Dissecting 1D Bin Packing: A Comparative Exploration of Four Algorithmic Paradigms

AREESHA AMIR, Computer Science Junior, Habib University
RANIA SIDDIQUI, Computer Science Junior, Habib University
SARRAH ALI AKBAR, Computer Science Junior, Habib University

ACM Reference Format:
Areesha Amir, Rania Siddiqui, and Sarrah Ali Akbar. 2024. **Dissecting 1D Bin Packing**: A Comparative Exploration of Four Algorithmic Paradigms. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 23 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

The bin packing problem is an NP-hard combinatorial optimization challenge where the goal is to efficiently pack items of various sizes into the fewest bins possible, considering constraints such as weight and priority. [1] This problem has wide-ranging applications in industries including logistics, manufacturing, and computing, where it is crucial for tasks like multiprocessor scheduling, transportation planning, and resource allocation.[2] At its heart, the problem aims to optimize space utilization and reduce costs in transportation and storage by minimizing the number of bins needed. Due to its complexity, finding an optimal solution for large problem instances is computationally intensive. Various algorithmic strategies, such as first-fit, best-fit, and metaheuristics like genetic algorithms and simulated annealing, have been developed to address this problem.

Understanding and tackling the bin packing problem is essential for optimizing resource use in real-world scenarios. In this paper, we discuss four distinct paradigms for solving the bin packing problem, focusing on their methodologies and efficiency in different scenarios:

(1) Brute Force
(2) Dynamic Programming
(3) Approximation Algorithms
    - First Fit Decreasing (FFD)
    - Best Fit
    - Next Fit
(4) Metaheuristic Algorithm:

Authors' Contact Information: Areesha Amir, Computer Science Junior, Karachi, Habib University, aa07613@st.habib.edu.pk; Rania Siddiqui, Computer Science Junior, Karachi, Habib University, rs07494@st.habib.edu.pk; Sarrah Ali Akbar, Computer Science Junior, Karachi, Habib University, sa07207@st.habib.edu.pk.

Manuscript submitted to ACM

- Ant Colony Optimization (ACO)

2 BACKGROUND STUDY

2.1 Brute Force

2.1.1 Algorithm Fundamentals. Brute force is a straightforward way of problem-solving that exhaustively tries all possible solutions and selects the best one. It systematically checks every possible combination or permutation of elements until the optimal solution is found. While brute force algorithms are simple to understand and implement, they can be inefficient for large problem instances due to the exponential growth in the number of combinations to explore.

To tackle the 1D bin packing problem using a brute force approach, the first step is exhaustively trying all possible combinations of item placements within bins. This means considering every permutation of items and all feasible assignments to bins. Simultaneously, each iteration undergoes scrutiny to verify its feasibility, ensuring that the total size of items within each bin does not surpass its capacity. Infeasible combinations are promptly discarded from consideration. Furthermore, the number of bins utilized is calculated for each feasible combination. The optimal solution is then determined by selecting the combination that employs the minimum number of bins.

2.2 Dynamic Programming

2.2.1 Algorithm Fundamentals. Dynamic Programming (DP) is a powerful problem-solving technique used in computer science and mathematics to efficiently solve optimization problems by breaking them down into smaller subproblems and solving each subproblem only once. It is based on the principle of optimal substructure, which states that an optimal solution to a problem contains optimal solutions to its subproblems. This approach is particularly useful for problems with over-lapping subproblems, where each subproblem is encountered multiple times during the computation. Dynamic programming problems often involve defining a state space, that represents the different configurations or states of the problem. Each state corresponds to a sub-problem, and the transition between the states represents the recursive relationship. Dynamic programming algorithms aim to reduce the time complexity of a problem from exponential to polynomial time by avoiding redundant computations.

Dynamic programming can be implemented by two main approaches:

- Memoization: In this approach, solutions to subproblems are stored in a data structure (often a dictionary or array) and retrieved when needed. Memoization is typically used in a top-down approach, where solutions to smaller subproblems are recursively computed and stored.
- Tabulation: In this approach, solutions to subproblems are filled in a table or array in a bottom-up manner, starting from the smallest subproblems and working towards the larger ones. Tabulation is more iterative and may be preferred for its simplicity and efficiency.

By exploiting this property, and storing the solutions to the subproblems in a table or memoization structure, dynamic programming can achieve significant efficiency gains compared to naive approaches.

2.3 Approximation Algorithms

2.3.1 Algorithm Fundamentals. Approximation algorithms are techniques used to find a near-optimal solution to a computational problem in situations when finding the exact optimal solution is computationally infeasible. The 3 main algorithms used for approximation are:

- First Fit Decreasing (FFD): The initial step in this algorithm is to sort the items in order of non-increasing size and then apply the First Fit packing rule [5]. Each item is placed into the first bin that can accommodate it. This approach attempts to reduce fragmentation and improve packing efficiency.
- Best Fit: In this algorithm, items are placed into the bin with the least space left. This means that for each item, the algorithm iterates through all bins to find the one with the smallest remaining capacity that can still accommodate the item. This method aims to minimize wasted space but may lead to more bins being used.
- Next Fit: This is perhaps the simplest algorithm for the classical one-dimensional bin packing problem [5]. When placing an item, if the current bin has enough space to accommodate the item, it's placed there. If not, a new bin is used. However, the main limitation is that once an item is placed in a bin, the algorithm does not search for other bins that might better fit the item.

The final output comprises a list of bins containing the packed items, representing an efficient solution to the 1D bin packing problem achieved through approximation algorithms.

2.4 Ant Colony Optimization

2.4.1 Algorithm Fundamentals. Ant Colony Optimization is a meta-heuristic that can be used to find approximate solutions to difficult optimization problems. [1] A meta-heuristic is a set of algorithmic concepts that can be used to define heuristic methods applicable to a wide set of different problems. ACO gives us a general-purpose algorithmic framework through which we can solve different optimization problems with few modifications. [2]

ACO (Ant Colony Optimization) is modeled after the behavior observed in actual ant colonies. When searching for food, ants explore various paths. The ant that discovers the shortest path can gather food more quickly and complete more trips to the food source than those that take longer routes. As ants travel, they release pheromones along their paths. Consequently, the shortest path, which receives the most traffic, ends up with higher levels of pheromones as the ants travelling through it are making more rounds efficiently. These pheromones are detected by other ants, guiding them to follow the same efficient path, thereby optimizing the route to the food source. It's an iterative algorithm. At each iteration, several artificial ants are considered, each of which builds a solution essentially a path from one vertex to another. The vertex to be visited is selected according to a stochastic mechanism biased by a pheromone. The pheromone values are then modified at the end of each iteration depending on the quality of the solution which then serves as a bias in the future iterations to construct the best solution.

PHEROMONE UPDATE RULE IN ANT COLONY OPTIMIZATION

After all ants have completed their tours, pheromone trails are updated according to the following rule:

$$\tau_{xy} \leftarrow (1-\rho)\tau_{xy} + \sum_{k=1}^{m} \Delta\tau_{xy}^{k}$$

where:

- τ_{xy} is the current pheromone level on the path from x to y.
- ρ is the pheromone evaporation coefficient.
- m is the number of ants.
- $\Delta\tau_{xy}^{k}$ is the amount of pheromone deposited by the k-th ant on the path from x to y, calculated as:

$$\Delta\tau_{xy}^{k} = \begin{cases} \frac{Q}{L_k} & \text{if ant } k \text{ uses path } xy \text{ in its tour} \\ 0 & \text{otherwise} \end{cases}$$

- Q is a constant.
- L_k is the cost or length of the k-th ant's tour.

2.4.2 Previous ACO Applications to solve 1D BPP. One research study improved the ant colony optimization (ACO) algorithm for the bin packing problem by using a tabu matrix tailored to track goods and vehicle interactions. Each column of the matrix corresponds to a specific good, with rows indicating the load visit state and the vehicle used. To further enhance this approach, incorporating an adaptive learning mechanism to dynamically adjust the pheromone evaporation rate based on real-time feedback can be beneficial. This adjustment optimizes the search process and prevents premature convergence by promoting the exploration of new areas in the solution space when stagnation is detected, potentially improving the efficiency and effectiveness of the solution in dynamic or complex packing scenarios. [3]

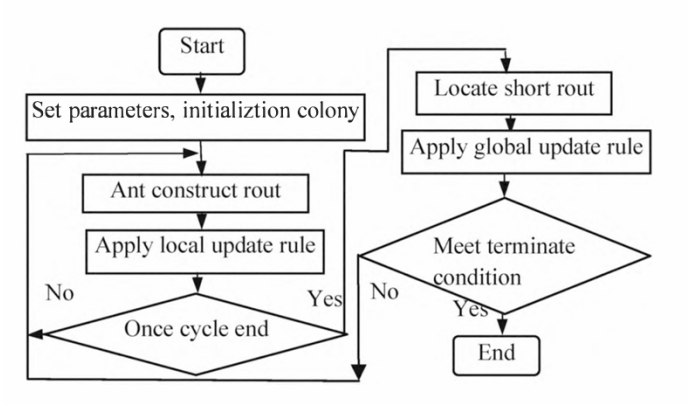

Fig. 1. ACO Procedure for BPP[3]

The procedure includes initializing parameters, placing ants, constructing routes using heuristic information, applying local and global pheromone updates, and iterating until termination conditions are met. The adaptive mechanism helps optimize efficiency and robustness in finding optimal solutions. We will use a similar approach in our implementation.

Algorithm 3 Construction focused on the number of bins (CFNB).

```
CFNB(I, W)
  B ←∅
  do
    N ← all elements of I
    b ← new empty bin
    do
      i←chooseItem(N, b, W)
      if (i ≠ null) then
        b ← b ∪ {i}
        I ← I − {i}
        N ← N − {i}
        for each j ∈N
          if (j does not fit into b) then
            N ← N − {j}
    while (N ≠∅) and (i ≠ null)
    B ← B ∪ {b}
  while (I ≠∅)
  LBH(B)
end
```

Fig. 2. Algorithm for CFNB[4]

Another research had a construction focused on the number of bins (CFNB). It adeptly combined the principles of Ant Colony Optimization (ACO) with the challenges of the Bin Packing Problem (BPP) as given in the algorithm above by minimizing the number of bins used while optimizing their load. CFNB features an innovative pheromone update mechanism tailored to enhance bin utilization; pheromone levels are adjusted based on how well each bin is filled, encouraging ants to select solutions that maximize space efficiency. This method incorporates a pheromone evaporation factor to facilitate continual exploration and avoidance of local optima, ensuring a comprehensive search of the solution space. Through its effective use of ACO strategies, CFNB offers a robust solution to BPP, achieving an efficient balance between bin utilization and load distribution. [4]

3 ALGORITHM APPROACHES AND IMPLEMENTATION

Dataset:

The 1D BPP takes a list of item weights as input whereas the length of this list is taken as the second input which determines the number of items. We designed an algorithm to produce 10,000 randomized weights and store them in a TXT file. The weights were set to be less than equal to 20. This is because we have assumed our bin capacities to be greater than equal to 20 to be able to accommodate all items in our input. We then used this dataset on all the algorithms to generate results and compare them. The code to generate the dataset is given below:

```python
import random

# Number of bins
num_bins = 10000

# Generate random weights for each bin
weights = [random.randint(1, 20) for _ in range(num_bins)]

# Write the data to a text file
with open("bin_sizes.txt", "w") as file:
    for weight in weights:
        file.write(str(weight) + "\n")

print("Data generation complete. Saved to bin_sizes.txt")
```

3.1 Brute Force

To analyse the performance of brute force, we employed the logic of an existing implementation in pascal and then derived a a python solution. Below is the breakdown of the code.
These are the utility functions being used by the functions that will generate all possible paths towards the solution. The can_fit_more function determines if there is at least one box that can fit into a given remaining capacity without already being placed in another bin. The clear_last removes the last appended item in the array.

```
def can_fit_more(boxes, capacity):
return any(box['size'] <= capacity and not box['taken'] for box in boxes)

def clear_last(arr):
    if arr:
        arr.pop()
```

The fit_to_boxes_rec function employs a recursive approach to optimally fit boxes into a bin. It operates by continually updating the "best" configuration when the current arrangement exceeds the best known. Each recursive call tries adding each box that fits and is available, and if no more boxes can be added, it backtracks by removing the last box to test other possible configurations. This method ensures all potential box combinations are considered to maximize the bin's utilized capacity.

```
def fit_to_boxes_rec(boxes, capacity, best, current_bin):
    if sum(current_bin) > sum(best[0]):
        best[0] = current_bin[:]
    if capacity == 0:
        return sum(current_bin)
    max_value = sum(best[0])
    for box in boxes:
        if box['size'] <= capacity and not box['taken']:
            box['taken'] = True
            current_bin.append(box['size'])
            max_value = max(max_value, fit_to_boxes_rec(boxes,
            capacity - box['size'], best, current_bin))
            current_bin.pop()
            box['taken'] = False
    return max_value
```

The fit_to_boxes function efficiently arranges boxes into bins without exceeding each bin's capacity limit. It repeatedly uses the fit_to_boxes_rec function above to fill bins with boxes. The process continues until no more boxes can be accommodated in a new bin. This function also keeps track of the number of bins utilized by incrementing a counter every time a bin is successfully filled, ensuring optimal use of space and resources.

```
    def fit_to_boxes(boxes, total_capacity):
    best = [[]]  # Only one 'best' array is needed as we're looking for any packing solution
    bins = 0

    while can_fit_more(boxes, total_capacity):
        current_bin = []
        value = fit_to_boxes_rec(boxes, total_capacity, best, current_bin)
        print(f"Bin {bins + 1}: {best[0]} with total size: {value}")
        for size in best[0]:
            for box in boxes:
                if box['size'] == size and not box['taken']:
                    box['taken'] = True
                    break
        best[0] = []  # Clear the best for the next round
        bins += 1

    return bins
```

3.1.1 Analysis of Brute Force: The brute force algorithm, exhibits an exponential time complexity, primarily due to its recursive nature and backtracking mechanism. The worst-case scenario, in terms of time complexity, approaches O(n!), stemming from the algorithm exploring every possible permutation of box placements to find an optimal solution. This exhaustive search is conducted by iterating over all boxes to see which can be placed in the current bin, leading to a high branching factor as it recursively attempts to fit the next box. In terms of space, the complexity is O(n), due to the depth of the recursion stack and the storage of arrays tracking current and best placements. Hence, while thorough in finding the best configuration, the algorithm can become computationally expensive with increasing numbers of boxes.

3.2 Dynamic Programming

The 1D bin backing problem is a classic optimization problem where items of different sizes must be packed into a minimum number of bins of fixed capacity. Each bin has a maximum capacity, and the goal is to minimize the number of bins used, while ensuring that all the items are packed without exceeding the capacity of any bin. Dynamic programming can be applied to the 1D bin packing problem by considering the optimal sub-structure and overlapping sub-problems that are inherent in this problem.

Here is below pseducode that shows how dynamic programming can be applied to the 1D bin packing problem:

```
    import time

def bin_packing(items, bin_capacity):
    n = len(items)
    dp = [float('inf')] * (n + 1)
    dp[0] = 0
    last_item_packed = [-1] * (n + 1)

    for i in range(1, n + 1):
        space_left = bin_capacity
        for j in range(i, 0, -1):
            space_left -= items[j - 1]
            if space_left >= 0 and dp[j - 1] + 1 < dp[i]:
                dp[i] = dp[j - 1] + 1
                last_item_packed[i] = j - 1

    # Backtrack to find items packed into each bin
    bins = []
    i = n
    while i > 0:
        j = last_item_packed[i]
        bins.append(items[j:i])
        i = j
    bins.reverse()
    return bins
```

3.3 Experimentation with the algorithm

```
    def read_item_sizes_from_file(file_name, n):
    with open(file_name, 'r') as file:
        items = [int(line.strip()) for line in file]
    return items[:n]

def experiment(n, bin_capacity):
    items = read_item_sizes_from_file(r"C:\Users\Lenovo\Downloads\bin_sizes.txt",n)
    start_time = time.time()
    packed_bins = bin_packing(items, bin_capacity)
    end_time = time.time()
    total_bins = len(packed_bins)
    print(f"Number of items: {n}, Bin capacity: {bin_capacity}")
    print(f"Total bins: {total_bins}")
    print(f"Execution time: {end_time - start_time:.6f} seconds\n")

# Example usage
n = 1000  # Specify the number of items you want to process
bin_capacity = 20  # Define the capacity of each bin
experiment(n, bin_capacity)
```

3.3.1 Analysis of the algorithm. The bin packing algorithm presented above employs a dynamic programming approach to efficiently solve the bin packing problem. Given a list of item sizes that is read from a file which contains 10,000 values(this is the dataset that is used for all algorithms), bin capacity, and number of items we want to process, the algorithm iterates through each item and calculates the minimum number of bins required to pack them.

The time complexity of this algorithm is $O(n \cdot C)$, where n is the number of items and C is the bin capacity. This complexity arises from the nested loops in the algorithm. The outer loop iterates over each item, and the inner loop iterates over the possible positions to pack the current item. Since there are n items and, in the worst case, C positions to consider for each item, the total number of iterations becomes $n \cdot C$.

Moreover, the space complexity of the algorithm is $O(n)$ due to the need to store the dynamic programming array, which has $n + 1$ elements.

However, the algorithm's performance heavily depends on the number of items and the bin capacity, as increasing either of these parameters could significantly increase the computation time.

3.4 Approximation Algorithms

As stated above, the approximation algorithm comprises 3 main ideas.

(1) **First Fit Decreasing Algorithm** The main issue with FFD is that packing large items is difficult, especially if they occur late in the sequence. However, to get rid of this problem we sort the input sequence and place the large items first.

```
def firstFit(weight, n, capacity):
    # Initialize result (Count of bins)
    result = 0
    # Create an array to store remaining space in bins as there can be at most n bins
    bin_rem = [0]*n

    # Place items one by one
    for i in range(n):
        # Find the first bin that can accommodate
        j = 0
        while( j < result):
            if (bin_rem[j] >= weight[i]):
                bin_rem[j] = bin_rem[j] - weight[i]
                break
            j+=1

        # If no bin could accommodate weight[i]
        if (j == result):
            bin_rem[result] = capacity - weight[i]
            result = result+1
    return result

# Returns number of bins required using first fit
def firstFitDec(weight, n, capacity):
    # First sort all weights in decreasing order
    weight.sort(reverse = True)
    # Now call first fit for sorted items
    return firstFit(weight, n, capacity)
```

3.4.1 Analysis of the algorithm. The First Fit Decreasing algorithm for the 1D bin packing problem offers a relatively straightforward approach to allocating items into bins efficiently. Its time complexity primarily depends on the number of items to be packed and the capacity of the bins. After sorting the input, in each iteration, the algorithm scans the existing bins to find the first one that accommodates the current item. While this results in a worst-case time complexity of O(n * m), where n is the number of items and m is the maximum number of bins required, it generally performs well in practice.

(2) **Best Fit Algorithm** The main idea behind this algorithm is to place the next item in such a way that the smallest space is left in the bin.

```
def bestFit(weight, n, c):
    # Initialize result (Count of bins)
    result = 0
    # Create an array to store remaining space in bins as there can be at most n bins
    bin_rem = [0]*n

    # Place items one by one
    for i in range(n):
        j = 0
        # Initialize minimum space left and index of best bin
        min_space_left = c + 1
        best_bin_index = 0

        for j in range(result):
            if (bin_rem[j] >= weight[i] and bin_rem[j] - weight[i] < min_space_left):
                best_bin_index = j
                min_space_left = bin_rem[j] - weight[i]

        # If no bin could accommodate weight[i], create a new bin
        if (min_space_left == c + 1):
            bin_rem[result] = c - weight[i]
            result += 1
        else: # Assign the item to best bin
            bin_rem[best_bin_index] -= weight[i]
    return result
```

3.4.2 Analysis of the algorithm. The time complexity of the bestFit function is primarily determined by its nested loops and the size of the input parameters n (the number of items) and c (the capacity of each bin). Initially, the function initializes an array to store the remaining space in bins, which takes $O(n)$ time. Subsequently, it iterates over each item in the input list, resulting in an outer loop with a time complexity of $O(n)$. Within this loop, there's an inner loop that searches for the best bin to accommodate the current item. In the worst-case scenario, where each item needs its own bin, this inner loop could iterate up to n times. Consequently, the overall time complexity of the bestFit function is $O(n^2)$. However, an optimization strategy involving maintaining a sorted list of bins and using binary search could reduce the time complexity to $O(nlogn)$ by improving the search for the best bin.

(3) **Next Fit Algorithm** This algorithm while choosing the next item checks if it fits in the same bin as the last item. It uses a new bin only if there isn't any capacity in the previous bin.

```python
def nextfit(weight, capacity):
    result = 0
    rem = capacity
    for _ in range(len(weight)):
        if rem >= weight[_]:
            rem = rem - weight[_]
        else:
            result += 1
            rem = capacity - weight[_]
    return result
```

3.4.3 Analysis of the algorithm. It is a simple algorithm. It requires only $O(n)$ time and $O(1)$ extra space to process n items. Next Fit is 2 approximate algorithm, that is the number of bins used by this algorithm is bounded by twice of optimal. Consider any two adjacent bins. The sum of items in these two bins must be > capacity; otherwise, NextFit would have put all the items in the second bin into the first. Thus, at most half the space is wasted, and so Next Fit uses at most 2M bins if M is optimal.

3.5 Ant Colony Optimization

In our implementation, we employed an existing implementation of ACO applied on a adaptation of BPP for load balancing and modified it to focus on minimum number of bins. To map ACO to 1D BPP, we made the following adjustments:

(1) **Bin Capacity:** We defined the bin capacity.
(2) **Pheromone Matrix:** Here we initialised a pheromone matrix. At every iteration, the ants with the best solutions are identified. Pheromones are updated to promote better solutions. Moreover, the pheromone evaporation is applied, over time the pheromone evaporates and this leads to the ants exploring newer solutions. This helps avoid premature convergence to suboptimal solutions.

```python
def GenerateSE(T, k, p, w_i, bin_capacity):
    SE = [[] for _ in range(p)]
    bin_current_weights = [[] for _ in range(p)]  # Dynamic list of bin weights

    for n in range(p):
        for i in range(k):
            b = len(bin_current_weights[n])  # Current number of bins
            probabilities = np.zeros(b + 1)  # Extra space for potentially new bin
            for j in range(b):
                if bin_current_weights[n][j] + w_i[i] <= bin_capacity:
                    if j >= len(T):
                        T.append([random.random()])
                        # Ensure pheromone trail exists for new bin
                    probabilities[j] = T[j][0]  # Use existing pheromone trail
            probabilities[-1] = 1
            # Base probability for opening a new bin

            probabilities /= probabilities.sum()  # Normalize probabilities
            chosen_bin = choice(range(len(probabilities)), p=probabilities)
            if chosen_bin == b:  # New bin
                bin_current_weights[n].append(w_i[i])
                SE[n].append(chosen_bin)
                T.append([random.random()])  # Initialize pheromone for new bin
            else:
                bin_current_weights[n][chosen_bin] += w_i[i]
                SE[n].append(chosen_bin)

    return SE, T
```

The GenerateSE function simulates multiple ants trying to pack items into bins. For each ant, it iteratively attempts to place each item into one of the existing bins based on pheromone trails which suggest the desirability of each bin. If no suitable bin is available, a new bin is created. This function uses probabilistic decision-making where the likelihood of choosing a bin is proportional to its pheromone level, ensuring dynamic learning through experience. The output is a list of decisions for each ant and the updated pheromone levels, enabling a feedback loop that continuously refines the solution approach.

(3) **Ant's Solution:** The number of ants are represented as 'p', and is passed in the ACO function. At every iteration, each ant chooses a bin depending on the pheromone levels. The choice is probabilistic and those bins with high pheromone levels are favored.

```
def ACO_BPP(p, e, item_sizes, bin_capacity):
    iteration = int(60 / p)  # Adjust iteration based on the number of ants
    k = len(item_sizes)
    w_i = item_sizes

    T = [[1.0]]  # Start with an initial pheromone matrix for one bin

    for itr in range(iteration):
        SE, T = GenerateSE(T, k, p, w_i, bin_capacity)
        d = Fitness(SE, p, bin_capacity)
        best_idx = d.index(min(d))

        # Update pheromones for better paths
        for i, bin_idx in enumerate(SE[best_idx]):
            if bin_idx >= len(T):
                T.append([1.0])  # Ensure pheromone list is long enough
            T[bin_idx][0] += 100 / (d[best_idx] + 1)  # Increase pheromone

        # Pheromone evaporation
        for i in range(len(T)):
            T[i][0] *= e

    return None
```

ACO_BPP serves as the main function to orchestrate the Ant Colony Optimization algorithm for the bin packing problem. It sets up initial conditions, including a uniform pheromone distribution, and runs multiple iterations where it continuously generates solutions, evaluates them, and updates pheromone trails based on the best solutions. The function adjusts pheromone levels using both reinforcement from good solutions and evaporation to avoid overly strong biases, facilitating a balance between exploration of new solutions and exploitation of known good ones. This iterative process aims to converge on an optimal or near-optimal solution over time.

(4) **Fitness Function:** The fitness of the solution is measured by the number of bins used. The lesser the bins used, the more the fitness of the solution. We calculate this by counting the non-empty bins. . This function operates by counting distinct bins for each ant's solution, effectively measuring the compactness and efficiency of packing, which are central to optimizing space and reducing material usage in real-world applications.

```
def Fitness(SE, p, bin_capacity):
d = []
for solution in SE:
    if None in solution:
        d.append(float('inf'))
    else:
        d.append(len(set(solution)))  # Fitness is the number of unique bins used
return d
```

3.5.1 Analysis of ACO:. The Ant Colony Optimization (ACO) algorithm for bin packing primarily sees its complexity driven by the `GenerateSE` function, which has a time complexity of $O(p \times k^2)$. This arises from nested loops iterating over p ants and k items, and potentially examining up to k bins per item. The space complexity for storing bin weights and ant assignments is $O(p \times k)$. Additionally, the `Fitness` function contributes a linear complexity of $O(p \times k)$ to evaluate solutions. The overarching `ACO_BPP` function iterates $\frac{60}{p}$ times and holds a dominant time complexity of $O(k^2)$ and space complexity of $O(p \times k)$, reflecting the intensive computational requirements for managing pheromone trails and ant solutions. These complexities highlight the algorithm's significant computational load, which can impact scalability and efficiency as problem sizes grow.

4 OBSERVATION ANALYSIS

4.1 Brute Force

Table 1. Summary of Test Results

No. of Items	Bin Capacity	Execution Time(s)	Total number of bins
5	10	0.152	3
8	15	0.121	4
10	20	0.105	6
20	20	0.113	11
50	20	19.084	29

The assessment of a brute force approach to the bin packing problem based on the provided data reveals several key insights:

(1) **Execution Time:** Execution times are minimal for smaller item counts but rise sharply for larger datasets, illustrating the exponential increase in computational demand. For instance, the execution time jumps from 0.113 seconds for 20 items to 19.084 seconds for 50 items, marking brute force methods as impractical for large scales.

(2) **Bin Utilization:** The number of bins used increases with the number of items. This trend underscores the inefficiency of brute force in optimizing bin space, particularly as item counts grow.

(3) **Scalability and Efficiency:** While computationally feasible for smaller datasets due to fewer combinations, the brute force approach faces significant scalability issues with larger datasets. The substantial rise in execution time highlights the need for more advanced algorithms capable of pruning the search space and learning from suboptimal configurations to enhance efficiency.

In conclusion, although the brute force method can deliver exact solutions for small datasets, its performance degrades significantly under larger scales. In our case, it exceeded time limit for 100 items.This inefficiency calls for employing more sophisticated algorithms that optimize computational resources while approaching or achieving optimal packing solutions.

4.2 Approximation Algorithms

To analyse the performance of the 3 different types of approximation algorithms in 1D BPP, We experimented by changing different parameters including the number of items and the bin capacities and generated the solutions. In each iteration, the bin capacity is kept constant at a value of 20 and the input size is changed, as shown in the table.

Table 2. Summary of Test Results

No. of Items	Total of bins - FFD	Total of bins - Best Fit	Total of bins - Next Fit
5	3	4	3
20	11	11	13
50	27	29	34
80	45	47	56
120	69	71	85

The Next Fit Algorithm offers simplicity and low computational overhead, making it suitable for low-end hardware with limited resources. However, it tends to use more bins and may not always find the optimal solution. Best Fit minimizes wasted space by selecting the smallest remaining capacity bin for each item, but it requires additional computational overhead and may not always achieve optimal results. First Fit Decreasing generally outperforms both algorithms by minimizing the number of bins used, striking a balance between performance and complexity. While it requires sorting the items, this overhead is manageable for moderate-sized inputs. Considering hardware constraints, Next Fit is best for low-end devices, while Best Fit and First Fit Decreasing are suitable for moderate to high-end hardware, with First Fit Decreasing often preferred for its superior bin utilization.

4.3 Dynamic Programming

To analyse the performance of dynamic programming in 1D BPP, We experimented by changing different parameters including the number of items and the bin capacities and generated worst and best solutions. For the size of each item, we utilized a common dataset for all our algorithms. The file contains 10,000 values each value indicating the size of the item.

Table 3. Summary of Test Results

No. of Items	Bin Capacity	Execution Time(s)	Total number of bins
500	20	0.003408	351
1000	20	0.031349	692
5000	20	0.582006	3430
8000	20	1.582916	5491
10000	20	2.377613	6857

The above table shows the results of bin packing problem experiment, on how changing the number of items affects the performance of dynamic programming in solving the bin packing problem. The table shows that the number of items affect both fitness and the execution time of the solution. Fitness, in this context refers to how good the solution is, which is measured by the number of bins used.

- **Number of items** : As the number of items increases, the number of bins needed to pack all the items also increases. This is because there is a limited amount of space in each bin, and so more bins are needed to hold more items.
- **Execution time**: The execution time also increases as the number of items increases. This is because the dynamic programming algorithm has to consider more possibilities as the number of items increases.

4.4 Ant Colony Optimization

To analyse the performance of ACO in 1D BPP, we generated the best and worst solutions produced by the artificial ants. We experimented by changing different parameters including the number of items, the bin capacities, the number of ants, the pheromone evaporation rate and the number of iterations.

Table 4. Summary of Test Results

Test ID	No. of Items	Capacity	No. of Ants	Evaporation Rate	Iterations	Best Solution	Execution Time (s)
1	10	20	20	0.5	10000	6	4.893
2	50	25	30	0.8	1500	18	2.36
3	100	20	25	0.4	1200	59	4.75
4	500	30	15	0.7	800	38	2.86
5	1000	20	15	0.8	800	571	69.11
6	5000	25	30	0.9	60	2226	121.086
7	8000	50	30	0.4	60	3544	169.455

4.5 Impact of Tuning Parameters on ACO Performance

The analysis of the Ant Colony Optimization (ACO) algorithm across various settings highlights the importance of tuning key parameters—number of ants, evaporation rate, and iterations—on its performance and efficiency:

- **Number of Ants:** Increasing the number of ants tends to improve solution quality by enhancing the exploration of the solution space, but this also leads to longer execution times. This trend is evident in test cases with more ants achieving better outcomes but requiring more time.
- **Evaporation Rate:** The evaporation rate dictates the persistence of pheromone trails. Lower rates increase exploration by reducing the impact of existing trails, potentially avoiding premature convergence at the cost of achieving less optimal solutions, as observed in higher bin counts with lower evaporation rates.
- **Iterations:** More iterations permit more thorough exploitation of the pheromone trails, improving solution quality. However, this benefit comes at the cost of increased computational time. Scenarios with fewer iterations show constrained algorithm performance, especially with large sets of items.

Conclusion: Properly balancing the number of ants, the evaporation rate, and the number of iterations is crucial for optimizing ACO performance. These parameters significantly affect both the computational load and the quality of the solutions, as demonstrated by the varying outcomes in different test configurations.

5 COMPARATIVE ANALYSIS

Table 5. Comparison of Bin Packing Algorithms

Test ID	No. of Items	Bin Capacity	BF	Approximation	DP	ACO
1	10	20	6	6	8	6
2	20	20	11	11	14	11
3	50	20	27	27	35	29
4	100	20	-	57	70	59
5	200	20	-	111	139	115
6	500	20	-	277	351	290
7	1000	20	-	547	692	576
8	5000	20	-	2664	3430	2226
9	10000	20	-	5299	6857	5545

5.1 Efficiency Comparison

To analyze the efficiency of the algorithms, we plotted graphs for the visual representation of the results generated by Brute Force, Approximation Algorithm (First-Fit Decreasing), Dynamic Programming (DP) and Ant Colony Optimization (ACO).

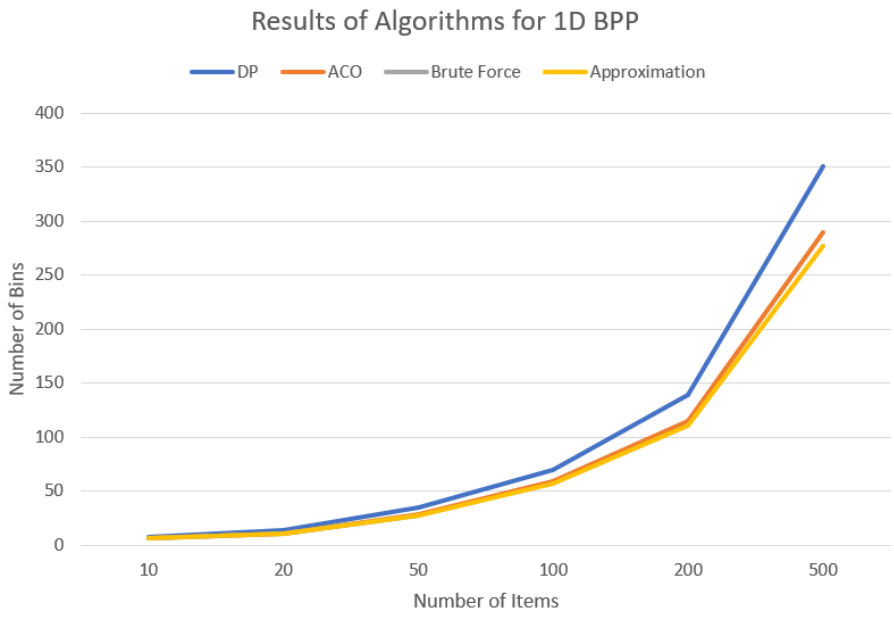

Fig. 3. Algorithm's Solutions for upto 500 items

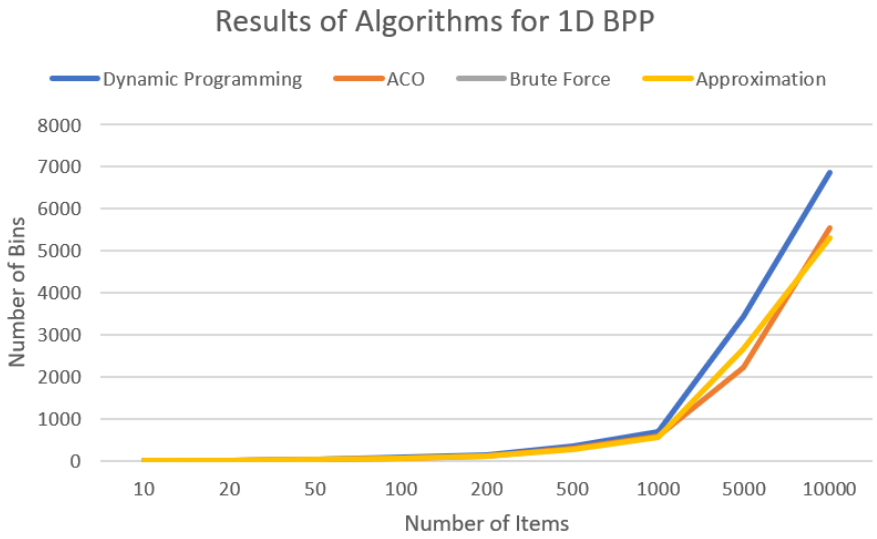

Fig. 4. Algorithm's Solutions for upto 10,000 items

The comparative efficiency analysis of different bin packing algorithms, as detailed in the Table 5 and Figure 3 and 4, highlights distinct performance trends across increasing dataset sizes. Both Dynamic Porgramming and Approximation algorithms (FFD) maintained consistent performance with Ant Colony Optimization (ACO), closely matching or slightly improving on the number of bins used, especially noticeable in smaller item counts. However, as the number of items grew, ACO and FFD showed a clearer advantage in terms of scalability and efficiency, particularly in larger problem instances in figure 4 (items ranging from 1,000 to 10,000). For instance, while Dynamic Programming (DP) scaled poorly, often requiring significantly more bins, ACO was more efficient, using fewer bins compared to both DP for all item sizes tested. Notably, in the largest dataset of 10,000 items, ACO used about 5545 bins and FFD used 5299 bins, both being substantially fewer than DP's 6857 bins, demonstrating its superior efficiency in managing larger scales effectively. This efficiency is indicative of ACO's ability to optimize solutions through iterative improvements and adaptive pheromone-based exploration, making it an advantageous approach for complex, high-volume bin-packing challenges and FFD's heuristic approach which involves sorting items in decreasing order of size and places each item into the first bin that can accommodate it, making it widely used to solve 1D bin-packing problem.

5.2 Accuracy and Optimality

(1) **Brute Force Algorithm**: Ensures the highest level of accuracy by exhaustively searching every possible configuration, thus delivering an optimal solution for small datasets. Its effectiveness is bound by its factorial time complexity, which restricts its practicality to smaller problems.

(2) **Dynamic Programming (DP)**: It is good for finding near optimal solutions and generates them efficiently however in terms of number of bins, it gaves a larger number compared to others.

(3) **Approximation Algorithms (First Fit Decreasing)**: FFD is efficient to implement. While it doesn't guarantee optimal solutions, it often provides good results quickly, making it suitable for large-scale instances, and thus effective in real-time applications where speed is critical. While this algorithm does not guarantee an optimal solution, it balances solution quality and computational efficiency, thus typically outperforming others in terms of minimizing the number of bins used as seen in the table above.

(4) **Ant Colony Optimization (ACO):** Offers solutions that are close to optimal by simulating natural processes. While it does not guarantee the absolute best solution, ACO is adept at exploring and exploiting complex problem spaces, making it highly effective for larger and more intricate datasets as seen in Figure 4.

5.3 Scalibity and Robustness

Brute Force Approach: Suffers from extremely poor scalability due to its exponential time complexity, rendering it impractical for large datasets.

Dynamic Programming: Offers moderate scalability for larger problems than brute force by utilizing a polynomial time complexity ($O(n \cdot C)$). However, it is limited by substantial memory requirements.

Approximation Algorithms: Exhibit excellent scalability with generally linear or near-linear time complexities. These algorithms maintain robust performance across various problem sizes, making them well-suited for real-time applications where speed is essential.

Ant Colony Optimization (ACO): Demonstrates superior scalability and robustness, efficiently managing larger and more complex datasets. The flexibility of ACO's parameter tuning enables dynamic adjustment to varying problem sizes, optimizing computational resources effectively.

6 CONCLUSION

We analyzed and implemented various bin packing algorithms, including Brute Force, Approximation algorithms, Dynamic Programming and Ant Colony Optimization (ACO). Our comparative analysis, showcased across different data visualizations, revealed that while Approximation algorithms consistently performed well across various dataset sizes, ACO demonstrated significant advantages in scalability and efficiency, particularly for larger problem instances. We observed that the Brute Force algorithm, though highly accurate for small datasets, is impractical for larger sizes due to its poor scalability. Dynamic Programming proved efficient for moderate-sized datasets but gave poor results as it returned large number of bins as compared to other algorithms. In contrast, ACO adapted dynamically to complex problem sizes, optimizing computational resources effectively and showcasing superior performance.

For future research, exploring additional diverse datasets will be crucial to further test the limits and capabilities of these algorithms. Exploring genetic algorithms could also uncover new insights into its adaptability and efficacy. Additionally, investigating hybrid approaches that combine features of multiple algorithms might lead to innovative solutions that enhance efficiency and accuracy in bin packing and other areas of optimization.

7 APPENDICES FOR CODE

Below is the github link that has the code for our project:
https://github.com/AAreesha/The-Bin-Packing-Problem

ACKNOWLEDGMENTS

To Professor Shah Jamal Alam <3

REFERENCES

[1] K. Fleszar and K. S. Hindi, "New heuristics for one-dimensional bin-packing," *Computers & Operations Research*, vol. 29, no. 7, pp. 821–839, 2002. doi: 10.1016/S0305-0548(00)00082-4.

[2] M. Dorigo, M. Birattari, and T. Stützle, *Ant Colony Optimization: Artificial Ants as a Computational Intelligence Technique*. Brussels, Belgium: Université Libre de Bruxelles, 200X.

[3] S. Wang and R. Shi, "Study on Improved Ant Colony Optimization for Bin-Packing Problem," in *Proceedings of the 2010 International Conference on Computer Design and Applications (ICCDA)*, Northeastern University at Qinhuangdao, Qinhuangdao Hebei, China, 2010.

[4] O. D. Lara and M. A. Labrador, "A Multiobjective Ant Colony-based Optimization Algorithm for the Bin Packing Problem with Load Balancing," in *Proceedings of the 2010 IEEE Congress on Evolutionary Computation (CEC)*, August 2010, doi: 10.1109/CEC.2010.5586259.

[5] C. Coffman, M. Garey, and D. Johnson, "Approximation Algorithms for Bin Packing," *Bin Packing Survey*, pp. 2, 1997, doi: 10.14778/3055543.3055550.

[6] GeeksforGeeks, "Bin Packing Problem - Minimize Number of Used Bins," GeeksforGeeks, Available online: https://www.geeksforgeeks.org/bin-packing-problem-minimize-number-of-used-bins/.