KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS

Programavimo kalbų teorija (P175B124)

Laboratorinių darbų ataskaita

Atliko:
IFF-6/6 gr. studentas
Ignas Jasonas
2019 m. kovo 20 d.
Priėmė:
Doc. Aštrys Kirvaitis

KAUNAS 2019

TURINYS

1. Paveikslėlių sąrašas

2. Scala (L2)

2.1. Darbo užduotis

Panaudojant programavimo įrankį / žaidimo kūrimo imitatorių „Scalatron“ parašyti scala kalba bot‘ą.

Reikalavimai programai/botui

1. Panaudoti bent kelis master boto išleidžiamus botų padėjėjų tipus
(pvz.: minos, raketos į priešus, "kamikadzės", rinkikai, masalas ir pan.)
2. Panaudoti bet kurį vieną iš kelio radimo algoritmų (DFS, BFS, A*, Greedy, Dijkstra).

pav 1. Reikalavimai darbui

2.2. Sprendimas

Suprogramuotas bot‘as atlieka šias funkcijas:

- Ieško maisto panaudojant trumpiausio kelio paieškos algoritmą A*
- Esant arti priešo paleidžia agresyvias, pasyvias arba gynybines raketas, kurios susinaikina esant prie pat priešo (panaudota iš pavyzdinio bot‘o)
- Aplinkoje esant daug maisto paleidžia padėjėją – rinkėją mini bot‘ą, kuris surinkęs tam tikrą kiekį maisto, sugrįžta pas pagrindinį botą. Maisto ieškojimui ir radimui naudoja tą patį algoritmą, kaip pagrindinis bot‘as

2.3. Programos tekstas

```
import Array._
import scala.collection.mutable.ListBuffer
object ControlFunction
{
    def forMaster(bot: Bot) {

        val (directionValue, nearestEnemyMaster, nearestEnemySlave) = analyzeViewAsMaster(bot, XY(-1, -1), true)

        val dontFireAggressiveMissileUntil = bot.inputAsIntOrElse("dontFireAggressiveMissileUntil", -1)
        val dontFireDefensiveMissileUntil = bot.inputAsIntOrElse("dontFireDefensiveMissileUntil", -1)
        val lastDirection = bot.inputAsIntOrElse("lastDirection", 0)

        bot.move(directionValue)

        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))
            }
        }
        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))
                    }
            }
        }
    }
```

```
    def forSlave(bot: MiniBot) {
        bot.inputOrElse("mood", "Lurking") match {
            case "Aggressive" => reactAsAggressiveMissile(bot)
            case "Defensive" => reactAsDefensiveMissile(bot)
            case "Collector" => reactAsCollector(bot)
            case s: String => bot.log("unknown mood: " + s)
        }
    }


    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" -> "Lurking", "target" -> "")
                    bot.say("Lurking")
                }
        }
    }
```

```
def reactAsCollector(bot: MiniBot) {
    if (bot.energy > 1500 && bot.offsetToMaster.x < 10 && bot.offsetToMaster.x > -10 && bot.offsetToMaster.y < 10 && bot.offsetToMaster.y > -10) {
        val (directionValue, nearestEnemyMaster, nearestEnemySlave) = analyzeViewAsMaster(bot, XY(10 + bot.offsetToMaster.x, 10 + bot.offsetToMaster.y))
        bot.move(directionValue)
    }
    else {
        val (directionValue, nearestEnemyMaster, nearestEnemySlave) = analyzeViewAsMaster(bot)
        bot.move(directionValue)
    }

}


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" -> "Lurking", "target" -> "")
                bot.say("Lurking")
            }
    }
}
```

```
def tracePath(bot: Bot, dest: XY, cellDetails: Array[Array[XY]]) =
{
    var row = dest.y;
    var col = dest.x;

    val Path = new ListBuffer[XY]()

    while (!(cellDetails(col)(row).parentROW == row
            && cellDetails(col)(row).parentCOL == col ))
    {
        Path += new XY(col, row)
        var temp_row = cellDetails(col)(row).parentROW;
        var temp_col = cellDetails(col)(row).parentCOL;
        row = temp_row;
        col = temp_col;
    }

    Path += new XY(col, row)
    //Get direction
    var src: XY = Path.last
    Path -= src
    var nextPoint: XY = Path.last

    XY((nextPoint.x - src.x), (nextPoint.y - src.y))
}
def analyzeViewAsMaster(bot: Bot, destination: XY = XY(-1, -1), master: Boolean = false) = {
    val view: View = bot.view

    var directionValue : XY = XY.Down
    var nearestEnemyMaster: Option[XY] = None
    var nearestEnemySlave: Option[XY] = None
    val cells = view.cells
    val cellCount = cells.length
    val N = math.sqrt(cellCount).toInt
    val center = (N - 1) / 2
    var cellDetails = ofDim[XY](N,N)
    var closedList = ofDim[Boolean](N,N)
    var src: XY = new XY(center, center)
    src.f = 0.0
    src.g = 0.0
    src.h = 0.0
    src.parentROW = center
    src.parentCOL = center
    src.f = 0
    var foodAround = 0

    cellDetails(center)(center) = src
    var dest: XY = destination

    for(i <- 0 until cellCount) {
        val cellRelPos = view.relPosFromIndex(i)

        if(cellRelPos.isNonZero) {

            var currentCOL = i % N
            var currentROW = i / N
            var addidionalF = 0.0
            val distance = src.distanceTo(XY(currentCOL, currentROW))
            closedList(currentCOL)(currentROW) = false
            val tempXY = new XY(currentCOL, currentROW)
            val additionalF = cells(i) match {
                case 'm' => // another master: not dangerous, but an obstacle
                    nearestEnemyMaster = Some(cellRelPos)
                    if(distance < 3) 1000 else 100

                case 's' => // another slave: potentially dangerous?
                    100

                case 'S' => // out own slave
                    100

                case 'B' => // good beast: valuable, but runs away
                    if ((!dest.isDefined || src.distanceTo(dest) > src.distanceTo(XY(currentCOL, currentROW))) && !destination.isDefined) dest = dest.update(currentCOL, currentROW)
                    foodAround += 1
                    100
                case 'P' => // good plant: less valuable, but does not run
                    if ((!dest.isDefined || src.distanceTo(dest) > src.distanceTo(XY(currentCOL, currentROW))) && !destination.isDefined) dest = dest.update(currentCOL, currentROW)
                    foodAround += 1
                    100
```

```
            case 'b' => // bad beast: dangerous, but only if very close
                1000

            case 'p' => // bad plant: bad, but only if I step on it
                if(distance < 2) 2 else 100

            case 'W' => // wall: harmless, just don't walk into it
                1000

            case '?' => 1000

            case 'M' => 100

            case _ => 100

        }

        tempXY.f = additionalF
        cellDetails(currentCOL)(currentROW) = tempXY
    }

}
val nextBotTime = bot.inputAsIntOrElse("nextAvailableBotTime", 0)
if(master && foodAround > 7 && bot.energy > 100 && nextBotTime < bot.time) { // deployCollector?
    bot.spawn(XY.Down, "mood" -> "Collector")
    bot.set("nextAvailableBotTime" -> (bot.time + 10))
}
//Jeigu neranda maisto, klajoja
if (!dest.isDefined)
{
    var i = 0
    var iterator = 0
    var yOffsetTop = 0
    var xOffsetRight = 0
    var yOffsetBot = 0
    var xOffsetLeft = 0
    var n = N
    var found = false
    while(!found) {
        for (i <- xOffsetLeft until n - xOffsetRight) {
            if (cellDetails(yOffsetTop)(i).f == 100)
            {
                found = true
                dest = cellDetails(yOffsetTop)(i)
            }
        }
        yOffsetTop += 1
        if (!found)
        {
          for (i <- yOffsetTop until n - yOffsetBot) {
            if (cellDetails(i)(n - 1 - xOffsetRight).f == 100)
            {
                found = true
                dest = cellDetails(i)(n - 1 - xOffsetRight)
            }
          }
          xOffsetRight += 1
        }
```

```
        if (!found)
        {
          iterator = n - 1 - xOffsetRight
          for (i <- xOffsetRight until n - xOffsetLeft) {
            if (cellDetails(n - 1 - yOffsetBot)(iterator).f == 100)
            {
                found = true
                dest = cellDetails(n - 1 - yOffsetBot)(iterator)
            }
            iterator -= 1
          }
          yOffsetBot += 1
        }
        if (!found)
        {
          iterator = n - 1 - yOffsetBot
          for (i <- yOffsetBot until n - yOffsetTop) {
            if (cellDetails(iterator)(xOffsetLeft).f == 100)
            {
                found = true
                dest = cellDetails(iterator)(xOffsetLeft)
            }
            iterator -= 1
          }
          xOffsetLeft += 1
        }
    }
}
val openList = new ListBuffer[XY]()
openList += src

var foundDest = false;

while(!openList.isEmpty) {
    var p: XY = openList.head
    openList -= p

    var currentROW = p.y
    var currentCOL = p.x
    closedList(currentROW)(currentCOL) = true
    var gNew = 0.0
    var hNew = 0.0
    var fNew = 0.0
    if (XY.isValid(currentCOL-1, currentROW, N) == true && !foundDest)
    {
        if (dest.x == currentCOL-1 && dest.y == currentROW)
        {
        // Set the Parent of the destination cell
            cellDetails(currentCOL-1)(currentROW).parentCOL = currentCOL
            cellDetails(currentCOL-1)(currentROW).parentROW = currentROW
            foundDest = true
            directionValue = tracePath(bot, dest, cellDetails)
        }
```

```
    else if (closedList(currentCOL-1)(currentROW) == false &&
         cellDetails(currentCOL-1)(currentROW).f < 1000)
    {
        gNew = cellDetails(currentCOL)(currentROW).g + 1.0
        hNew = XY.calculateHValue(currentCOL-1, currentROW, dest)
        fNew = gNew + hNew

        if (cellDetails(currentCOL-1)(currentROW).f > fNew)
        {
            // Update the details of this cell
            cellDetails(currentCOL-1)(currentROW).f = fNew
            cellDetails(currentCOL-1)(currentROW).g = gNew
            cellDetails(currentCOL-1)(currentROW).h = hNew
            cellDetails(currentCOL-1)(currentROW).parentCOL = currentCOL
            cellDetails(currentCOL-1)(currentROW).parentROW = currentROW
            openList += cellDetails(currentCOL-1)(currentROW)
        }
    }
}
if (XY.isValid(currentCOL+1, currentROW, N) == true && !foundDest)
{
    if (dest.x == currentCOL+1 && dest.y == currentROW)
    {
    // Set the Parent of the destination cell
        cellDetails(currentCOL+1)(currentROW).parentCOL = currentCOL
        cellDetails(currentCOL+1)(currentROW).parentROW = currentROW
        foundDest = true
        directionValue = tracePath(bot, dest, cellDetails)
    }
    else if (closedList(currentCOL+1)(currentROW) == false &&
         cellDetails(currentCOL+1)(currentROW).f < 1000.0)
    {
        gNew = cellDetails(currentCOL)(currentROW).g + 1.00
        hNew = XY.calculateHValue(currentCOL+1, currentROW, dest)
        fNew = gNew + hNew

        if (cellDetails(currentCOL+1)(currentROW).f > fNew)
        {
            // Update the details of this cell
            cellDetails(currentCOL+1)(currentROW).f = fNew
            cellDetails(currentCOL+1)(currentROW).g = gNew
            cellDetails(currentCOL+1)(currentROW).h = hNew
            cellDetails(currentCOL+1)(currentROW).parentCOL = currentCOL
            cellDetails(currentCOL+1)(currentROW).parentROW = currentROW

            openList += cellDetails(currentCOL+1)(currentROW)
        }
    }
}
```

```
if (XY.isValid(currentCOL, currentROW+1, N) == true && !foundDest)
{
    if (dest.x == currentCOL && dest.y == currentROW+1)
    {
    // Set the Parent of the destination cell
        cellDetails(currentCOL)(currentROW+1).parentCOL = currentCOL
        cellDetails(currentCOL)(currentROW+1).parentROW = currentROW
        foundDest = true
        directionValue = tracePath(bot, dest, cellDetails)
    }
    else if (closedList(currentCOL)(currentROW+1) == false &&
         cellDetails(currentCOL)(currentROW+1).f < 1000)
    {
        gNew = cellDetails(currentCOL)(currentROW).g + 1.00
        hNew = XY.calculateHValue(currentCOL, currentROW+1, dest)
        fNew = gNew + hNew

        if (cellDetails(currentCOL)(currentROW+1).f > fNew)
        {
            // Update the details of this cell
            cellDetails(currentCOL)(currentROW+1).f = fNew
            cellDetails(currentCOL)(currentROW+1).g = gNew
            cellDetails(currentCOL)(currentROW+1).h = hNew
            cellDetails(currentCOL)(currentROW+1).parentCOL = currentCOL
            cellDetails(currentCOL)(currentROW+1).parentROW = currentROW

            openList += cellDetails(currentCOL)(currentROW+1)
        }
    }
}

if (XY.isValid(currentCOL, currentROW-1, N) == true && !foundDest)
{
    if (dest.x == currentCOL && dest.y == currentROW-1)
    {
    // Set the Parent of the destination cell
        cellDetails(currentCOL)(currentROW-1).parentCOL = currentCOL
        cellDetails(currentCOL)(currentROW-1).parentROW = currentROW
        foundDest = true
        directionValue = tracePath(bot, dest, cellDetails)
    }
    else if (closedList(currentCOL)(currentROW-1) == false &&
         cellDetails(currentCOL)(currentROW-1).f < 1000.0)
    {
        gNew = cellDetails(currentCOL)(currentROW).g + 1.00
        hNew = XY.calculateHValue(currentCOL, currentROW-1, dest)
        fNew = gNew + hNew

        if (cellDetails(currentCOL)(currentROW-1).f > fNew)
        {
            // Update the details of this cell
            cellDetails(currentCOL)(currentROW-1).f = fNew
            cellDetails(currentCOL)(currentROW-1).g = gNew
            cellDetails(currentCOL)(currentROW-1).h = hNew
            cellDetails(currentCOL)(currentROW-1).parentCOL = currentCOL
            cellDetails(currentCOL)(currentROW-1).parentROW = currentROW
```

```
            openList += cellDetails(currentCOL)(currentROW-1)
        }
    }
}
if (XY.isValid(currentCOL-1, currentROW+1, N) == true && !foundDest)
{
    if (dest.x == currentCOL-1 && dest.y == currentROW+1)
    {
    // Set the Parent of the destination cell
        cellDetails(currentCOL-1)(currentROW+1).parentCOL = currentCOL
        cellDetails(currentCOL-1)(currentROW+1).parentROW = currentROW
        foundDest = true
        directionValue = tracePath(bot, dest, cellDetails)
    }
    else if (closedList(currentCOL-1)(currentROW+1) == false &&
         cellDetails(currentCOL-1)(currentROW+1).f < 1000)
    {
        gNew = cellDetails(currentCOL)(currentROW).g + 1.44
        hNew = XY.calculateHValue(currentCOL-1, currentROW+1, dest)
        fNew = gNew + hNew

        if (cellDetails(currentCOL-1)(currentROW+1).f > fNew)
        {
            // Update the details of this cell
            cellDetails(currentCOL-1)(currentROW+1).f = fNew
            cellDetails(currentCOL-1)(currentROW+1).g = gNew
            cellDetails(currentCOL-1)(currentROW+1).h = hNew
            cellDetails(currentCOL-1)(currentROW+1).parentCOL = currentCOL
            cellDetails(currentCOL-1)(currentROW+1).parentROW = currentROW

            openList += cellDetails(currentCOL-1)(currentROW+1)
        }
    }
}
if (XY.isValid(currentCOL-1, currentROW-1, N) == true && !foundDest)
{
    if (dest.x == currentCOL-1 && dest.y == currentROW-1)
    {
    // Set the Parent of the destination cell
        cellDetails(currentCOL-1)(currentROW-1).parentCOL = currentCOL
        cellDetails(currentCOL-1)(currentROW-1).parentROW = currentROW
        foundDest = true
        directionValue = tracePath(bot, dest, cellDetails)
    }
    else if (closedList(currentCOL-1)(currentROW-1) == false &&
         cellDetails(currentCOL-1)(currentROW-1).f < 1000.0)
    {
        gNew = cellDetails(currentCOL)(currentROW).g + 1.44
        hNew = XY.calculateHValue(currentCOL-1, currentROW-1, dest)
        fNew = gNew + hNew
```

```
        if (cellDetails(currentCOL-1)(currentROW-1).f > fNew)
        {
            // Update the details of this cell
            cellDetails(currentCOL-1)(currentROW-1).f = fNew
            cellDetails(currentCOL-1)(currentROW-1).g = gNew
            cellDetails(currentCOL-1)(currentROW-1).h = hNew
            cellDetails(currentCOL-1)(currentROW-1).parentCOL = currentCOL
            cellDetails(currentCOL-1)(currentROW-1).parentROW = currentROW

            openList += cellDetails(currentCOL-1)(currentROW-1)
        }
    }
}
if (XY.isValid(currentCOL+1, currentROW+1, N) == true && !foundDest)
{
    if (dest.x == currentCOL+1 && dest.y == currentROW+1)
    {
    // Set the Parent of the destination cell
        cellDetails(currentCOL+1)(currentROW+1).parentCOL = currentCOL
        cellDetails(currentCOL+1)(currentROW+1).parentROW = currentROW
        foundDest = true
        directionValue = tracePath(bot, dest, cellDetails)
    }
    else if (closedList(currentCOL+1)(currentROW+1) == false &&
         cellDetails(currentCOL+1)(currentROW+1).f < 1000.0)
    {
        gNew = cellDetails(currentCOL)(currentROW).g + 1.44
        hNew = XY.calculateHValue(currentCOL+1, currentROW+1, dest)
        fNew = gNew + hNew

        if (cellDetails(currentCOL+1)(currentROW+1).f > fNew)
        {
            // Update the details of this cell
            cellDetails(currentCOL+1)(currentROW+1).f = fNew
            cellDetails(currentCOL+1)(currentROW+1).g = gNew
            cellDetails(currentCOL+1)(currentROW+1).h = hNew
            cellDetails(currentCOL+1)(currentROW+1).parentCOL = currentCOL
            cellDetails(currentCOL+1)(currentROW+1).parentROW = currentROW

            openList += cellDetails(currentCOL+1)(currentROW+1)
        }
    }
}
if (XY.isValid(currentCOL+1, currentROW-1, N) == true && !foundDest)
{
    if (dest.x == currentCOL+1 && dest.y == currentROW-1)
    {
    // Set the Parent of the destination cell
        cellDetails(currentCOL+1)(currentROW-1).parentCOL = currentCOL
        cellDetails(currentCOL+1)(currentROW-1).parentROW = currentROW
        foundDest = true
        directionValue = tracePath(bot, dest, cellDetails)
    }
```

```
                else if (closedList(currentCOL+1)(currentROW-1) == false &&
                     cellDetails(currentCOL+1)(currentROW-1).f < 1000)
                {
                    gNew = cellDetails(currentCOL)(currentROW).g + 1.44
                    hNew = XY.calculateHValue(currentCOL+1, currentROW-1, dest)
                    fNew = gNew + hNew

                    if (cellDetails(currentCOL+1)(currentROW-1).f > fNew)
                    {
                        // Update the details of this cell
                        cellDetails(currentCOL+1)(currentROW-1).f = fNew
                        cellDetails(currentCOL+1)(currentROW-1).g = gNew
                        cellDetails(currentCOL+1)(currentROW-1).h = hNew
                        cellDetails(currentCOL+1)(currentROW-1).parentCOL = currentCOL
                        cellDetails(currentCOL+1)(currentROW-1).parentROW = currentROW

                        openList += cellDetails(currentCOL+1)(currentROW-1)
                    }
                }
            }
        }
        (directionValue, nearestEnemyMaster, nearestEnemySlave)
    }
}



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

    // 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
    def move(direction: XY) : Bot
    def log(text: String): Bot
}
```

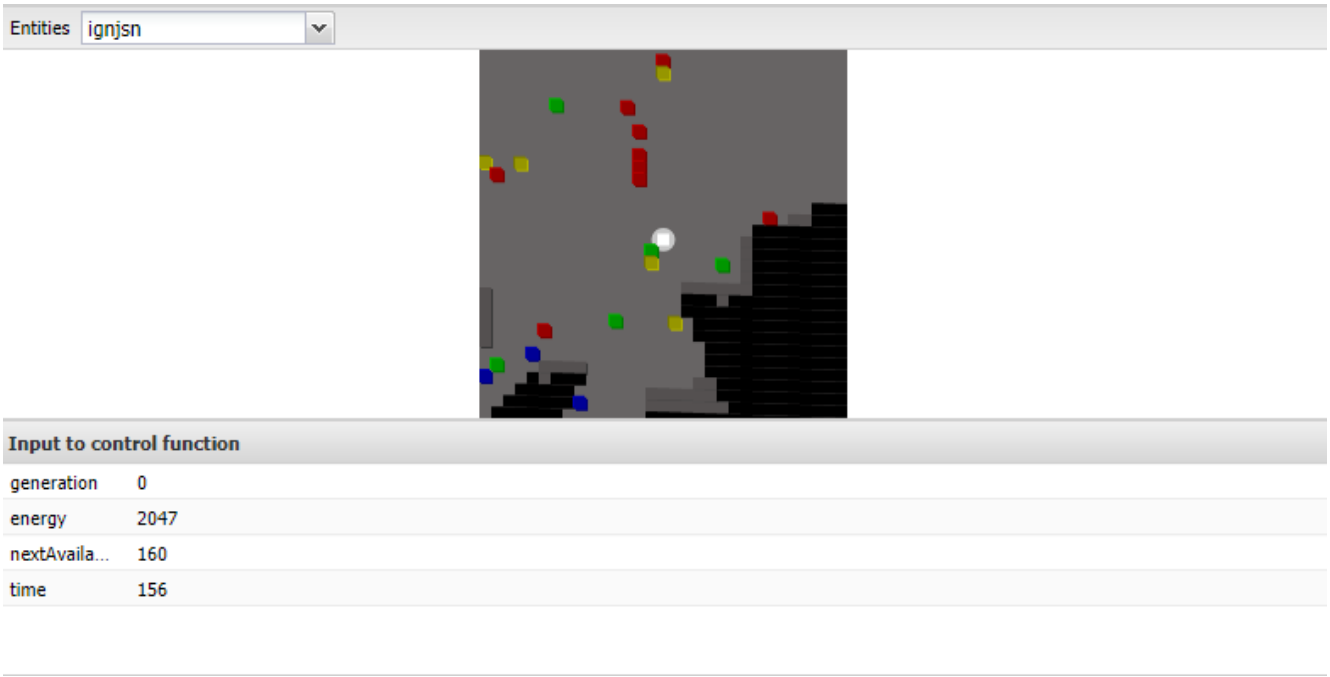
```
case class BotImpl(inputParams: Map[String, String]) extends MiniBot {
    // input
    def inputOrElse(key: String, fallback: String) = inputParams.getOrElse(key, fallback)
    def inputAsIntOrElse(key: String, fallback: Int) = inputParams.get(key).map(_.toInt).getOrElse(fallback)
    def inputAsXYOrElse(key: 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)


    // 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 }
}
```

```
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

    var parentROW = -1
    var parentCOL = -1
    var f = 10000.0
    var g = 10000.0
    var h = 10000.0

    def isNonZero = x != 0 || y != 0
    def isZero = x == 0 && y == 0
    def isNonNegative = x >= 0 && y >= 0
    def isDefined = x != -1 && y != -1

    def updateX(newX: Int) = XY(newX, y)
    def updateY(newY: Int) = XY(x, newY)
    def update(newX: Int, newY: Int) = XY(newX, 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)
}
```

```
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 isValid (x: Int, y: Int, N: Int): Boolean =
        (x >= 0) && (x < N) &&
           (y >= 0) && (y < N)

    def calculateHValue(row: Int, col: Int, dest: XY): Double =
        // Return using the distance formula
        math.sqrt((row-dest.x)*(row-dest.x)
                            + (col-dest.y)*(col-dest.y))


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

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

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)
        }
    }
}
```

2.4. Pradiniai duomenys ir rezultatatai



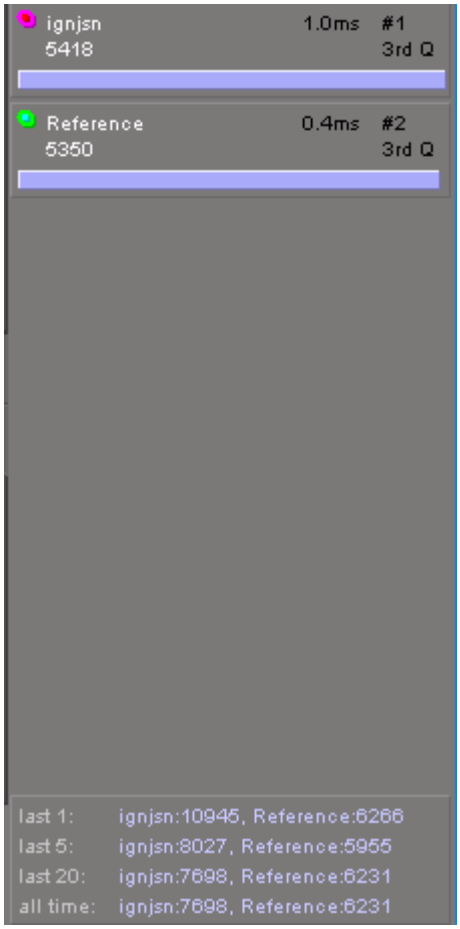
pav 2. Bot'as renka maistą

pav 3. Bot'as išleidžia pagalbininką-rinkėją

pav 4. Pagalbininko matymo laukas

pav 5. Bot'o paleidimas į areną prie pavyzdinį bot'a (reference)

pav 6. Boto raketų paleidimas



pav 7.Bot'o surinktų taškų rezultatai