach {case (c, i) =>

val cellRelPos = view.relPosFromIndex(i)

if (cellRelPos.isNonZero){

val stepDistance = cellRelPos.stepCount

val value: Double = c match{

case 'W' =>

if (stepDistance < 2) -1000 else 0

case 'm' =>

nearestEnemyMaster = Some(cellRelPos)

-100 / stepDistance

case 's' =>

nearestEnemyBot = Some(cellRelPos)

-100 / stepDistance

case 'S' =>

-50 / stepDistance

case 'M' =>

master = Some(cellRelPos)

0.0

case 'B' =>

if (stepDistance == 1) 600

else if (stepDistance == 2) 400

else (150 - stepDistance \* 15).max(10)

case 'P' =>

if (stepDistance == 1) 500

else if (stepDistance == 2) 300

else (150 - stepDistance \* 10).max(10)

case 'b' =>

if (stepDistance < 4) -400 / stepDistance else -50 / stepDistance

case 'p' =>

if (stepDistance < 2) -1000 else 0

case \_ =>

0.0

}

val direction45 = cellRelPos.toDirection45

directionValue(direction45) += value

}

}

(directionValue, nearestEnemyMaster, nearestEnemyBot, master)

}

def analyzeViewAsReturningBot(bot: MiniBot, view: View) = {

val directionValue = Array.ofDim[Double](8)

var nearestEnemyMaster: Option[XY] = None

view.cells.zipWithIndex foreach {case (c, i) =>

val cellRelPos = view.relPosFromIndex(i)

if (cellRelPos.isNonZero){

val stepDistance = cellRelPos.stepCount

val value: Double = c match{

case 'W' =>

if (stepDistance < 2) -1000 else 0

case 'm' =>

nearestEnemyMaster = Some(cellRelPos)

-100 / stepDistance

case 's' =>

-100 / stepDistance

case 'S' =>

-100 / stepDistance

case 'M' =>

1000

case 'B' =>

if (stepDistance == 1) 600 else 0

case 'P' =>

if (stepDistance == 1) 500 else 0

case 'b' =>

if (stepDistance < 4) -400 / stepDistance else -50 / stepDistance

case 'p' =>

if (stepDistance < 2) -1000 else 0

case \_ =>

0.0

}

val direction45 = cellRelPos.toDirection45

directionValue(direction45) += value

}

}

(directionValue, nearestEnemyMaster)

}

}

// -------------------------------------------------------------------------------------------------

// Framework

// -------------------------------------------------------------------------------------------------

class ControlFunctionFactory {

def create = (input: String) => {

val (opcode, params) = CommandParser(input)

opcode match {

case "React" =>

val bot = new BotImpl(params)

if( bot.generation == 0 ) {

ControlFunction.forMaster(bot)

} else {

ControlFunction.forSlave(bot)

}

bot.toString

case \_ => "" // OK

}

}

}

// -------------------------------------------------------------------------------------------------

trait Bot {

// inputs

def inputOrElse(key: String, fallback: String): String

def inputAsIntOrElse(key: String, fallback: Int): Int

def inputAsXYOrElse(keyPrefix: String, fallback: XY): XY

def view: View

def energy: Int

def time: Int

def generation: Int

def viewString: String

// outputs

def move(delta: XY) : Bot

def say(text: String) : Bot

def status(text: String) : Bot

def spawn(offset: XY, params: (String,Any)\*) : Bot

def set(params: (String,Any)\*) : Bot

def log(text: String) : Bot

}

trait MiniBot extends Bot {

// inputs

def offsetToMaster: XY

// outputs

def explode(blastRadius: Int) : Bot

}

case class BotImpl(inputParams: Map[String, String]) extends MiniBot {

// input

def inputOrElse(key: String, fallback: String) = inputParams.getOrElse(key, fallback)

def inputAsIntOrElse(key: String, fallback: Int) = inputParams.get(key).map(\_.toInt).getOrElse(fallback)

def inputAsXYOrElse(key: String, fallback: XY) = inputParams.get(key).map(s => XY(s)).getOrElse(fallback)

val view = View(inputParams("view"))

val energy = inputParams("energy").toInt

val time = inputParams("time").toInt

val generation = inputParams("generation").toInt

def offsetToMaster = inputAsXYOrElse("master", XY.Zero)

val viewString = inputParams("view")

// output

private var stateParams = Map.empty[String,Any] // holds "Set()" commands

private var commands = "" // holds all other commands

private var debugOutput = "" // holds all "Log()" output

/\*\* Appends a new command to the command string; returns 'this' for fluent API. \*/

private def append(s: String) : Bot = { commands += (if(commands.isEmpty) s else "|" + s); this }

/\*\* Renders commands and stateParams into a control function return string. \*/

override def toString = {

var result = commands

if(!stateParams.isEmpty) {

if(!result.isEmpty) result += "|"

result += stateParams.map(e => e.\_1 + "=" + e.\_2).mkString("Set(",",",")")

}

if(!debugOutput.isEmpty) {

if(!result.isEmpty) result += "|"

result += "Log(text=" + debugOutput + ")"

}

result

}

def log(text: String) = { debugOutput += text + "\n"; this }

def move(direction: XY) = append("Move(direction=" + direction + ")")

def say(text: String) = append("Say(text=" + text + ")")

def status(text: String) = append("Status(text=" + text + ")")

def explode(blastRadius: Int) = append("Explode(size=" + blastRadius + ")")

def spawn(offset: XY, params: (String,Any)\*) =

append("Spawn(direction=" + offset +

(if(params.isEmpty) "" else "," + params.map(e => e.\_1 + "=" + e.\_2).mkString(",")) +

")")

def set(params: (String,Any)\*) = { stateParams ++= params; this }

def set(keyPrefix: String, xy: XY) = { stateParams ++= List(keyPrefix+"x" -> xy.x, keyPrefix+"y" -> xy.y); this }

}

// -------------------------------------------------------------------------------------------------

/\*\* Utility methods for parsing strings containing a single command of the format

\* "Command(key=value,key=value,...)"

\*/

object CommandParser {

/\*\* "Command(..)" => ("Command", Map( ("key" -> "value"), ("key" -> "value"), ..}) \*/

def apply(command: String): (String, Map[String, String]) = {

/\*\* "key=value" => ("key","value") \*/

def splitParameterIntoKeyValue(param: String): (String, String) = {

val segments = param.split('=')

(segments(0), if(segments.length>=2) segments(1) else "")

}

val segments = command.split('(')

if( segments.length != 2 )

throw new IllegalStateException("invalid command: " + command)

val opcode = segments(0)

val params = segments(1).dropRight(1).split(',')

val keyValuePairs = params.map(splitParameterIntoKeyValue).toMap

(opcode, keyValuePairs)

}

}

// -------------------------------------------------------------------------------------------------

/\*\* Utility class for managing 2D cell coordinates.

\* The coordinate (0,0) corresponds to the top-left corner of the arena on screen.

\* The direction (1,-1) points right and up.

\*/

case class XY(x: Int, y: Int) {

override def toString = x + ":" + y

def isNonZero = x != 0 || y != 0

def isZero = x == 0 && y == 0

def isNonNegative = x >= 0 && y >= 0

def updateX(newX: Int) = XY(newX, y)

def updateY(newY: Int) = XY(x, newY)

def addToX(dx: Int) = XY(x + dx, y)

def addToY(dy: Int) = XY(x, y + dy)

def +(pos: XY) = XY(x + pos.x, y + pos.y)

def -(pos: XY) = XY(x - pos.x, y - pos.y)

def \*(factor: Double) = XY((x \* factor).intValue, (y \* factor).intValue)

def distanceTo(pos: XY): Double = (this - pos).length // Phythagorean

def length: Double = math.sqrt(x \* x + y \* y) // Phythagorean

def stepsTo(pos: XY): Int = (this - pos).stepCount // steps to reach pos: max delta X or Y

def stepCount: Int = x.abs.max(y.abs) // steps from (0,0) to get here: max X or Y

def signum = XY(x.signum, y.signum)

def negate = XY(-x, -y)

def negateX = XY(-x, y)

def negateY = XY(x, -y)

/\*\* Returns the direction index with 'Right' being index 0, then clockwise in 45 degree steps. \*/

def toDirection45: Int = {

val unit = signum

unit.x match {

case -1 =>

unit.y match {

case -1 =>

if(x < y \* 3) Direction45.Left

else if(y < x \* 3) Direction45.Up

else Direction45.UpLeft

case 0 =>

Direction45.Left

case 1 =>

if(-x > y \* 3) Direction45.Left

else if(y > -x \* 3) Direction45.Down

else Direction45.LeftDown

}

case 0 =>

unit.y match {

case 1 => Direction45.Down

case 0 => throw new IllegalArgumentException("cannot compute direction index for (0,0)")

case -1 => Direction45.Up

}

case 1 =>

unit.y match {

case -1 =>

if(x > -y \* 3) Direction45.Right

else if(-y > x \* 3) Direction45.Up

else Direction45.RightUp

case 0 =>

Direction45.Right

case 1 =>

if(x > y \* 3) Direction45.Right

else if(y > x \* 3) Direction45.Down

else Direction45.DownRight

}

}

}

def rotateCounterClockwise45 = XY.fromDirection45((signum.toDirection45 + 1) % 8)

def rotateCounterClockwise90 = XY.fromDirection45((signum.toDirection45 + 2) % 8)

def rotateClockwise45 = XY.fromDirection45((signum.toDirection45 + 7) % 8)

def rotateClockwise90 = XY.fromDirection45((signum.toDirection45 + 6) % 8)

def wrap(boardSize: XY) = {

val fixedX = if(x < 0) boardSize.x + x else if(x >= boardSize.x) x - boardSize.x else x

val fixedY = if(y < 0) boardSize.y + y else if(y >= boardSize.y) y - boardSize.y else y

if(fixedX != x || fixedY != y) XY(fixedX, fixedY) else this

}

}

object XY {

/\*\* Parse an XY value from XY.toString format, e.g. "2:3". \*/

def apply(s: String) : XY = { val a = s.split(':'); XY(a(0).toInt,a(1).toInt) }

val Zero = XY(0, 0)

val One = XY(1, 1)

val Right = XY( 1, 0)

val RightUp = XY( 1, -1)

val Up = XY( 0, -1)

val UpLeft = XY(-1, -1)

val Left = XY(-1, 0)

val LeftDown = XY(-1, 1)

val Down = XY( 0, 1)

val DownRight = XY( 1, 1)

def fromDirection45(index: Int): XY = index match {

case Direction45.Right => Right

case Direction45.RightUp => RightUp

case Direction45.Up => Up

case Direction45.UpLeft => UpLeft

case Direction45.Left => Left

case Direction45.LeftDown => LeftDown

case Direction45.Down => Down

case Direction45.DownRight => DownRight

}

def fromDirection90(index: Int): XY = index match {

case Direction90.Right => Right

case Direction90.Up => Up

case Direction90.Left => Left

case Direction90.Down => Down

}

def apply(array: Array[Int]): XY = XY(array(0), array(1))

}

object Direction45 {

val Right = 0

val RightUp = 1

val Up = 2

val UpLeft = 3

val Left = 4

val LeftDown = 5

val Down = 6

val DownRight = 7

}

object Direction90 {

val Right = 0

val Up = 1

val Left = 2

val Down = 3

}

// -------------------------------------------------------------------------------------------------

case class View(cells: String) {

val size = math.sqrt(cells.length).toInt

val center = XY(size / 2, size / 2)

def apply(relPos: XY) = cellAtRelPos(relPos)

def indexFromAbsPos(absPos: XY) = absPos.x + absPos.y \* size

def absPosFromIndex(index: Int) = XY(index % size, index / size)

def absPosFromRelPos(relPos: XY) = relPos + center

def cellAtAbsPos(absPos: XY) = cells.charAt(indexFromAbsPos(absPos))

def indexFromRelPos(relPos: XY) = indexFromAbsPos(absPosFromRelPos(relPos))

def relPosFromAbsPos(absPos: XY) = absPos - center

def relPosFromIndex(index: Int) = relPosFromAbsPos(absPosFromIndex(index))

def cellAtRelPos(relPos: XY) = cells.charAt(indexFromRelPos(relPos))

def offsetToNearest(c: Char) = {

val matchingXY = cells.view.zipWithIndex.filter(\_.\_1 == c)

if( matchingXY.isEmpty )

None

else {

val nearest = matchingXY.map(p => relPosFromIndex(p.\_2)).minBy(\_.length)

Some(nearest)

}

}

}

case class Node(key: Int, value: XY) extends (Int, XY)(key, value)

case class BreadthFirstSearch(bot: Bot, view: String){

val viewMatrix = Array.ofDim[Char](31,31)

var adjacentNodes = new Array[ListBuffer[Node]](961)

var v = View(view)

var center = 480

var directions: List[XY] = List(XY.RightUp, XY.Up, XY.UpLeft, XY.Right,XY.Left,XY.DownRight,XY.Down,XY.LeftDown)

var path = new StringBuilder

def goodTarget(c: Char) = (c == '\_' || c == 'P' || c == 'B')

def stringViewToMatrix() = for (i <- 0 to 30) viewMatrix(i) = bot.viewString.substring(i\*31, (i+1) \* 31).toArray

def findAdjacentNodes(i: Int, j: Int){

var dir = 0

adjacentNodes(i\*31 + j) = new ListBuffer[Node]()

var list = adjacentNodes(i \* 31 + j)

for (ii <- i-1 to i+1; jj <- j-1 to j+1){

if(!(ii == i && jj == j)){

if(ii >= 0 && jj >= 0 && ii < 31 && jj < 31 && goodTarget(viewMatrix(ii)(jj))){

list += Node(ii\*31 + jj, directions(dir))

list.toList

}

dir += 1

}

}

adjacentNodes(i\*31 + j) = list

}

def findAllAdjacentNodes() = for(i <- 0 to 30; j <- 0 to 30) findAdjacentNodes(i, j)

def findPath(source: Int, destination: Int){

if (source == destination) return

var shortestPath: ArrayBuffer[Node] = new ArrayBuffer[Node]()

var visited: Array[Boolean] = new Array[Boolean](961)

for (i <- 0 to 960) visited(i) = false

var queue = Queue[Node]()

var pathStack = Stack[Node]()

var sourceNode = Node(source, XY(0,0))

queue.enqueue(sourceNode)

pathStack.push(sourceNode)

visited(source) = true

val inner = new Breaks;

val outer = new Breaks;

while(!queue.isEmpty){

var list = adjacentNodes(queue.dequeue.key).toList

inner.breakable{

for(ppp <- list){

if(visited(ppp.key) == false){

visited(ppp.key) = true

queue.enqueue(ppp)

pathStack.push(ppp)

if(ppp.key == destination){

inner.break;

}

}

}

}

}

var node: Node = Node(0, XY(0,0))

var currentSource = Node(destination, XY(0,0))

shortestPath += currentSource

outer.breakable{

while(!pathStack.isEmpty){

node = pathStack.pop()

var list = adjacentNodes(currentSource.key).toList

inner.breakable{

for(ppp <- list){

if(ppp.key == node.key){

shortestPath += node

currentSource = node

if (node.key == source){

inner.break

outer.break

}

}

}

}

}

}

for (vvv <- shortestPath.toArray.reverse) path ++= vvv.value.toString() + ";"

path.toString

}

}

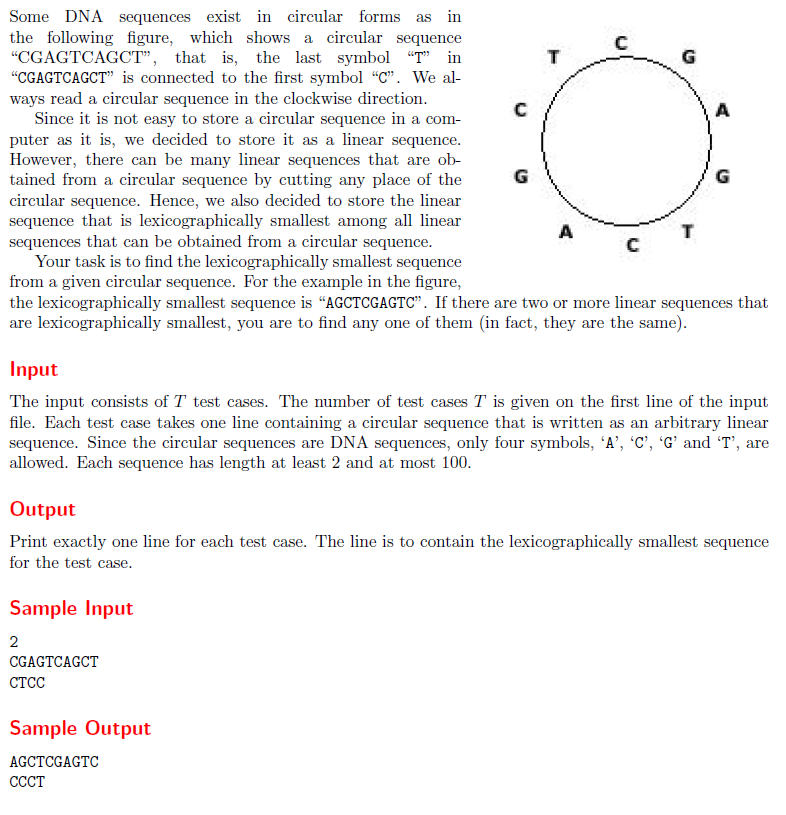
## Kai kurių sukurto roboto programos kodo vietų paaiškinimai

Žaidimo robotas buvo kuriamas jau paties Scalatron pateikto „reference bot“ pagrindu, todėl roboto kūrimui buvo naudojamas jau sukurtas karkasas, aprašantis roboto judėjimo kryptis bei žaidimo lauką (aplinką). Papildomai prie roboto kodo buvo pridėta galimybė robotui į aplinką paleisti mini-robotus, kurie juda link maisto ir jį parneša pagrindiniam robotui (master). Tam tiklsui buvo sukurtas papildomos mini-robotų būsenos „Gathering“ ir „Returning“ bei funkcijos „reactAsGatheringBot(bot)“ bei „reactAsReturningBot(bot)“. Taip pat reikėjo papildomai sukurti funkcijas, reikalingas mini-robotų aplinkos analizei, t.y. „analyzeViewAsGatheringBot“ bei „analyzeViewAsReturningBot“. Analizės funkcijos robotams padeda pasirinkti palankiausią kryptį, naudojamą kitame ėjime (kitos iteracijos metu). „Gathering“ būsenos mini-robotui aukščiausias prioritetas – maistas, kurį jie turi surinkti, kartu išvengiant priešininko mini robotų bei pavojingų pabaisų. „Returning“ mini boto aukščiausias prioritetas yra surinkus maistą grįžti pas didįjį robotą (master). Tačiau, jei pakeliui link „master“ roboto pasitaiko maisto (poros žingsnių atstumu), „Returning“ mini robotas jį stengiasi paimti.

Taip pat žaidime buvo implementuotas paieškos į plotį algoritmas, padedantis „master“ robotui surasti artimiausią kelią iki maisto (nejudančio). Tam buvo sukurta papildoma klasė „BreadthFirstSearch“, aprašanti paieškos į plotį algoritmą, ir klasė „Node“, aprašanti vieną grafo mazgą.

# Haskell (L3)

## Darbo užduotis

Pasirinkta 1584 užduotis, kurios formuluotė tokia:

## Programos tekstas

{-

Mangirdas Kazlauskas IFF-4/1

L3, Haskell

Užduotis: 1584 - Circular Sequence

-}

import System.IO

import Data.List

-- Add all possible circular sequence orders to list

testCaseToList :: String -> Int -> [String]

testCaseToList caseString counter =

if counter == 0

then []

else nextWord : (testCaseToList nextWord (counter - 1))

where nextWord = ((tail caseString) ++ ((head caseString):[]))

-- Find lexicographically smallest sequence

findSmallestSequence :: String -> String

findSmallestSequence word = head (sort (testCaseToList word (length word)))

-- Main recursive loop

findAllSmallestSequences :: [String] -> [String]

findAllSmallestSequences [] = []

findAllSmallestSequences (x:xs) = (findSmallestSequence x) : (findAllSmallestSequences xs)

-- Main program

main = do

input <- readFile "data.txt"

putStr (unlines(findAllSmallestSequences [x | x <- (tail (lines input))]))

## Pradiniai duomenys ir rezultatai

Pradiniai duomenys:

5

CGAGTCAGCT

CTCC

ABBAA

BBAZZ

ABABA

Rezultatai:

AGCTCGAGTC

CCCT

AAABB

AZZBB

AABAB

# Prolog (L4)

## Darbo užduotis

Darbo užduoties variantas nustatomas pagal gimimo datą. Mano gimimo data: 1996-07-14, todėl užduoties variantas – 1+9+9+6+0+7+1+4 = 37, 3+7 = 10 – reikia atlikti 10 ir 11 užduotis.

10 užduotis: Suskaičiuokite dviejų dimensijų sąrašo bendrą elementų ilgį.

11 užduotis: Padidinkite sąrašo elementų skaičių pagal pirmą skaičių, pvz.: [[2, "a"], [3, 1]] -> [["aa"][111]]

## Programos tekstas

**10 užduotis:**

/\* Mangirdas Kazlauskas IFF-4/1 \*/

/\* 10 užduotis \*/

/\* Programos paleidimas: main([1,2,3,4,[5,6,7]]). \*/

calculate2DListLength([], Length) :-

Length is 0.

calculate2DListLength([Head|Tail], FinalLength) :-

calculate2DListLength(Tail, Length),

sublistLength(Head, SubListLength),

FinalLength is Length + SubListLength.

sublistLength(X, Answer) :-

is\_list(X) -> length(X, Answer); Answer is 1.

main(List) :-

calculate2DListLength(List, Answer),

write(Answer), nl.

**11 užduotis:**

/\* Mangirdas Kazlauskas IFF-4/1 \*/

/\* 11 užduotis \*/

/\* Programos paleidimas: main([[1, 'a'], [2, 1]]). \*/

buildListOfLists([],[]).

buildListOfLists([Head|Tail], FinalList) :-

buildListOfLists(Tail, List),

[Times|Symbol] = Head,

buildListWithSameValues(Symbol, Times, SubList),

FinalList = [SubList|List].

buildListWithSameValues(\_,0,[]).

buildListWithSameValues(X,N1,[X|L]) :-

N1 > 0,

N is N1 - 1,

buildListWithSameValues(X,N,L).

buildFinalList([],[]).

buildFinalList([Head|Tail], FinalList) :-

buildFinalList(Tail, List),

flatten(Head, Temp),

atomic\_list\_concat(Temp, '', String),

FinalList = [String|List].

main(List) :-

buildListOfLists(List, ListOfLists),

buildFinalList(ListOfLists, FinalList),

write(FinalList), nl.