Bin-Packing with Ant Colony Optimisation

ECM3412 - Nature-Inspired Computation

680033128 - 161678

December 17, 2021

I certify that all material in this report which is not my own work has been identified and that no material has previously been submitted and approved for the award of a degree by this or any other university.

1 Introduction and Motivation

The Bin Packing Problem is a NP-hard optimisation problem with many potential applications. Being NP-hard, there is no known optimal algorithm for BPP running in polynomial time. The objective is to equally distribute items of different weights into a finite set of bins. Exact solution methods can only be used for very small instances, so for real-world problems, we have to rely on heuristic methods. In recent years, researchers have started to apply evolutionary approaches to these problems, including Genetic Algorithms and Evolutionary Programming [1]. This paper discusses various nature-inspired algorithms that have been applied to the Bin Packing problem and analyses results from Ant Colony Optimisation (ACO) as a heuristic approach for BPP to find the approximate best solutions for the specified problems.

2 Applications of BPP

The Bin Packing Problem has application in storage [2], transportation planning, multiprocessor scheduling optimisation [3], resource allocation, real-world planning, cutting and packing, and capital budgeting [4]. Many algorithms are used to deal with BPP, such as improved approximation algorithms, particle swarm optimization, and ant colony optimization, genetic approach.

3 Ant Colony Optimisation

In 1991 Dorigo, Maniezzo and Colorni published the first ACO algorithm, known as the Ant System (AS) to solve combinatorial problems [5]. Based on the path-finding abilities of real ants, the Traveling Salesman Problem (TSP) was solved by using a colony of artificial ants using a combination of heuristic information and an artificial pheromone trail to build new solutions. As the quality of solutions built by the ants increases, the pheromone trail becomes stronger. Some smaller TSP instances were able to be solved optimally using AS. Numerous improvements to the original AS have been proposed since its publication, and the AS has been successfully applied to a wide range of problems.

ACO algorithms were originally inspired by the ability of real ants to find the shortest path between their nest and a food source as part of their foraging behaviour when acting as a colony [6]. While walking from food sources to the nest and vice versa, ants deposit on the ground a substance called pheromones, these form a pheromone trail. Other ants can smell these pheromones, and when choosing their way they tend to choose paths marked by strong pheromone concentrations. When several paths are available from the nest to a food source, a colony of ants may be able to exploit the pheromone trails left by the individual ants to discover the shortest path from the nest to the food source and back. There will be more pheromones on the shortest path, influencing other ants to follow this path. After some time, this results in the whole colony following the shortest path. ACO is a multi agent heuristic search approach to difficult combinatorial optimization problems [7].

4 Experiments

Table 1: BPP1: 10 bins and weight i between 1 and 500

Experiment 1: p = 100, e = 0.9				
Run Num	**Final optimal**	**Min**	**Max**	**Avg**
1	2819	2342	3298	2817.2
2	3156	1880	3231	2826.2
3	2202	1557	2857	2298.6
4	3276	2353	3363	2895.6
5	2947	2277	3130	2702.6

Experiment 2: p = 100, e = 0.5				
Run Num	**Final optimal**	**Min**	**Max**	**Avg**
1	2697	1874	2984	2361.0
2	3007	1805	3007	2230.0
3	1950	1950	2608	2276.0
4	2660	2010	2660	2326.0
5	2694	1352	2694	2071.2

Experiment 3: p = 10, e = 0.9				
Run Num	**Final optimal**	**Min**	**Max**	**Avg**
1	949	894	1289	1036.2
2	693	693	1614	1299.2
3	1357	1153	1583	1382.6
4	890	649	1658	1106.8
5	1302	801	1450	1166.0

Experiment 4: p = 10, e = 0.5				
Run Num	**Final optimal**	**Min**	**Max**	**Avg**
1	1372	1027	1627	1347.6
2	2158	1036	2694	1939.8
3	2818	789	2818	1450.0
4	1977	551	1977	1492.0
5	1905	945	2131	1651.2

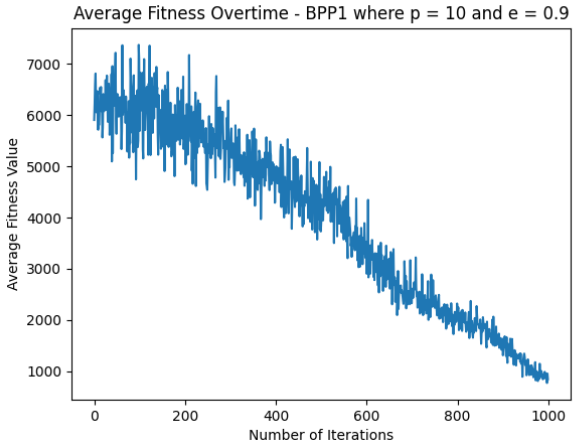


Figure 1: Gradual decrease of average fitness.

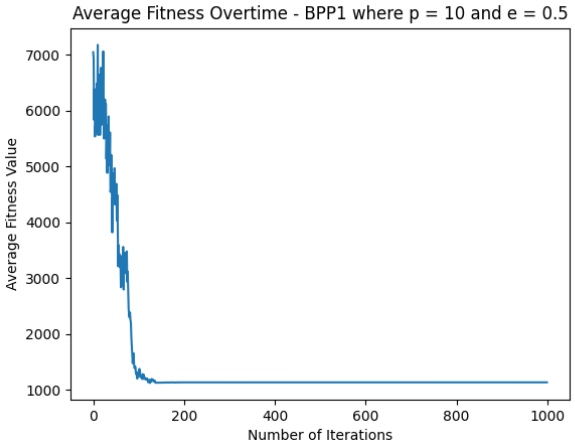


Figure 2: Fast decrease of average fitness

Table 2: BPP2: 50 bins and weight i^2 between 1 and 500

Experiment 1: p = 100, e = 0.9				
Run Num	**Final optimal**	**Min**	**Max**	**Avg**
1	1191200	1061756	1191200	1126174.6
2	1119919	964004	1195869	1076361.4
3	1089146	1076516	1218526	1125957.8
4	1063537	907411	1155050	1048738.6
5	1151138	1037324	1151138	1085560.8

Experiment 2: p = 100, e = 0.5				
Run Num	**Final optimal**	**Min**	**Max**	**Avg**
1	1083963	881531	1083963	1029922.8
2	1090040	859352	1090040	993183.0
3	1072993	835940	1072993	1003413.4
4	965973	938964	1028432	978278.0
5	1125786	889987	1125786	987187.6

Experiment 3: p = 10, e = 0.9				
Run Num	**Final optimal**	**Min**	**Max**	**Avg**
1	1009095	839690	1009095	910594.2
3	904404	904404	1112721	986270.8
3	942044	880360	1162912	992104.2
4	904470	814961	1037024	935891.2
5	1039250	1004102	1133900	1054766.8

Experiment 4: p = 10, e = 0.5				
Run Num	**Final optimal**	**Min**	**Max**	**Avg**
1	1250448	1122200	1389999	1241368.6
2	1122451	1122451	1168417	1142524.6
3	1080041	915596	1290082	1092163.4
4	1326193	1122667	1656517	1345153.2
5	1362849	1104401	1362849	1226250.0

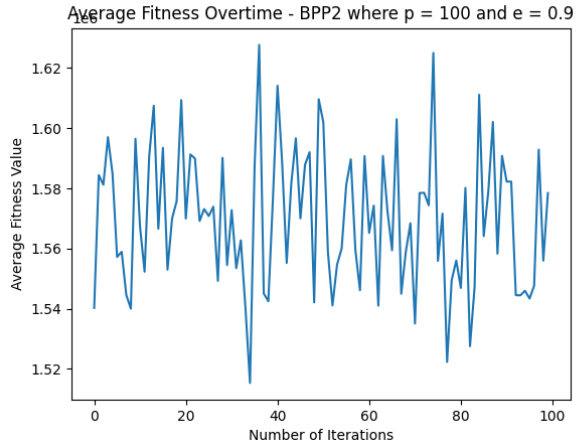


Figure 3: Irregular decrease of average fitness.

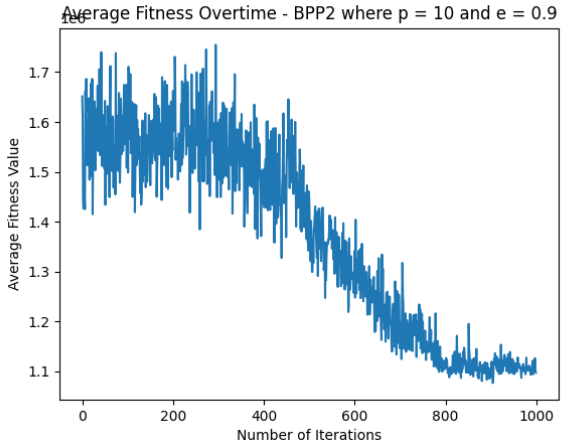


Figure 4: Fast decrease of average fitness

5 Analysis

5.1 Question 1

Which combination of parameters produces the best results? The best combination of parameters found for BPP1 were p = 10 and e = 0.9. For BPP2 the best combination was again p = 10 and e = 0.9.

5.2 Question 2

What do you think is the reason for your findings in Question 1? Both problems demonstrate that higher evaporation values, and hence a lower rate of evaporation, produces better results. This is due to the smoother construction graph that results in a slower rate of convergence. At each decision point, the pheromone values of the next node are similar, giving an approximately equal weight to the selection of each. Thus, the ants traverse more of the graph and, although it may converge slower, it is more likely to approximate the optimal solution.

The graphs show how average fitness results improve with each generation of ants in order to visualise progress of the ants.

Comparing Figures 1 and 2, using pheromone evaporation helps eliminate bad solutions from previous environments. However, too high evaporation rate can result in the loss of good paths as well. From Figure 1 the higher evaporation value and hence lower evaporation rate results in a marginally better solution. However, we can clearly see that Figure 2 converges far more quickly at around 150 generations. This trade-off in accuracy against speed of convergence could be one instance where the evaporation parameter can be balanced to improve the performance of the algorithm.

For BPP1 the smaller population size with either evaporation parameter performed better than the larger ant colonies, as these combinations were better suited to the smaller graph size.

When 100 ant pathways are constructed for p = 100, the ants explore the graph, largely overwriting prior best and worst paths. If a large enough subset of the total population end up traversing a sub-optimal path then each iteration the surroundings routes will be evaporated. With e = 0.5 the better pathways are evaporated more quickly leaving little time to explore other less travelled, but potentially more optimal paths. For the same reason it may be worth exploring trials with a greater number of fitness evaluation iterations to achieve the optimal outcomes, as for BPP2 and harder problems the current limit of 10000 may be insufficient.

For BPP2 I was surprised to see that increasing the population size did not lead to better performance for BBP2, where there are more paths to traverse due to the greater number of bins. One reason for this could be the vast number of ants effectively hiding good solutions as there are many routes to evaluate so the best ants are overlooked. This could be because we are not increasing the size of the graph all that drastically, even though we are now using the square of the item weights, the number of bins has only increases by five times. Perhaps an even larger graph would allowed the ant size to iterate more effectively through the graph or we could experiment with reducing the population size to say 50 ants.

Figure 3 shows how for the large population size and high evaporation parameter, there was essentially little change between generations. The ants appear to be randomly investigating the construction graph. The best fitness was reached around the 35th generation due to the low evaporation rate, after which the result appears to be lost in the noise and tend to deteriorate.

Figure 4 shows a fast rate of convergence which tapers as the evaluation limit is reached for the maximum number of generations. We see here again, that the smaller population has more favourable results.

5.3 Question 3

How do each of the parameter settings influence the performance of the algorithm? All of the pheromone weights in the graph are multiplied by the evaporation value. The lower the value, the fewer bad paths or paths that have not been travelled are found, and the algorithm converges faster. The greater the value, the smoother the construction graph is, and the slower the pace of convergence. If the ants identify a good path, it will be travelled more frequently. A path that was not initially detected will have substantially reduced pheromones and thus is not chosen as often if the evaporation value is smaller. Every iteration, the chance to choose this path is lowered further. Because the pheromone value is halved every evaporation step (e = 0.5), the paths that are not initially chosen or visited as much are reduced more heavily. This leads to the sharp convergence as shown in Figure 2 but does not necessarily lead to the global optimum as can be seen in Figure 1 which has a lower average fitness value meaning a more optimal solution. In essence, we can use the evaporation parameter to influence convergence rate.

The greater ant population size permits them to explore the construction graph more thoroughly. A larger number of ants can travel more pathways in broader setting. This ensures that the majority of paths are travelled and that better ones are more likely to be discovered. If the ant colony is not large enough, pheromones on less-traveled paths will evaporate, further reducing the pheromone weight. Because the pheromones have been decreased to such a little amount, as a result of not being examined as frequently, this could result in a local optimum, as ants will most likely follow a single path. The pheromones for alternative paths are so low, the ants never choose them, resulting in the ants only selecting few paths for the rest of the iterations. Ant size affects graph coverage, resulting in improved fitness and reducing the chances of encountering local optima.

5.4 Question 4

Do you think that one of the algorithms in your literature review might have provided better results? Explain your answer. Note that for the following approaches they include the additional constraint of each bin having a maximum capacity, this is not modelled in our implementation.

Hybrid Grouping Genetic Algorithm (HGGA) is a variant of GA that is modified to accommodate problems in which the goal is to approximate the optimal partition of a set [8]. This combines a grouping based genetic algorithm with simple but effective local search based on the Dominance Criterion of Martello and Toth [9].

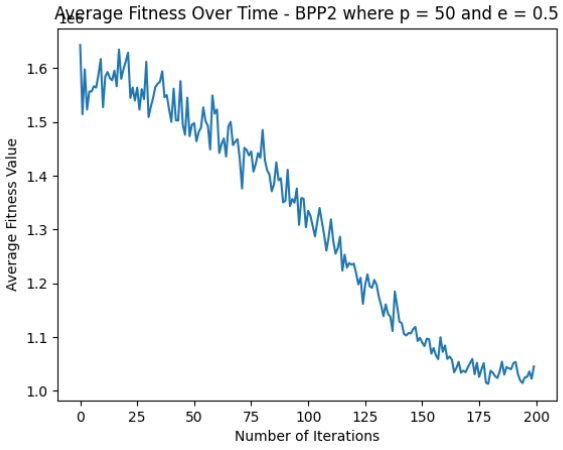
The problem of grouping items into bins is characterised by cost functions that depend on the composition of the groups that is, where one item taken isolatedly has little or no meaning. The grouping genetic algorithm (GGA) is a GA heavily modified to suit the particular structure of grouping problems.

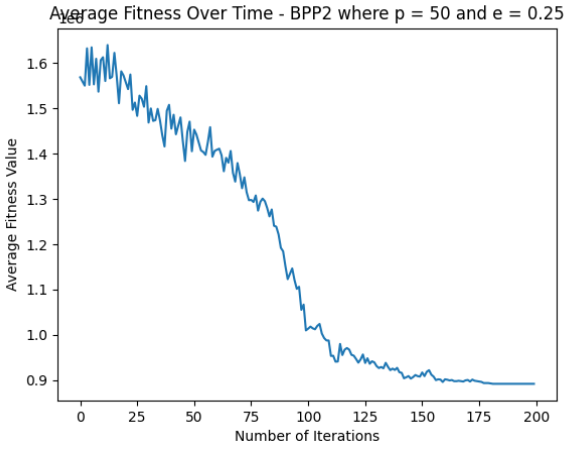
By moving some items around, each ant's solution is improved, and the improved solutions are used to update the pheromone trail. ACO and local search can work together as a complementary partnership [10]. ACO performs a rather coarse-grained search, providing good starting points for local search to refine the results.

Overall, the proposed approach appears to perform well when applied to smaller BPPs. Falkenauer's HGGA is still regarded as one of the best methods for solving the BPP since it combines genetic algorithms with local search. This ACO implementation is quite slow in general though this can be improved by combining it with a simple local search algorithm [1].

In response to Falkenauer's HGGA, Liang *et al* proposed an Evolutionary Algorithm (EA) for the Cutting Stock Problem (CSP) [11]. Both BPP and CSP are classical combinatorial optimisation problems, with BPP being a special case of CSP. This allows us to represent each BPP as an instance of CSP. The problem was represented by chromosomes, as a vectorised, ordered list of all items. A series of cut points split the vector into bins. Much like BPP the items cannot change and hence a basic operator such as the 3-Point Swap (3PS) described in this paper mutates the chromosomes to converge on a global optimum [11]. From here the algorithm follows the same process as other EAs by generating an initial population, selecting the best portion of the population using the fitness function and generating new offspring with the 3PS. This repeats until termination.

6 Further Experiments & Conclusions





Applying the following parameters to BPP2. The two plots appear nearly identicle with the right plot converging slightly sooner. The reduction in population size appears to display a small improvement in the best fitness. Interestingly the evaporation rate for both is significantly smaller than the previous parameters in Figure 4. This is likely due to the the increase in the population size which much be balanced with the evaporation rate. Testing with evaporation 0.9 showed a similar sporadic plot as Figure 3. This could indicate that population is more important than evaporation in determining the fitness of a Bin Packing Problem.

Ultimately, the construction graph for ACO must be considered, as well as the efficiency of the algorithm in determining the optimal parameters. For lengthier searches, a bigger ant population and evaporation value is preferable since it allows for a wider coverage of the construction, resulting in a slower convergence but a better approximate answer. Shorter searches can be accomplished by employing a lower evaporation rate and a smaller ant size to avoid overpopulation. Since simple local search algorithm has proven to greatly improve results [1], further work should combine the approaches.

References

[1] J. Levine and F. Ducatelle, "Ant colony optimization and local search for bin packing and cutting stock problems," *Journal of the Operational Research society*, vol. 55, no. 7, pp. 705–716, 2004.

[2] N. Hoare and J. E. Beasley, "Placing boxes on shelves: a case study," *Journal of the Operational Research Society*, vol. 52, no. 6, pp. 605–614, 2001.

[3] E. G. Coffman, Jr, M. R. Garey, and D. S. Johnson, "An application of bin-packing to multiprocessor scheduling," *SIAM Journal on Computing*, vol. 7, no. 1, pp. 1–17, 1978.

[4] B. Brugger, K. F. Doerner, R. F. Hartl, and M. Reimann, "Antpacking–an ant colony optimization approach for the one-dimensional bin packing problem," in *European Conference on Evolutionary Computation in Combinatorial Optimization*. Springer, 2004, pp. 41–50.

[5] M. Dorigo and L. M. Gambardella, "Ant colony system: a cooperative learning approach to the traveling salesman problem," *IEEE Transactions on evolutionary computation*, vol. 1, no. 1, pp. 53–66, 1997.

[6] M. Dorigo, V. Maniezzo, and A. Colorni, "Ant system: optimization by a colony of cooperating agents," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 26, no. 1, pp. 29–41, 1996.

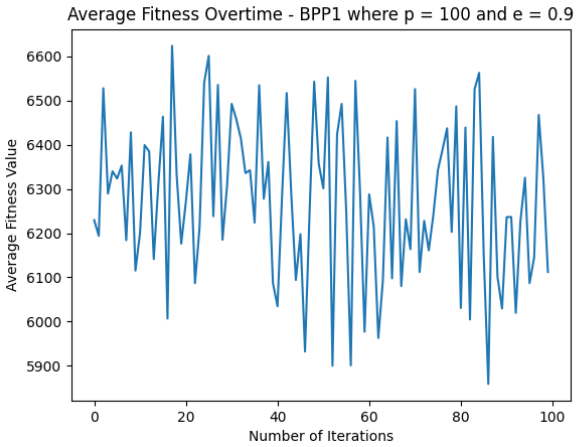
[7] D. S. Johnson, "A brief history of np-completeness, 1954–2012," *Documenta Mathematica*, pp. 359–376, 2012.

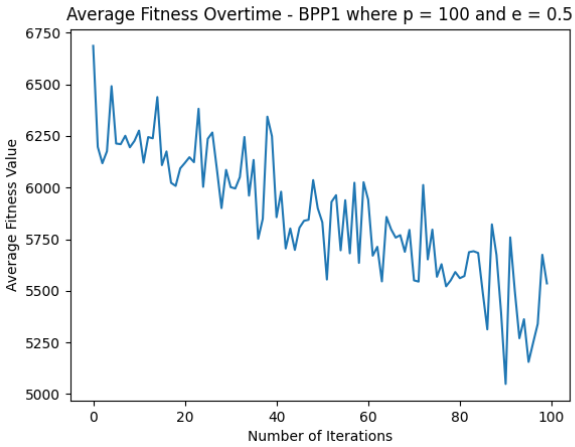
[8] E. Falkenauer, "A hybrid grouping genetic algorithm for bin packing," *Journal of heuristics*, vol. 2, no. 1, pp. 5–30, 1996.

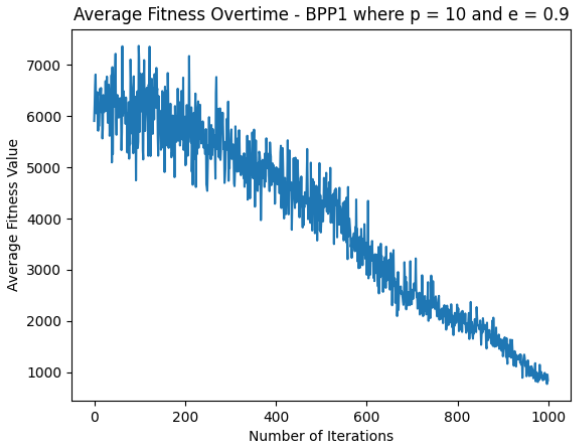
[9] S. Martello, "Knapsack problems: algorithms and computer implementations," *Wiley-Interscience series in discrete mathematics and optimiza tion*, 1990.

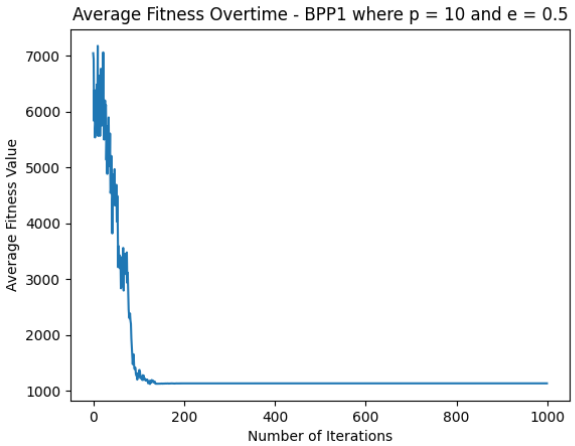
[10] M. Dorigo and T. Stützle, "The ant colony optimization metaheuristic: Algorithms, applications, and advances," in *Handbook of metaheuristics*. Springer, 2003, pp. 250–285.

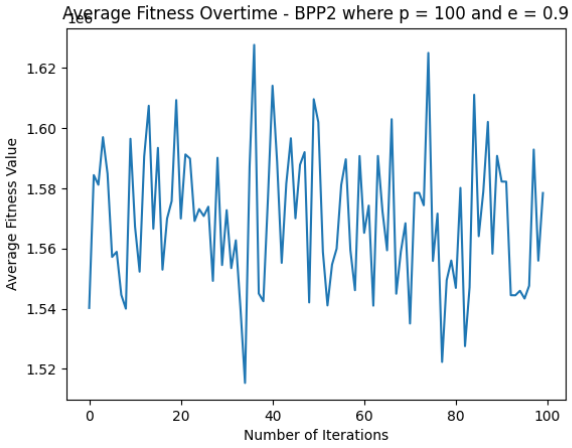
[11] K.-H. Liang, X. Yao, C. Newton, and D. Hoffman, "A new evolutionary approach to cutting stock problems with and without contiguity," *Computers & Operations Research*, vol. 29, no. 12, pp. 1641–1659, 2002.

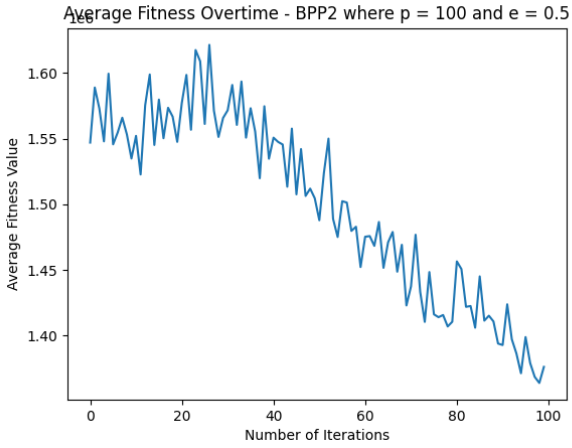


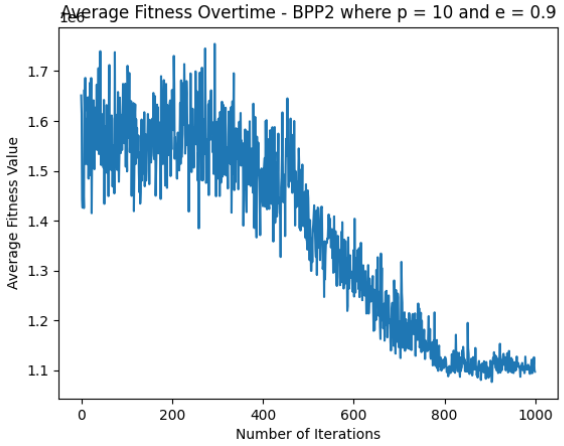


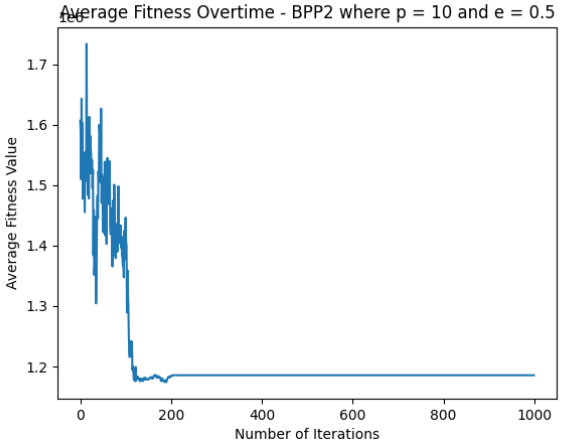


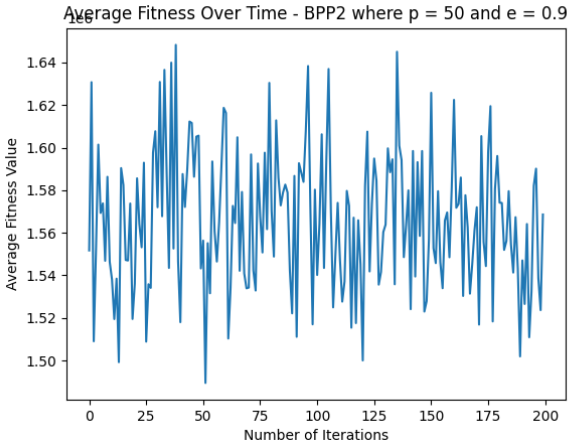


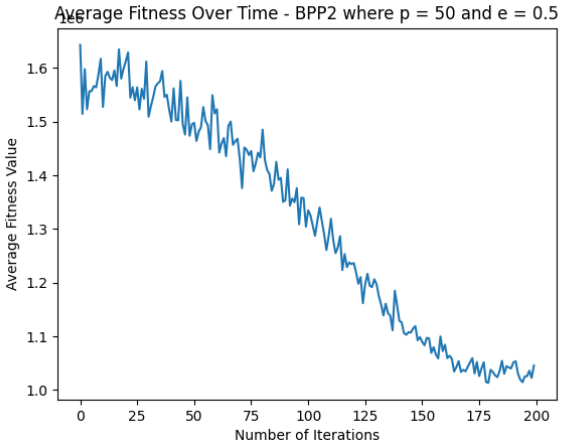

Average Fitness Overtime - BPP2 where p = 100 and e = 0.9
1e6
Average Fitness Value
Number of Iterations

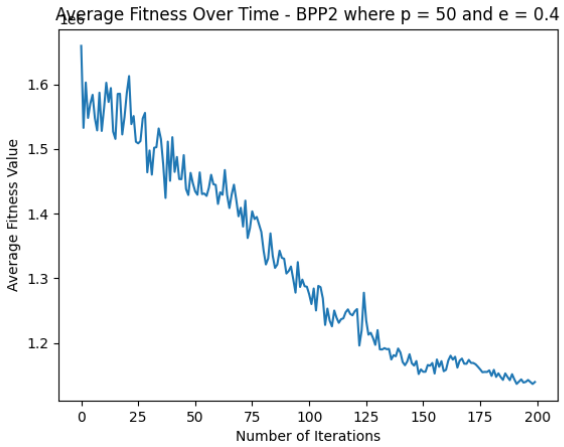

Average Fitness Overtime - BPP2 where p = 100 and e = 0.5
1e6
Average Fitness Value
Number of Iterations

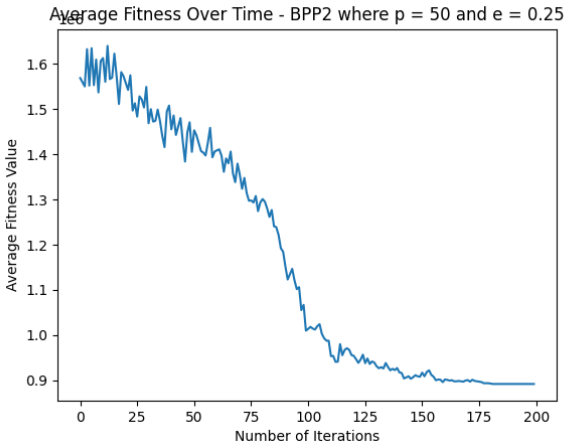

Average Fitness Overtime - BPP2 where p = 10 and e = 0.9
1e6
Average Fitness Value
Number of Iterations


Average Fitness Overtime - BPP2 where p = 10 and e = 0.5
1e6
Average Fitness Value
Number of Iterations


Average Fitness Over Time - BPP2 where p = 50 and e = 0.9
1e6
Average Fitness Value
Number of Iterations


Average Fitness Over Time - BPP2 where p = 50 and e = 0.5
1e6
Average Fitness Value
Number of Iterations


Average Fitness Over Time - BPP2 where p = 50 and e = 0.4
1e6
Average Fitness Value
Number of Iterations


Average Fitness Over Time - BPP2 where p = 50 and e = 0.25
1e6
Average Fitness Value
Number of Iterations

```
from random import random
from time import time
import numpy as np
import matplotlib.pyplot as plt


class Ant(object):
    """
    This class represents an ant and stores the information about its route
and fitness.
    Attributes
    ----------
    route : array(tuple(int, int))
        a list of coordinates that represent the bin-item configuration.
    fitness : int
        the current fitness of the ants route.
    bins : array(Bin)
        holds the bin configuration if the ant is chosen as a generational
champion.
    Methods
    -------
    distribute_pheromones(graph)
        Distributes a pheromone weight on the graph at the positions
defined in the route attribute.
    copy()
        Similar to deepcopy - creates a replica object of the ant.
    get_route_as_str()
        Format the route in a human readable format.
    """
    route = []
    bins = []
    fitness = -1

    def distribute_pheromones(self, graph):
        """Distributes a pheromone weight on the graph at the positions
defined in the route attribute."""
        pheromone_weight = 100.0 / self.fitness
        previous_bin = 0
        for new_bin, item in self.route:
            graph.graph[previous_bin, item, new_bin] += pheromone_weight
            previous_bin = new_bin

    def copy(self):
        """Creates a replica object of the ant."""
        new_ant = Ant()
        new_ant.route = [r for r in self.route]
        new_ant.bins = self.bins.copy()
        new_ant.fitness = self.fitness
        return new_ant

    def get_route_as_str(self):
        """Format the route in a human readable format."""
        return " -> ".join("Item %d in Bin %d" % (point[1] + 1, point[0])
for point in self.route)


class Bin(object):
    """
    This class represents bin that can hold items and caches the current
fitness.
    Attributes
```

```
    ----------
    total_weight : int
        the sum of the items in the bin.
    items : array(int)
        the item weights currently in the bin.
    Methods
    -------
    add_item(item)
        Add an item to the bin and increase the total weight.
    copy()
        Similar to deepcopy - creates a replica object of the bin.
    empty()
        Reset the contents of the bin.
    """
    total_weight = 0
    items = []

    def __repr__(self):
        return "Bin with %d items weighing %d: %s" \
            % (len(self.items), self.total_weight, self.items)

    def add_item(self, item):
        """Add an item to the bin and increase the total weight."""
        self.items.append(item)
        self.total_weight += item

    def copy(self):
        """Creates a replica object of the bin."""
        bin_copy = Bin()
        bin_copy.total_weight = self.total_weight
        bin_copy.items = [item for item in self.items]
        return bin_copy

    def empty(self):
        """Reset the contents of the bin."""
        self.items = []
        self.total_weight = 0


class Graph(object):
    """
    This class represents the pheromone weights across the bin-item matrix.
    Attributes
    ----------
    graph : np.array(int)
        3-d array of pheromone weights.
    evaporation_rate : float
        Scalar value to evaporate the pheromone weights.
    Methods
    -------
    evaporate()
        Reduce the pheromone weights across the graph.
    """
    def __init__(self, bins, items):
        self.graph = np.random.rand(bins, items, bins)
        # print(self.graph)
        # print(np.shape(self.graph))

    def evaporate(self, evaporation):
        """Reduce the pheromone weights across the graph."""
        self.graph *= evaporation
```

```
def generate_bin_items(quantity=500, bpp1=True):
    """This function creates an array of bin weights."""
    if bpp1:
        return [i for i in range(1, quantity + 1)]
    return [(i ** 2) for i in range(1, quantity + 1)]


def create_bins(quantity):
    """This function creates a number of bins and returns them as an
array"""
    return [Bin() for _ in range(quantity)]


class ACO(object):
    """This class holds all relevant information for objects required for
running the ACO algorithm.
    ...
    Attributes
    ----------
    bins : Bin
        a bin object that holds items and a total weight.
    items : array(int)
        an array of integers representing the weights of items.
    ants : array(Ant)
        an array of Ant objects to be controlled during the algorithms run.
    best_ant : Ant
        an ant object - the best ant of the final generation of a algorithm
run.
    graph : Graph
        a graph object to store the pheromone weights.
    num_of_evaluations : int
        the number of routes evaluated.
    limit : int
        the maximum number of evaluations allowed.
    verbose : boolean
        whether or not to print to the console when log() is called.
    ran : boolean
        has the ACO been run.
    runtime : float
        time duration of the last run.
    avg_fitness : array(float)
        the timeseries of average fitnesses over each cycle.
    Methods
    -------
    summary()
        prints a summary of the last run if there is one.
    stats()
        returns the best fitness and time elapsed over last run if there is
one.
    run()
        runs the ACO algorithm.
    explore()
        runs one cycle of route creation and evaporation.
    ant_run(ant)
        reset the ant and recreate its route.
    create_route(ant)
        create a route through the graph of pheromones.
    route_step(prev_bin, item)
        return a step from the current bin to the next bin position.
```

```
    route_fitness()
        calculate the fitness for the current bin configuration.
    set_champion()
        set the best ant for the current generation.
    empty_bins()
        reset all bins.
    log(message)
        prints to the console if verbose is true.
    graph_averages()
        create a graph using the data from avg_fitness.
    """

    def __init__(self, bins, items, population, evaporation, limit=10000,
verbose=False):
        """Initialise the ACO object with the required parameters."""
        self.bins = bins
        self.items = items

        self.ants = [Ant() for _ in range(population)]
        self.best_ant = None

        self.graph = Graph(len(bins), len(items))

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

        self.num_of_evaluations = 0
        self.ran = False
        self.runtime = 0
        self.best_ant = None
        self.avg_fitness = []

    def summary(self):
        """Print a summary of the last run if there is one."""
        if hasattr(self, 'ran') and self.ran:
            print("Run was successful and took %d seconds." %
int(self.runtime))
            print("--- Best fitness: %d" % self.best_ant.fitness)
            # displays the full bin configuration for the best ant in the
final set
            # print("--- Best bin config:")
            # for i, b in enumerate(self.best_ant.bins):
            #     print("%4d. %s" % (i + 1, b))
        return self.best_ant.fitness

    def stats(self):
        """Return the best fitness achieved in the final generation and the
time taken to run the ACO"""
        if hasattr(self, 'ran') and self.ran:
            return self.best_ant.fitness, self.runtime

    def run(self):
        """Runs a full ACO run."""
        self.ran = False
        self.best_fits = []
        self.avg_fitness = []
        start_time = time()

        while self.num_of_evaluations < self.limit:
```

```
            self.explore()

        self.set_champion()

        self.ran = True
        self.runtime = time() - start_time

    def explore(self):
        """Create a route for all ants and evaporate the graph."""
        self.ants = [*map(self.create_path, self.ants)]
        best = None
        for ant in self.ants:
            ant.distribute_pheromones(self.graph)
        fitnesses = [ant.fitness for ant in self.ants]
        self.best_fits.append(min(fitnesses) / sum(self.items))
        self.avg_fitness.append(sum(fitnesses) / len(fitnesses))
        self.graph.evaporate(self.evaporation)

    def create_path(self, ant):
        """Reset the bins and create a route for the given ant.

        :param ant: ant object
        :returns: ant with new path
        """
        self.empty_bins()
        ant = self.create_route(ant)
        ant.bins = self.bins.copy()
        return ant

    def create_route(self, ant):
        """Calculate a route through the pheromone graph."""
        current_bin = 0
        ant.route = []
        for item in enumerate(self.items):
            current_bin, item = self.route_step(current_bin, item)
            ant.route.append((current_bin, item))

        ant.fitness = self.route_fitness()
        self.num_of_evaluations += 1
        return ant

    def route_step(self, current_bin, item):
        """Get the index of the next bin to place the item in.

        :param currentBin: index of the current bin
        :param item: item weight
        :returns: next bin index
        """
        column = self.graph.graph[current_bin][item[0]].tolist()
        total = sum(column)
        threshold = total * random()

        current = 0.0
        for index, weight in enumerate(column):
            if current + weight >= threshold:
                self.bins[index].add_item(item[1])
                return index, item[0]
            current += weight

    def route_fitness(self):
        """Calculate the fitness of the current bin configuration.
```

```
        :returns: current fitness
        """
        max_weight = self.bins[0].total_weight
        min_weight = self.bins[0].total_weight

        for b in self.bins:
            if b.total_weight > max_weight:
                max_weight = b.total_weight
            if b.total_weight < min_weight:
                min_weight = b.total_weight

        return max_weight - min_weight

    def set_champion(self):
        """Allocate the best ant of the generation to the best_ant."""
        for ant in self.ants:
            if self.best_ant and ant.fitness < self.best_ant.fitness:
                    self.best_ant = ant.copy()
            elif not self.best_ant:
                self.best_ant = ant.copy()

    def empty_bins(self):
        """Resets the bin configuration."""
        [b.empty() for b in self.bins]

    def log(self, message):
        """Prints a message to the console if verbose is true."""
        if self.verbose:
            print(message)

    def graph_averages(self):
        """Output a graph to the user based on the values in avg_fitness"""
        plt.plot(self.avg_fitness)
        plt.title("Average Fitness Over Time - BPP" + str(val) + " where p
= " + str(
            population) + " and e = " + str(evaporation))
        plt.xlabel('Number of Iterations')
        plt.ylabel('Average Fitness Value')
        plt.show()


if __name__ == '__main__':
    val = input("Please select BPP problem. Write 1 to run BPP1, 2 to run
BPP2. ")

    if val == "1":
        bins = create_bins(10)
        items = generate_bin_items(quantity=500)
    elif val == "2":
        bins = create_bins(50)
        items = generate_bin_items(quantity=500, bpp1=False)
    # print(items)

    print("Running single ACO trial with %d bins and %d items."
          % (len(bins), len(items)))

    population = int(input("Specify population size: "))
    evaporation = float(input("Specify evaporation rate: "))

    trial = ACO(bins, items, population, evaporation)
```

```
    trial.run()
    trial.graph_averages()

    print("Run took %d seconds." % int(trial.runtime))
    print("Best fitness: %d" % trial.best_ant.fitness)
```

```
from operator import itemgetter
from time import time

from aco import ACO
from aco import create_bins
from aco import generate_bin_items


def run_bpp(num_of_bins=10, bpp1=True, num_of_items=500, num_of_trials=5):
    """Run a full bin packing problem test with the specified parameters
and return the results."""

    results = []
    rules = [
        {'population': 100, 'evaporation': 0.9},
        {'population': 100, 'evaporation': 0.5},
        {'population': 10,  'evaporation': 0.9},
        {'population': 10,  'evaporation': 0.5},
    ]

    # Test each rule configuration.
    for rule in rules:
        print("Current trial Parameters: Bins=%d, Items=%d, BPP1=%s,
Population=%d, Evaporation=%.1f" %
              (num_of_bins, num_of_items, bpp1, rule['population'],
rule['evaporation'])
              )

        result = run_test(num_of_bins, num_of_items, rule['population'],
rule['evaporation'], bpp1=bpp1,  num_of_trials=num_of_trials, verbose=True)
        results.append(result)

        print(" -- Achieved average fitness %.1f in %.2f seconds.\n" %
(result['average_fitness'], result['average_time']))

        print("MIN: %s, MAX: %s, Fitness AVG: %.1f" % (
            result['min_fitness'],
            result['max_fitness'],
            result['average_fitness']
        ))

        print("MIN: %.2f, MAX: %.2f, Time AVG: %.2f\n" % (
            result['min_time'],
            result['max_time'],
            result['average_time']
        ))
    return results


def run_test(num_of_bins, num_of_items, population, evaporation, bpp1=True,
num_of_trials=5, verbose=False):
    """Run ACO trial and return an object of the calculated results.

    :param num_of_bins: b number of bins
    :param num_of_items: k number of items
    :param population: p population size
    :param evaporation: e evaporation
    :param bpp1: boolean indicating which problem to run
    :param numOfTrials: number of wanted trials for each rule
    """
    results = []
```

```
    average_fitness = 0
    average_time = 0
    final_best_ant = 0

    total_time = 0

    # Run 5 tests of the ACO algorithm and compile a set of averages.
    for i in range(num_of_trials):
        bins = create_bins(num_of_bins)
        items = generate_bin_items(quantity=num_of_items, bpp1=bpp1)

        trial = ACO(bins, items, population, evaporation, verbose=False)
        trial.run()

        final_best_ant = trial.summary() # shows the best ant for final
generation of last run

        fitness, time = trial.stats()
        results.append((fitness, time))
        average_fitness += fitness * 0.2
        average_time += time * 0.2

    log("Test finished in %d seconds." % total_time, verbose)
    log("Stats:", verbose)
    log(" -- Average Fitness of Test: %f" % average_fitness, verbose)
    log(" -- Average Time Per Test Run: %f" % average_time, verbose)
    log(" -- Final Best Ant: %d\n" % final_best_ant, verbose)

    return {
        'raw_results': results,
        'bins': num_of_bins,
        'items': num_of_items,
        'population': population,
        'evaporation': evaporation,
        'bpp1': bpp1,
        'final_best_ant': final_best_ant,
        'min_fitness': min(results, key=itemgetter(0))[0],
        'max_fitness': max(results, key=itemgetter(0))[0],
        'average_fitness': average_fitness,
        'min_time': min(results, key=itemgetter(1))[1],
        'max_time': max(results, key=itemgetter(1))[1],
        'average_time': average_time,
        'total_time': total_time
    }


def pretty_print_results(results):
    """This function helps to format the results object in a readable
format."""
    for i, bpp_results in enumerate(results):
        print("\nResult Set For BPP%s\n" % str(i+1))
        for j, test in enumerate(bpp_results):
            print("Test Conditions: - B=%d, I=%d, P=%d, E=%f, BPP1=%s" %
                  (test['bins'], test['items'], test['population'],
test['evaporation'], test['bpp1'])
                  )
            print("Best Ant: %i,  MIN: %d,  MAX: %d,  Fitness - AVG: %6.1f,
MIN: %6.2f,  MAX: %6.2f,  Time - AVG: %6.2f,\n" % (
                test['final_best_ant'],
                test['min_fitness'],
                test['max_fitness'],
```

```
                test['average_fitness'],
                test['min_time'],
                test['max_time'],
                test['average_time']
            ))


def log(message, verbose=False):
    """Output to the console if verbose is true."""
    if verbose:
        print(message)


if __name__ == "__main__":
    print("Starting Full Test...")
    start_time = time()
    total_res = []
    print("Running Bin-Packing Problem 1\nBPP1 Parameters: 10 Bins and 500
Items with i weights\n")
    total_res.append(run_bpp())
    print("Finished BPP1.")
    print("Running Bin-Packing Problem 2\nBPP2 Parameters: 50 Bins and 500
Items with i^2 weights\n")
    total_res.append(run_bpp(50, False))
    print("Finished BPP2.")

    print("Full program executed in %.2f" % float(time() - start_time))
    print("Results...\n\n")
    pretty_print_results(total_res)
```