Stack Cheese

ECE578

Brass, Brooks, Hull, Mayers, Minko

Portland State
UNIVERSITY

Electrical and Computer Engineering
Portland State University
2019-11-06

Contents

1 Overview

As a new design scheme, we changed the *Policemen and Thief* to the less violent, *Mice and Cheese*.

Design a game with 3 “Viking Bots”: two dressed as mice and the third as a wedge of cheese. The goal of the game is to keep the cheese away from the hungry maws of the two little mice. The game uses object recognition to find the location of each game piece on the board. The program builds a game board and determines the proper strategy so that the agents can move effectively in the right direction. The user goes first and can control the course of the cheese, and the mice will continuously move towards the cheese after each turn. Will the cheese escape the mice? Or will it get eaten? Tune in and find out next time on Perkowskis comedy cartoon hour.

2 Hardware

The hardware for our project consists of 3 robots and all their associated parts. In the past, this project utilized two Viking Bots and a single, more massive hexapod robot. Our requirements for the project were to replace the hexapod robot with another Viking Bot and to improve the robustness of the hardware. This would ensure that our hardware can run more consistently, at a faster pace, and in more diverse conditions than were possible in previous implementations of the project.

2.1 Goals

1. Purchase and assemble a new Viking Bot
2. Assess what hardware was left over from previous implementations of project
3. Upgrade and standardize battery packs for all robots
4. Improve wiring reliability and cable management
5. Ensure that the robots can run consistently at top speed for fast gameplay

2.2 Design

We assembled our projects robot cars from an inexpensive kit, which was quick to assemble. The Viking Bot robot kit consists of the following parts:

- An acrylic sheet with mounting holes and cutouts for wires
- A set of two DC motors and wheels
- A swivel caster for a back wheel
- A battery pack (AA battery size)
- An L298N H-Bridge Module
- A Raspberry Pi microprocessor board

Due to the nature of using a “kit” robot, we didnt have a lot of latitude to design the hardware we were using. However, we undertook the development of a better battery and power management system for the robots after discovering the following problems:

- Batteries we inherited had differing voltages, causing robots to function differently from one another
- Some robots used multiple battery packs to achieve uniform voltages, adding extra weight to the robot
- The V_{IN} port powered the Raspberry Pi’s, causing damage to the boards.
- H-Bridge modules were providing inconsistent outputs and current limiting the Raspberry Pi’s

To solve these problems, we designed a rechargeable battery and power system using three 18650 batteries that provided a stable voltage for the motors and Raspberry Pi. This design increased our run time, avoided damage to the Raspberry Pi’s, and was easier to implement across all three robots.

2.3 Implementation

Initially, we were told by a member of the previous team that their principal concern was using non-identical battery packs. This irregularity caused one of the robots to turn more quickly than the other. Also, we knew that we were replacing the hexapod robot with a Viking Bot, which required ordering a new robot kit and assembling it.

2.4 Assess Project Hardware

Our first task was to assess what hardware the previous team left. Unfortunately, the robots we found in the old project locker were in a sad state. The locker included all sorts of batteries, servos, screws, electronics, and other parts in a large pile. The two Viking Bot robots we inherited had loose jumper wires and tape all over them. There were three types of battery packs in the locker for the robots: a USB pack (too low voltage for constant use), AA battery packs (not rechargeable), and a sizable Li-Po battery pack (no charger available). We were not able to test any of the hardware since we did not have a battery of the correct voltage that we could recharge. These irregularities left us in the position of needing to correct the power issues immediately so we could test the robots. In order to correct the mess in the locker and to avoid losing vital parts, we cleaned out all the components. We organized them by type into labeled boxes. This organization has significantly improved the storage locker and made the project run more smoothly.

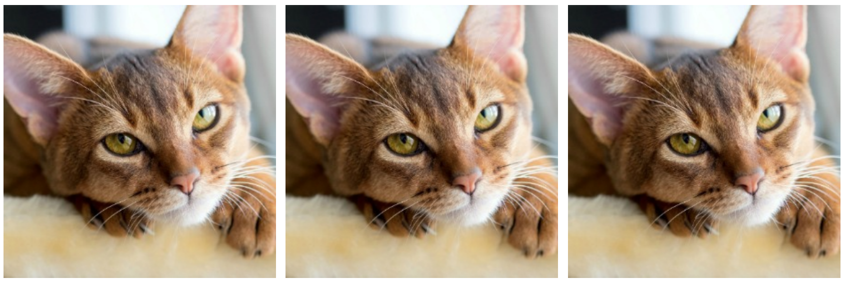

Figure 1: bat and servo parts

2.5 Battery and Power System

After learning that the Viking Bots use the L298N H-Bridge module for powering the motors and the Raspberry Pi, we were able to determine that we needed to be able to provide close to 12V for the system. Additionally, the robots were set up to use the 5V output from the H-Bridge module to power the Raspberry Pi. However, after inspection of the L298N schematic, we discovered two problems with this setup:

- The L298N requires a separate 5V power supply if supplying the module with more than 12V in order to protect the H-Bridge chip.
- The 5V onboard voltage regulator for 5V output (L78M05) can only supply up to 0.5A of current on the output.

Therefore, using a battery pack that is greater than 12V requires a different power supply for obtaining 5V. Initially, this was our only consideration, and we purchased rechargeable batteries in the 18650 size for the system. We selected these batteries their high mAh ratings and high current output ability. Then, we purchased battery holders, batteries for all the robots, and a battery charger. Using three batteries in series produces $3.7V \cdot 3 = 11.1V$ nominal voltage for the system. However, when fully charged, these batteries can reach 4.2V and provide 12.6V to the system, potentially damaging the onboard voltage regulator and the Raspberry Pi.

In addition, the Raspberry Pi requires up to 2.5A for peak power. This requirement is 2A over the maximum available from the onboard voltage regulator supply. Due to this rating, we needed to find a different solution for powering the Raspberry Pi in order to provide a stable voltage with enough amperage to function safely.

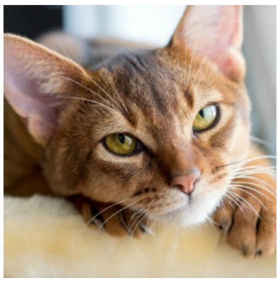

Figure 2: Schematic

Our solution was to add a separate voltage regulator that adjusts to the correct voltage for the Raspberry Pi and provides enough amperage for it to run without brownouts or damage to the board. We chose the DROK adjustable voltage regulator for this task as we could tune them to precisely 5V with the potentiometer on the underside, and they can handle up to 3A (0.5A over the Raspberry Pis max requirements).

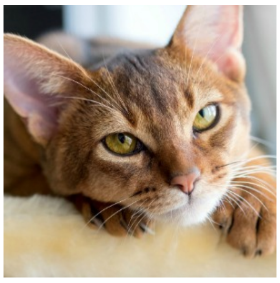

Figure 3: Voltage Regulator

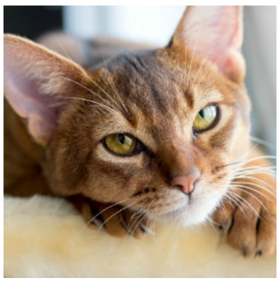

Figure 4: H-Bridge Module

2.6 Building the Third Viking Bot

Assembling the robot kit was very straight forward. The kit contained instructions and required only a screwdriver to assemble. Building a new Viking Bot from scratch allowed us to understand better how to make improvements to the wiring and set up to make the robot more robust.

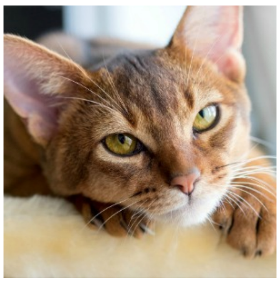

Figure 5: Robot kit fully assembled with H-Bridge module

We standardized the setup of the robots, putting the battery pack over the back swivel wheel for better weight distribution. We mounted the H-Bridge between the motors underneath the deck, the Raspberry Pi at the front, and the voltage regulator between it and the battery pack. We also attached a power switch for each robot to save on battery life. Finally, we used shrink wrap for solder connections to avoid short circuits. Then we zip-tied wires to the chassis paying close attention to keeping them out of the way. We mounted major components using velcro for ease of access. Later, we added each robots IP address to the bottom for easy identification.

Figure 6: Finished robot with robust wiring and updated power system

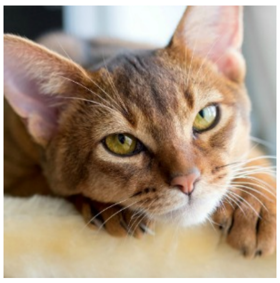

Figure 7: Final wiring diagram for Viking Bots

2.7 Fixing damaged Raspberry Pi's

When our battery packs first arrived, we were still attempting to utilize the 5V regulator on the H-Bridge modules. However, when we attempted to power on the robots, we discovered that one of the Raspberry Pis we inherited was not powering on. We then tried using our own Raspberry Pi in the system, and it immediately started smoking and was damaged.

After investigating the cause of the issue, we determined that the 5V regulator on the H-Bridge module was damaged. This regulator was causing some increased current draw or voltage spike that fed into the Raspberry Pi. Since the system was set up to feed power to the Pi through the VIN pin, there was no protection on the circuit, and it fried the diode that typically protects the Raspberry Pi. We assume that the H-Bridges voltage regulator became damaged from being supplied with more than 12V as the system under test had been using the higher-rated Li-Po battery during the previous iteration.

This situation was what prompted us to switch to a dedicated voltage regulator for the Pi, but it left us without a control board for the robots. After inspecting the other non-working pi we inherited, we saw that it too had damage to the diode. Thankfully we were able to order more diodes and replace them. This diode reinstallation fixed the boards and allowed us to use them for the robots. However, we still had an issue with using the VIN pin as it was not protected.

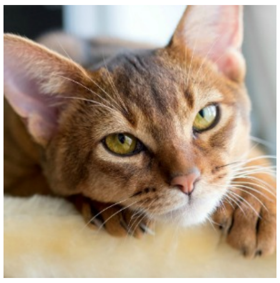
Figure 8: Diode replaced just above the Micro USB port

We cannibalized three micro USB cables to get power from our new voltage regulators and send it into the power port of the Raspberry Pi. To do this, we cut and stripped the USB cable and found the V+ and V- wires with a multimeter. After finding these, we wired them to the correct lines on the voltage regulator output, then plugged the micro-USB into the Raspberry Pi power port. This solution gave us multiple layers of protection against more voltage issues.

2.8 Building an Enclosure for the Robots

During the project, we realized the need for an additional task in the hardware category. One of our robots needed an enclosure in order for our cameras to differentiate it from the others during training. We decided that the project would transition from police chasing a thief to rats chasing cheese. So we needed to develop and cheese-like enclosure that was light enough for the Viking Bot to be still able to move quickly.

Our initial thought was to purchase a foam Green Bay Packers Cheesehead hat and modify it for our project. Unfortunately, due to time constraints and availability, we were not able to get one in time for this to work. We instead decided to try to create a custom enclosure ourselves.

A trip to the hardware store yielded some purple insulation sheets, gorilla glue, and yellow spray paint. The next step was to trace out a cheese wedge on the sheets, then cut out four of them, like a layer cake. After making a rough cut of the shape, a smaller template was made using a sheet of bamboo. This bamboo template was then taped to each rough cut piece of foam and used as a cut guide through a hot wire foam slicer.

Figure 9: Foam sheets and bamboo template

Next, the bottom two wedges had the center removed to make space for the robot electronics and wheels. We cut slots between these two sheets to make space for bamboo mounting brackets for attaching to the robot. We glued the sheets together, clamped the lot of them, and left it to dry. After the glue dried, we used a heat gun to melt classic holes into the cheese for a more recognizable look. Then, we used plastic wood putty to smooth out any differences between the sheets and to add some texture to the enclosure. We repeated this process three times to ensure reliable putty coverage, and then spray painted the entire thing yellow. Lastly, we mounted the robot inside.

Figure 10: Foam wedge with melt spots and wood putty

Figure 11: Finished cheese wedge from side and top view

We ordered rat stuffed animals for the enclosures of our other Viking Bots. The original plan was to order rats that were large enough to unsew and wrap around our robots. With this in mind, we ordered the most enormous stuffed rats we could find within our price range, which ended up being a length of 19 inches. However, upon their arrival, we learned that they included the tail in the length measurement, and the tails were quite long. That meant that we did not quite have enough space to wrap them around the robots, so we decided to mount them to the top of the robots instead, like rats piloting the Viking Bots.

We ordered two white rats, but we soon realized that we should make them different colors so the vision model could recognize them more easily. To accomplish this, we attempted to use grey and brown fabric dye to give the rats a darker color. However, the dye had an unusual reaction to the fabric of our rats. After washing and drying, the "grey" rat turned blue, and the "brown" rat turned pink. Still, this provided enough difference for us to feel comfortable training the vision model with them.

Figure 12: White rat getting ready for dyeing

Figure 13: Hand-dying to avoid dunking

Figure 14: Rats on Roman and rats on cheese

Figure 15: Blue and pink rats on cheese

2.9 Challenges

- The complete disarray of the project equipment in the locker introduced much confusion as we tried to determine what we needed to fix and what we needed to order. This chaos resulted in us spending more money to order parts and way more time than we anticipated just assessing what was available to us and what was working.
- A member from the previous group initially told us that everything was up and running and just needed battery replacement and assorted electrical fixes. This miscommunication further added to the confusion while getting started and resulted in us not understanding the actual state of the project right away. Overall, it wasted a large chunk of our time.
- Our battery holders came with wire rated for fewer amps than we needed. During our initial test, we used the H-Bridge that we did not know was faulty. This problem caused an increased current draw and ended up melting a battery pack wire and almost blowing up the battery. This situation caused us to think very carefully about how we should improve the power system on the robots.

- Frying the diodes on the Raspberry Pi was a significant setback for us. We were not able to test multiple robots until we had more controllers set up. We were able to order new parts and replace the diodes quickly, so we did not waste too much time on this.
- Figuring out what the previous group built with the wiring was a big challenge, as well. There were no diagrams or schematics for the wiring left over from the last team. We had to deal with different types of batteries, taped connections, strange switch setups, and daisy-chained jumper wires that were all contributing to inconsistent results.
- Correcting the hardware issues and making the system more robust took far more time than we had allocated for it. We were under the impression that the hardware was all working and needed minor adjustments. Instead, it required us to rebuild it completely. This issue meant we had a lot less time than we thought to work on computer vision and game concepts than we originally planned.

2.10 Future improvements

- A shield board for the Raspberry Pi, so wires use either screw terminals or solder connections instead of jumper wires.
- A better H-Bridge module that allowed for powering the Raspberry Pi without the need for an additional voltage regulator.
- Larger motors could replace the current versions to get increased torque and speed.
- Different wheels with softer tires would make an excellent addition to getting better traction on slick floors.
- Laser cutting a smaller deck for the robots would allow us to fit the rats to over the top for better aesthetics.
- If we use a standalone voltage regulator, we can upgrade the 18650 battery pack size to a four-battery holder version for even longer runtime. This change would also provide extra power for adding sensors or cameras to the robots.

3 Movement

Each Viking Bot has a Raspberry Pi that sends signals to an L298N H-Bridge. We created the controls using simple python functions that send commands which control the direction, power, and speed of the motors. It is possible to sign into the board remotely and send instructions using the wifi capabilities of the Raspberry Pi.

3.1 Goals

- Control each robot remotely
- Control speed, timing, and direction of robots
- Streamline all hardware to allow identical control schemes for each robot

3.2 Design

The Viking Bots are functionally Braitenberg vehicles. The control is simple and sent to the H-Bridge via the Raspberry Pi. We control the movement of the robots by the applied power and the direction of the motors rotation. The robots are fundamentally simple but need a mechanism to control them remotely.

3.3 Implementation

The Viking Bots are feedback-driven agents that we control through the overall game program. The user gives the program instructions in which way they want the bots to move. The program interprets this command and decides the turn direction and motor-driven distance.

3.4 Challenges

The lack of documentation regarding movement is a considerable challenge. The primary control program on each robot is simple and easy to follow; however, the mechanics of the game and functionality have been the biggest hurdle.

- No documentation
- No instructions for game use

- No instructions on setup
- No comments in code

3.5 Planned Improvements

- Better documentation
- Example or walkthrough of how to set up a game
- Inline code comments explaining how each function operates

4 Vision

4.1 Introduction

The vision module allows the Viking Bots to take whatever information they have around them in their environment and implement it into the software. The software used for the vision portion and in preparation for training was OpenCV (open-source computer vision and machine learning library), MATLAB programming language, Darknet (open-source neural network framework), YOLO (real-time object detection system), YOLO Mark (GUI for marking bounded boxes of objects in images for training neural network Yolo). We use all these packages in preparation and integration of the training.

4.2 Goals

1. Determine how we want to set up the game board
2. Set up the camera at an angle where it will capture the whole game board
3. Take images of the robots on each one of the tiles that make up the game board
4. Generate text files for each image containing the coordinates of each robot
5. Write a program that generates permutations of each image and the text files along with them
6. Add the images and text files into Darknet for training

4.3 Design

We stuck with using the same design method as the previous group that worked on this project, with some modifications and the addition of the MATLAB program. We changed the Hexapod to another Viking Bot for faster mobility. We also designed and created costumes for each robot to make it easier for object detection. The computer differentiates the rats by color, and the cheese looks strikingly different on its own. These modifications required us to re-train the machine vision algorithm.

4.4 Implementation

In order to train the robots, we needed images of the robots on the game board, as well as text files containing coordinates of each robot. The process for this required us to set up the game board and camera, as it would be set up for the demonstration. The game board is made up of 16 triangle tiles that are connected to make up one giant triangle.

We began taking images of the robots on the game board. We needed to make sure to have images of each robot being in each of the small triangle tiles. Another essential approach was taking pictures of the robots on each tile from different angles. We decided to rotate the robots 45 degrees in each small triangle tile and capture pictures of all angles of the robots. We also took images of the robots being obstructed by other robots to train the neural network for those kinds of situations.

Once we captured all of the images (201 images), we used a GUI called Yolo_Mark to create bounding boxes around each robot in each image to generate text files for the coordinates of each robot. As mentioned before, training requires these text files.

In order to improve accuracy for training, Kai wrote a program in Matlab that generated permutations of each image. This program reduces issues in detecting robots. Some permutations adjusted the coordinates of the robots, so the MATLAB program.

4.5 MATLAB and permutation training

The associated MATLAB r2018a file generates permutations of all images in the training folder, such as color-shifting, adding grain, de-noising, and rotating. These permutations ensure the training algorithm learns to identify the robots in sub-optimal conditions, or learns to identify them more fundamentally. For example, adding noise to the image requires more advanced line-recognition. Rotating the images forces the algorithm to understand what the robots look like from different angles, for instance, if future groups set up the webcam differently. Color-shifting forces the algorithm to learn more about the shape and structure of the robots, instead of a simple color-matching.

The program also copies and outputs the associated text files, so it is imperative that the text files of the training (input) pictures already exist from Yolo_Mark. User-adjustable settings, such as the number of permutations per image, have labels at the beginning of the program.

While none of these are technically necessary to train the algorithm, training using permutated images results in more consistent identification in suboptimal environments.

.

nts.

Note that we used MATLAB arbitrarily as a quick way to generate permutations without needing to set up various Python-related virtual environments and manage external dependencies. There is no practical reason why permutations require MATLAB specifically.

5 Useful Links

Link to GIT repository: https://github.com/mmayers88/Robotics

OpenCV installation: https://docs.opencv.org/3.4.7/d7/d9f/tutorial_linux_install.html

Darknet/YOLO: https://github.com/AlexeyAB/darknet

YOLO training - https://medium.com/@manivannan_data/how-to-train-yolov3-to-detect-custom-objects-ccbcafeb13d2

Yolo_Mark: https://github.com/AlexeyAB/Yolo_mark

6 MATLAB Code of Mathematical Analysis

```
% Robot cop/thief model
% Kai Brooks
% github.com/kaibrooks
% 2019
%
```

```
% Does a thing
%

% 3 axes of motion
% action space for agent:
% stay, left, right, up/down
% action space = 4
%
% state actions for agent:
% independent
% dependent on 1 robber
% dependent on 2 robber
% dependent on both
%
% total state space = 16


% Init
clc
close all
clear all
format
rng('shuffle')


% generate initial chromosome

lengthX = 6;
lengthY = 6;
maxPop = 10;

% generated internal vars
board = zeros(lengthX);
chromLength = lengthX * lengthY * 3; % size of the board, *3
    for 3 bits

% generate chromosome
for n = 1:maxPop
    population(n,:) = round(rand(1,chromLength));
end




% mix

board(3,1) = 1; % starting position
lastPosY = 3;
lastPosX = 1;
```

```

% get nearby spaces
nextY = [lastPosY-1 lastPosY+1];
nextX = [lastPosX-1 lastPosX+1];

xw5x
% zero moves to large (off the board)
nextY(nextY>=lengthY) = 0;
nextY(nextY<=1) = 0;

% zero moves too small (off the board)
nextX(nextX>=lengthX) = 0;
nextX(nextX<=1) = 0;


% evaluate population
for i = 1:maxPop



end % 1:maxPop


% rebreed

% display
```

7 Python Code for A Thing

```python
from camera_system import Camera
from object_detector import Detector
from strategy import Strategy
from graph_builder import GraphBuilder
from control_system import Controller
import logging
import sys
import time
import json
import random

WEIGHT_PATH = '../model/custom_tiny_yolov3.weights'
NETWORK_CONFIG_PATH = '../cfg/custom-tiny.cfg'
OBJECT_CONFIG_PATH = '../cfg/custom.data'
ROBOTS_CONFIG_PATH = '../cfg/robots.json'

logger = logging.getLogger(__name__)


```

```
class FakeGame:
    def __init__(self):
        self.camera = Camera(None, draw=False)
        self.display_camera = Camera(None, window_name='
            labeled')
        centers = []
        with open('centers.txt', encoding='utf-8', mode='r')
            as file:
             for line in file:
                center = tuple(map(float, line.strip().split(
                    ' ')))
                centers.append(center)
        self.centers = centers
        self.graph_builder = GraphBuilder(self.centers)
        self.orders = ['thief', 'policeman1', 'policeman2']
        self.strategy = Strategy(self.orders)
        self.object_list = {
            "thief": {
                "confidence": 0.99,
                "center": self.centers[6],  # (width,height)
                "size": (0.15, 0.10),  # (width,height)
            },
            "policeman1": {
                "confidence": 0.99,
                "center": self.centers[1],  # (width,height)
                "size": (0.15, 0.05),  # (width,height)
            },
            "policeman2": {
                "confidence": 0.99,
                "center": self.centers[3],  # (width,height)
                "size": (0.15, 0.05),  # (width,height)
            }
        }
        self.counter = 0
        self.thief_movements = [13, 14, 15, 16]
        self.escape_nodes = {10}
        self.graph = None
        self.objects_on_graph = None
        self.instructions = None

    def forward(self):

        image = self.camera.get_fake_gaming_board()
        self.display_camera.draw_boxes(image, self.
            object_list)
        self.display_camera.display(image)

        # build a graph based on object list
        graph, objects_on_graph = self.graph_builder.build(
                self.object_list)

        self.graph = graph
        self.objects_on_graph = objects_on_graph

        # generate instructions based on the graph
        instructions = self.strategy.
            get_next_steps_shortest_path(graph,
            objects_on_graph)
        logger.info('instructions:{}'.format(instructions))

        # instructions['thief'] = [objects_on_graph['thief'],
             self.thief_movements[self.counter]]
        self.instructions = instructions

        self.counter += 1
        for key, value in instructions.items():
            self.object_list[key]['center'] = self.centers[
                value[1] - 1]
        time.sleep(1)

        image = self.camera.get_fake_gaming_board()
        self.display_camera.draw_boxes(image, self.
            object_list)
        self.display_camera.display(image)

    def is_over(self):
        """
        Check if the game is over.

        Returns
        -------
        game_over: bool
            True if the thief is at the escape point or the
                policemen have caught the thief, otherwise
                False.
        """
        game_over = False
        if self.instructions is None or self.objects_on_graph
             is None or self.graph is None:
            return game_over
        if 'thief' in self.objects_on_graph:
            if self.objects_on_graph['thief'] in self.
                escape_nodes:
                game_over = True
                logger.info('The thief wins!')
            else:
                for name, instruction in self.instructions.
                    items():
```

```python
                    if name != 'thief':
                        if self.instructions['thief'][1] == instruction[1]:
                            game_over = True
                            logger.info('The policemen win!')
        return game_over

    def get_report(self):
        """
        Generate a game report(json, xml or plain text).

        Returns
        -------
        game_report: object or str
            a detailed record of the game
        """
        game_report = None
        return game_report

    def shuffle(self):
        random.randint(5, 10)


class Game:
    """
    Each game is an instance of class Game.
    """

    def __init__(self, weight_path, network_config_path, object_config_path, robots_config_path):
        """
        Load necessary modules and files.

        Parameters
        ----------
        weight_path: str
            file path of YOLOv3 network weights
        network_config_path: str
            file path of YOLOv3 network configurations
        object_config_path: str
            file path of object information in YOLOv3 network
        robots_config_path: str
            file path of robots' remote server configuration
        """

        # fix robot movement order
        self.orders = ['thief', 'policeman1']
        # self.orders = ['policeman1', 'policeman2']
        # self.orders = ['thief', 'policeman1', 'policeman2']

        # initialize internal states
        self.graph = None
        self.objects_on_graph = None
        self.instructions = None

        # set up escape nodes
        self.escape_nodes = set()

        # construct the camera system
        self.camera = Camera(1)

        # construct the object detector
        self.detector = Detector(weight_path,
           network_config_path, object_config_path)

        # load gaming board image and get centers'
           coordinates of triangles
        self.gaming_board_image = self.camera.get_image()
        self.centers = self.detector.detect_gaming_board(self
           .gaming_board_image)

        # construct the graph builder
        self.graph_builder = GraphBuilder(self.centers)

        # construct the strategy module
        self.strategy = Strategy(self.orders)

        # construct the control system
        self.controller = Controller(self.detector, self.
           camera.get_image, robots_config_path)

        # connect to each robot
        self.controller.connect()

    def is_over(self):
        """
        Check if the game is over.

        Returns
        -------
        game_over: bool
            True if the thief is at the escape point or the
               policemen have caught the thief, otherwise
               False.
        """
        game_over = False
        if self.instructions is None or self.objects_on_graph
            is None or self.graph is None:
```

```python
            return game_over
        if 'thief' in self.objects_on_graph:
            if self.objects_on_graph['thief'] in self.escape_nodes:
                game_over = True
                logger.info('The thief wins!')
            else:
                for name, instruction in self.instructions.items():
                    if name != 'thief':
                        if self.instructions['thief'][1] == instruction[1]:
                            game_over = True
                            logger.info('The policemen win!')
        return game_over

    def shuffle(self):
        random.randint(5, 10)

    def forward(self):
        """
        Push the game to the next step.
        """
        # get objects' coordinates and categories
        image = self.camera.get_image()
        object_list = self.detector.detect_objects(image)

        # build a graph based on object list
        graph, objects_on_graph = self.graph_builder.build(object_list)
        self.graph = graph
        self.objects_on_graph = objects_on_graph

        # generate instructions based on the graph
        instructions = self.strategy.get_next_steps_shortest_path(graph, objects_on_graph)
        self.instructions = instructions
        logger.info('instructions:{}'.format(instructions))

        if self.is_over():
            return
        # move robots until they reach the right positions
        while not self.controller.is_finished(self.centers, object_list, instructions):
            # obtain feedback from camera
            image = self.camera.get_image()
            object_list = self.detector.detect_objects(image)

            # calculate control signals
            control_signals = self.controller.
               calculate_control_signals(
                self.centers, object_list, instructions)

            # cut extra signals
            real_signals = []
            for name in self.orders:
                for signal in control_signals:
                    if signal['name'] == name:
                        # if True:
                        real_signals.append(signal)
                if len(real_signals) > 0:
                    break

            # update internal states
            self.controller.update_state(object_list)

            # move robots
            self.controller.move_robots(real_signals)

            # obtain feedback from camera
            image = self.camera.get_image()
            object_list = self.detector.detect_objects(image)

            # update internal states
            self.controller.update_state(object_list)

    def get_report(self):
        """
        Generate a game report(json, xml or plain text).

        Returns
        -------
        game_report: object or str
            a detailed record of the game
        """
        game_report = None
        return game_report


def main():
    # set up logger level
    logger.setLevel(logging.DEBUG)
    handler = logging.StreamHandler(sys.stdout)
    handler.setLevel(logging.DEBUG)
    logger.addHandler(handler)

    # parse config file
    if len(sys.argv) > 1:
        config_path = sys.argv[1]
    else:
        config_path = '../cfg/game_config.json'
    with open(config_path, encoding='utf-8', mode='r') as file:
        config = json.load(file)

    # load game parameters
    weight_path = config['weight_path']
    network_config_path = config['network_config_path']
    object_config_path = config['object_config_path']
    robots_config_path = config['robots_config_path']

    # construct a game logic
    game = Game(weight_path, network_config_path, object_config_path, robots_config_path)
    # game = FakeGame()
    # start the game logic
    while True:
        input('Press ENTER to the start a game:')

        # keep running until game is over
        while not game.is_over():
            game.forward()

        # get the game report
        report = game.get_report()

        # display the game report
        print(report)

        # shuffle the robots on the gaming board
        # TODO: finish shuffle() function
        game.shuffle()


if __name__ == '__main__':
    main()
```