EVCO Open Assessment

Exam no: Y0076159

17th January 2017

Contents

List of Figures

1 Introduction & Problem Definition

i.

2 Existing work

i. General

General research GP AI study $https://www.gamedev.net/resources/_technical/artificial-intelligence/application-of-genetic-programming-to-the-snake-r1175$

ii. Genetic Algorithms for snake

Ehlis [2] has provided the only comprehensive article specific to the applications of genetic algorithms to Snake. The definitions of the problem diverge from our assessment in a few non-essential ways: snake movements can be inheritely defined relative to the snake's current direction (i.e: a turn_right and turn_left terminal is defined), and the playing field of the game is a larger rectangle than the 1212 field. In all other respects, Ehlis provides an overview of a method which functions on the same game as our case. Given this, we will first analyse this approach to the problem, and aim to build an improvement both in terms of computational performance and quality of the solution.

A generic set of terminals allowing the snake to navigate the world is defined: Forward, Left and Right. The latter two allow the snake to make a change of direction relative to it's current heading, whilst the first argument does not affect the heading. The relative movement provided by the two turn operators does allow for a simpler evolution, as implementing an equivalent relative turn using absolute directions requires a significant number of operators, as we will later discuss [CITE DESIGN SECTION]

Non terminals can be categorised as either food sensing (ifFoodAhead, ifFoodUp, ifFoodRight) or danger sensing (ifDangerAhead, ifDangerRight, ifDangerLeft, ifDangerTwoAhead). The use of absolute food sensing (ifFoodUp, ifFoodRight) does however require the addition of 4 sensing methods to orient the snake, allowing it to undertake the correct action given the context. This introduces some complexity into the set of operators, potentially limiting the efficiency of the algorithm.

These sets of operators therefore define a generally greedy behaviour: the snake may be progressing towards the food, either at the root of the decision tree or further down once a given situation occurs (such as the 'wall-slitherer' describes in [2]). This evokes a classical approach to the game, similar to what we could expect from the average playerś strategy. However, given that the last move of a game-winning situation would involve fully covering the map, a greedy approach would need to be very attentive to surrounding dangers to correctly navigate a path to the wall. As such, the difficulty a greedier snake may have in adapting to every potential situation may be the cause for why the algorithm may not produce optimal solutions: an example situation is stated by Ehlis. This issue forms a basis for the design as later explained.

Additional details of the algorithm are however lacking: a large population of 10000 individuals is evolved for 500 generations; the large population is potentially a couterweight to the lack of mutation, as unique genomes are created at the start of the evolution rather than during it; this most likely results in an equivalent effect. with a maximum tree size of 150 nodes to limit bloating. Crossover probability is stated as .10 for leaves, 0.80 for nodes, with a mutation probability of 0, in addition to the use of primed populations from previously unsuccesful runs. Additional details concerning which mutation or crossover functions were used are omitted, and reasonable assumptions of these are discussed in subsection iii. to evaluate the method. The algorithm did however account for the randomisation of food placement, mitigating it by evaluating each individual over multiple runs per generation; this was also considered in our algorithm, as we will later discuss.

The proposed solution results in the creation of a game-winning decision tree (which scores the maximal 211 points). This algorithm was re-implemented to establish a benchmark for our research, revealing a

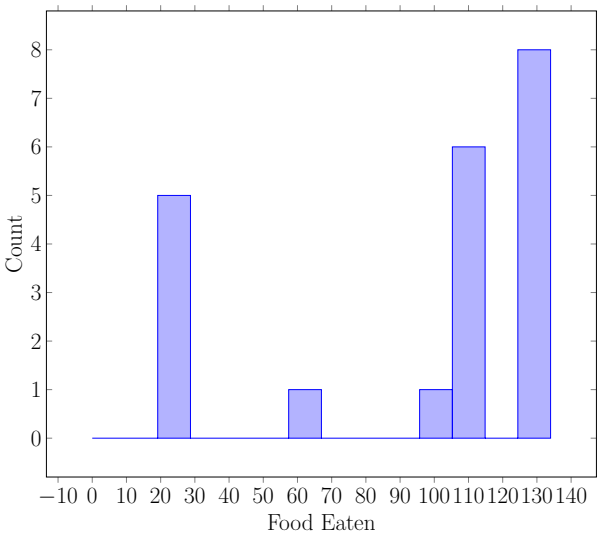

Figure 1: Histogram of best individuals from Ehlis's algorithm, 150 generations

significant requirement in computational time, as discussed in subsection iii..

iii. Implementation, assumptions and results

In order to obtain benchmark fitness values, tree sizes and computational times against which to evaluate our new algorithms, the algorithm developed by Ehlis was implemented using the Python DEAP [1] package; the code for this can be found in appendix A.
The algorithm was ran 20 times in order to obtain a reliable statistic of it's performance, but given the extensive computation time (20 runs at 150 generations required approximately 3 days to run using a 32 core linux machine with 256GB of ram) required for such a large population, each run was capped at 150 generations, and was executed using the python multiprocessing module for parallel evaluation of individuals. The results of this can be extrapolated to some extent, but we will compare further algorithms on the results they achieve on the same number of generations or less. Some assumptions regarding the type of mutation (uniform tree insertion was utilised, which did not permit a variation in the mutation probability between leaf nodes and non-leaf nodes) and crossover was set with a probability of 0.0. Finally, we assumed that the fitness tournament was used to drive the fitness upwards: this was implemented with a size of 2, and that a standard evolution algorithm was used (DEAP's eaSimple, which "takes a population and evolves it in place using varAnd()).

As seen in Figure 1, the algorithm produced 8 optimal solutions (a maximal score of 133), and an average best individual of 95.38 (std: 44.24). This was achieved using an average tree size of 92 nodes. One should note that the algorithm only has a 40% success rate in produce game-beating A.I; this is certainly related to the shortening of the algorithm by 350 generations, and the algorithm may therefore be more sucessful than the results above. Nevertheless, we can establish a benchmark against which we can compare future algorithms, keeping in mind the scale of resources provided to the algorithms is limited given available computing power. This can most easily be described as an average best score of 95.38 (std: 44.24), given 150 generations with a population of 10 000, with an average tree size of 92.

3 Testing & Design methodology

In order to evaluate the effects of operators or fitness functions, each candidate fitness function will be evaluated 20 times, with a consistant configuration: a population of 300 individuals evolved over 150 generations. We will aim to create an algorithm which requires less computational power, and produces optimal results more consistently with a smaller population and tree size. Fitness functions will be evaluated using the same algorithm (eaSimple), although a later comparison

i. Discussion

4 Design and implementation

- As we cannot implement a turn_left and turn_right function, detection of the direction of motion (ifMovingUp) is necessary - Why no food sensing?

Bibliography

[1] F.-M. De Rainville, F.-A. Fortin, M.-A. Gardner, M. Parizeau, and C. Gagné, "Deap: A python framework for evolutionary algorithms," in *Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation*, ser. GECCO '12. New York, NY, USA: ACM, 2012, pp. 85–92. [Online]. Available: http://doi.acm.org/10.1145/2330784.2330799

[2] T. Ehlis, "Application of genetic programming to the snake game - artificial intelligence - articles - articles - GameDev.net," https://www.gamedev.net/resources/_/technical/artificial-intelligence/application-of-genetic-programming-to-the-snake-r1175, 9 Aug. 2000, accessed: 2017-1-16.

A Benchmark algorithm

Based on the design document by Ehlis [2].

```
import curses
import random
import operator
import numpy

from functools import partial

from deap import algorithms
from deap import base
from deap import creator
from deap import tools
from deap import gp
import multiprocessing

S_UP, S_RIGHT, S_DOWN, S_LEFT = 0, 1, 2, 3
XSIZE, YSIZE = 14, 14
INIT_SIZE = -1

# NOTE: YOU MAY NEED TO ADD A CHECK THAT THERE ARE ENOUGH SPACES LEFT FOR
# THE FOOD (IF THE TAIL IS VERY LONG)
NFOOD = 1
GENERATIONS = 150
POP = 10000
NUM_EVALS = 3
cxpb = 0.0
mutpb = 0.8

def if_then_else(condition, out1, out2):
    out1() if condition() else out2()

def progn(*args):
    for arg in args:
        arg()

def prog2(out1, out2):
    return partial(progn,out1,out2)

def prog3(out1, out2, out3):
    return partial(progn,out1,out2,out3)

# This class can be used to create a basic player object (snake agent)


class SnakePlayer(list):
    global S_RIGHT, S_LEFT, S_UP, S_DOWN
    global XSIZE, YSIZE
    global INIT_SIZE
    def __init__(self):
        self.direction = S_RIGHT
        self.body = [[4, 10], [4, 9], [4, 8], [4, 7], [4, 6],
                     [4, 5], [4, 4], [4, 3], [4, 2], [4, 1], [4, 0]]
        INIT_SIZE = len(self.body)
        self.score = 0
        self.ahead = []
        self.food = []

    def _reset(self):
        self.direction = S_RIGHT
        self.body[:] = [[4, 10], [4, 9], [4, 8], [4, 7], [4, 6],
                        [4, 5], [4, 4], [4, 3], [4, 2], [4, 1], [4, 0]]
        self.score = 0
        self.ahead = []
        self.food = []

    def is_tile_dangerous(self, tile):
        return (tile in self.body) or (tile[1]==0) or (tile[1]==XSIZE-1) or (tile[0]==0)
             or (tile[0]==YSIZE-1)

    def get_right_location(self):
        tile = [self.body[0][0], self.body[0][1]]
        if self.direction == S_LEFT:
```

```
            tile[0]-=1
        elif self.direction == S_UP:
            tile[1]+=1
        elif self.direction == S_RIGHT:
            tile[0]+=1
        elif self.direction == S_DOWN:
            tile[1]-=1
        return tile

    def get_left_location(self):
        tile = [self.body[0][0], self.body[0][1]]
        if self.direction == S_LEFT:
            tile[0]+=1
        elif self.direction == S_UP:
            tile[1]-=1
        elif self.direction == S_RIGHT:
            tile[0]-=1
        elif self.direction == S_DOWN:
            tile[1]+=1
        return tile

    def get_ahead_location(self):
        self.ahead = [ self.body[0][0] + (self.direction == S_DOWN and 1) + (self.
            direction == S_UP and -1), self.body[0][1] + (self.direction == S_LEFT and
            -1) + (self.direction == S_RIGHT and 1)]

    def get_ahead_2_location(self):
        tile = [self.body[0][0], self.body[0][1]]
        if self.direction == S_LEFT:
            tile[1]-=2
        elif self.direction == S_UP:
            tile[0]-=2
        elif self.direction == S_RIGHT:
            tile[1]+=2
        elif self.direction == S_DOWN:
            tile[0]+=2
        return tile

    def updatePosition(self):
        self.get_ahead_location()
        self.body.insert(0, self.ahead)

    # You are free to define more sensing options to the snake

    def turn_left(self):
        self.direction = (self.direction - 1) % 4
    # You are free to define more sensing options to the snake

    def turn_right(self):
        self.direction = (self.direction + 1) % 4

    def go_straight(self):
        pass

    def snake_has_collided(self):
        self.hit = False
        if self.body[0][0] == 0 or self.body[0][0] == (
                    YSIZE - 1) or self.body[0][1] == 0 or self.body[0][1] == (XSIZE - 1)
                        :
            self.hit = True
        if self.body[0] in self.body[1:]:
            self.hit = True
        return (self.hit)

    def sense_danger_right(self):
        tile = self.get_right_location()
        return self.is_tile_dangerous(tile)

    def sense_danger_left(self):
        tile = self.get_left_location()
        return self.is_tile_dangerous(tile)

    def sense_danger_ahead(self):
        self.get_ahead_location()
        return self.is_tile_dangerous(self.ahead)

    def sense_danger_2_ahead(self):
```

```
        tile = self.get_ahead_2_location()
        return self.is_tile_dangerous(tile)

    def sense_wall_ahead(self):
        self.get_ahead_location()
        return( self.ahead[0] == 0 or self.ahead[0] == (YSIZE-1) or self.ahead[1] == 0
            or self.ahead[1] == (XSIZE-1) )

    def sense_food_ahead(self):
        self.get_ahead_location()
        return self.ahead in self.food

    def sense_tail_ahead(self):
        self.get_ahead_location()
        return self.ahead in self.body

    def sense_food_up(self):
        return self.food[0][0]<=self.body[0][0]

    def sense_food_right(self):
        return self.food[0][1]>=self.body[0][1]

    def sense_food_below(self):
        return not self.sense_food_up()

    def sense_food_left(self):
        return not self.sense_food_right()

    def sense_moving_up(self):
        return self.direction == S_UP

    def sense_moving_right(self):
        return self.direction == S_RIGHT

    def sense_moving_down(self):
        return self.direction == S_DOWN

    def sense_moving_left(self):
        return self.direction == S_LEFT

    def if_wall_ahead(self, out1, out2):
        return partial(if_then_else, self.sense_wall_ahead, out1, out2)

    def if_food_ahead(self, out1, out2):
        return partial(if_then_else, self.sense_food_ahead, out1, out2)

    def if_tail_ahead(self, out1, out2):
        return partial(if_then_else, self.sense_tail_ahead, out1, out2)

    def if_danger_right(self, out1, out2):
        return partial(if_then_else, self.sense_danger_right, out1, out2)

    def if_danger_left(self, out1, out2):
        return partial(if_then_else, self.sense_danger_left, out1, out2)

    def if_danger_ahead(self, out1, out2):
        return partial(if_then_else, self.sense_danger_ahead, out1, out2)

    def if_danger_2_ahead(self, out1, out2):
        return partial(if_then_else, self.sense_danger_2_ahead, out1, out2)

    def if_food_up(self, out1, out2):
        return partial(if_then_else, self.sense_food_up, out1, out2)

    def if_food_right(self, out1, out2):
        return partial(if_then_else, self.sense_food_right, out1, out2)

    def if_food_left(self, out1, out2):
        return partial(if_then_else, self.sense_food_left, out1, out2)

    def if_food_down(self, out1, out2):
        return partial(if_then_else, self.sense_food_down, out1, out2)

    def if_moving_up(self, out1, out2):
        return partial(if_then_else, self.sense_moving_up, out1, out2)

    def if_moving_right(self, out1, out2):
```

```
        return partial(if_then_else, self.sense_moving_right, out1, out2)

    def if_moving_down(self, out1, out2):
        return partial(if_then_else, self.sense_moving_down, out1, out2)

    def if_moving_left(self, out1, out2):
        return partial(if_then_else, self.sense_moving_left, out1, out2)

def is_tile_empty(snake, food, tile):
    return not ((tile in snake.body) or (tile in food))

# This function places a food item in the environment
# TODO convert to using a spiral search around the point
def place_food(snake):
    food = []
    while len(food) < NFOOD:
        free_spaces = []
        for x in range(1, XSIZE - 2):
            for y in range(1, YSIZE - 2 ):
                if [x, y] not in snake and [x,y] not in food:
                    free_spaces.append([x,y])
        rand = random.randint(0, len(free_spaces)-1)
        food.insert(0, free_spaces[rand])
        # randx = random.randint(1, (XSIZE - 2))
        # randy = random.randint(1, (YSIZE - 2))
        # rand_food_tile = [randy, randx]


        # if is_tile_empty(snake, food, rand_food_tile):
        #     food.insert(0, rand_food_tile)
        # else:
        #     closest_free_tile = rand_food_tile
        #     min_d = -1
        #     for x in range(XSIZE-1):
        #         for y in range(YSIZE-1):
        #             # print x,y, is_tile_empty(snake, food, [y,x])
        #             if is_tile_empty(snake, food, [y,x]):
        #                 d = numpy.sqrt(numpy.power(y-randy,2) + numpy.power(x-randx,2)
        )
        #                 if d<min_d:
        #                     min_d = d
        #                     closest_free_tile[0] = y
        #                     closest_free_tile[1] = x
        #     food.insert(0, closest_free_tile)

    snake.food = food  # let the snake know where the food is
    return (food)



def runGame(individual):
    global snake
    global pset

    routine = gp.compile(individual, pset)

    total_score = 0
    for run in range(0,NUM_EVALS):
        snake._reset()
        food = place_food(snake)
        timer = 0


        while not snake.snake_has_collided() and not timer == XSIZE * YSIZE:
            ## EXECUTE THE SNAKE'S BEHAVIOUR HERE ##
            if len(food)<NFOOD:
                food = place_food(snake)
            routine()
            snake.updatePosition()

            if snake.body[0] in food:
                snake.score += 1
                timer = 0
            else:
                snake.body.pop()
                timer += 1  # timesteps since last eaten
```

```
        if snake.score==0:
            snake.score = -(abs(snake.body[0][0] - food[0][0]) + abs(snake.body[0][1] -
                food[0][1]))
        total_score += snake.score
    return total_score/NUM_EVALS,

def run_debug(individual):
    global snake
    global pset

    routine = gp.compile(individual, pset)

    total_score = 0
    for run in range(0,NUM_EVALS):
        snake._reset()
        food = place_food(snake)
        timer = 0


        while not snake.snake_has_collided() and not timer == XSIZE * YSIZE:
            ## EXECUTE THE SNAKE'S BEHAVIOUR HERE ##
            print snake.direction, snake.body[0], snake.sense_danger_ahead()
            routine()
            print snake.direction
            snake.updatePosition()
            if snake.body[0] in food:
                snake.score += 1
                food = place_food(snake)
                timer = 0
            else:
                snake.body.pop()
                timer += 1  # timesteps since last eaten
        if snake.score==0:
            snake.score = -(abs(snake.body[0][0] - food[0][0]) + abs(snake.body[0][1] -
                food[0][1]))
        total_score += snake.score


    collided = snake.snake_has_collided()
    hitBounds = snake.body[0][0] == 0 or snake.body[0][0] == (
                    YSIZE - 1) or snake.body[0][1] == 0 or snake.body[0][1] == (XSIZE -
                        1)
    print "Collided:␣", collided
    print "Hit␣wall:␣", hitBounds
    return total_score/NUM_EVALS,




# This outline function is the same as runGame (see below). However,
# it displays the game graphically and thus runs slower
# This function is designed for you to be able to view and assess
# your strategies, rather than use during the course of evolution
def displayStrategyRun(individual):

    routine = gp.compile(individual, pset)

    curses.initscr()
    win = curses.newwin(YSIZE, XSIZE, 0, 0)
    win.keypad(1)
    curses.noecho()
    curses.curs_set(0)
    win.border(0)
    win.nodelay(1)
    win.timeout(120)

    snake._reset()
    food = place_food(snake)

    for f in food:
        win.addch(f[0], f[1], '@')

    timer = 0
    collided = False
    tmp = []
    while not collided and not timer == ((2 * XSIZE) * YSIZE):
```

```
        # Set up the display
        win.border(0)
        win.addstr(0, 2, 'Score␣:␣' + str(snake.score) + '␣')
        win.getch()

        ## EXECUTE THE SNAKE'S BEHAVIOUR HERE ##
        routine()
        snake.updatePosition()

        if snake.body[0] in food:
            snake.score += 1
            for f in food:
                win.addch(f[0], f[1], '␣')
            food = place_food(snake)
            for f in food:
                win.addch(f[0], f[1], '@')
            timer = 0
        else:
            last = snake.body.pop()
            win.addch(last[0], last[1], '␣')
            timer += 1  # timesteps since last eaten
        win.addch(snake.body[0][0], snake.body[0][1], 'o')

        collided = snake.snake_has_collided()
        hitBounds = snake.body[0][0] == 0 or snake.body[0][0] == (
                    YSIZE - 1) or snake.body[0][1] == 0 or snake.body[0][1] == (XSIZE -
                        1)
    curses.endwin()
    print "Collided:␣", collided
    print "Hit␣wall:␣", hitBounds
    print "Score:␣", snake.score
    raw_input("Press␣to␣continue...")

    return snake.score,


def main():
    global snake
    global pset

    snake = SnakePlayer()

    pset = gp.PrimitiveSet("MAIN", 0)
    # pset.addPrimitive(operator.and_, 2, name="aNd")
    # pset.addPrimitive(operator.or_, 2, name="Or")
    # pset.addPrimitive(operator.not_, 2, name="not")
    # pset.addPrimitive(prog2, 2)
    # pset.addPrimitive(prog3, 3)

    pset.addPrimitive(snake.if_food_ahead, 2, name="if_food_ahead")
    pset.addPrimitive(snake.if_danger_ahead, 2, name="if_danger_ahead")
    pset.addPrimitive(snake.if_danger_right, 2, name="if_danger_right")
    pset.addPrimitive(snake.if_danger_left, 2, name="if_danger_left")
    pset.addPrimitive(snake.if_danger_2_ahead, 2, name="if_danger_2_ahead")
    pset.addPrimitive(snake.if_food_up, 2, name="if_food_up")
    pset.addPrimitive(snake.if_food_right, 2, name="if_food_right")
    pset.addPrimitive(snake.if_moving_up, 2, name="if_moving_up")
    pset.addPrimitive(snake.if_moving_right, 2, name="if_moving_right")
    pset.addPrimitive(snake.if_moving_down, 2, name="if_moving_down")
    pset.addPrimitive(snake.if_moving_left, 2, name="if_moving_left")



    pset.addTerminal(snake.turn_right, name="turn_right")
    pset.addTerminal(snake.turn_left, name="turn_left")
    pset.addTerminal(snake.go_straight, name="go_straight")

    creator.create("FitnessMax", base.Fitness, weights=(1.0,))
    creator.create("Individual", gp.PrimitiveTree, fitness=creator.FitnessMax)

    toolbox = base.Toolbox()


    pool = multiprocessing.Pool()
    toolbox.register("map", pool.map)
```

```
    toolbox.register("expr", gp.genHalfAndHalf, pset=pset, min_=2, max_=4)
    toolbox.register("individual", tools.initIterate,
                     creator.Individual, toolbox.expr)
    toolbox.register("population", tools.initRepeat, list, toolbox.individual)
    toolbox.register("compile", gp.compile, pset=pset)

    toolbox.register("evaluate", runGame)
    toolbox.register("select", tools.selTournament, tournsize=2)
    toolbox.register("mate", gp.cxOnePoint)
    toolbox.register("expr_mut", gp.genHalfAndHalf, min_=1, max_=3, pset=pset)
    toolbox.register("mutate", gp.mutUniform, expr=toolbox.expr_mut, pset=pset)

    toolbox.decorate("mate", gp.staticLimit(
        key=operator.attrgetter("height"), max_value=15))
    toolbox.decorate("mutate", gp.staticLimit(
        key=operator.attrgetter("height"), max_value=15))


    stats_fit = tools.Statistics(lambda ind: ind.fitness.values)
    stats_size = tools.Statistics(len)
    mstats = tools.MultiStatistics(fitness=stats_fit, size=stats_size)
    mstats.register("avg", lambda x: float(int(numpy.mean(x)*100))/100)
    mstats.register("std", lambda x: float(int(numpy.std(x)*100))/100)
    # mstats.register("min", numpy.min)
    mstats.register("max", numpy.max)



    ## THIS IS WHERE YOUR CORE EVOLUTIONARY ALGORITHM WILL GO #
    random.seed()
    pop = toolbox.population(n=POP)
    hof = tools.HallOfFame(5)
    try:
        pop, log = algorithms.eaSimple(
            pop, toolbox, cxpb , mutpb, GENERATIONS, stats=mstats, halloffame=hof,
                verbose=True)

    except KeyboardInterrupt:
        pool.terminate()
        pool.join()
        raise KeyboardInterrupt
    return mstats.compile(pop), mstats, hof


if __name__ == "__main__":
    try:
        for x in range(0,5):
            record = main()[0]
            print record
            row = (record['fitness']['avg'], record['fitness']['max'], record['fitness'
                ]['std'], record['size']['avg'], record['size']['max'], record['size']['
                std'], "\r")
            fd = open('approach1results.csv', 'a')
            fd.write(",".join(map(str, row)))
            fd.close()
    except KeyboardInterrupt:
        print "Terminated␣by␣user,␣after␣%s␣iterations" % str(x)
```

B Stream.py