Code for ACO

ant.py

```
class Ant(object):
    '''
    Class representing an ant

    :attr route: list of co-ordinates that represent the bin-item config
    :attr fitness: current fitness of the ant's route
    :attr bins: the bin config if the ant is the best in a generation

    :method lay_pheromones(graph):
    :method copy():
    :method get_route_str():
    '''

    route = []
    fitness = -1
    bins = []

    def lay_pheromones(self, graph):
        '''
        Distribute a pheremone weight on the graph at positions defined in route
        :param graph: Construction graph of the problem
        :return: None
        '''

        weight = 100.0 / self.fitness
        previous_bin = 0

        for bin, item in self.route:
            graph.graph[previous_bin, item, bin] += weight
            previous_bin = bin

    def copy(self):
        '''
        Creates a copy of the ant
        :return: copy of ant
        '''

        n_ant = Ant()
        n_ant.route = [r for r in self.route]
        n_ant.bins = self.bins.copy()
        n_ant.fitness = self.fitness
        return n_ant

    def get_route_str(self):
        '''
        Give a string representing the ant's route
        :return: String representing the ant's route
        '''

        return " -> ".join("Item %d in Bin %d" % (point[1] + 1, point[0]) for point in self.
```

ant_colony_optimisation.py

```
from random import random
from time import time
from itertools import repeat
from matplotlib import pyplot as plt

from graph import Graph
from ant import Ant


class AntColonyOpt(object):
    '''
    All relevant info for objects required for running ACO

    :attr bins: bin object holding items and total weight
    :attr items: list of item weights
    :attr ants: list of ant objects to be controlled
    :attr b_ant: best ant of final gen of a run
    :attr graph: graph to store pheromone weights
    :attr num_paths: number of routes evaluated
    :attr limit: generation limit
    :attr verbose: whether or not to print to console when log called
    :attr ran: has the ACO been run
    :attr runtime: duration of last run
    :attr avg_fitnesses: timeseries of averages fitnesses over each cycle

    :method summary():
    :method stats():
    :method run():
    :method explore():
    :method ant_run(a):
    :method create_route(a):
    :method route_step(p_bin, i):
    :method route_fitness():
```

```
:method set_best():
:method empty_bins():
:method log(message):
:method graph_averages():
'''

def __init__(self, b, i, p, e, limit=10000, verbose=False):

    self.bins = b
    self.items = i

    self.p = p
    self.e = e

    self.ants = [Ant() for _ in range(p)]
    self.b_ant = None

    self.graph = Graph(len(b), len(i), e)

    self.num_paths = 0
    self.limit = limit
    self.verbose = verbose

    self.ran = False
    self.runtime = 0

    self.avg_fitnesses = []

def summary(self):
    '''
    Give summary data on the run
    :return: None
    '''

    if hasattr(self, 'ran') and self.ran:
        print("Run Successful")
        print("Runtime: %d seconds" % int(self.runtime))
        print("Best Fitness: %d" % self.b_ant.fitness)
        print("Best Configuration: ")
        for i, b in enumerate(self.b_ant.bins):
            print("%4d. %s" % (i + 1, b))

def stats(self):
    '''
    Give stats about the run
    :return: best fitness of run and runtime
```

```
        '''

        if hasattr(self, 'ran') and self.ran:
            return self.b_ant.fitness, self.runtime

    def run(self):
        '''
        Run the ACO
        :return: None
        '''

        self.log("Starting Run")
        self.ran = False
        self.b_fitnesses = []
        self.avg_fitnesses = []
        start = time()

        while self.num_paths < self.limit:
            self.explore()

        self.set_best()
        self.ran = True
        self.runtime = time() - start

    def explore(self):
        '''
        Run a cycle of route creation and evaporation
        :return: None
        '''

        self.ants = [*map(self.ant_run, self.ants)]
        best = None

        for a in self.ants:
            a.lay_pheromones(self.graph)

        fitnesses = [a.fitness for a in self.ants]
        self.b_fitnesses.append(min(fitnesses) / sum(self.items))
        self.avg_fitnesses.append(sum(fitnesses) / len(fitnesses))
        self.graph.evaporate()

    def ant_run(self, a):
        '''
        Reset and recreate route of ant
        :param a: Ant to recreate route of
        :return: Ant with new route
        '''
        self.empty_bins()
        a = self.create_route(a)
        a.bins = self.bins.copy()
        return a

    def create_route(self, a):
        '''
        Create route through graph
        :param a: Ant to travel the route
        :return: Ant
        '''

        p_bin = 0
        a.route = []

        for i in enumerate(self.items):
            p_bin, i = self.route_step(p_bin, i)
            a.route.append((p_bin, i))

        a.fitness = self.route_fitness()
        self.num_paths += 1

        return a

    def route_step(self, p_bin, i):
        '''
        Step from current bin to next bin solution
        :param p_bin: The previous bin
        :param i: The item
        :return: The next bin
        '''

        col = self.graph.graph[p_bin][i[0]].tolist()
        total = sum(col)
        threshold = total * random()

        cur = 0.0

        for ind, w in enumerate(col):
            if cur + w >= threshold:
                self.bins[ind].add(i[1])
                return ind, i[0]

            cur += w
```

```
def route_fitness(self):
    '''
    Calculate fitness of the route
    :return: The fitness value
    '''

    max_w = self.bins[0].total_weight
    min_w = self.bins[0].total_weight

    for b in self.bins:
        if b.total_weight > max_w:
            max_w = b.total_weight
        if b.total_weight < min_w:
            min_w = b.total_weight

    return max_w - min_w

def set_best(self):
    '''
    Set best ant of generation
    :return: The best ant
    '''

    for a in self.ants:
        if self.b_ant and a.fitness < self.b_ant.fitness:
            self.b_ant = a.copy()
        elif not self.b_ant:
            self.b_ant = a.copy()

def empty_bins(self):
    '''
    Empty the bins
    :return: List of emptied bins
    '''

    [b.empty() for b in self.bins]

def log(self, message):
    '''
    Print a message if running in verbose mode
    :param message: Message to be printed
    :return: None
    '''

    if self.verbose:
        print(message)
```

```python
    def graph_averages(self):
        '''
        Graph the averages of each generation
        :return: None
        '''

        plt.title("Pop: %d; ER: %2f" % (self.p, self.e))
        plt.plot(self.avg_fitnesses)
        plt.show()


if __name__ == "__main__":
    from bin import gen_bins
    from item import gen_items

    b = gen_bins(10)
    i = gen_items(n=200)
    p = 10
    e = 0.4

    t = AntColonyOpt(b, i, p, e, verbose=True)
    t.run()
    t.graph_averages()
```

bin.py

```python
class Bin(object):
    '''
    Represents a bin that can hold items

    :attr total_weight: sum of items in the bin
    :attr items: item weights currently in bin

    :method add:
    :method copy:
    :method empty:
    '''


    total_weight = 0
    items = []

    def __repr__(self):
        return "Item Count: %d ; Weight: %d ; Items: %s" \
                % (len(self.items), self.total_weight, self.items)

    def add(self, item):
        '''
        Add item to bin
        :param item:  item to be added
        :return: None
        '''

        self.items.append(item)
        self.total_weight += item

    def copy(self):
        '''
        Creates a copy of the bin
        :return: copy of bin
        '''

        n_bin = Bin()
        n_bin.total_weight = self.total_weight
        n_bin.items = [i for i in self.items]
        return n_bin

    def empty(self):
        '''
        Resets contents of bin
        :return:  None
        '''

        self.items = []
        self.total_weight = 0


def gen_bins(n):
    '''
    Creates number of bins given
    :param n: number of bins to create
    :return: list of generated bins
    '''

    return [Bin() for _ in range(n)]


if __name__ == "__main__":
    print("Generating 10 bins")
    b = gen_bins(10)
    print(b)
    print(len(b))
```

graph.py

```
import numpy as np


class Graph(object):
    '''
    Represents pheromone weights across bin-item matrix

    :attr graph: 3D array of pheromone weights
    :attr e: Pheromone evaporation rate

    :method evaporate():
    '''

    def __init__(self, b, i, e):
        self.graph = np.random.rand(b, i, b)
        self.e = e

    def __repr__(self):
        return "Graph: " + str(self.graph)

    def evaporate(self):
        '''
        Reduce pheromone weights across the graph
        :return: None
        '''

        self.graph = self.graph * self.e


if __name__ == "__main__":
    print("Generating graph [10 bins, 200 items, 0.9 evap. rate]")
    g = Graph(10, 200, 0.9)
    print(g)
```

item.py

```
def gen_items(lower=1, upper=200, n=200, scale=False, test=False):
    '''
    Creates array of int weights
    :param n: number of items to be generated
    :param scale: boolean to decide how to generate items
    :param test: boolean to decide how to generate items
    :return: list of items
    '''

    if not (scale or test):
        return [i for i in range(1, n+1)]

    if scale:
        return [int((i**2) / 2) for i in range(1, n+1)]

    return [1 for _ in range(n)]


if __name__ == "__main__":
    print("Generating items [DEFAULT]")
    i = gen_items()
    print(i)
    print(len(i))

    print("Generating items [SCALE]")
    i = gen_items(scale=True)
    print(i)
    print(len(i))

    print("Generating items [TEST]")
    i = gen_items(test=True)
    print(i)
    print(len(i))
```

test.py

```
from operator import itemgetter
from time import time

from ant_colony_optimisation import AntColonyOpt
from bin import gen_bins
from item import gen_items


def run_tests(b=10, scale=False):
    '''
    Run all the tests
    :param b: Number of bins
```

```python
    :param scale: Boolean determining the item generation
    :return: List of results
    '''

    res_list = []

    i = 500
    rules = [
        {'p': 100, 'e': 0.9},
        {'p': 100, 'e': 0.6},
        {'p': 10, 'e': 0.9},
        {'p': 10, 'e': 0.6},
    ]

    for r in rules:
        res = run_test(b, i, r['p'], r['e'], scale=scale, verbose=True)
        res_list.append(res)
        print("Test [B=%d, I=500, S=%s, P=%d, E=%.1f]" %
              (b, scale, r['p'], r['e'])
              )
        print("Average fit: %.1f; Average runtime: %.2f s" %
              (res['avg_fitness'], res['avg_runtime']))

    return res_list


def run_test(b_num, i_num, p, e, scale=False, verbose=False):
    '''
    Run a test
    :param b_num: Number of bins
    :param i_num: Number of items
    :param p: Population size
    :param e: Evaporation rate
    :param scale: Boolean determining the item generation
    :param verbose: Boolean determining whether to log all steps
    :return: Dictionary containing results of test
    '''

    res = []
    avg_fitness = 0
    avg_runtime = 0

    total_runtime = 0

    for i in range(5):
        b = gen_bins(b_num)
        i = gen_items(n=i_num, scale=scale)

        t = AntColonyOpt(b, i, p, e, verbose=False)
        t.run()
        t.graph_averages()

        fitness, runtime = t.stats()
        res.append((fitness, runtime))

        avg_fitness += fitness * 0.2
        avg_runtime += runtime * 0.2

        total_runtime += runtime

    log("Test Complete; Time: %d s" % total_runtime, verbose)
    log("Stats: ", verbose)
    log("Average Fitness: %f" % avg_fitness, verbose)
    log("Average Runtime: %f" % avg_runtime, verbose)

    return {
        "res" : res,
        "b_num" : b_num,
        "i_num" : i_num,
        "p" : p,
        "e" : e,
        "scale" : scale,
        "avg_fitness" : avg_fitness,
        "max_fitness" : max(res, key=itemgetter(0))[0],
        "min_fitness" : min(res, key=itemgetter(0))[0],
        "avg_runtime" : avg_runtime,
        "max_runtime" : max(res, key=itemgetter(1))[1],
        "min_runtime" : min(res, key= itemgetter(1))[1],
        "total_runtime" : total_runtime
    }


def show_results(res_list):
    '''
    Show results
    :param res_list: The list of results
    :return: None
    '''

    for i, res in enumerate(res_list):
        print("Results for run %s\n" % str(i+1))
        for j, test in enumerate(res):
            print("PARAMETERS")
            print("B=%d, I=%d, P=%d, E=%f, S=%s" %
                  (test['b_num'], test['i_num'], test['p'], test['e'], test['scale'])
                  )
            print("Fitness - AVG: %6.1f; MAX: %6s; MIN:%6s" %
                  (test['avg_fitness'], test['max_fitness'], test['min_fitness']))
            print("Time - AVG: %6.2f; MAX: %6.2f; MIN: %6.2f\n" %
                  (test['avg_runtime'], test['max_runtime'], test['min_runtime'])
                  )


def log(message, verbose=False):
    '''
    Print message if running in verbose mode
    :param message: Message to print
    :param verbose: Boolean indicating verbose mode
    :return: None
    '''

    if verbose:
        print(message)


if __name__ == "__main__":
    print("Starting")
    start = time()
    total_res = []
    print("Starting First run")
    total_res.append(run_tests())
    print("Finished First run")
    print("Starting Second run")
    total_res.append(run_tests(50, True))
    print("Finished Second run")

    print("Full test complete; Runtime: %.2f" % float(time() - start))
    print("Results\n")
    show_results(total_res)
```