The chosen representation for the snakes' representation is a list of weights for a neural network. For each movement decision, the snake will take values for the snake's current direction, location of food, and what the state of the coordinates around the head of the snake are. These inputs are fed into the neural network and based on the weights of the neural network, it outputs a value of which direction the snake should turn.

Representing the snakes as a set of neural network weights is a good way to provide solutions to the problem as each decision will be based on all the inputs as each layer is fully connected. This adds to the complexity of the solutions which the snakes can evolve, with multiple layers being able to combine to produce intricate and complex solutions to perceived states.

For example, instead of trying to move always just towards the food, the snake's can evolve to zig-zag to try and create space for itself as it reaches higher lengths.

To evaluate the fitness of each set of weights for the neural network, the weights will be loaded into the network and the game will be ran with the network as the controller. The fitness value will be the score at the end of the game (i.e. how many food items the snake ate), as the goal is to evolve a snake which gives the highest score and this value should be maximised. As the game has a large degree of randomness in the position of where the food objects appear, it makes sense that for each snake being evaluated, there should be multiple runs of the game for each evaluation with the fitness value being the average score of each run. This reduces the chance of high fitness snakes being discarded over one bad run unnecessarily.

To increase the stability of evolution, I also used elitism to keep the top 10% of the top individuals in the next generation unchanged. This prevents excessive mutation resulting in good solutions being lost over generations.

To evolve the weights of the snakes, the weights can be mutated with a probability of 5% with a random gaussian value, and tournament selection is used to select the highest fitness individuals to reproduce in the next generation. This results in the fitness of the individuals in each generation increasing and new found improved reproducing.

I produced three different versions of my evolutionary algorithm to compare. All algorithms use tournament selection, elitism and mutation to evolve a new generation with improved fitness individuals, however one variant uses a network with a lower number of hidden nodes and the other implements crossover to investigate the effect this has. In my main solution, I decided against using crossover due to concern about how the competing conventions problem could affect evolution of the neural network weights, which is where a sections of weights in different individuals can affect different decisions in the network and therefore this could have a higher likelyhood of losing existing good solutions and evolution becoming unstable.

This section of the code is unchanged and sets up the grid and display class:

In [1]:
```
import random
import time
import turtle
```

In [2]:
```
XSIZE = YSIZE = 16 # Number of grid cells in each direction (do not change this)
```

In [3]:
```
class DisplayGame:
    def __init__(self, XSIZE, YSIZE):
        # SCREEN
        self.win = turtle.Screen()
        self.win.title("EVCO Snake game")
        self.win.bgcolor("grey")
```

```
        self.win.setup(width=(XSIZE*20)+40,height=(YSIZE*20)+40)
        self.win.tracer(0)

        #Snake Head
        try:
            self.head = turtle.Turtle()
        except:
            self.head = turtle.Turtle()
        self.head.shape("square")
        self.head.color("black")

        # Snake food
        try:
            self.food = turtle.Turtle()
        except:
            self.food = turtle.Turtle()
        self.food.shape("circle")
        self.food.color("yellow")
        self.food.penup()
        self.food.shapesize(0.55, 0.55)
        self.segments = []

    def reset(self, snake):
        self.segments = []
        self.head.penup()
        self.food.goto(-500, -500)
        self.head.goto(-500, -500)
        for i in range(len(snake)-1):
            self.add_snake_segment()
        self.update_segment_positions(snake)

    def update_food(self,new_food):
        self.food.goto(((new_food[1]-9)*20)+20, (((9-new_food[0])*20)-10)-20)

    def update_segment_positions(self, snake):
        self.head.goto(((snake[0][1]-9)*20)+20, (((9-snake[0][0])*20)-10)-20)
        for i in range(len(self.segments)):
            self.segments[i].goto(((snake[i+1][1]-9)*20)+20, (((9-snake[i+1][0])*2(

    def add_snake_segment(self):
        try:
            self.new_segment = turtle.Turtle()
        except:
            self.new_segment = turtle.Turtle()
        self.new_segment.speed(0)
        self.new_segment.shape("square")
        self.new_segment.color(random.choice(["green",'black','red','blue']))
        self.new_segment.penup()
        self.segments.append(self.new_segment)
```

# In the snake class I have made multiple changes in how the snake perceives its environment:

## For example, for each direction I have added a 'sense' function, e.g. 'sense_left'. These functions returns a value of 1 if there is part of its tail is one move in this direction, or if one move in this direction is a wall. Otherwise, the functions return a value of 0.

## The other sensing functions include sense_food_row and sense_food_column. These functions respond a value of 1 if the x/y coordinates of the snake are higher than the x/y coordinates of the food, -1 if the x/y coordinates are lower and 0 if the food and snake is on the same row/column. This is to help the snake find which direction it needs to head towards to find the food.

## The last function I have added which gets used in the neural network input, is a simple function which returns a number which corresponds to the direction that the snake is currently moving.

In [4]:
```
class snake:
    def __init__(self, _XSIZE, _YSIZE):
        self.XSIZE = _XSIZE
        self.YSIZE = _YSIZE
        self.directions = ["left","right","up","down"]
        self.reset()

    def reset(self):
        self.snake = [[8,10], [8,9], [8,8], [8,7], [8,6], [8,5], [8,4], [8,3], [8,
        self.food = self.place_food()
        self.ahead = []
        self.snake_direction = "right"

    def place_food(self):
        self.food = [random.randint(1, (YSIZE-2)), random.randint(1, (XSIZE-2))]
        while (self.food in self.snake):
            self.food = [random.randint(1, (YSIZE-2)), random.randint(1, (XSIZE-2)
        return( self.food )

    def update_snake_position(self):
        self.snake.insert(0, [self.snake[0][0] + (self.snake_direction == "down" a

    def food_eaten(self):
        if self.snake[0] == self.food:
            return True
        else:
            last = self.snake.pop()  # [1] If it does not eat the food, it moves fo
            return False

    def snake_turns_into_self(self):
        if self.snake[0] in self.snake[1:]:
            return True
        else:
            return False

    def snake_hit_wall(self):
        if self.snake[0][0] == 0 or self.snake[0][0] == (YSIZE-1) or self.snake[0]
            return True
        else:
            return False


    def sense_up(self):
        snake_x = self.snake[0][1]
        snake_y = self.snake[0][0]
        if [snake_y+1,snake_x] in self.snake or (snake_y+1)==YSIZE-1: #if tail abo
            return 1
```

```
        else:
            return 0

    def sense_down(self):
        snake_x = self.snake[0][1]
        snake_y = self.snake[0][0]
        if [snake_y-1,snake_x] in self.snake or (snake_y-1)==0:
            return 1
        else:
            return 0

    def sense_left(self):
        snake_x = self.snake[0][1]
        snake_y = self.snake[0][0]
        if [snake_y,snake_x-1] in self.snake or (snake_x-1)==0:
            return 1
        else:
            return 0

    def sense_right(self):
        snake_x = self.snake[0][1]
        snake_y = self.snake[0][0]
        if [snake_y,snake_x+1] in self.snake or (snake_x+1)==XSIZE-1:
            return 1
        else:
            return 0

    def sense_food_row(self):
        if self.food[0] == self.snake[0][0]:
            return 0
        elif self.food[0] < self.snake[0][0]:
            return 1
        else:
            return -1

    def sense_food_column(self):
        if self.food[1] == self.snake[0][1]:
            return 0
        elif self.food[1] < self.snake[0][1]:
            return 1
        else:
            return -1

    def get_direction_number(self):
        return self.directions.index(self.snake_direction) + 1
```

## In the run_game function I have made a couple of changes. Firstly, the decision of which direction to move is decided by passing the 7 sensing functions into the snake's neural network and making the decision based on which of the 4 returned softmaxed values is the largest.

## The other change I have made to this function is to add the variable ticks_since_food. This variable is a counter of the time it has been since the snake last ate, and if the snake goes 500

## ticks without having eaten a food item, ends the game. This acts as a stopping condition and is to help prevent the snake looping infinitely without eating any of the food items which would mean that evolution couldn't continue.

In [5]:
```
def run_game(display,snake_game,AI,headless,test=False,speed=0.1):

    score = 0
    snake_game.reset()
    if not headless:
        display.reset(snake_game.snake)
        display.win.update()
    snake_game.place_food()
    game_over = False
    snake_direction = "right"
    directions = ["left","right","up","down"]
    snake_game.set_snake_direction = "right"

    flag = True

    ticks_since_food = 0

    while not game_over:


        output = AI.feedForward( [snake_game.get_direction_number(), snake_game.se
                                  snake_game.sense_food_column(),
                                  snake_game.sense_left(),snake_game.sense_right()
                                  snake_game.sense_up(), snake_game.sense_down()]

        decision = np.argmax(output, axis=0)

        new_snake_direction = directions[decision]


        snake_direction = new_snake_direction
        snake_game.snake_direction = snake_direction


        snake_game.update_snake_position()


        # Check if food is eaten
        if snake_game.food_eaten():
            snake_game.place_food()
            score += 1
            ticks_since_food = 0
            if not headless: display.add_snake_segment()
        else:
            ticks_since_food+=1

        # Game over if the snake runs over itself
        if snake_game.snake_turns_into_self():
            game_over = True
            if test:
                print("Snake turned into itself!")

        # Game over if the snake goes through a wall
        if snake_game.snake_hit_wall():
            game_over = True
            if test:
                print("Snake hit a wall!")
```

```
        if ticks_since_food >= 500:
            game_over = True
            if test:
                print("Too long since food!")

        if not headless:
            display.update_food(snake_game.food)
            display.update_segment_positions(snake_game.snake)
            display.win.update()
            time.sleep(speed) # Change this to modify the speed the game runs at wh

    if not headless: turtle.done()
    return score
```

# Below is the start of the evolutionary algorithms:

In [6]:
```
from deap import base
from deap import creator
from deap import tools
import numpy as np
```

## A simple neural network structure to load and process the weights of the snakes

In [7]:
```
class MLP(object):
    def __init__(self, numInput, numHidden1, numHidden2, numOutput):
        self.fitness = 0
        self.numInput = numInput + 1
        self.numHidden1 = numHidden1
        self.numHidden2 = numHidden2
        self.numOutput = numOutput

        self.inp_to_hidden1_weights = np.random.randn(self.numHidden1, self.numInpu
        self.hidden1_to_hidden2_weights = np.random.randn(self.numHidden2, self.nu
        self.hidden2_to_output_weights = np.random.randn(self.numOutput, self.numH

        self.ReLU = lambda x : max(0,x)

    def softmax(self, x):
        e_x = np.exp(x - np.max(x))
        return e_x / e_x.sum()

    def feedForward(self, inputs):
        inputsBias = inputs[:]
        inputsBias.insert(len(inputs),1)

        h1 = np.dot(self.inp_to_hidden1_weights, inputsBias)         # feed input
        h1 = [self.ReLU(x) for x in h1]              # Activate hidden layer1

        h2 = np.dot(self.hidden1_to_hidden2_weights, h1)                # feed lay
        h2 = [self.ReLU(x) for x in h2]              # Activate hidden layer 2

        output = np.dot(self.hidden2_to_output_weights, h2)             # feed to

        return self.softmax(output)
```

```
    def setWeights(self, genome_weights):
        numWeights_I_H1 = self.numHidden1 * self.numInput
        numWeights_H1_H2 = self.numHidden2 * self.numHidden1
        numWeights_H2_O = self.numOutput * self.numHidden2

        self.inp_to_hidden1_weights = np.array(genome_weights[:numWeights_I_H1])
        self.inp_to_hidden1_weights = self.inp_to_hidden1_weights.reshape((self.nu

        self.hidden1_to_hidden2_weights = np.array(genome_weights[numWeights_I_H1:
        self.hidden1_to_hidden2_weights = self.hidden1_to_hidden2_weights.reshape(

        self.hidden2_to_output_weights = np.array(genome_weights[(numWeights_H1_H2
        self.hidden2_to_output_weights = self.hidden2_to_output_weights.reshape((s
```

# My final solution evolutionary code:

## This code calculates the size of the individual needed based on the number of nodes in each layer. The input layer has 7 nodes as there are 7 inputs, and the output layer has 4 outputs based on the 4 directions which the snake can choose.

## The size of the hidden layers are carefully chosen and are a balance between having a value too small that there are not complex enough behaviours that can evolve, and values too big that it firstly struggles to converge and secondly increases the processing time significantly so that it takes a long time to run.

## In analysing variants of this evolutionary code, I will show how a lower number of hidden nodes affects the evolution.

In [8]:
```
numInputNodes = 7
numHiddenNodes1 = 14
numHiddenNodes2 = 14
numOutputNodes = 4

IND_SIZE = ((numInputNodes+1) * numHiddenNodes1) + (numHiddenNodes1 * numHiddenNod
```

## initialises the network and snake game variables

In [9]:
```
snake_net = MLP(numInputNodes, numHiddenNodes1, numHiddenNodes2, numOutputNodes)
```

In [10]:
```
snakeGame = snake(XSIZE,YSIZE)
```

## Here FitnessMax is the fitness variable which has weights= ((1.0),). This means that we will be maximising the value of the fitness from the evaluation function.

## The representation of each individual is also defined here as a list of floats which ranges from -1.0 to 1.0

In [11]:
```
creator.create("FitnessMax", base.Fitness, weights=((1.0),))
creator.create("Individual", list, fitness=creator.FitnessMax)

toolbox = base.Toolbox()
toolbox.register("attr_float", random.uniform, -1.0, 1.0)
toolbox.register("individual", tools.initRepeat, creator.Individual,
                 toolbox.attr_float, n=IND_SIZE)
```

## The evaluation function loads the weights of the current individual being evaluated and runs the game a set number of times based on the num_evals parameter. The fitness value is then returned on the average of the scores from the runs.

In [12]:
```
def evaluate(indiv, snake_net, num_evals):
    fitness = 0
    headless = True
    snake_net.setWeights(indiv)
    for i in range(num_evals):
        fitness += run_game(display,snakeGame,snake_net,headless)

    return (fitness/num_evals),
```

## This section registers the different toolbox functions

## Included in this is the 'select' function which runs a selection tournament of size 5. A tournament size of 5 was chosen as this was a value which was found to help convergence to good solutions quicker, while still allowing enough variance in the population that it was less likely to get stuck on local optima.

## The mutate function was also registered here and I found that a gaussian mutation with a low value of 0.05 for the individual probability and a sigma value of 0.2 were good values as this allowed small changes to be made to the snake's weights to slowly improve the fitness without mutating too much that it had the tendancy to lose good solutions.

In [13]:
```
toolbox.register("evaluate", evaluate)
toolbox.register("select", tools.selTournament, tournsize=5)

toolbox.register("mutate", tools.mutGaussian, mu=0.0, sigma=0.2, indpb=0.05)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
```

## This section sets up some stats to log for each generation to track the progress of the evolution

In [14]:
```
stats = tools.Statistics(key=lambda ind: ind.fitness.values[0])
stats.register("avg", np.mean)
stats.register("std", np.std)
```

```
stats.register("min", np.min)
stats.register("max", np.max)
```

## This code resets the population for a new run of the algorithm

In [15]:
```
def resetPopulation(pop_size):
    pop = toolbox.population(n=pop_size)
    return pop
```

## This sets up a list which will contain the logbooks for each of the different variations to test

In [16]:
```
algorithm_variants_logbooks = []
```

# Main solution

## For testing each of these variants, I have ran the evolutionary code for 300 generations. This does not always result in a fully converged solution, however running for more generations for testing was unfeasible given how long the code took to run for each run. I wanted to have 15 runs for each variant so that there was a large enough sample size for analysis.

## This first solution uses mutation, elitism and tournament selection to evolve the snakes' network weights.

In [17]:
```
NGEN, NRUNS, NEvals = 300, 15, 2

logbook = tools.Logbook()

pop = toolbox.population(n=200)

fitnesses = [toolbox.evaluate(indiv, snake_net,NEvals) for indiv in pop]
for ind, fit in zip(pop, fitnesses):
    ind.fitness.values = fit
```

In [18]:
```
for run in range(NRUNS):
    print("-- RUN %i --" % (run+1))
    pop = resetPopulation(len(pop))
    elite_indv_size = int(len(pop)/10)
    for g in range(NGEN):
        print("-- Generation %i --" %(g))

        elite_indvs = tools.selBest(pop, elite_indv_size)

        offspring = toolbox.select(pop, len(pop)-elite_indv_size)
        offspring = list(map(toolbox.clone, offspring))

        for mutant in offspring:
            toolbox.mutate(mutant)
            del mutant.fitness.values

        for elite_ind in elite_indvs:
            del elite_ind.fitness.values
```

```
        offspring = offspring + elite_indvs

        invalid_ind = [ind for ind in offspring if not ind.fitness.valid]
        fitnesses = [toolbox.evaluate(indiv, snake_net, NEvals) for indiv in inval
        for ind, fit in zip(invalid_ind, fitnesses):
             ind.fitness.values = fit

        pop[:] = offspring
        record = stats.compile(pop)
        logbook.record(gen=g,run=run+1, **record)

        if ((g+1)%25==0):
            print (record)

algorithm_variants_logbooks.append(logbook)
```

```
-- Generation 278 --
-- Generation 279 --
-- Generation 280 --
-- Generation 281 --
-- Generation 282 --
-- Generation 283 --
-- Generation 284 --
-- Generation 285 --
-- Generation 286 --
-- Generation 287 --
-- Generation 288 --
-- Generation 289 --
-- Generation 290 --
-- Generation 291 --
-- Generation 292 --
-- Generation 293 --
-- Generation 294 --
-- Generation 295 --
-- Generation 296 --
-- Generation 297 --
-- Generation 298 --
-- Generation 299 --
{'avg': 18.1125, 'std': 10.06558958779862, 'min': 0.0, 'max': 45.5}
```

In [19]:
```
bestInd = tools.selBest(pop, 1)[0]
```

In [20]:
```
print (bestInd)
```

[1.3758843546268258, 0.6017425951186124, -0.6158859519611705, -0.3753082371194586,
0.9373460124222237, -0.5646177760123969, -0.18457321825520084, -1.127942861087426
7, 0.21296684087683682, -0.8347585671297941, 0.13076593785468704, -0.1730670377020
2508, -0.5761639781964791, 0.37542605814691404, 2.3387831467176334, 1.389228763916
7429, 1.251136889664596, -0.4681295097003123, -1.4191157431288155, -0.096045601221
35439, 0.6740531587652154, 0.08094083981826194, -0.004478287955576282, 0.183189529
2094956, -0.7378694463581058, -0.4174116138425573, 0.28747357834195764, 1.13548117
54579244, -0.42054976711203174, -1.260539916789189, 1.7350595519478054, 1.67101727
64449482, 0.7828639930221507, -0.054269692779993606, -1.4388853371301231, 0.254967
1407876447, -0.46463549478602717, 0.4440406648169165, 0.5404798998617963, -1.43784
49569569167, -0.6681529312086083, -0.3088771205592125, -1.2600251473043522, 0.7590
295762757354, -0.7236260106182827, 1.8448158145415166, 1.1996534925545808, -0.1837
0232511976348, 1.3096665017095246, 0.5579210102913317, 0.8597544288500097, -1.6348
542540620454, 0.1413520464795015, 1.180874266910247, 0.8234326149872214, 0.3134800
210503315, -0.04802649417523275, -0.10408677675806009, -1.689778746908961, 1.58486
81820187858, -1.693846337840802, -0.5893815914535399, 0.49979653623700016, -0.0888
6840348105804, 0.7617310526402685, 0.13037420533290978, 0.33515184789394736, -0.36
80656921238889, -0.006357397811233012, -1.2401865500160214, -1.1645962517449344,
0.256557729671053, -0.5903422497755426, -1.1952413965650701, 0.7176729320639113, -
0.44889408257831837, 0.4844269535109774, -0.7188095359575424, 0.6777543203011223,
-0.9105347585573419, 0.531214374420722, 0.2336938570201878, 0.4085471010424436, -
0.019844265253641566, 0.6404451416469177, 1.9455603153203134, -1.881334444466428,
1.1003228917604406, 0.3733052989375666, -0.1986044636475328, -0.01726071256625622,
-0.016824688845532967, 0.664503770806785, 1.5654725475234401, -0.7552876888032353,
-0.23167213629742303, -2.779827139262017, -0.24712227437788625, -0.606852336702505
9, 1.22997858535836, -0.6990921555914124, 0.7196217861357324, -0.1444608424533010
4, -0.338194800738734, -0.5009070688741243, 0.5276718774462432, 0.806762373076532
5, -1.254876292462085, 0.290249785068998, -0.8096623562899972, -0.742432431118138
3, -0.41978334042086335, 0.24928948508682292, 0.29598730706095117, -0.371514599433
55386, 0.6122617558406372, -1.3002287652953575, 0.2443627073274876, -0.46430721110
63895, -1.3475938151259277, -0.6846244312827585, -0.5025648822934549, -0.038283429
261528126, 1.449168486133432, -0.9088935241883923, 2.2054575945757686, -0.06565291
00184281, -0.0324658974238535, 0.4279808721837375, 0.10518755672116636, -1.1313550
255715787, -1.161183965747191, -1.0133742512823338, -0.08354902912697354, -0.49019
25890217689, -0.03942085404183607, -0.1878368467071314, 0.6307702722512026, 1.1914
290766319222, 0.2008091402004633, -0.5334422828413174, -0.22335929829519685, -0.01
4471865298911257, 0.8219329943815233, -0.8455536963180628, -0.2892323415525852, 0.
15950709980061373, 0.9410526072094958, -0.11195706236628522, -1.5455402371750762,
-0.4036947703984445, -0.3711392406325591, -2.4879270193814675, 0.0853784676666935
5, -0.18373001814136197, -1.5030483990706565, 1.4825491387806333, 1.09279281088039
41, 1.6057637747640907, -0.011547339620920305, -0.31259843405213317, 0.17207467020
5933, 1.6536099973224812, -0.21543616555629444, -0.0792645188928079, 0.06311432567
219086, 0.4620564552172459, -0.21061389671193184, -0.831312640360886, 0.3810201957
1895085, 0.57338483908939, 0.3855373202136157, 0.021500502631344842, 2.01274330418
36333, 0.0031383716445427534, -0.536925281165598, 1.1497576709319723, 1.6690198298
388856, 0.30724207875588694, 0.15951219720225318, 0.24489035772602, -0.87136323927
28524, -0.2808306957733444, -0.020003419642373732, -1.3309121875879837, 0.23312995
554506155, -1.0995638521651308, -1.3340983161943, 1.426860998570127, 0.44411147644
96247, 0.058474565377577, -1.55881365058312, -1.466277083072317, 0.577890042537007
7, 0.5118134230110081, 1.765594601987238, 0.7829203269517349, 0.17171478519783348,
0.4887320385192645, 1.3821513435635369, -0.5988180735274454, 0.023688487385330137,
0.03791400675768336, -0.12762628703110185, 0.6763290520358238, -0.616602878876007
8, -0.9086123723121288, -0.5743038532788522, 0.2630058150411768, 0.148239622661977
02, -0.22870805948364453, 0.406224779356236, -1.7409270250004043, -0.7226004097628
349, 0.11727685104721366, -1.1588156496158637, -2.104211661353046, 0.2909847740821
8076, -1.5348574770449508, 1.2076283622989887, -0.5767783062489273, 0.953450882203
2288, -0.9650401299562761, -0.5359986978758384, -0.5296699766068377, 0.57104465260
46953, 0.6902494251664067, 0.7463133227298925, -0.42132407818676554, 0.98228145220
66989, 0.31633057330829695, -0.07709772585371681, -0.34005333903044127, -0.1779128
6488727032, -0.5352457132599395, 1.3779823645204246, 2.4078076051102206, 1.0827847
052510684, 0.35593656631720905, -1.2592297837809658, -1.0580242885854263, 1.389534
736943673, -0.16665932297342337, 1.058484649511904, 0.21785724375010262, -0.029717
237646423533, 0.9431761700893161, 0.5616846845580893, 0.7391506072282709, 1.723645
600796973, -0.593532252977548, 0.6066465255255278, 0.36014329109934184, -2.0623877

```
41288243, 0.4236581895617889, 0.3348637539660471, -0.5881514862599183, 1.023717042
5013273, 0.24376828489225863, -0.0768982929081716, 0.6536031940241978, 0.033844149
273971835, -0.6210269037399084, 0.7777431887976755, -0.7876210170828092, 0.3948878
6079587956, 1.1583669788993687, -1.4420639768153878, 2.1402310592258322, 0.9302133
08135629, 0.8929410780006577, 1.0333496135572615, 0.6806903245646482, 0.4720913834
1008994, -0.5636285273328796, 1.284716827931043, -0.35880950170685055, 0.229767364
58810954, 0.04749742563288337, 1.2293472082830255, -0.5487215346928741, 0.32208051
071494004, -1.2825221411175778, -0.02935125885107584, -0.41393637056081256, -0.267
1272336439271, 0.9587260882623454, -0.291278041257365, -0.4364012395746964, -0.138
7902700102882, 1.136602333297518, -0.6200323972837781, 0.2648334547036768, -0.2491
0146881883857, -1.7529329555899529, -0.22533016646942614, 0.20711151738586897, -1.
272900875524516, -1.0720337915509062, -1.0605196267850863, -0.3531252713634708, 0.
1169434228869273, 1.6603830326639117, 0.32158723916082077, -0.36723336325023453,
0.1901327940602336, 1.6530981039662227, -1.3148884361674882, -0.40976329141205026,
1.4540947095553465, -0.8075926738676571, -0.5235997786254591, 0.03321686685617969
4, 1.4654670781474857, 0.6164811492413202, -0.8946527095365875, 0.833796868635580
9, -0.28183497088304443, -0.4560561346176928, 0.2925542181329894, 0.63492125344924
24, -0.05740572593859267, -0.7534579537266664, -0.43117472439073923, -1.4581398382
045907, -0.5229765166988334, -0.19336282818996145, 0.40471712616190636, -1.4832708
330233348, -0.8082427308254371, 0.32251544962520523, -0.9702131519449385, -0.07980
39327757043, 0.9732601071206727, 0.6438371550793555, 0.1967026778239365, 0.1382712
6309785986, -0.7271301585352726, 0.32509983181120294, -0.048331381439437826, 0.570
2263053247726, -1.5945469371872136, -0.03596410636659986, -0.9243740810924245, -0.
5904375581982645, 0.8232870721965563, -0.648354097969281, 0.2957654348317561, 1.27
38914572847404, 0.9633509430949545, 1.058758750240226, -0.5664934869792313, 1.7473
280663666104, 0.1829958729672559, -0.7628948994509496, -0.3903012759263454, 1.0686
080367619697, -0.943031842261541, 0.40389547056913394, -1.4626862293917298, 1.5874
120638536224, 0.3791529503598102, 1.250813923631485, 0.8504986748941383]
```

In [ ]:
```
#to test the evolved solution on the previous run, uncomment the next line and copy
#bestInd = []
```

In [21]:
```
snake_net.setWeights(bestInd)
```

## To view the evolved snake's behaviour, uncomment the two commented lines

In [77]:
```
snake_game = snake(XSIZE,YSIZE)
headless = True
#display = DisplayGame(XSIZE,YSIZE)
#headless = False
run_game( display,snake_game, snake_net, headless=headless,test=False,speed=0.01)
```

Out[77]:
```
49
```

In [71]:
```
total = 0

snake_game = snake(XSIZE,YSIZE)
for i in range(100):
    score = run_game( display,snake_game, snake_net, headless=True,test=False,speed
    total+=score

print (total/100)
```

```
36.27
```

## Variant #1 testing a network with a smaller number of hidden nodes per layer

# This variant of the algorithm uses a smaller number of 7 nodes per hidden layer. There are less weights to evolve, however there is the risk that the lower number of hidden layer nodes may reduce the complexity and effectiveness of the evolved solutions.

In [24]:
```
numInputNodes = 7
numHiddenNodes1 = 7
numHiddenNodes2 = 7
numOutputNodes = 4

IND_SIZE = ((numInputNodes+1) * numHiddenNodes1) + (numHiddenNodes1 * numHiddenNod
```

In [25]:
```
snake_net = MLP(numInputNodes, numHiddenNodes1, numHiddenNodes2, numOutputNodes)
```

In [26]:
```
snakeGame = snake(XSIZE,YSIZE)
```

In [27]:
```
creator.create("FitnessMax", base.Fitness, weights=((1.0),))
creator.create("Individual", list, fitness=creator.FitnessMax)

toolbox = base.Toolbox()
toolbox.register("attr_float", random.uniform, -1.0, 1.0)
toolbox.register("individual", tools.initRepeat, creator.Individual,
                 toolbox.attr_float, n=IND_SIZE)

toolbox.register("evaluate", evaluate)
toolbox.register("select", tools.selTournament, tournsize=5)

toolbox.register("mutate", tools.mutGaussian, mu=0.0, sigma=0.2, indpb=0.05)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
```

```
C:\Users\traub\anaconda3\lib\site-packages\deap\creator.py:138: RuntimeWarning: A
class named 'FitnessMax' has already been created and it will be overwritten. Cons
ider deleting previous creation of that class or rename it.
  warnings.warn("A class named '{0}' has already been created and it "
C:\Users\traub\anaconda3\lib\site-packages\deap\creator.py:138: RuntimeWarning: A
class named 'Individual' has already been created and it will be overwritten. Cons
ider deleting previous creation of that class or rename it.
  warnings.warn("A class named '{0}' has already been created and it "
```

In [28]:
```
NGEN, NRUNS, NEvals = 300, 15, 2

logbook = tools.Logbook()

pop = toolbox.population(n=200)

fitnesses = [toolbox.evaluate(indiv, snake_net,NEvals) for indiv in pop]
for ind, fit in zip(pop, fitnesses):
    ind.fitness.values = fit
```

In [29]:
```
for run in range(NRUNS):
    print("-- RUN %i --" % (run+1))
    pop = resetPopulation(len(pop))
    elite_indv_size = int(len(pop)/10)
    for g in range(NGEN):
        print("-- Generation %i --" %(g))

        elite_indvs = tools.selBest(pop, elite_indv_size)

        offspring = toolbox.select(pop, len(pop)-elite_indv_size)
```

```
        offspring = list(map(toolbox.clone, offspring))

        for mutant in offspring:
            toolbox.mutate(mutant)
            del mutant.fitness.values

        for elite_ind in elite_indvs:
            del elite_ind.fitness.values

        offspring = offspring + elite_indvs

        invalid_ind = [ind for ind in offspring if not ind.fitness.valid]
        fitnesses = [toolbox.evaluate(indiv, snake_net, NEvals) for indiv in inval
        for ind, fit in zip(invalid_ind, fitnesses):
             ind.fitness.values = fit

        pop[:] = offspring
        record = stats.compile(pop)
        logbook.record(gen=g,run=run+1, **record)

        if ((g+1)%25==0):
            print (record)

algorithm_variants_logbooks.append(logbook)
```

```
-- Generation 278 --
-- Generation 279 --
-- Generation 280 --
-- Generation 281 --
-- Generation 282 --
-- Generation 283 --
-- Generation 284 --
-- Generation 285 --
-- Generation 286 --
-- Generation 287 --
-- Generation 288 --
-- Generation 289 --
-- Generation 290 --
-- Generation 291 --
-- Generation 292 --
-- Generation 293 --
-- Generation 294 --
-- Generation 295 --
-- Generation 296 --
-- Generation 297 --
-- Generation 298 --
-- Generation 299 --
{'avg': 0.9275, 'std': 0.6049741730024514, 'min': 0.0, 'max': 3.0}
```

# Variant #2: Using crossover

## In this variant of the evolutionary algorithm, crossover will be used between the list of network weights. An initial concern with this variant would be how the competing conventions problem affects how effective crossover is, although this might become less of an issue as the population converges to all become more similar to each other and there is less variation in the population.

In [30]:
```
numInputNodes = 7
numHiddenNodes1 = 14
numHiddenNodes2 = 14
numOutputNodes = 4

IND_SIZE = ((numInputNodes+1) * numHiddenNodes1) + (numHiddenNodes1 * numHiddenNod
```

In [31]:
```
snake_net = MLP(numInputNodes, numHiddenNodes1, numHiddenNodes2, numOutputNodes)
```

In [32]:
```
snakeGame = snake(XSIZE,YSIZE)
```

In [33]:
```
creator.create("FitnessMax", base.Fitness, weights=((1.0),))
creator.create("Individual", list, fitness=creator.FitnessMax)

toolbox = base.Toolbox()
toolbox.register("attr_float", random.uniform, -1.0, 1.0)
toolbox.register("individual", tools.initRepeat, creator.Individual,
                 toolbox.attr_float, n=IND_SIZE)

toolbox.register("evaluate", evaluate)
toolbox.register("select", tools.selTournament, tournsize=5)

toolbox.register("mutate", tools.mutGaussian, mu=0.0, sigma=0.2, indpb=0.05)

toolbox.register("mate", tools.cxOnePoint)
```

```
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
```

In [34]:
```
NGEN, NRUNS, NEvals = 300, 15, 2

logbook = tools.Logbook()

pop = toolbox.population(n=200)

fitnesses = [toolbox.evaluate(indiv, snake_net,NEvals) for indiv in pop]
for ind, fit in zip(pop, fitnesses):
    ind.fitness.values = fit
```

In [35]:
```
CXProb = 0.5
for run in range(NRUNS):
    print("-- RUN %i --" % (run+1))
    pop = resetPopulation(len(pop))
    elite_indv_size = int(len(pop)/3)
    for g in range(NGEN):
        print("-- Generation %i --" %(g))

        elite_indvs = tools.selBest(pop, elite_indv_size)

        offspring = toolbox.select(pop, len(pop)-elite_indv_size)
        offspring = list(map(toolbox.clone, offspring))

        for child1, child2 in zip(offspring[::2], offspring[1::2]):
            if random.random() < CXProb:
                toolbox.mate(child1, child2)
                del child1.fitness.values
                del child2.fitness.values

        for mutant in offspring:
            toolbox.mutate(mutant)
            del mutant.fitness.values

        for elite_ind in elite_indvs:
            del elite_ind.fitness.values

        offspring = offspring + elite_indvs

        invalid_ind = [ind for ind in offspring if not ind.fitness.valid]
        fitnesses = [toolbox.evaluate(indiv, snake_net, NEvals) for indiv in inval
        for ind, fit in zip(invalid_ind, fitnesses):
             ind.fitness.values = fit

        pop[:] = offspring
        record = stats.compile(pop)
        logbook.record(gen=g,run=run+1, **record)

        if ((g+1)%25==0):
            print (record)

algorithm_variants_logbooks.append(logbook)
```

```
-- Generation 278 --
-- Generation 279 --
-- Generation 280 --
-- Generation 281 --
-- Generation 282 --
-- Generation 283 --
-- Generation 284 --
-- Generation 285 --
-- Generation 286 --
-- Generation 287 --
-- Generation 288 --
-- Generation 289 --
-- Generation 290 --
-- Generation 291 --
-- Generation 292 --
-- Generation 293 --
-- Generation 294 --
-- Generation 295 --
-- Generation 296 --
-- Generation 297 --
-- Generation 298 --
-- Generation 299 --
{'avg': 26.245, 'std': 17.5031418608203, 'min': 0.0, 'max': 68.0}
```

## The function avgStatsOverRuns() takes a logbook as input and returns the average of each stat per generation over all the runs in the logbook. This should give a good insight into how effective each of the evolutionary algorithms are, and whether we can determine if one variant produces higher fitness solutions than others.

In [36]:
```
def avgStatsOverRuns(logbook,NRuns,NGens):
    gen = [i for i in range(NGens)]
    avgList = [0 for i in range(max(gen)+1)]
    stdList = [0 for i in range(max(gen)+1)]
    minList = [0 for i in range(max(gen)+1)]
    maxList = [0 for i in range(max(gen)+1)]
    for runindx in range(NRUNS):
        runRecords = [record for record in logbook if record['run']==(runindx+1)]
        tempAvgList = [record['avg'] for record in runRecords]
        tempStdList = [record['std'] for record in runRecords]
        tempMinList = [record['min'] for record in runRecords]
        tempMaxList = [record['max'] for record in runRecords]
        avgList = [total_avg + temp_avg for total_avg, temp_avg in zip(avgList, te
        stdList = [total_std + temp_std for total_std,temp_std in zip(stdList, tem
        minList = [total_min + temp_min for total_min,temp_min in zip(minList, tem
        maxList = [total_max + temp_max for total_max,temp_max in zip(maxList, tem

    avgList = [avg/NRuns for avg in avgList]
    stdList = [std/NRuns for std in stdList]
    minList = [minVal/NRuns for minVal in minList]
    maxList = [maxVal/NRuns for maxVal in maxList]

    return gen,avgList,stdList,minList,maxList
```

## The function getStatsLastGen gets the list of the average last generation score for each run for a variant. This will be used to determine how effective the evolution has been at 300 generations.

In [37]:
```
def getStatsLastGen(logbook,gens):
    lastGenAvgs = [record['avg'] for record in logbook if record['gen']==(gens-1)]
    return lastGenAvgs
```

In [38]:
```
import matplotlib.pyplot as plt
%matplotlib inline

plt.rc('axes', labelsize=14)
plt.rc('xtick', labelsize=14)
plt.rc('ytick', labelsize=14)
plt.rc('legend', fontsize=14)
```

In [61]:
```
gen,mainAvgs,mainStd,mainMin,mainMax = avgStatsOverRuns(algorithm_variants_logbook
gen,variant1Avgs,variant1Std,variant1Min,variant1Max = avgStatsOverRuns(algorithm_
gen,variant2Avgs,variant2Std,variant2Min,variant2Max = avgStatsOverRuns(algorithm_
```

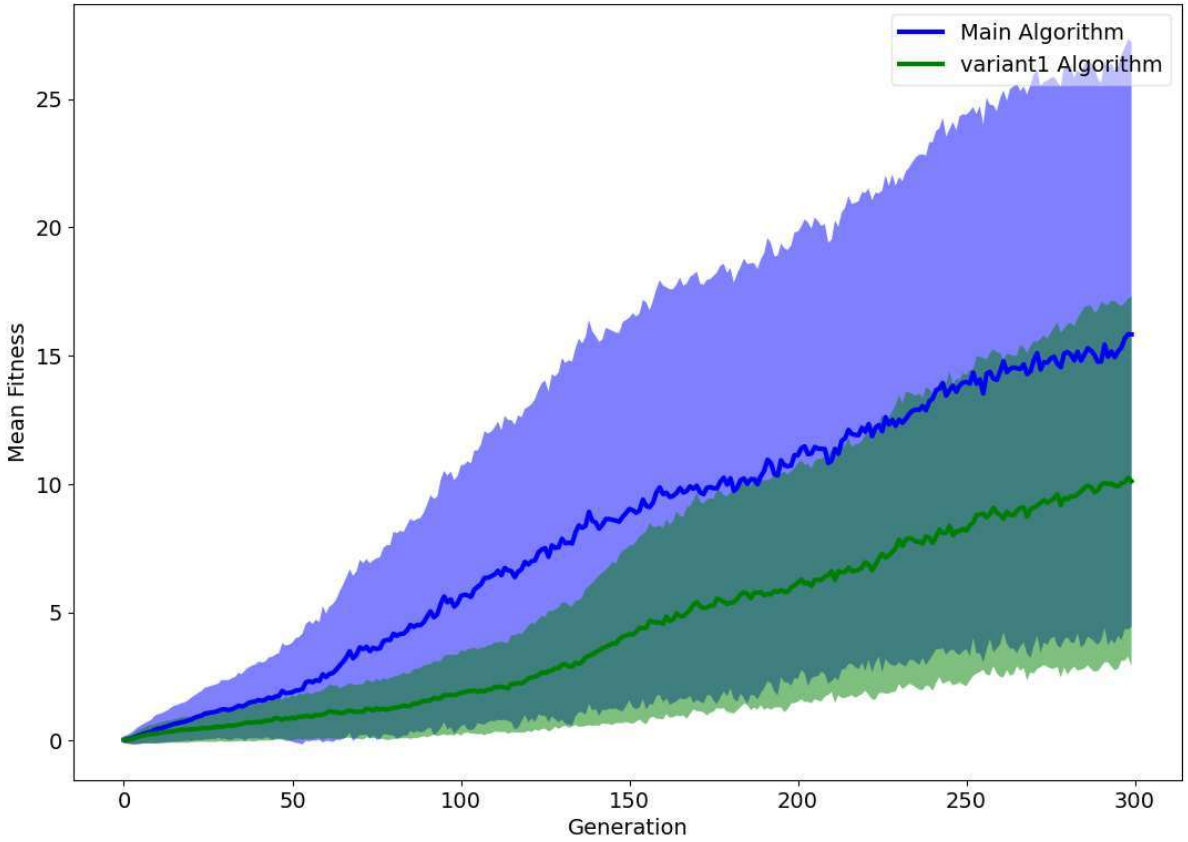
## The graph below shows the mean fitness at each generation for both the main algorithm and the first variant of the algorithm. The shaded areas show the area covered by the standard deviation below and above the mean.

## What we can see is that the standard deviation for both algorithms are both very large, and over the 15 runs of the algorithm there was a lot of variation in how fast the evolution picked up speed. A possible way to improve this and decrease the standard devation of the averages would be to have a higher population size so that the evolution is less affected by the initial randomness of the population as there is more variety.

## Although the standard deviation is large for both algorithms, there is a clear gap in the effectiveness of the two algorithms with the gap with the variant algorithm increasing over time.

In [62]:
```
plt.rcParams.update({'font.size': 20})

fig = plt.figure(figsize =(10, 7))
ax = fig.add_axes([0, 0, 1, 1])
ax.set_xlabel('Generation')
ax.set_ylabel('Mean Fitness')

ax.plot(gen, mainAvgs, lw=3, label='Main Algorithm', color='blue')
ax.fill_between(gen, np.array(mainAvgs)+mainStd, np.array(mainAvgs)-mainStd, facec

ax.plot(gen, variant1Avgs, lw=3, label='variant1 Algorithm', color='green')
ax.fill_between(gen, np.array(variant1Avgs)+variant1Std, np.array(variant1Avgs)-va



ax.legend(loc='best', fancybox=True, framealpha=0.5)
```

Out[62]:
```
<matplotlib.legend.Legend at 0x211f16592e0>
```



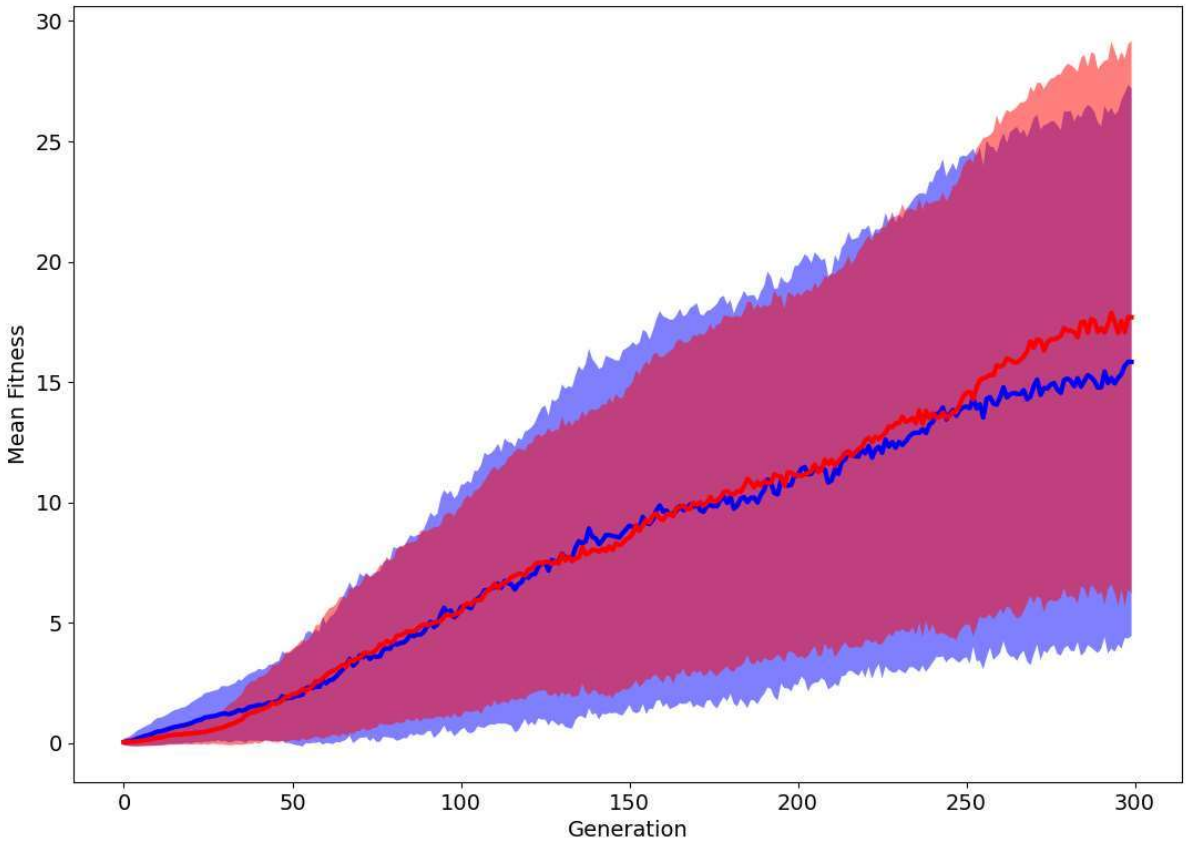

This second plot shows the main algorithm compared to the algorithm with crossover implemented.

Interestingly, the crossover variant and the main algorithm were very similar in how the evolution progressed over the generations, with the crossover variant even outperforming the main algorithm towards the end of the generations. This suggests that the competing conventions problem does not cause too much of an issue for this problem, and potentially causes less of a problem as the population converges as suggested.

However, it is very difficult to argue that one algorithm is better than another due to the very high standard deviations of the average results for both algorithms, and it would require many more results to determine fully whether one was better than the other.

In [63]:
```
plt.rcParams.update({'font.size': 20})
fig = plt.figure(figsize =(10, 7))
ax = fig.add_axes([0, 0, 1, 1])
ax.set_xlabel('Generation')
ax.set_ylabel('Mean Fitness')

ax.plot(gen, mainAvgs, lw=3, label='Main Algorithm', color='blue')
ax.fill_between(gen, np.array(mainAvgs)+mainStd, np.array(mainAvgs)-mainStd, facec

ax.plot(gen, variant2Avgs, lw=3, label='variant2 Algorithm', color='red')
ax.fill_between(gen, np.array(variant2Avgs)+variant2Std, np.array(variant2Avgs)-va
```

Out[63]: <matplotlib.collections.PolyCollection at 0x211f1614790>

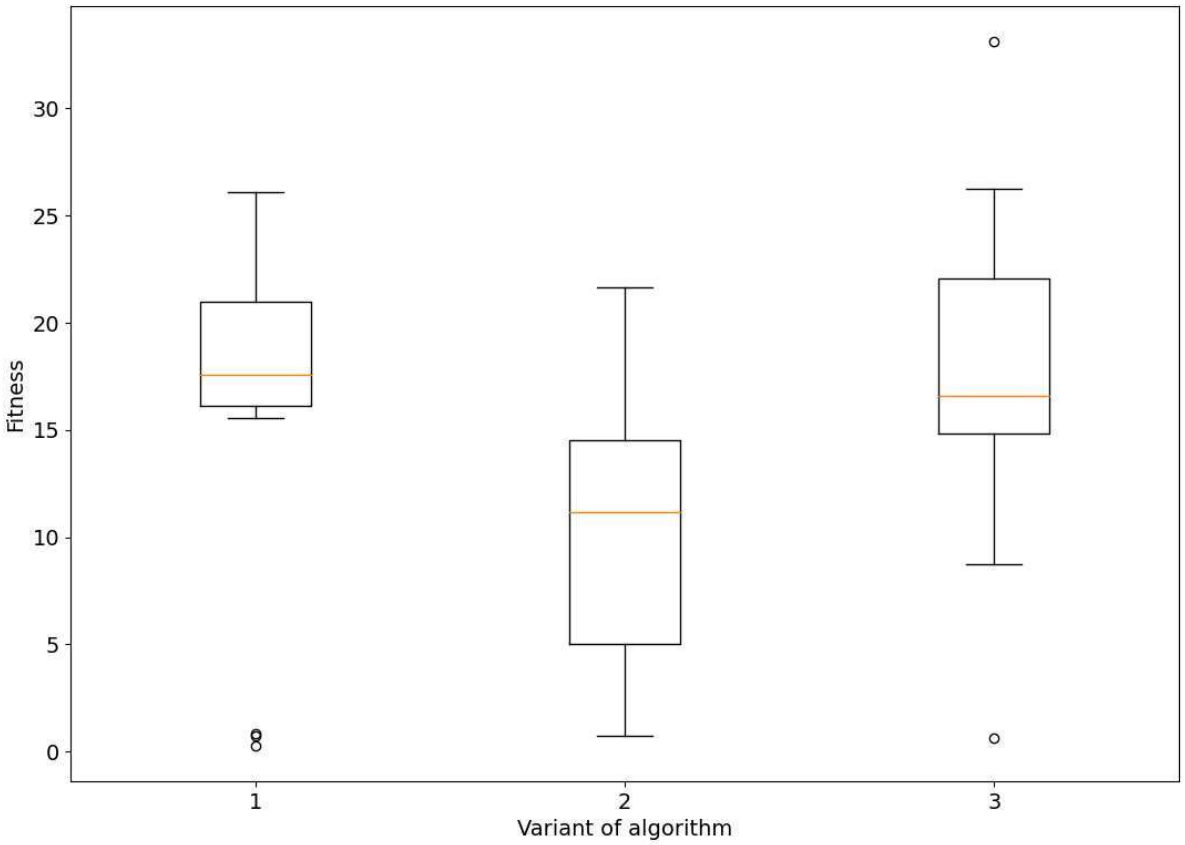



This next plot is a boxplot which summarises the distribution of the last generation averages for each variation of the evolutionary alogrithms. From this plot we can see that the smaller network size variant has a significantly lower median than the other two variants and the upper quartile for this variant is lower than the lower quartile for the other two variants.

The median for the main algorithm and the crossover variant are very similar which supports the idea that the two algorithms are similar in quality. The main algorithm had two outliers where the evolution never took off in the way that it usually did compared to this happening once with the crossover algorithm. Over 15 runs this is not something which you can draw conclusions from as there is a high degree in randomness in the initial variance of the populations of which a bad initial variation can make evolution difficult.

In [73]:
```
mainlastGenAvgs = getStatsLastGen(algorithm_variants_logbooks[0],300)
variant1lastGenAvgs = getStatsLastGen(algorithm_variants_logbooks[1],300)
variant2lastGenAvgs = getStatsLastGen(algorithm_variants_logbooks[2],300)

variantsAvgs = [mainlastGenAvgs, variant1lastGenAvgs, variant2lastGenAvgs]



fig = plt.figure(figsize =(10, 7))
ax = fig.add_axes([0, 0, 1, 1])
```

```
ax.set_xlabel('Variant of algorithm')
ax.set_ylabel('Fitness')
bp = ax.boxplot(variantsAvgs )
```




We can also analyse the last generation averages of the difference variants using the Mann-Whitney U test. This tests the likelyhood that two lists of data come from the same distribution. With the null hypothesis that the two sets of results come from the distribution, I will be using a significance level of 0.05 to measure whether this null hypothesis can be rejected in favour of the H1 hypothesis that one variant produces better fitness individuals than another.

From this first Mann-Whitney test comparing the average fitnesses of the main algorithm compared to the smaller number of hidden nodes variant. This gave a P-value of 0.02019. This value is statistically significant as 0.02019 < 0.05, and we can therefore reject the null hypothesis in favour of the alternate hypothesis that they are not from the same distribution and the main variant algorithm produces higher fitness populations.

In [68]:
```
from scipy.stats import mannwhitneyu

stat, p_value = mannwhitneyu(mainlastGenAvgs, variant1lastGenAvgs)
print('Statistics=%.3f, p=%.5f' % (stat, p_value))
```

```
Statistics=169.000, p=0.02019
```

From this test comparing the main algorithm to the variant using crossover, we cannot reject the null hypothesis as the P-

value is a lot higher than 0.05 at 0.96691. Therefore we cannot say that with high probability that the results are not drawn from the same distribution.

In [69]:
```
stat, p_value = mannwhitneyu(mainlastGenAvgs, variant2lastGenAvgs)
print('Statistics=%.3f, p=%.5f' % (stat, p_value))
```

```
Statistics=111.000, p=0.96691
```

# Summary

To summarise, the main algorithm and the variant using crossover were difficult to compare in effectiveness due to the unpredictability of evolution and initial variation in population. In future work, if I were able to run the algorithms for significantly more time, running the algorithms with a higher population size would help mitigate this issue, although this would obvious increasing processing time.

For future developments, a method of ensuring that there was diversity in the population using techniques such as fitness sharing could be something to look into. This would help ensure that the evolution doesn't get stuck in local optima as easily, as would help the algorithms produce unique diverse solutions.

Futher work could also look to add more sensing functions to the snake to feed into the networks, such as a more global view of the map and being able to sense whether moving in a certain direction is more likely to get the snake stuck surrounded by its tail.