
Project Title

Natural Computing Final Project

Derek Stotz

Christopher Smith

May 6, 2015

Contents

Title	i
Contents	iii
List of Figures	v
List of Tables	vii
List of Algorithms	ix
Document Preparation and Updates	xi
1 Introduction	1
1.1 Overview	1
1.2 Background	1
2 Problem	3
3 Implementation	5
3.1 Implementation	5
3.2 The Game	5
3.3 Evolutionary Algorithm	5
3.3.1 Genetic Algorithm	5
3.3.2 Tournament Selection	5
3.4 The ANNs	5
3.4.1 The Inputs	6
3.4.2 The Outputs	6
3.4.3 Example ANN	6
3.5 Training the ANNs	8
3.6 The Main	8
4 Issues	9
4.1 Issues	9
5 Results	11
Bibliography	11
A Supporting Materials	13
B Code	15
L^AT_EX Example	33

List of Figures

3.1	A example of a building phase ANN	7
-----	---	---

List of Tables

List of Algorithms

1	Evolve ANNs	5
---	-----------------------	---

Document Preparation and Updates

Current Version [X.X.X]

Prepared By:
Team Member #1
Team Member #2
Team Member #3

Revision History

<i>Date</i>	<i>Author</i>	<i>Version</i>	<i>Comments</i>
<i>2/2/15</i>	<i>Team Member #1</i>	<i>1.0.0</i>	<i>Initial version</i>
<i>3/4/15</i>	<i>Team Member #3</i>	<i>1.1.0</i>	<i>Edited version</i>

1

Introduction

Introduce your project.

1.1 Overview

Provide a description.

1.2 Background

Literature review here. Background work.

Problem

The board game Puerto Rico is a beloved Euro game of critical acclaim and widespread popularity. It involves indirect player interaction through the building of a Caribbean island colony, where the players compete for money and victory points via exports and sales of their cash crops. Puerto Rico is simple to learn but hard to master because of all the decision making the player has to do throughout the game. This means that the actions, while simple to represent, are very difficult to pick in different situations... making the choices of actions perfect for one of the techniques discussed in class. Our goal is to develop an AI capable of playing puerto rico with a human with 3 players.

3

Implementation

3.1 Implementation

3.2 The Game

3.3 Evolutionary Algorithm

The evolutionary algorithm 1 for this program starts off randomly creating a population of two thousand and then evaluating two thousand generations. The population is first initialized and evaluated. The best individual is kept and stored. All individuals equal in fitness are also kept between the generations. After the best fitness is found the all the bad fitness individuals are selected for recombination based on the the best individuals that population. A new generation is created based on the best fitness individuals and the ones recombined based off of their weights. The genetic algorithm takes care of selecting weights to recombine as well as the mutation rate for individual offspring, while the tournament selection handles the evaluation of an individual during the game based on how many times they win and become runner up.

Algorithm 1 Evolve ANNs

```
initialize population
evaluate population
for i in range( number_of_generations ) do
    get top fitness
    for p in population do
        for k in range( 0 , keep ranks) do
            if keep[k] < p.fitness and not (p.fitness in keep) then
                keep[k] = p.fitness
            end if
        end for
    end for
end for
```

3.3.1 Genetic Algorithm

3.3.2 Tournament Selection

3.4 The ANNs

The Artificial Neural Network used for this project was a modified ANN from Dr. Pyeatte's code. We converted his ANN to python and removed the back-propagation function and saving and loading from files.

We added a mutation function and a recombination function. The mutation function takes the number of weights that are to be mutated then randomly chooses that number of weights and randomly assigns them a value. The recombination function takes to lists of weights as arguments and gives a 50-50 chance of choosing a weight from one or the other to create its own weights.

Each AI contains a neural network for each role card that can be chosen. So there is one for the captain, trader, settler, builder, mayor, craftsman, and prospector. Each one of these phases take the same number of inputs, however they have different outputs because of the different things that are accomplished in each phase.

3.4.1 The Inputs

Each phase takes the entire game board as inputs. Player 1's colonists, buildings, plantations, victory points, doubloons, goods, and player 3 and player 2 inputs. Also what is currently on the trade ship, cargo ship, colonists ship, victory points remaining, colonists left, and doubloons left as well are all inputs to each phase.

3.4.2 The Outputs

The difference between the phases lies with the outputs. Since each phase allows a player to do something else, the outputs needed to be different. For example the building phase outputs would be the possible buildings that the player can buy. They would be ranked from highest to lowest. The player would start with the highest building that the ANN outputted and try to buy it. If it doesn't have enough doubloons it would go to the next highest ranked building. This will continue until a building is bought or the next highest building is buy nothing option. The same thing would happen with other phases where it will try to do the highest ranked output.

3.4.3 Example ANN

See Figure 3.1. This is an example of a building phase ANN. The whole game board is given as inputs to the neural network. The number of colonists, each building and plantation type, number of each resource, victory points, and the amount of dabloons they have are all given under the player inputs. Player1 inputs used to save space in the figure. All of this information fed through the hidden layers and then outputs in the building phase are ranked. The higher the output the better the option is. The player will start with the best ranked option and try to do it. If it doesn't have enough dabloons to buy it the next option is chosen and repeats until something is bought or buy nothing is chosen.

Player inputs consist of: All their coloninits, All buildings, Plantations, Resources, Dabloons, and Victory points. Condensed to save space

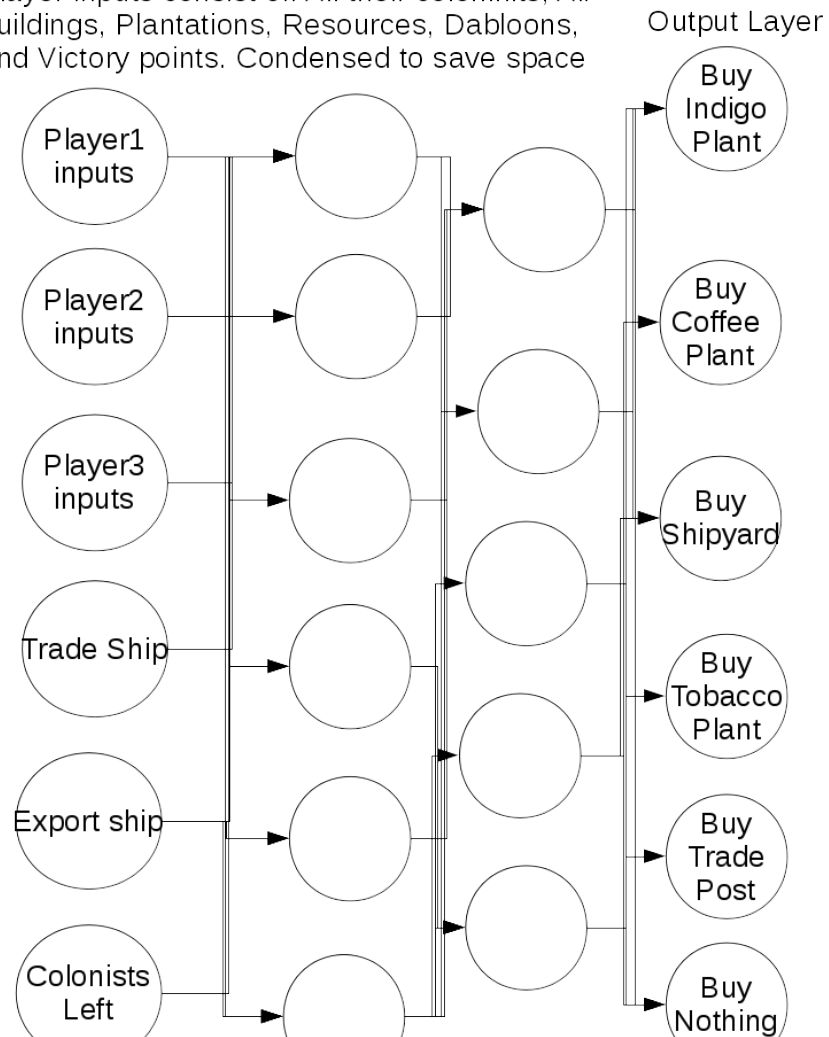


Figure 3.1: A example of a building phase ANN

3.5 Training the ANNs

3.6 The Main

4

Issues

4.1 Issues

5

Results

Your results.

A

Supporting Materials

This document will contain several appendices used as a way to separate out major component details, logic details, or tables of information. Use of this structure will help keep the document clean, readable, and organized.

B

Code

```
#!/*****  
#/*  
#/* Copyright (c) 1994  
#/* Larry D. Pyeatt  
#/* Computer Science Department  
#/* Colorado State University  
#/*  
#/* Permission is hereby granted to copy all or any part of  
#/* this program for free distribution. The author's name  
#/* and this copyright notice must be included in any copy.  
#/*  
#/* Contact the author for commercial licensing.  
#/*  
  
import random  
from math import exp  
import pickle  
class phase_ann:  
    #Takes two lists of weights and randomly chooses between the two weight to pick  
    def combine_weights( self, weights1, weights2 ):  
        for i in range( 0, self.numlayers-1 ): # (i = 0 ; i < (numlayers - 1) ; i++)  
            for j in range( 0, self.size[i+1] ): # (j = 0 ; j < size[i+1] ; j++)  
                for k in range( 0, self.size[i]+1 ): # (k = 0 ; k < size[i]+1 ; k++)  
// +1 for the bias input  
                    tmp = random.random()  
                    if( tmp < 0.5 ):  
                        self.weights[i][j][k] = weights1[i][j][k]  
                    else:  
                        self.weights[i][j][k] = weights2[i][j][k]  
  
    def mutate_weights( self, num_weights ):  
        for i in range( num_weights ):  
            j = random.randint(0, self.numlayers-2)  
            k = random.randint(0, self.size[j+1]-1 )  
            l = random.randint(0, self.size[j])  
            self.weights[j][k][l] = random.random()
```

```

/* define the activation (transfer) function and its derivative */

def xferfunc(self, x, theta):
    return 1.0 / (1.0 + exp(-(theta*x)));

def xferfuncprime( self, xprime, theta):
    return theta * (xprime * (1.0 - xprime));

/* Methods for scaling the input and output data */
def scale( self, x ):
    return ((x - self.xfermin) / (self.xfermax - self.xfermin) * (self.outmax-self.outmin));

def unscale( self, x ):
    return ((x / (self.outmax - self.outmin)) * (self.xfermax - self.xfermin));

/* Allocate storage for the bpnet */
def new_all(self):
    /* allocate storage for the layers */
    self.activation = [0 for i in range(self.numlayers)] #new
    self.dotprod = [0 for i in range(self.numlayers)] #new double*[numlayers];
    self.weights = [0 for i in range(self.numlayers-1)] #new double**[numlayers-1];
    self.sigma = [0 for i in range(self.numlayers)] #new double*[numlayers];
    self.delta = [0 for i in range(self.numlayers-1)] #new double**[numlayers-1];
    for i in range( self.numlayers ):
        self.activation[i] = [0 for k in range( self.size[i] ) ] #new double[size[i]];
        self.dotprod[i] = [0 for k in range( self.size[i] ) ] #new double[size[i]];

        self.sigma[i] = [0 for k in range( self.size[i] ) ] #new double[size[i]];

        if( i < ( self.numlayers - 1 ) ):
            self.weights[i] = [0 for k in range( self.size[i+1] ) ] #new double*[size[i+1]];
            self.delta[i] = [0 for k in range( self.size[i+1] ) ] #new double*[size[i+1]];
            for j in range( self.size[i+1] ): #(j = 0 ; j < size[i+1] ; j++)
                /* add one for the bias input
                self.weights[i][j] = [ 0 for k in range( self.size[i] + 1 ) ] #new double[size[i]+1];
                self.delta[i][j] = [ 0 for k in range( self.size[i] + 1 ) ] #new double[size[i]+1];

/******
/* Constructor for the backpropagation networks */
/* arguments: */
/*     number of layers, */
/*     size of input layer, */
/*     size of first hidden layer, */
/*     ... */
/*     size of output layer */
def __init__(self, layers, *args):
    self.set_defaults();
    self.numlayers = layers;
    self.weights = []
    self.size = [ 0.0 for i in range(self.numlayers) ]
    j = 0
    for i in args:
        self.size[j] = i;
        j = j + 1

```

```

        self.new_all();
        /* randomize the weights and set deltas to zero */
        for i in range( 0, self.numlayers-1 ): #(i = 0 ; i < (numlayers - 1) ; i++)
            for j in range( 0, self.size[i+1] ): #(j = 0 ; j < size[i+1] ; j++)
                for k in range( 0, self.size[i]+1 ): #(k = 0 ; k < size[i]+1 ; k++)
// +1 for the bias input
                    self.weights[i][j][k] = random.random()
                    self.delta[i][j][k] = 0.0;

    def set_defaults(self):
        self.fitness = -1.0;
        self.THETA = 1.0;
        self.STEP = 0.01;
        self.MOMENTUM = 0.0;
        self.outmax = 1.0;
        self.outmin = -1.0;
        self.xfermin = 0;
        self.xfermax = 1;
        self.LINEAR_OUTPUT = 0;

    /* evaluate the network from inputs to outputs */
    def evaluate( self, input_vector, output_vector ):
        /* Don't copy input vector. Just set the pointer */
        self.activation[0] = input_vector;

        for i in range( self.numlayers - 1 ): #(i = 0 ; i < numlayers - 1 ; i++)
// i = layer number
            for j in range( self.size[i+1] ): #(j = 0 ; j < self.size[i+1] ; j++)
// j = to node
                self.dotprod[i+1][j] = 0.0;
                for k in range( self.size[i] ): #(k = 0 ; k < size[i] ; k++)
// k = from node
                    self.dotprod[i+1][j] += (self.weights[i][j][k] * self.activation[i][k])
                self.dotprod[i+1][j] += self.weights[i][j][self.size[i]] * 1.0;
// bias input
            if( ( i < self.numlayers-2 ) or ( not self.LINEAR_OUTPUT ) ):
                self.activation[i+1][j] = self.xferfunc(self.dotprod[i+1][j], self.THETA)
            else:
                self.activation[i+1][j] = self.dotprod[i+1][j] * self.THETA;

        for i in range( self.size[self.numlayers - 1] ): #(i = 0 ; i < size[numlayers-1]
            output_vector[i] = self.scale( self.activation[self.numlayers-1][i] );
    def write_weights(self, filename):
        pickle.dump( self.weights, open(filename, 'wb'))
        pickle.dump( self.size, open(filename + '_size.pickle', 'wb'))

    def get_ann(filename):
        rfile = open(filename, 'r')
        weights = pickle.load(rfile)
        size = pickle.load(open(filename+'_size.pickle', 'r'));
        layers = len(size)
        player = phase_ann( layers, *size )
        player.weights = weights
        return player

```

```

def main():
    layers = 4
    layer1 = 4
    layer2 = 4
    layer3 = 7
    layer4 = 7
    input_vector = [0 for i in range(layer4) ]
    input_vector[0] = [ 0, 0, 0, 0]
    input_vector[1] = [ 0, 0, 0, 1]
    input_vector[2] = [ 0, 0, 1, 0]
    input_vector[3] = [ 0, 0, 1, 1]
    input_vector[4] = [ 0, 1, 0, 0]
    input_vector[5] = [ 0, 1, 0, 1]
    input_vector[6] = [ 0, 1, 1, 0]
    ans = [0 for i in range(layer4)]
    for i in range( layer4 ):
        ans[i] = input_vector[i][0]*2 + input_vector[i][1] + input_vector[i][2]*2 + input_v
    results = [ -1 for i in range(layer4)]
    test = phase_ann(layers, layer1, layer2, layer3, layer4 )
    test.write_weights('my_test')
    new_test = get_ann('my_test')
    new_test.write_weights('test2')
    while( min(results) == -1 ):
        results = [ -1 for i in range(layer4)]
        for i in range(layer4):
            output_vector = [0 for k in range(layer4)]
            test.evaluate(input_vector[i], output_vector)
            test.mutate_weights(1)
            if( output_vector[ans[i]] == max(output_vector)):
                results[i] = 1
        print(results)
    test.write_weights('my_test')
    new_test = get_ann('my_test')
    new_test.write_weights('test2')
if __name__ == "__main__":
    main()

```

```

from math import *
from random import *
from sys import *
#from game import *
from phase_ann2 import *
import operator

# This file is used to generate the AI file through tournament selection
# and an evolutionary program.

def create_ann():
    return phase_ann(2, 4, 7)

def fit_ann(ann, input_vector, printy):
    # if no input vector passed, make it random
    #if input_vector == None:
    #    input_vector = [getrandbits(1), getrandbits(1), getrandbits(1), getrandbits(1)]
    #ans = input_vector[0]*2 + input_vector[1] + input_vector[2]*2 + input_vector[3]
    a_out = [0]*7
    ann.evaluate(input_vector, a_out)
    #if printy:
    #    print("\n||" + str(a_out) + "||\n")
    #if ans is 0:
    #    ann.fitness = abs(1 - abs((a_out.index(max(a_out))+1)-(ans+1))/(abs(a_out.index(max(a_out)))))
    #else:
    #    ann.fitness = abs(1 - abs(a_out.index(max(a_out))-ans)/(abs(a_out.index(max(a_out)))))
    #print("\nfitness: " + str(ann.fitness) + " ans = " + str(ans) + " guess = " + str(abs(
    return a_out.index(max(a_out))

def run_selection(anns):
    for ann in anns:
        ann.fitness = 0
        ann.fitness += int(fit_ann(ann, [0, 0, 0, 0], False) == 0)
        ann.fitness += int(fit_ann(ann, [0, 0, 0, 1], False) == 1)
        ann.fitness += int(fit_ann(ann, [0, 0, 1, 0], False) == 2)
        ann.fitness += int(fit_ann(ann, [0, 0, 1, 1], False) == 3)

        ann.fitness += int(fit_ann(ann, [0, 1, 0, 0], False) == 1)
        ann.fitness += int(fit_ann(ann, [0, 1, 0, 1], False) == 2)
        ann.fitness += int(fit_ann(ann, [0, 1, 1, 0], False) == 3)
        ann.fitness += int(fit_ann(ann, [0, 1, 1, 1], False) == 4)

        ann.fitness += int(fit_ann(ann, [1, 0, 0, 0], False) == 2)
        ann.fitness += int(fit_ann(ann, [1, 0, 0, 1], False) == 3)
        ann.fitness += int(fit_ann(ann, [1, 0, 1, 0], False) == 4)
        ann.fitness += int(fit_ann(ann, [1, 0, 1, 1], False) == 5)

        ann.fitness += int(fit_ann(ann, [1, 1, 0, 0], False) == 3)
        ann.fitness += int(fit_ann(ann, [1, 1, 0, 1], False) == 4)
        ann.fitness += int(fit_ann(ann, [1, 1, 1, 0], False) == 5)
        ann.fitness += int(fit_ann(ann, [1, 1, 1, 1], False) == 6)

```

```

# starts running tournament selection to improve the weight sets given
# sorts them by rank and returns them
def run_tournament_selection(anns, max_iterations, input_vector):
    wincounts = [0] * len(anns)
    runnerupcounts = [0] * len(anns) # use for tie breaking
    competitor_indecies = [0, 0, 0]
    for i in range(0, max_iterations):
        # select three random anns
        # run a single 3-AI game and get the winner and runner-up. increment the values in t
        for k in range(0,3):
            competitor_indecies[k] = randrange(0, len(anns))
            while competitor_indecies[0] == competitor_indecies[1] or competitor_indecies[0] == c
                for m in range(0,3):
                    competitor_indecies[m] = randrange(0, len(anns))

        competitors = [anns[competitor_indecies[0]], anns[competitor_indecies[1]], anns[comp
        for j in competitors:
            fit_ann(j, None, False)
        max_index, max_value = max(enumerate(competitors), key=lambda p: p[1].fitness)
        max_index = competitor_indecies[max_index]
        wincounts[max_index] += 1
        anns[max_index].fitness = 0
        max_index_2, max_value_2 = max(enumerate(competitors), key=lambda p: p[1].fitness)
        max_index_2 = competitor_indecies[max_index_2]
        runnerupcounts[max_index_2] += 1
        anns[max_index].fitness = max_value

        #print("\n\n")
        #print(wincounts)
        #print("RUNNERUP")
        #print(runnerupcounts)

    for k in range(0, len(anns)):
        anns[k].fitness = wincounts[k] + runnerupcounts[k]/float(max(runnerupcounts))

# fills a new population with mates, fits, mutates and returns it
def mate_population(population, n, mutation_rate):
    children = []
    for i in range(0, n):
        a = randrange(0, len(population))
        b = randrange(0, len(population))
        while a == b: # make sure that a dude doesn't breed with itself
            b = randrange(0, len(population))
        child = create_ann()
        child.combine_weights(population[a].weights, population[b].weights)
        if(random.random() < mutation_rate):
            child.mutate_weights(1)
        children.append(child)
    return children

if __name__ == "__main__":

    seed()

```

```

population = []
breeding_population = []
keep_ranks = 2
population_size = 2000
number_of_iterations = 2000
mutation_rate = .5
selection_rate = .1 # selection is deterministic
input_vector = [0, 1, 0, 0]
tournament_rounds = 500

if(len(argv)>4):
    selection_rate = float(argv[4])
if(len(argv)>3):
    mutation_rate = float(argv[3])
if(len(argv)>2):
    number_of_iterations = int(argv[2])
if(len(argv)>1):
    population_size = int(argv[1])

# generate initial population
for i in range(0, population_size):
    population.append(create_ann())

run_selection(population)
best = population[0]

# begin generations
for i in range(0, number_of_iterations):
    # population.sort(key = lambda i: i.fitness)
    #if best.fitness < population[len(population)-1].fitness:
    #    best = population[len(population)-1]
    # get the top fitnesses
    max_index, max_value = max(enumerate(population), key=lambda p: p[1].fitness)
    if(population[max_index].fitness > best.fitness):
        best = population[max_index]

    keep = [0] * keep_ranks
    for p in population:
        for k in range(0, keep_ranks):
            if keep[k] < p.fitness and not (p.fitness in keep):
                keep[k] = p.fitness

    # weed out the shitty fits
    for j in range(0, len(population)):
        if j >= len(population):
            break
        if not (population[j].fitness in keep):
            del population[j]

    print(keep)
    print(len(population))

    new_population = mate_population(population, population_size - len(population), mutation_rate)
    population = population + new_population

```

```

    run_selection(population)
    print("Best fitness after " + str(i) + " iterations: " + str(best.fitness) + " out of " + str(population.fitness))

best_out = [0]*7
best.evaluate(input_vector, best_out)
print("-----\n Final best:\noutput = " + str(best_out))

# print out adder results
print("\n-----\n")
print("0 + 0 = " + str(fit_ann(best, [0, 0, 0, 0], True)) + "\n")
print("0 + 1 = " + str(fit_ann(best, [0, 0, 0, 1], True)) + "\n")
print("0 + 2 = " + str(fit_ann(best, [0, 0, 1, 0], True)) + "\n")
print("0 + 3 = " + str(fit_ann(best, [0, 0, 1, 1], True)) + "\n\n")

print("1 + 0 = " + str(fit_ann(best, [0, 1, 0, 0], True)) + "\n")
print("1 + 1 = " + str(fit_ann(best, [0, 1, 0, 1], True)) + "\n")
print("1 + 2 = " + str(fit_ann(best, [0, 1, 1, 0], True)) + "\n")
print("1 + 3 = " + str(fit_ann(best, [0, 1, 1, 1], True)) + "\n\n")

print("2 + 0 = " + str(fit_ann(best, [1, 0, 0, 0], True)) + "\n")
print("2 + 1 = " + str(fit_ann(best, [1, 0, 0, 1], True)) + "\n")
print("2 + 2 = " + str(fit_ann(best, [1, 0, 1, 0], True)) + "\n")
print("2 + 3 = " + str(fit_ann(best, [1, 0, 1, 1], True)) + "\n\n")

print("3 + 0 = " + str(fit_ann(best, [1, 1, 0, 0], True)) + "\n")
print("3 + 1 = " + str(fit_ann(best, [1, 1, 0, 1], True)) + "\n")
print("3 + 2 = " + str(fit_ann(best, [1, 1, 1, 0], True)) + "\n")
print("3 + 3 = " + str(fit_ann(best, [1, 1, 1, 1], True)) + "\n\n")

```

```
from math import *
from random import *
from sys import *
from enum import *

class Role(Enum):
    none = 0
    captain = 1
    trader = 2
    builder = 3
    settler = 4
    craftsman = 5
    mayor = 6

# because of strange issues I was having
RoleList = [ Role.none, Role.captain, Role.trader, Role.builder, Role.settler, Role.craftsm

# building ID is used when applying modifiers
class BID(Enum):
    none = 0
    small_indigo_plant = 1
    small_sugar_mill = 2
    small_market = 3
    hacienda = 4
    construction_hut = 5
    small_warehouse = 6
    indigo_plant = 7
    sugar_mill = 8
    hospice = 9
    office = 10
    large_market = 11
    large_warehouse = 12
    tobacco_storage = 13
    coffee_roaster = 14
    factory = 15
    university = 16
    harbor = 17
    wharf = 18
    guild_hall = 19
    residence = 20
    fortress = 21
    customs_house = 22
    city_hall = 23

# the .value of the crop is equivalent to its base sale value
class Crop(Enum):
    none = -2
    quarry = -1
    corn = 0
    indigo = 1
    sugar = 2
    coffee = 3
    tobacco = 4
```

```

# lists of these are in the store and on each player's board
class Building:
    def __init__(self, size, cost, workers, name, production_building = False):
        self.size = size
        self.cost = cost
        self.workers = workers
        self.name = name
        self.assigned = 0
        self.production_building = production_building

class Ship:
    def __init__(self, capacity):
        self.capacity = capacity
        self.crop = Crop.none
        self.cargo = 0

    # try to fill the ship with all of one crop, return what doesn't fit
    def fill(self, crop, amount):
        if self.crop == Crop.none:
            self.crop = crop
        self.cargo = min(self.capacity, self.cargo + amount)
        return max(0, self.cargo + amount - self.capacity)

    # depart, clearing all crops
    def depart(self):
        self.crop = Crop.none
        self.cargo = 0

class City:
    def __init__(self):
        self.capacity = 12
        self.used = 0
        self.buildings = []
        self.unemployed = 0

    def add_building(self, building):
        if (self.capacity < self.used + building.size):
            return false
        self.buildings.append(building)
        self.used += building.size
        return true

    def assign_worker(self, building_no):
        if self.buildings[building_no].assigned < self.buildings[building_no].workers and self.unemployed > 0:
            self.buildings[building_no].assigned += 1
            self.unemployed -= 1

    def get_blank_spaces(self):
        blanks = 0
        for bld in self.buildings:
            blanks += (bld.workers - bld.assigned)
        return blanks

```

```

class Console:
    def get_role(self, player_roles, player_num, role_gold):
        print("Player " + str(player_num) + ": Pick a role number\n")
        for i in range(1, 7):
            if not Role(i) in player_roles:
                print(str(i) + ". " + str(Role(i)) + "(" + str(role_gold[i]) + " Doubloons)")
        # fish for input until input is valid
        while True:
            temp = input(str(player_num) + ">>")
            if temp.isdigit() and int(temp) < 7 and int(temp) > 0:
                temp = Role(int(temp))
            if not temp in player_roles:
                return temp

    def get_building(self, store, player_num):
        print("Player " + str(player_num) + ": Pick a building number")
        for i in range(1, 24):
            if BID(i) in store and store[BID(i)][1]>0: # if the building is available
                print(str(i) + ". " + store[BID(i)][0].name + " (" + str(store[BID(i)][1]) + ")")
        # fish for input until input is valid
        while True:
            temp = input(str(player_num) + ">>")
            if temp.isdigit() and int(temp) < 20 and int(temp) > 0: #?
                temp = BID(int(temp))
            if temp in store and store[temp][1]>0: # if the building is available
                return temp

    def get_ship(self, ships, player_num):
        print("Player " + str(player_num) + ": Pick a ship number")
        for i in range(1, len(ships)):
            print(str(i) + ". Crop: " + str(ships[i].crop) + " Cargo: " + str(ships[i].cargo))
        # fish for input until input is valid
        while True:
            temp = input(str(player_num) + ">>")
            if temp.isdigit() and int(temp) < len(ships) and int(temp) > 0: #?
                return ships[temp]

    def get_crop(self, player_crops, crops):
        print("Player " + str(player_num) + ": Pick a crop number")
        for i in range(1, len(crops)):
            print(str(i) + ". " + str(crops[i]))
        # fish for input until input is valid
        while True:
            temp = input(str(player_num) + ">>")
            if temp.isdigit() and int(temp) < len(crops) and int(temp) > 0:
                return crops[temp]

    def get_worker_space(self, city, player_num):
        print("Player " + str(player_num) + ": Pick a building number")
        for i in range(0, len(city.buildings)):
            if city.buildings[i].workers != city.buildings[i].assigned:
                print(str(i) + ". " + str(city.buildings[i].name) + " (" + str(city.buildings[i]

```

```
# fish for input until input is valid
while True:
    temp = input(str(player_num) + ">>")
    if temp.isdigit() and int(temp) < len(city.buildings) and int(temp) > 0 and city.l
        return i
#todo
return 0
```

```

from game_objects import *

# a simulation of a 3-player game of puerto rico
#
# Some assumptions that we're making here:
# 1. The players are placing their buildings in an efficient manner, such that
#    the number of spaces left in their city is enough to determine placement availability
#
# 2. The players take turns arranging their colonists in the mayor phase

class Game:
    def __init__(self, num_players):
        self.winner = None
        self.num_players = num_players
        self.roles = [Role.none] * num_players
        self.gold = [200] * num_players
        self.victory_points = [0] * num_players
        self.victory_points_max = 75
        self.console = Console()
        self.role_gold = [0] * 7

        self.governor = 0 # 0th player starts first
        self.current_player = 0
        self.colonists_left = 55 # for 3 players
        self.trade_house = [Crop.none] * 4
        self.ships = [None]*(5) # ships[num players + 1] is for the player with the wharf
        self.plantations = []
        self.cities = [City()]*num_players
        self.available_roles = [ Role.trader, Role.builder, Role.settler, Role.craftsman, Role]

        self.ships[0] = Ship(4)
        self.ships[1] = Ship(5)
        self.ships[2] = Ship(6)
        self.ships[3] = Ship(7)
        self.ships[4] = Ship(8)

        self.store = \
            { #[size, cost, workers, name], amount available, number of quarries which can be
              BID.small_indigo_plant : [Building(1, 1, 1, "Small Indigo Plant"), 4, 1], \
              BID.small_market : [Building(1, 1, 1, "Small Market"), 2, 1], \
              BID.small_sugar_mill : [Building(1, 2, 1, "Small Sugar Mill"), 4, 1], \
              BID.hacienda : [Building(1, 2, 1, "Hacienda"), 2, 1], \
              BID.construction_hut : [Building(1, 2, 1, "Construction Hut"), 2, 1], \
              BID.small_warehouse : [Building(1, 3, 1, "Small Warehouse"), 2, 1], \
              BID.indigo_plant : [Building(1, 3, 3, "Indigo Plant"), 3, 2], \
              BID.sugar_mill : [Building(1, 4, 3, "Sugar Mill"), 3, 2], \
              BID.hospice : [Building(1, 4, 1, "Hospice"), 2, 2], \
              BID.office : [Building(1, 5, 1, "Office"), 2, 2], \
              BID.large_market : [Building(1, 5, 1, "Large Market"), 2, 2], \
              BID.large_warehouse : [Building(1, 6, 1, "Large Warehouse"), 2, 2], \
              BID.tobacco_storage : [Building(1, 5, 3, "Tobacco Storage"), 3, 3], \
              BID.coffee_roaster : [Building(1, 6, 2, "Coffee Roaster"), 3, 3], \
              BID.factory : [Building(1, 7, 1, "Factory"), 2, 3], \

```

```

        BID.university : [Building(1, 8, 1, "University"), 2, 3], \
        BID.harbor : [Building(1, 8, 1, "Harbor"), 2, 3], \
        BID.wharf : [Building(1, 9, 1, "Wharf"), 2, 3], \
        BID.guild_hall : [Building(2, 10, 1, "Guild Hall"), 1, 4], \
        BID.residence : [Building(2, 10, 1, "Residence"), 1, 4], \
        BID.fortress : [Building(2, 10, 1, "Fortress"), 1, 4], \
        BID.customs_house : [Building(2, 10, 1, "Customs House"), 1, 4], \
        BID.city_hall : [Building(2, 10, 1, "City Hall"), 1, 4] \
    }

    temp = []
    for i in range(0, 8):
        temp.append(Crop.coffee)
    for i in range(0, 9):
        temp.append(Crop.tobacco)
    for i in range(0, 10):
        temp.append(Crop.corn)
    for i in range(0, 11):
        temp.append(Crop.sugar)
    for i in range(0, 12):
        temp.append(Crop.indigo)
    shuffle(temp)
    self.plantation_deck = [temp[0:11], temp[12:24], temp[25:37], temp[38:50]]
    self.quarries_remaining = 8

def role_turn(self, role):
    role_player = self.roles.index(role)
    currentplayer = role_player
    colonist_ship = max(3, self.cities[0].get_blank_spaces() + self.cities[1].get_blank_spaces())

    while(True):
        if (role is Role.captain):
            self.captain_phase(currentplayer)
        elif (role is Role.trader):
            self.trader_phase(currentplayer)
        elif (role is Role.craftsman):
            self.craftsman_phase(currentplayer)
        elif (role is Role.builder):
            self.builder_phase(currentplayer)
        elif (role is Role.settler):
            self.settler_phase(currentplayer)
        elif (role is Role.mayor):
            self.mayor_phase(currentplayer, colonist_ship)
        else:
            print("\nError: no role\n")
            currentplayer = (currentplayer + 1)%num_players
            if(currentplayer is role_player):
                return

# Returns whether or not to end the game
def game_end_contition(self):
    if ( self.roles[self.current_player] == Role.captain ) and (sum(self.victory_points) > 0):
        return true
    if ( self.colonists_left <= 0):

```

```

        return true
    if ( self.cities[0].used == self.cities[0].capacity or self.cities[1].used == self.cities[1].capacity ):
        return true

def end_game(self):
    self.winner = self.victory_points.index(max(self.victory_points))

def end_game_turn(self):
    self.roles = [Role.none] * self.num_players
    self.governor = (self.governor + 1)%num_players
    self.current_player = self.governor

# Returns whether or not to continue the game turn
def end_player_turn(self):
    if self.game_end_contition():
        self.end_game()
    if (((self.governor == 0) and (self.current_player == (self.num_players - 1))) or (self.current_player == self.governor)):
        self.end_game_turn()
        return False
    else:
        self.current_player = (self.current_player + 1) % self.num_players
        return True

def game_turn(self):
    selector = self.governor
    self.roles[selector] = self.console.get_role(self.roles, selector, self.role_gold)
    self.gold[selector] += self.role_gold[RoleList.index(self.roles[selector])]
    self.role_gold[RoleList.index(self.roles[selector])] = 0;
    selector = (selector + 1) % 3
    while selector != self.governor:
        self.roles[selector] = self.console.get_role(self.roles, selector, self.role_gold)
        self.gold[selector] += self.role_gold[RoleList.index(self.roles[selector])]
        self.role_gold[RoleList.index(self.roles[selector])] = 0;
        selector = (selector + 1) % 3

# throw doubloons on all roles which were not chosen
for i in range(0, 7):
    if not (Role(i) in self.roles):
        self.role_gold[i] += 1

self.current_player = self.governor
while True:
    # do the phase of the current player
    self.role_turn(self.roles[self.current_player])
    if ( not self.end_player_turn()):
        return

def captain_phase(self, player):
    print("CAPTAIN PHASE")
    return

def trader_phase(self, player):
    print("TRADER PHASE")
    return

```

```

def craftsman_phase(self, player):
    print("CRAFTSMAN PHASE")
    return

def builder_phase(self, player):
    print("BUILDER PHASE for player " + str(player))
    choice = self.console.get_building(self.store, player)
    while (self.gold[player] < (self.store[choice][0].cost - min(self.store[choice][2], 0)):
        print("Not enough doubloons.")
        choice = self.console.get_building(self.store, player)
    return

def settler_phase(self, player):
    print("SETTLER PHASE")
    return

def mayor_phase(self, player, colonist_ship):
    print("MAYOR PHASE for player " + str(player))
    take = colonist_ship // 3
    if self.roles[player] == Roles.mayor:
        take += 1
    for i in range(0, take):
        if self.cities[player].get_blank_spaces() == 0:
            self.cities[player].unemployed += (take - i)
            return
        choice = self.console.get_worker_space(self.cities[player], player)
        self.cities[player].unemployed += 1
        self.cities[player].assign_worker(choice)
    # now give them the opportunity to assign unemployed citizens
    print("Player " + str(player) + " assign unemployed citizens")
    take = self.cities[player].unemployed
    for i in range(0, take):
        choice = self.console.get_worker_space(self.cities[player], player)
        self.cities[player].assign_worker(choice)
    return

if __name__ == "__main__":
    num_players = 3
    game = Game(num_players)

    while game.winner == None:
        game.game_turn()

```

```
from game import *
from phase_ann import *
from sys import *

# This is the main which should be run for the
# Puerto Rico AI
if __name__ == "__main__":
    # ask for number of players (0 - 3)

    # load weights from file

    # select the AIs (either randomly, or deterministically, or let the user pick)

    # begin the game
    pass
```

L^AT_EX Example
