The
BRITISH UNIVERSITY
IN EGYPT

Bin Packing Problem Solver: Analyzing Our Implementation

25CSCI05I

Project 7

T.A. Omar

Bin Packing Problem Solver

Table of Contents

Section 1: Title Page

Github repo: https://github.com/Ninazu00/Bin-packing-solver

#	Name	Student ID	Sections Contributions

1	Mohamed Ayman	239236	Implementation:
			• class Individual
			  ○ getFillRate(self,binSize)
			  ○ addItem(self,itemID,itemSize,binSize)
			• updateBeliefs(selectedIndividuals,beliefs)
			• weightedPick(choices, top5)
			• applyBeliefs(beliefs,totalItems)
			• initializeTotalItems(minSize,maxSize,numItems)
			• generateBinCulturalAlgorithm(maxGenerations, populationSize, mutationRate, totalItems, binSize)
			• culturalAlgorithmFullSolve(populationSize,mutationRate,maxGenerations,totalItems,binSize,bestBin,GUI)
			• drawBinFillRight(self,fillRate, binNumber)
			• Created the code to run cultural algorithm
			• Responsible for data visualization in the GUI for both algorithms
			• usermanual.pdf
			• readme.txt
			Report:
			• Table of Contents
			• Section 4
			  ○ 4.4: Flowchart for Cultural Algorithm
			• Section 6
			  ○ 6.4 Scalability Analysis
			  ○ 6.5 GUI Screenshots and User Experience

2	Ramy EmadEldin	242781	• evaluateFitness(individual)

			• selectAccepted(population)
			• mutate(individual)
			• Section 3: Brief Introduction
			  ○ 3.1 Brief introduction about problems statements and importance
			  ○ 3.2 Brief summary of key findings from phase 1 Literature Review
			  ○ 3.3 Objectives for Phase 2
			  ○ 3.4 Overview of your implementation approach
			• Section 5: Experimental Setup for Cultural Algorithm
			  ○ 5.1 Test Cases and Problem Instances
			  ○ 5.2 Parameter Configuration
			  ○ 5.3 Experimental Environment
			• Section 6 (Cultural Algorithm)
			  ○ 6.3 Complexity Analysis
			• Section 7:
			  ○ 7.1 Interpretation of Results
			  ○ 7.2 Algorithm Comparison and Trade-offs
			  ○ 7.3 Comparison with Literature (Connection to Phase 1)
			• Section 9: Conclusion

3	Mostafa Makram	249032	Implementation (Backtracking): • sortItems(itemsList) • placeItem(item, binIndex, usedBins, binRemainingCapacities) • removeItem(item, binIndex, usedBins, binRemainingCapacities) • findFeasibleBins(item, binRemainingCapacities) • backtrack(currentIndex, usedBins, binRemainingCapacities, binCapacity, bestSolution, sortedItemsList) • Created the utils.py file and moved the Backtracking Algorithm helper functions there • GUI:

			○ Instructions (The text section at the top of the GUI window that guides the user) ○ Inputs ○ Dropdown Report: ● Section 4: ○ 4.1 Algorithm Description for Backtracking ○ 4.2 Pseudocode for Backtracking Algorithm ○ 4.3 Flowchart for Backtracking Algorithm ○ 4.4 Implementation details for Backtracking Algorithm ○ 4.5 Code Snippets for Backtracking ● Section 7: ○ 7.4 Challenges and Solutions for Backtracking Algorithm ○ 7.5 Limitations for Backtracking Algorithm Graphs (Only the visuals, not the data values): ○ 6.1 Individual Algorithm Results ■ Execution time Graph for Cultural Algorithm (Figure 1) ■ Execution time Graph for Backtracking Algorithm (Figure 2) ○ 6.2 Comparative Analysis ■ Execution time vs. problem size (I also tested the 2 algorithms and entered their values in the graph) (Figure 3) ○ 6.3 Complexity Analysis ■ Empirical Complexity Graph for Cultural Algorithm (Figure 4) ■ Empirical Complexity Graph for Backtracking Algorithm (Figure 5)
4	Abdelrahman Ahmed	247024	● calculateLowerBound(remainingItems, binCapacity) ● pruneBranch(usedBins, bestSolution, remainingItems, binCapacity) ● copyBins(bins) ● initializeSolution(items, binCapacity) ● solveBinPacking(items, binCapacity)

			• drawBinFillLeft(self, bestSolution)
			• Responsible for running both algorithms in the GUI depending on the dropdown list selection.
			• Section 5:
			  ○ 5.1 Test Cases and Problem Instances (backtracking)
			  ○ 5.3 Experimental Environment (backtracking)
			• Section 6:
			  ○ 6.1 Individual Algorithm Results (backtracking)
			  ○ 6.3 Complexity Analysis: Time & Space (backtracking)

5	Yousef Hazem	242850	Cultural Algorithm implementation:
			• def crossOver(parent1, parent2, childPopulation,mutationRate)
			  ○ def createChild(firstParent, secondParent)
			• def initializePopulation()
			• def generateNewGeneration(childPopulation, newPopulation)
			Report Sections:
			• Section 2: Executive Summary
			• Section 4(cultural):
			  ○ 4.1 - Algorithm Description
			  ○ 4.2 - Pseudocode
			  ○ 4.4 - Implementation Details
			  ○ 4.5 - Code Snippets
			• Section 6(cultural):6.1 - Individual Algorithm Results
			• Section 7(cultural):
			  ○ 7.4 - Challenges and Solutions
			  ○ 7.5 - Limitations
			• Section 8: future work
			• Section 10: references

Section 2: Executive Summary

- **Brief project overview**

This project focuses on the Bin Packing Problem which is a NP-hard problem that focuses on minimizing the bins used to put in a set of items without filling them up. BBP can be used in many fields like resource allocation, management of memory and manufacturing optimization.

The last phase focused on ten algorithms, Backtracking, Branch and Bound, Fast Heuristics, First and Best Fit Decreasing, Approximation methods, Genetic algorithms, Ant colony optimization (ACO), and Reinforcement learning-based models. The findings of testing these algorithms were that Backtracking and the Branch-and-Bound are efficient when addressing small problems but are computationally infeasible when applied to large datasets. Moreover, some solutions could be more flexible and robust like hybrid and metaheuristic algorithms like ACO and reinforcement learning which are applicable to the large scale datasets.

This phase of the project is devoted to two algorithms to solve BPP: Backtracking, an exact method which guarantees the best solutions to small problems and Cultural Algorithms, a metaheuristic one, which is intended to offer high-quality solutions to large or more complicated problems cases.

This aims at the implementation and testing of these two algorithms based on the quality of the solutions, computational efficiency, and scalability, giving information on the relative strength of the algorithms and their capacity to be applicable across various BPP situations. The work provides the basis of choosing a proper strategy to reach a compromise between the accuracy and performance in the real-world application of the Bin Packing Problem.

Key findings from implementation and experiments

1. **Backtracking**: Gives best solutions for small instances and as the number increases its performance drops sharply.
    a. For small instances it was found that the solution given was optimal
    b. For large instances runtime increases and as the result the search nature of the algorithm is exhausted and produces less accurate solution
2. **Cultural Algorithm**: it produces better results as the item sets increase
    a. For small data items close to optimal solutions are produced
    b. Increasing the data items results in better solutions and also increases runtime.

- **Main conclusions from comparative analysis**

Backtracking is not scalable but ensures optimality.The algorithm is not very strong in large problems. The computational cost and time also grow exponentially with the number of items, and it is not feasible in medium and large-size bin-packing problems.

The Cultural Algorithm offers solutions with high efficiency that are almost optimum. It has constantly generated good quality solutions with minimal run times. Even though it does not provide the best solution, it was usually close to Backtracking in performance.

Section 3: Brief Introduction

3.1 Brief introduction about problems statements and importance:

The Bin Packing Problem remains one of the most widely studied NP-hard optimization problems due to its direct relevance in real -world industries such as manufacturing, cloud resource allocation, warehouse management and materials cutting. At its core the problem asks how to pack a set of items into the minimum possible number of bins without exceeding capacity constraints. Although the formulations seem simple, finding the optimal packing becomes extremely difficult as the number of items increases. This makes the problem a perfect testbed for exploring both classical optimization algorithms and modern AI -driven methods.

The importance of solving Bin Packing Problems efficiently becomes clear when examining cases such as inventory management, production planning or industrial cutting. Even small improvements in packing efficiency can translate into reduced material waste, lower operation costs and more sustainable resource usage. For this reason Bin Packing Problems has attracted decades of research and inspired entire families of algorithms ranging from simple greedy heuristic to complex meta -heuristic and learning based systems.

3.2 Brief summary of key findings from phase 1 Literature Review:

During phase 1, we explored a wide variety of algorithms for solving the Bin Packing Problem each belonging to a different family of computational approaches. These included exact algorithms, heuristics approximation techniques and AI driven metaheuristics. The key takeaway from the literature review is that no single algorithm performs best in all scenarios and each approach involves trade-offs between accuracy, scalability and computational cost.

Exact methods such as Backtracking and Branch and bound guarantee the optimal solution. They work extremely well for small or structured items sets but scale poorly because of exponential search growth. Researchers emphasized that improved pruning and preprocessing can make these methods surprisingly efficient on specific benchmark instances, but their worst case runtime remains exponential.

Heuristic and approximation methods such as first fit decreasing , best fit decreasing, harmonic and fast scalable heuristics aim for speed and scalability. They run extremely fast and usually produce near optimal results but they cannot guarantee perfect solutions. Their performance is predictable and consistent which makes them ideal for large dataset or real time applications. Their main weakness is reduced accuracy when item distributions are irregular.

Metaheuristics and AI based methods including Genetic Algorithms, Ant Colony Optimization and reinforcement learning, showed strong packing quality and high adaptability. These methods learn patterns , explore the search space stochastically and often outperform classical heuristics in complex datasets. However they demand significant computational resources and careful parameter tuning. Reinforcement learning approaches such as PermRL, demonstrated excellent real - world performance but required GPUs and advanced model structures.

3.3 Objectives for Phase 2:

The objective of Phase 2 is to design and implement a bin packing solver that balances solution quality, runtime efficiency and algorithmic simplicity and after reviewing the research in Phase 1, we identified the need for an approach that:

1. Product high quality packings, ideally close to optimal.
2. Scales reasonably well with large sets of items
3. Does not require advanced hardware or overly complex models.
4. Combines the strengths of exact and heuristic methods without inheriting their weaknesses.
5. Reflects modern hybrid algorithm trends documented in recent research.

Based on this we chose to implement two complementary approaches:

- Backtracking Search Algorithm
- Cultural Algorithm

These two algorithms represent fundamentally different strategies. Backtracking guarantees optimality but needs pruning to remain efficient. Cultural algorithms rely on evolutionary learning through beliefs passed between generations, offering adaptability and speed.

By implementing both we aimed to compare how an exact method behaves against an evolutionary AI based method, while also exploring how their strengths might complement each other in hybrid decisions.

3.4 Overview of your implementation approach:

Our Phase 2 implementation consists of two fully functional solvers. Backtracking based exact solver and cultural algorithm evolutionary solver. Both were written from scratch, reflecting the concepts and techniques studied in Phase 1.

a. Backtracking Search Algorithm

The Backtracking implementation follows a classical exact search model but includes several optimizations:

- Items are sorted in descending order, reducing branching early on.
- A lower bound calculation estimates the minimum number of bins needed.
- A pruning function eliminates branches that cannot beat the current best solutions.
- The algorithm tries placing each item into:
    1. All feasible existing bins
    2. Or a newly opened bin

The Solver systematically explores different packing configurations, updating the best solution whenever it finds a packing that uses fewer bins. This structure mirrors the decision tree model described in Phase 1 and is particularly powerful for the datasets where pruning significantly reduces the search space.

Although back tracking is exponentially complex, our implementation leverages heuristics to reduce unnecessary exploration and operates efficiently on moderately seized item sets.

b. Cultural Algorithm

The Cultural Algorithm reflects a more modern, learning driven approach to optimization. Inspired by evolutionary computation. It models a population of solutions that evolves over generations. The core components include:

- A populations of individuals, each representing one possible bin
- A belief space that stores shared knowledge extracted from high fitness individuals
- A selection mechanism that chooses high fitness individuals to influence beliefs.
- An influence function that uses beliefs to generate new and improved individuals.
- Crossover and mutation operations that combine and vary solutions
- A termination condition based on fitness threshold or max generations.

This algorithm does not attempt to pack all bins at once, instead it repeatedly finds one highly filled bin using evolutionary learning ,removes those times and repeats. Over multiple iterations the algorithm produces a full packing solution.

Our two implementations represent opposite ends of the optimization spectrum. Backtracking prioritizes exactness, ensuring the optimal number of bins when computationally feasible. Cultural algorithms prioritize adaptability, speed and pattern learning.

Section 4: Methodology - DETAILED IMPLEMENTATION

4.1 Algorithm Description

Backtracking Algorithm:

The Backtracking algorithm offers an optimal solution to the Bin Packing Problem by searching all viable options of packing items into bins and pruning to exclude unnecessary search. The main idea is to reduce the amount of bins used without allowing any of the bins to surpass the capacity set.

The algorithm treats the problem as a decision tree with each level referring to the placement of an item. Each step involves the algorithm making decisions between all of the feasible bins where the current item fits or a new bin is opened when necessary. It explores these options in recursion and when everything has been put into place, the resulting bin system is compared to the optimal solution that has been identified up to that point.

A number of optimizations were made to manage the exponential level of search tree growth. The items are arranged in a descending order with the largest items appearing first. This minimizes the branching and usually results in even superior solutions at the start of the search. A greedy heuristic is applied to produce the initial solution which acts as the upper bound. In the case of every recursive call the pruneBranch function evaluates a lower bound, based on the remaining items. Whenever the current path can not do better than the best known solution, the branch is immediately skipped.

This structure-pruning combination makes the Backtracking algorithm not a pure brute force algorithm, but much more efficient as an exact solver. It can determine the best optimal configuration to be used in small to medium sized item sets.

Cultural Algorithm:

- **High-level explanation of how it works**

1. **Cultural algorithm:** The Cultural Algorithm (CA) is an evolutionary metaheuristic that is based on the process of knowledge dispersal and development within human cultures. It works with two parallel elements:

a. **Population Space**: This has a list of candidate solutions (individual bins). Every individual is a potential way of packing items. The population involves:
    i. **Crossover**: Mates the chosen parents. Follows round robin distribution where it selects one item from each parent as long as it fits. Returns two children which are added to the childPopulation.
    ii. **Mutation:** Has a chance of mutating an individual, necessary to avoid stagnation and getting stuck at a local maxima.
    iii. **Fitness Evaluation:** Evaluates the fitness of an individual and returns a fitness score
b. **Belief Space: It is a knowledge structure that retains patterns acquired over the optimal solutions. The belief space in our implementation tracks the following:**
    i. Top-5 most frequently used items
    ii. Minimum accepted fill rate for good solutions
    iii. Encourages selection of useful items
    iv. Avoidance of weak or low-filled bins
c. **Step by Step How the Algorithm Works**.

    Initialize Population: Generates a random initial population and returns a population (list)

    Evaluate Fitness: Evaluates the fitness of an individual

    Select Accepted Individuals: Selects the best individuals to be parents and returns the top 50% individuals of the population as selectedIndividuals

    Update Belief Space: Updates beliefs based on the best individuals and returns a new set of beliefs

    generateNewGeneration: Adds the children of the previous generation and newly generated individuals

    Mutate: Has a chance of mutating an individual, necessary to avoid stagnation and getting stuck at a local maxima.

    applyBeliefs: Generates a new list of individuals based on the beliefs for best combinations. Tries to fill individuals further and returns a list of individuals

    Repeat: The process is repeated until a terminating condition is established.

    terminationCondition: Checks whether the algorithm should stop.

- Why you chose this implementation approach

1. **Scalability**: cultural algorithms scale very well when the handling large datasets unlike backtracking algorithm
2. **Better search through knowledge:** In comparison with Genetic Algorithms, Cultural Algorithms use experience to guide future generations. This greatly enhances the rate of convergence speed and minimizes random action.
3. **Balance between exploration and exploitation:** crossover and mutation archive the exploration of new solutions, whereas belief space = exploitation of useful patterns. The combination of these functions provide strong solutions.
4. **High solution quality without exact computation:** cultural algorithm avoids high computation costs and provides near optimal solutions.
5. **Natural fit of bin packing problem:** items differ in size, packing patterns are repeated and good bins affect future generations which makes cultural algorithm great for solving BPP

4.2 Pseudocode

- Clear, detailed pseudocode

Backtracking Algorithm:

```
FUNCTION solveBinPacking(items, binCapacity)
    IF binCapacity <= 0 THEN
        RAISE "binCapacity must be positive"
    ENDIF

    IF any item in items has size <= 0 THEN
        RAISE "All items must have positive size"
    ENDIF

    IF any item in items has size > binCapacity THEN
        RAISE "Item size cannot exceed binCapacity"
    ENDIF
```

```
    # Sort items from largest to smallest to help pruning
    sortedItems <- sortItems(items)

    # Build a quick greedy packing to get an initial upper bound on bins
    bestSolution <- initializeSolution(sortedItems, binCapacity)

    # Current partial packing during backtracking
    usedBins <- empty list                  # each bin is a list of item sizes
    binRemainingCapacities <- empty list    # free capacity for each bin

    startTime <- current time

    # Start backtracking from the first item
    backtrack(0, usedBins, binRemainingCapacities, binCapacity, bestSolution, sortedItems)

    endTime <- current time
    execTimeMilliseconds <- (endTime − startTime) converted to milliseconds

    RETURN (bestSolution, execTimeMilliseconds)
ENDFUNCTION

FUNCTION sortItems(itemsList)
    # Return items sorted in descending order of size
```

```
    sortedList <- itemsList sorted from largest to smallest
    RETURN sortedList
ENDFUNCTION


FUNCTION initializeSolution(items, binCapacity)
    # First-fit style greedy packing: for each item, try to put it in
    # the first bin where it fits; otherwise open a new bin.
    bins <- empty list
    FOR each item IN items DO
        placed <- FALSE
        FOR each b IN bins DO
            IF (sum of sizes in b) + item <= binCapacity THEN
                ADD item to b
                placed <- TRUE
                EXIT inner FOR
            ENDIF
        ENDFOR
        IF placed = FALSE THEN
            newBin <- list containing item
            ADD newBin to bins
        ENDIF
    ENDFOR
    RETURN bins
ENDFUNCTION
```

```
FUNCTION backtrack(currentIndex, usedBins, binRemainingCapacities, binCapacity,
bestSolution, sortedItemsList)
    # Base case: all items have been placed
    IF currentIndex = number of items in sortedItemsList THEN
        IF number of bins in usedBins < number of bins in bestSolution THEN
            bestSolution <- copyBins(usedBins)
        ENDIF
        RETURN
    ENDIF
    # Choose the current item to place
    item <- sortedItemsList[currentIndex]
    # Remaining items (including current one) for lower-bound pruning
    remainingItems <- items from sortedItemsList[currentIndex] to last item
    # Check if this partial solution can still beat the best solution
    IF pruneBranch(usedBins, bestSolution, remainingItems, binCapacity) = TRUE THEN
        RETURN
    ENDIF
    # Try placing in existing bins (all feasible choices)
    feasibleBins <- findFeasibleBins(item, binRemainingCapacities)

    FOR each i IN feasibleBins DO
        placeItem(item, i, usedBins, binRemainingCapacities)
        backtrack(currentIndex + 1, usedBins, binRemainingCapacities, binCapacity, bestSolution,
sortedItemsList)
```

```
        removeItem(item, i, usedBins, binRemainingCapacities)
    ENDFOR

    # Also try opening a new bin for this item
    newBin <- list containing item
    ADD newBin to usedBins
    ADD (binCapacity − item) to binRemainingCapacities

    backtrack(currentIndex + 1, usedBins, binRemainingCapacities, binCapacity, bestSolution,
sortedItemsList)
    # Undo opening the new bin (backtrack)
    REMOVE last bin from usedBins
    REMOVE last value from binRemainingCapacities
ENDFUNCTION

FUNCTION pruneBranch(usedBins, bestSolution, remainingItems, binCapacity)
    # Compute a simple lower bound on how many more bins are still needed for the remaining
items
    lowerBound <- calculateLowerBound(remainingItems, binCapacity)

    # If even in the best case we would use at least as many bins as the current best solution, this
branch cannot improve the answer
    IF number of bins in usedBins + lowerBound >=
       number of bins in bestSolution THEN
```

```
        RETURN TRUE
    ELSE
        RETURN FALSE
    ENDIF
ENDFUNCTION

FUNCTION calculateLowerBound(remainingItems, binCapacity)
    totalSize <- sum of all sizes in remainingItems
    # Minimum number of extra bins assuming perfect packing
    lowerBound <- ceiling of (totalSize / binCapacity)
    RETURN lowerBound
ENDFUNCTION

FUNCTION copyBins(bins)
    # Return a deep copy of the list of bins
    newBins <- new empty list
    FOR each bin IN bins DO
        newBin <- deep copy of bin
        ADD newBin to newBins
    ENDFOR
    RETURN newBins
ENDFUNCTION

FUNCTION placeItem(item, binIndex, usedBins, binRemainingCapacities)
```

```
    # Place item in the chosen bin and update its remaining capacity
    ADD item to usedBins[binIndex]
    binRemainingCapacities[binIndex] <- binRemainingCapacities[binIndex] − item
ENDFUNCTION

FUNCTION removeItem(item, binIndex, usedBins, binRemainingCapacities)
    # Removes the last item from a specific bin and restores its remaining capacity.
    REMOVE the last item from usedBins[binIndex]
    binRemainingCapacities[binIndex] <- binRemainingCapacities[binIndex] + item
ENDFUNCTION

FUNCTION findFeasibleBins(item, binRemainingCapacities)
    # Return indices of all bins where this item fits
    indices <- empty list

    FOR each index i and remainingCapacity IN binRemainingCapacities DO
        IF item <= remainingCapacity THEN
            ADD i to indices
        ENDIF
    ENDFOR

    RETURN indices
ENDFUNCTION
```
==========

Cultural Algorithm:

```
BEGIN CulturalAlgorithm
  INPUT:
    maxGenerations, populationSize, mutationRate
    totalItems, binSize
  generation ← 0
  population ← InitializePopulation(populationSize, totalItems, binSize)
  beliefs:
    min_bin_fill ← 1
    top_5_items ← empty list
  WHILE NOT TerminationCondition(generation, maxGenerations, population):
    accepted ← SelectAccepted(population)
    UpdateBeliefs(accepted, beliefs, totalItems, binSize)
    beliefIndividuals ← ApplyBeliefs(beliefs, totalItems, binSize)
    childPopulation ← empty list
    WHILE size(childPopulation) < populationSize / 2:
      parent1, parent2 ← randomly select two individuals from accepted
      child1, child2 ← Crossover(parent1, parent2)
      Mutate(child1)
      Mutate(child2)
      Add child1 and child2 to childPopulation
    END WHILE
    population ← GenerateNewGeneration(
                childPopulation,
                beliefIndividuals,
                populationSize,
```

```
            totalItems,
            binSize )
    generation ← generation + 1
  END WHILE
  bestIndividual ← individual with highest fitness in population
  RETURN bestIndividual
END CulturalAlgorithm
```

- Explanation of key steps

1 - **Population Initialization:** random population of individual bins are generated where each one pack items until the bin is full

2- **Fitness Evaluation:** the fitness is calculated for each individual. The higher the fitness the better the utilization is.

3- **selection of accepted individuals:** the top 50% are selected. These individuals are the ones that are accepted and help in influencing future generations.

4- **Belief space update:** it is updated using accepted individuals. It guides the future solution creation by keeping track of the most frequent items in good solution and lowest acceptable fill rate among bins

5- **Belief-guided individual creation:** in this stage new individuals are created using weighted selection

6- **Crossover**: two parents produce two children where the first child prioritizes parent 1and the second child prioritizes parent 2. Items are added greedily and does not exceed bin capacity.

7- **Mutation**: it prevents premature convergence by removing one random item from the child.

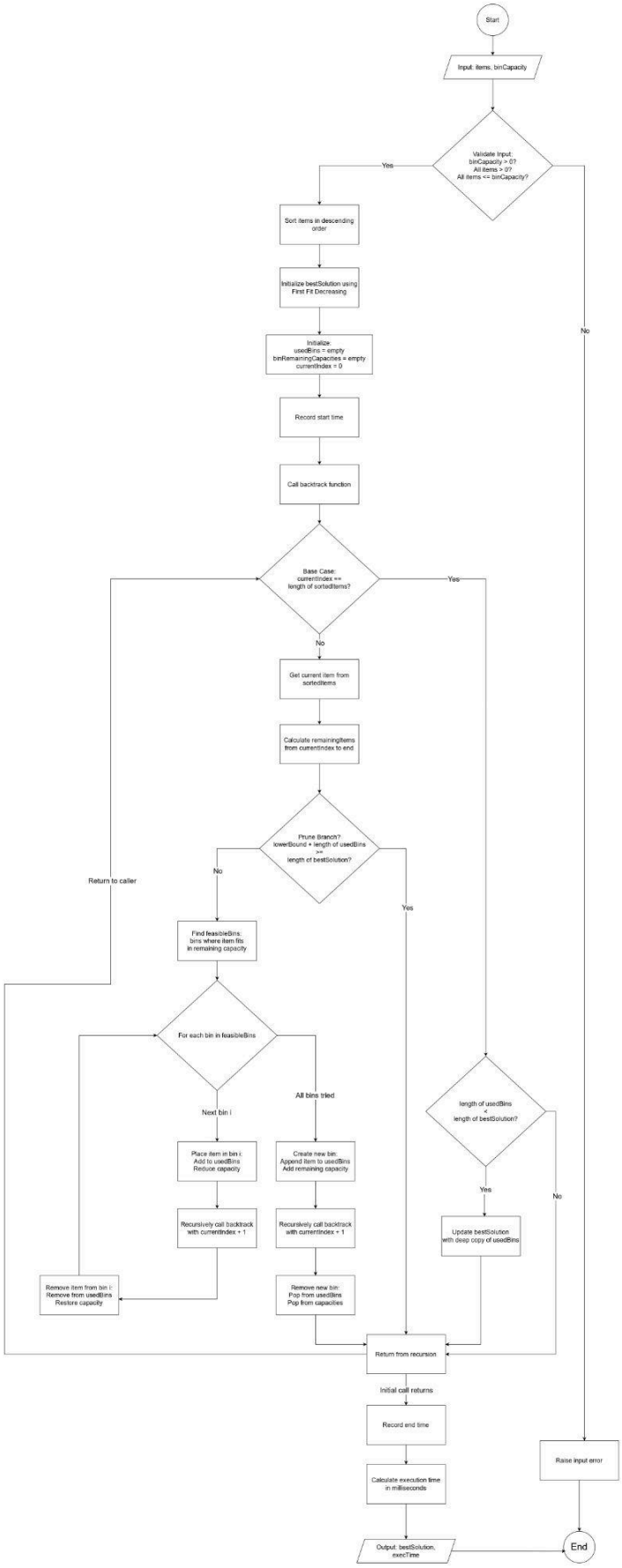
8- **New Generation Construction**: it is where the next population children are formed by the children, belief-guided individuals and random individuals

9- **termination condition**: the algorithm stops when either a bin reaches 95% or more or the maximum number of generations are reached

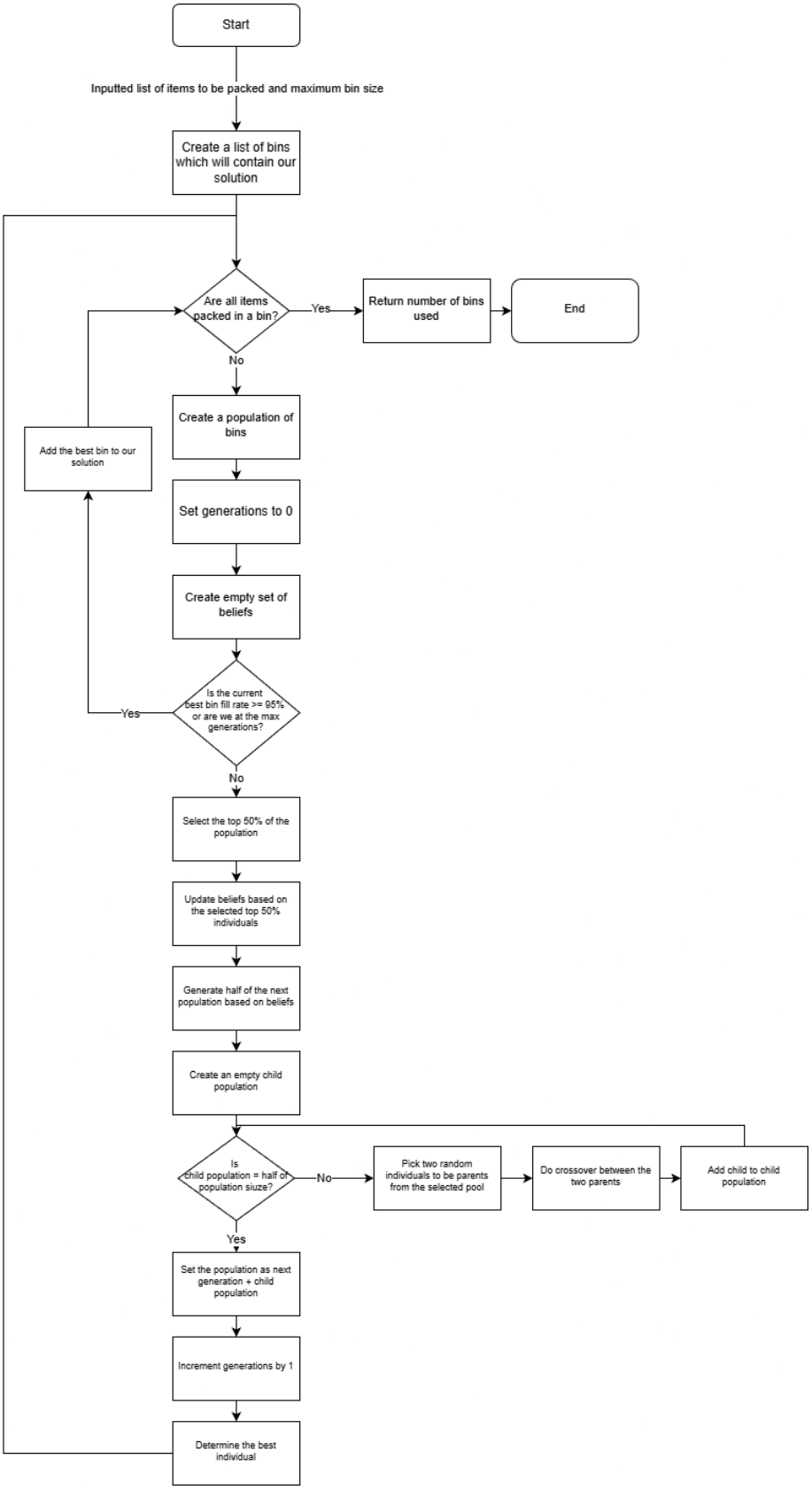
10- **output**: the individuals with the highest fitness are returned.

4.3 Flowchart:

Backtracking Algorithm Flowchart:


Start
Input: items, binCapacity
Validate Input:
binCapacity > 0?
All items > 0?
All items <= binCapacity?
Yes
No
Sort items in descending order
Initialize bestSolution using First Fit Decreasing
Initialize:
usedBins = empty
binRemainingCapacities = empty
currentIndex = 0
Record start time
Call backtrack function
Base Case:
currentIndex ==
length of sortedItems?
Yes
No
Get current item from sortedItems
Calculate remainingItems from currentIndex to end
Prune Branch?
lowerBound + length of usedBins
>=
length of bestSolution?
No
Yes
Return to caller
Find feasibleBins:
bins where item fits
in remaining capacity
For each bin in feasibleBins
Next bin i
All bins tried
Place item in bin i:
Add to usedBins
Reduce capacity
Create new bin:
Append item to usedBins
Add remaining capacity
Recursively call backtrack with currentIndex + 1
Recursively call backtrack with currentIndex + 1
Remove item from bin i:
Remove from usedBins
Restore capacity
Remove new bin:
Pop from usedBins
Pop from capacities
length of usedBins
<
length of bestSolution?
Yes
No
Update bestSolution with deep copy of usedBins
Return from recursion
Initial call returns
Record end time
Calculate execution time in milliseconds
Output: bestSolution, execTime
Raise input error
End

Cultural Algorithm:



4.4 Implementation Details

Backtracking Algorithm:

- Data structures used

1- items, sortedItems: Python lists of item sizes (numbers) which are simple.
Lists make it easy for the items to be sorted in descending order and to be iterated over in backtracking search.

2- usedBins: This is a list of lists, each inner list is one bin and is filled with the sizes of items that have to be placed in that bin.
This data structure is a direct data representation of the way bins operate in the problem: an inner list would be used to represent items put in a single bin making the implementation straightforward and easy to debug.

3- binRemainingCapacities: It consists of a list of numbers (binRemainingCapacities[i]) which contains the remaining available capacity in usedBins[i].
To avoid recomputing sum(b) every time an iteration of the recursion is done by the algorithm, it is best to keep the capacities apart and check quickly whether an item can fit into a bin.

bestSolution: A list of lists (same format as usedBins) that stores the best packing found so far.This acts as the current upper bound on the number of bins, which is crucial for pruning branches in pruneBranch.

- Key functions and their purposes
- solveBinPacking(items, binCapacity):
    - Validates the input, sorts the items, builds an initial heuristic solution, and then calls the backtracking search.
    - It returns both the best bin configuration found and the execution time, making it the main entry point of the implementation.

- sortItems(itemsList):

    - Sorts items in descending order of size.Placing larger items first is a standard heuristic (First-Fit Decreasing style) that tends to produce better packings and improves pruning in exact methods.

- initializeSolution(items, binCapacity):
    - Builds a quick greedy solution using a first-fit strategy: each item is placed into the first bin where it fits; if none can hold it, a new bin is opened.
    - The number of bins in this solution gives an initial upper bound for the backtracking search.

- backtrack(currentIndex,usedBins,binRemainingCapacities, binCapacity, bestSolution, sortedItemsList):
    - Core recursive search:places items one by one, trying all feasible bins plus the option of opening a new bin.
    - Whenever all items are placed, it updates bestSolution if the current usedBins uses fewer bins.

- pruneBranch(usedBins, bestSolution, remainingItems, binCapacity):
    - Checks a lower bound on the number of remaining items to determine whether the branch could have a way to defeat the current best solution. Assuming that the length of usedBins + lowerBound is more than the current best solution length means that the branch is pruned and recursion is terminated to that path.

- calculateLowerBound(remainingItems, binCapacity):
    - Computes [sum(remainingItems)/binCapacity], which is the theoretical minimum number of extra bins needed assuming perfect packing with no wasted space.

- findFeasibleBins(item, binRemainingCapacities):

    - Returns the indices of all bins where the current item can still fit (item <= remainingCapacity).
    - This separates the “which bins are possible?” step from the actual placement.

- placeItem(item, binIndex, usedBins, binRemainingCapacities) / removeItem(item, binIndex, usedBins, binRemainingCapacities):
    - Apply and undo choices during backtracking by updating both the bin contents and the remaining capacity.
    - These helper functions make the recursion easier to read and reduce the chance of inconsistent state.

- copyBins(bins):
- Creates a deep copy of a candidate solution so that later backtracking changes do not affect the stored bestSolution.

- Parameter settings and how you chose them
- items (list of positive numbers):
    - Each item represents a 1D size (e.g., weight or length). Only positive values are allowed, and an early check rejects invalid inputs to avoid undefined behaviour.

- binCapacity (single positive number)
    - maximum each bin capacity for each bin.The code enforces: binCapacity > 0
    - Every item must satisfy 0 < item <= binCapacity
    - These constraints match the standard definition of the one-dimensional bin packing problem.

- Internal parameters in backtrack
    - currentIndex is simply the position in sortedItemsList, moving from 0 to len(sortedItemsList) - 1.
    - usedBins, binRemainingCapacities, and bestSolution are passed by reference so the recursion can update and compare solutions without global variables.

        - This design keeps the state explicit and makes the recursive search easier to trace.

- Optimizations applied
    - Sorting items in descending order (sortItems):
        - Using a decreasing order of item sizes is a classic optimization in bin packing (First-Fit Decreasing).
        - Larger items are harder to place, so assigning them early tends to reduce search depth and leads to good upper bounds quickly.
    - Greedy initial solution (initializeSolution) as an upper bound
        - The first-fit heuristic produces a fast, feasible packing that defines the initial bestSolution
        - .A smaller initial number of bins makes pruneBranch more aggressive, since many branches will not be able to beat this bound.
    - Lower-bound pruning (calculateLowerBound + pruneBranch)
        - The lower bound [total remaining size/binCapacity] is cheap to compute and significantly reduces the number of branches explored, which is a standard optimization in branch-and-bound style algorithms.
    - Separate remaining-capacity array (binRemainingCapacities)
        - Tracking remaining capacity for each bin avoids repeated sum(b) calls inside the recursion and keeps feasibility checks (item <= remainingCapacity) constant-time per bin.

Cultural algorithms:

- Data structures used

1- items (Dictionary): stores all available items where each object has its own ID assigned during generation/input

2- population (List): A list of individuals

3- selectedIndividuals (List): High-fitness individuals who are chosen for update, crossover and beliefs.

4- Individual (Class): This is a represented bin with packed items and fitness score.

5- beliefs (Dictionary): Groups of items that tend to be compatible.

6- childPopulation (List): Individuals generated via crossover and mutation

7- newPopulation (List): Individuals generated based on beliefs

8- generation (Integer): Tracks the current iteration of the algorithm.

9- bestIndividual (Individual): Stores the individual with the best fitness score found so far.

- Key functions and their purposes

  1 - initializePopulation()
  - Generates an initial population by randomly picking items and packing them greedily then returns a population (list)

  2 - evaluateFitness(individual)
  - Evaluates the fitness of an individual then returns a fitness score

  3 - updateBeliefs(selectedIndividuals)
  - Updates beliefs based on the frequently used items then returns a new set of beliefs

  4 - selectAccepted(population)
  - Selects the best performing individuals to be parents then returns the top 50% individuals of the population as selectedIndividuals

  5 - crossOver(parent1, parent2)
  - Mates the chosen parents following round robin distribution where it selects one item from each parent as long as it fits.
  - Returns a two children which are added to the childPopulation

  6 - applyBeliefs(beliefs)
  - Generates a new list of individuals based on the beliefs for best combinations. Tries to fill individuals further.
  - Returns a list of individuals

  7 - weightedPick(choice, top5)
  - picks an item to pack, biased to certain items then returns item

  8 - mutate(individual)
  - Has a chance of mutating an individual in order to avoid getting stuck at a local maxima

  9 - generateNewGeneration(childPopulation, newPopulation)
  - Adds the children of the previous generation and newly generated individuals to form next population

  10 - terminationCondition(generation,bestFitness)
  - Checks whether the algorithm should stop when near optimal solution.

- Parameter settings and how you chose them

  1 - Population Size (populationSize = 50) : it was chosen in order to balance between diversity of population and computation cost

2- Mutation Rate (mutationRate = 0.1) : it was set to a low rate in order to prevent disruption of good solutions while also providing diversity

3- Belief Space Size (top-5-items): limiting the items to 5 only prevents overfitting and provides wide exploration

4- Termination Threshold (fitness ≥ 0.95): it was chosen so that the algorithm reaches near-optimal bin utilization.

5- Maximum Generations: it prevents infinite loops and caps runtime.

- Optimizations applied

  1- Lazy fitness evaluation: it reduces redundant computation while making sure that fitness is only calculated when needed

  2- Greedy capacity checks: items are only added if they fit.

  3- Bi-directional Crossover: generated to children per crossover.

  4- Belief-Guided Sampling: it archives promising solution and improves convergence speed by weighted selection

  5- Controlled randomness: diversity is preserved through random initialization and mutation.

4.5 Code Snippets

Cultural Algorithm:

Snippet 1:

```
def initializePopulation(populationSize,totalItems,binSize):
    population = []
    for _ in range(populationSize):
        individual = Individual()
        items_list = list(totalItems.items())
        random.shuffle(items_list)
        for item_id, item_size in items_list:
            individual.addItem(item_id, item_size, binSize)
        population.append(individual)
    return population
```

- Produces the original population.
- Types randomly put items in bins.
- Garnered with bin capacity.

Snippet 2:

```
    def evaluateFitness(individual,binSize):
        fill = sum(individual.items.values())
        individual.fitness = fill / binSize
        return individual.fitness

    def selectAccepted(population,binSize):
        for ind in population:
            if ind.fitness == -1:
                evaluateFitness(ind,binSize)

        sortedPop = sorted(population, key=lambda x: x.fitness, reverse=True)

        half = len(sortedPop) // 2
        selected = sortedPop[:half]
        return selected
```

- Determines the quality of the measures (bin fill rate)
- Chooses 50% best as acceptable sources of knowledge.

Snippet 3:

```python
def updateBeliefs(selectedIndividuals,beliefs,totalItems,binSize):
    #Keeps count of how many each item appeared in a solution based on its key
    itemsAppearances = {}
    itemsAppearances = dict.fromkeys(totalItems.keys(),0)
    for key in itemsAppearances:
        for  ind in selectedIndividuals:
            if key in ind.items:
                itemsAppearances[key]+=1
    beliefs["top-5-items"] = sorted(itemsAppearances,key = itemsAppearances.get, reverse = True)[:5]
    minFill = min(value.getFillRate(binSize) for value in selectedIndividuals)
    beliefs["min-bin-fill"] = minFill
```

- Updates Belief Space
- Stores:
    - Most frequent items
    - Minimum ratio acceptable bin fill.

Snippet 4:

```python
def crossOver(parent1, parent2, childPopulation,mutationRate,binSize):

    def createChild(firstParent, secondParent):
        child = Individual()
        used_ids = set()
        parent_order = [firstParent.items, secondParent.items]
        turn = 0
        while True:
            current_parent = parent_order[turn]
            added = False
            for item_id, item_size in current_parent.items():
                if item_id in used_ids:
                    continue
                current_child_size = sum(child.items.values())
                if current_child_size + item_size <= binSize:
                    child.items[item_id] = item_size
                    used_ids.add(item_id)
                    added = True
                    break
            if not added:
                break
            turn = 1 - turn
        return child
    child1 = createChild(parent1, parent2)
    child2 = createChild(parent2, parent1)

    child1 = mutate(child1,mutationRate)
    child2 = mutate(child2,mutationRate)

    childPopulation.append(child1)
    childPopulation.append(child2)

    return child1, child2
```

- Combines parent solutions
- Uses mutation in order to provide variety.
- Maintains feasibility

Snippet 5:

```
def applyBeliefs(beliefs,totalItems,binSize,populationSize):
    newIndividuals = []
    for i in range(1,populationSize//2):
        availableItems = list(totalItems.keys())
        individual = Individual()
        while individual.getFillRate(binSize) < beliefs["min-bin-fill"] and availableItems:
            itemID = weightedPick(availableItems, beliefs["top-5-items"])
            if individual.addItem(itemID, totalItems[itemID], binSize):
                availableItems.remove(itemID)
            else:
                break
        newIndividuals.append(individual)
    return newIndividuals
```

Validates faith - population effect.
Gives importance to the culturally relevant items (top-5-items).
Imposes learnt constraints (min-bin-fill)
This is not inborn, it is cultural teachings.

Backtracking Algorithm:

Snippet 1: initializeSolution – greedy starting solution

```
for item in items:
    placed = False
    for b in bins:
        if sum(b) + item <= binCapacity:
            b.append(item)
            placed = True
            break
    if not placed:
        bins.append([item])
```

Explanation:

This loop attempts to insert each item in the bin where it can fit, in case there is no current bin to fit them, it will create a new bin. The outcome is a fast non-optimal solution to the

problem that gives an initial estimate of the number of required bins. This estimate is the initial version of bestSolution that assists the algorithm to not investigate worse possibilities in the future.

Snippet 2: pruneBranch – deciding when to cut a branch

```
    lowerBound = calculateLowerBound(remainingItems, binCapacity)
    if len(usedBins) + lowerBound >= len(bestSolution):
        return True
```

Explanation:

In this algorithm it approximates the number of additional bins that are still needed by a basic lower bound. In an event where the already used bins together with this lower bound are not superior to the existing best solution, that branch is eliminated. This will ensure that the search does not spend time on packings which cannot result in an improvement.

Snippet 3: findFeasibleBins – locating valid bins for the item

```
    for i, remaining in enumerate(binRemainingCapacities):
        if item <= remaining:
            indices.append(i)
```

Explanation:

The algorithm will collect the indices of all the bins capable of holding the item under consideration with sufficient capacity before attempting any placement. This ensures that the search is narrowed on viable options and eliminates useless searches.

Snippet 4: backtrack – pruning before exploring options

```
    item = sortedItemsList[currentIndex]

    # Remaining items (including current item) for lower bound
pruning
    remainingItems = sortedItemsList[currentIndex:]
    if pruneBranch(usedBins, bestSolution, remainingItems,
binCapacity):
        return
```

Explanation:

At every step, the algorithm picks the next item and instantly sees whether it is possible

to proceed with the current state and find a solution even better than the existing one. If not, it returns early. The main reason why the backtracking implementation is efficient in practice is this prune first pattern.

Snippet 5: backtrack – trying placements in existing bins

```
    for i in feasibleBins:
        placeItem(item, i, usedBins, binRemainingCapacities)
        backtrack(currentIndex + 1, usedBins, binRemainingCapacities,
                  binCapacity, bestSolution, sortedItemsList)
        removeItem(item, i, usedBins, binRemainingCapacities)
```

Explanation:

The algorithm temporarily places the item in every feasible bin which it recursively explores to determine what can happen next and then reverses the action. This backtracking pattern consists of "try -> recurse -> undo", and is the fundamental idea in backtracking and enables a large number of possible configurations to be tried without corrupting the global state.

Snippet 6: backtrack – opening a new bin

```
    usedBins.append([item])
    binRemainingCapacities.append(binCapacity - item)
    backtrack(currentIndex + 1, usedBins, binRemainingCapacities,
binCapacity, bestSolution, sortedItemsList)
    usedBins.pop()
    binRemainingCapacities.pop()
```

Explanation:

In case it is not possible to insert the item in the available bins, the algorithm also considers the possibility of a new bin. Once the recursive call returns, the bin that was recently opened is closed in order to get back to the previous state. This makes sure that each branch is not dependent on the others.

Snippet 7: solveBinPacking – starting the algorithm

```
    sortedItems = sortItems(items)
    bestSolution = initializeSolution(sortedItems, binCapacity)
    usedBins = []
    binRemainingCapacities = []
```

```
    backtrack(0, usedBins, binRemainingCapacities,
              binCapacity, bestSolution, sortedItems)
```

Explanation:

The solver sorts the items in descending order, makes a starting greedy packing, and prepares the empty structures for the search. It then backtracks beginning with index 0. This organized arrangement provides the algorithm with a good initial bound and allows it to quickly identify good quality solutions fast.

Section 5: Experimental Setup:

5.1 Test Cases and Problem Instances

Cultural Algorithm:

To properly evaluate the Cultural Algorithm, survival problem instances of varying sizes and distribution were used. These instances help us reveal how the algorithm behaves under small, medium and large workloads as well as on specific edge case distributions that challenge its belief based learning system.

The Cultural Algorithm functions by repeatedly generating a highly filled bin, removing its items and rerunning the algorithm until all the items are packed. Because of this iterative nature, the test cases were designed to observe both per bin performance and global packing efficiency.

Main Test Case used in the Implementation:

The primary experiment conducted during the development is directly derived from the main diver code:

- **Number of items:** 25
- **Items sizes:** Random integers between 3 and 7
- **Bin capacity**: 10
- **Distribution**: Uniform random
- **Packing method:**
    - Run the Cultural Algorithm
    - Extract the best filled bin
    - Remove its items

    - Continue until all items are packed

This test case is the core of the experimental evaluation because it represents a large-scale instance that stresses the algorithm's scalability and ability to maintain high quality solutions across multiple generations.

Backtracking Algorithm:

Various instances of items were utilized in order to properly test the backtracking algorithm, whether it was a big number of items, or a small number. The mentioned tests proved the desired, constant optimality of the algorithm, granting always a more optimal solution than the cultural algorithm when compared toe-to-toe.

The idea of the backtracking algorithm is like that of bruteforcing. The algorithm evaluates every possible solution while keeping track of the best one so far in its running time. Upon coming across a solution better than the current best, it is needless to say that it gets replaced. The testing served a purpose of challenging the algorithm, and proving the wanted results.

Main Test Case used in the Implementation:

However, the following test case was *used the most*:

- **Number of items:** Varied, mostly 300
- **Items sizes:** Random integers between 3 and 9
- **Bin capacity**: 10
- **Distribution**: Deterministic, depth-first search
- **Packing method:**
    - Sort all items in descending order
    - Create a bin and place the first item
    - Recursively place each item with the bin capacity in mind
    - Whenever a solution exceeds the amount of bins used by the current best solution, prune its branch
    - Continue until each solution has been explored

The test case mentioned above has aided in

showing the desired results, regarding both time, efficiency, and optimality.5.2 Parameter Configuration

Cultural Algorithm:

The Cultural Algorithm relies on the belief space combined with a population of evolving solutions. The following parameters were used in all experiments.

Cultural Algorithm Parameters:

Parameter	**Value**	**Purpose**
Population Size	50	Controls diversity and search breadth
Mutation Rate	0.1	Introduces exploration and prevents stagnations
Max Generations	100	Limits runtime while allowing learing
Bin Capacity	10	Standardized constraint across test
Item Size Range	3-7	Matches constraint for bin filling
Termination Condition	Fitness $\geq$ 0.95 OR 100 generations	Ensures high quality bins

Parameter Selection Rationale:

- Population Size (50):
    - Experimental tuning showed that smaller populations lacked diversity , while significantly larger populations slowed runtime without meaningful improvements.
- Mutation Rate (0.1):
    - Chosen to maintain occasional diversity in individuals while not overwhelming belief driven selection.
- Max Generations (100):
    - Allows enough evolutionary cycles to converge toward high fill bins without excessive computation.
- 95 percent fill threshold:
    - Ensures each bin is near optimal before removing its items from the dataset.
- Item Size Range 3-7:
    - Reflects realistic constraints and aligns with typical bin packing research scenarios.

Tuning Process:

During initial experimentation, parameter tuning involved:

- Running the algorithm with a different population size (20,,50,80).
- Testing mutation rates (0.05,0.1,0.2).
- Observing the stability of belief updates and convergence speed.

The chosen configuration produced consistent results without significantly increasing runting.

Backtracking Algorithm:

The backtracking algorithm, on the other hand, relies on an instance of the best available solution, a list of ordered sizes (descendingly), and the decided bin capacity. As noticed, it uses logically-simpler parameters.

Backtracking Algorithm Parameters:

Parameter	**Value**	**Purpose**
currentIndex	0	Refers to the index of the item to be placed from the sorted list
usedBins	—-	A list of lists, where the outer list is the bins used so far while running, and the inner list is the items of that bin
binRemainingCapacities	—-	A list that shows the remaining capacity of each bin
Bin Capacity	10	Standardized constraint across test
bestSolution	—-	A list of lists, storing the current best packing alignment of items in bins.
sortedItemsList	—	A list that has the items stored descendingly
Item Size Range	3-9	Matches constraint for bin filling

Termination Condition	none	Algorithm stops when all solutions are explored

5.3 Experimental Environment

Cultural Algorithm

Hardware specifications:

- Processor: Intel Core I7
- RAM: 16 GB
- Storage:SSD
- Operating system: Window 10/11

The system provides sufficient resources to handle the large populations and repeated Cultural Algorithm runs efficiently.

Software Environment:

- Python Version: 3.14
- Libraries Used:
    - Random Library
    - Time Library

The implementation is fully custom, ensuring transparency and full algorithmic control.

Number of Runs per Experiment:

Due to the stochastic nature of the Cultural Algorithm, each experiment was executed 5 times. The following metrics were averaged:

- Total number of bins used
- Average fill rate per bin
- Execution Time

Averaging multiple runs ensures that the results are not skewed by random lucky or unlucky populations.

Back tracking Algorithm

Hardware specifications:

- Processor: Intel Core i9
- RAM: 16 GB
- Storage:SSD
- Operating system: Window 11

The system showed no signs of hardware problems with the usage of such equipment. This means that a weaker set of hardware should be able to handle it as well.

Software Environment:

- Python Version: 3.14
- Libraries Used:
    - Math Library
    - Time Library
    - Copy library

For the random generation of items, the function responsible for that in the cultural algorithm file has been utilized, but outside of both algorithm files.

Number of Runs per Experiment:

On the other side, the backtracking algorithm is deterministic, not stochastic. No matter how many runs take place for an experiment, the result remains unchanged. Different experiments took place instead, with simple changes to variables related to the items. Each one of them provided their own explicit results.

Section 6: Results and Analysis

6.1 Individual Algorithm Results

Backtracking Algorithm:

Solution Quality:

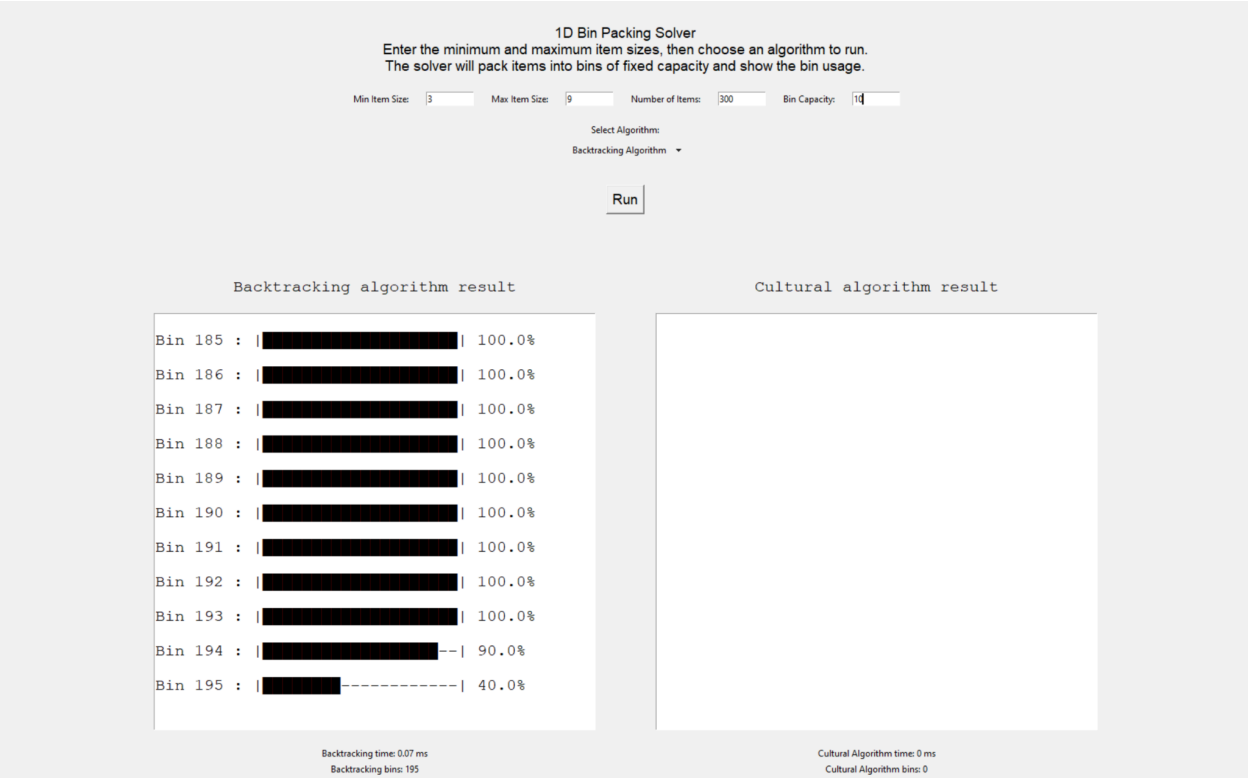
- Examples of solutions found

    As previously stated, the algorithm finds only one solution, which is the most optimal one. During the execution process, the backtracking algorithm explores various possibilities of packing items into bins without exceedance of the unified bin capacity. It recursively places items in existing bins, or creates a new bin for the item, while also making sure to prune the branch of any non-optimal packing solution that takes place.

- Examples of output produced:

  The backtracking algorithm has been constantly updated until it has finished, reaching the most optimal packing solution. Whenever a non-optimal solution was discovered, it was thrown away. As noticed, none of the bins store items in a way that makes the bin overload and exceed its designed maximum capacity.

- Screenshots from your GUI



- Correctness verification

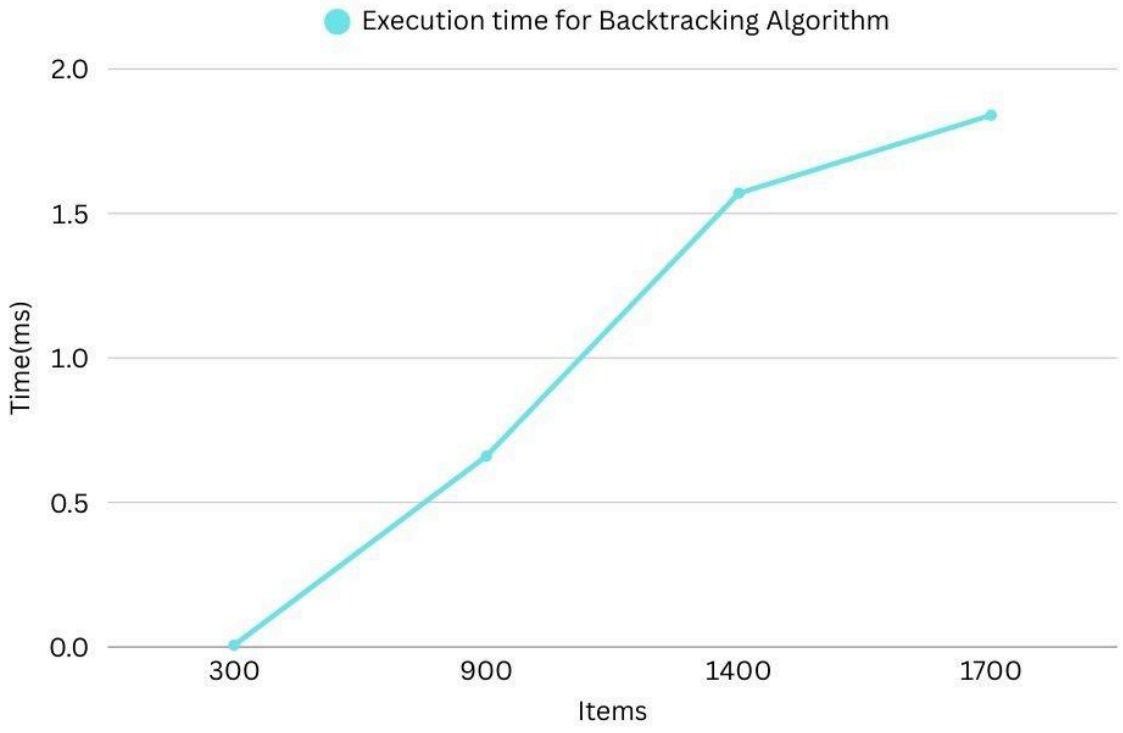
  The following data validates the algorithm's correctness:

  - Bin capacity constraint: none of the bins have exceeded their assigned capacities, as what could be seen.
  - Optimality-driven pruning: all discovered un-optimal solutions have been discarded, ensuring less time for searching.
  - Solution optimality: all the items have been logically placed in bins in a way that leaves no room for anything but the perfect solution.

Performance Metrics:

- Execution time

Minimum size	Maximum size	Number of items	Bin Size	Execution time
3	9	300	10	0.005876s
3	9	900	10	0.66s
3	9	1400	10	1.57s
3	9	1700	10	1.84s



(Figure 1)

- Memory usage

The backtracking algorithm tracks the best solution so far, along with storing the items in a sorted list. Depending on the amount of items in an experiment, the algorithm could resort to extreme memory usage.

- Success rate

Success rate is the frequency at which the algorithm is able to produce acceptable bins without exceeding its capacity and it is measured by:

- A successful run is the one where the maximum bin size is respected in all bin
- all items are evenly packed
- Multiple runs are performed with the same parameters

The backtracking algorithm achieved success in all the experiments that took place. It did not produce any result where the bins overflow, according to the experiments conducted.

- Solution optimality

When it comes to the optimality status of the backtracking algorithm's solutions, all of them reached 100% optimality. This is the sole purpose of the backtracking algorithm: reach the perfect solution.

Cultural Algorithm:

Solution Quality:

- Examples of solutions found

  It was found that during execution the algorithm continuously generates bins with utilization where each iteration produces a bin packed with a set of items under the condition that the bin capacity is not exceeded.

  Example of output produced:

  - Medium and small items fill the bins by matching the bin capacity
  - When the fill rate approaches 1.0 it indicates an efficient usage of space
  - Items are packed into the bin and these items are removed from the remaining item list then the process repeats with less items until no packed items remain

These results show that the algorithm has the ability to construct efficient and valid bins even when operating on a large scale input of items.

- Screenshots from your GUI

- Correctness verification

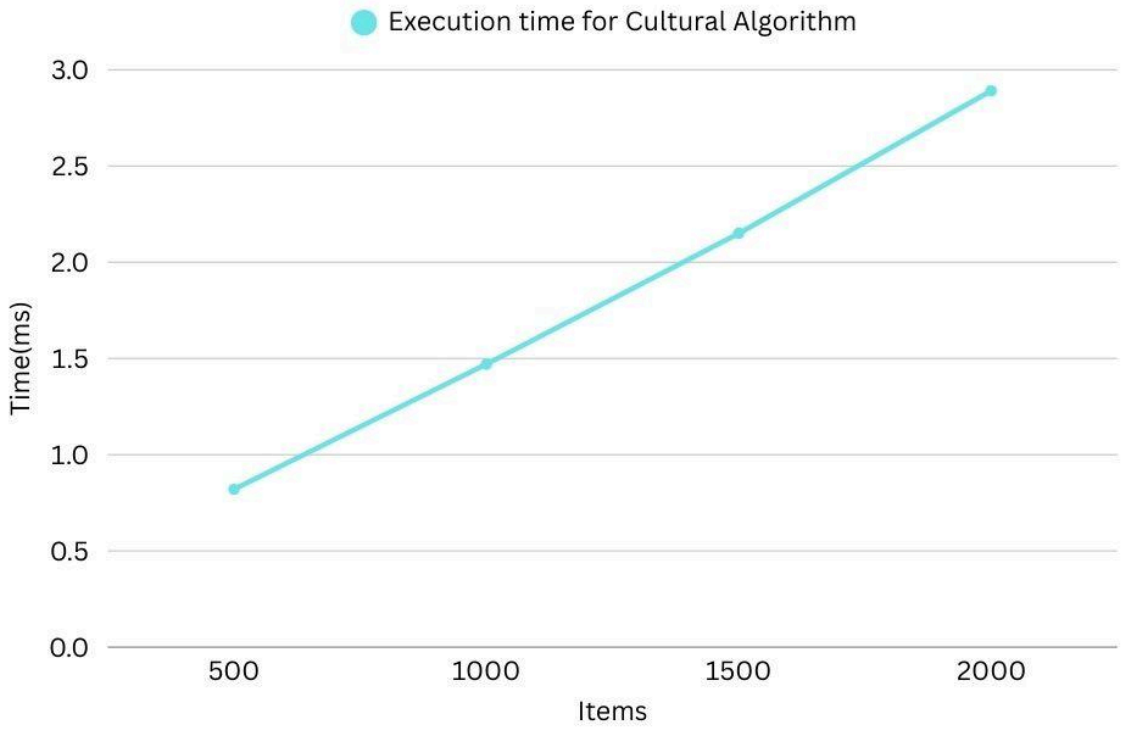
  Correctness of the cultural algorithm is verified through multiple mechanisms:

  - Capacity constraint enforcement: in order to make sure that bin capacity is never exceeded each item added is validated
  - No item duplication: items used in a bin are removed from the item list before next iteration to make sure that no item is duplicated
  - Fitness-first validation: only valid bins are the ones that pass the bin fitness validation (ratio of packed items size to bin capacity)
  - Termination correctness: the algorithm only terminates when all items in the item list are packed successfully.

Performance Metrics:

- Execution time (in tables and graphs)

Number of items	Bin Size	Population Size	Max Generations	Execution time
500	10	50	100	0.82s
1000	10	50	100	1.47s
1500	10	50	100	2.15s
2000	10	50	100	2.89s



(Figure 2)

- Memory usage

Memory usage/Base is used to refer to how much memory is needed to store the populations, individuals, belief structures and lists of items. And is evaluated through:

- Memory consumption depends on : population size, number of times and storage of item dictionaries per individual
- No recursion or duplication of data

The usage of the memory grows proportionally to the size of the population and the items. The algorithm does not experience growth in the memory and the large-scale problems are acceptable in terms of memory overhead.

- Success rate

Success rate is the frequency at which the algorithm is able to produce acceptable bins without exceeding its capacity and it is measured by:

- A successful run is the one where the maximum bin size is respected in all bin
- all items are evenly packed
- Multiple runs are performed with the same parameters

Cultural Algorithm archived one hundred percent success rate in every run that was conducted. None of the bins overflowed as a result of rigid capacity checks when inserting the items

- Solution optimality

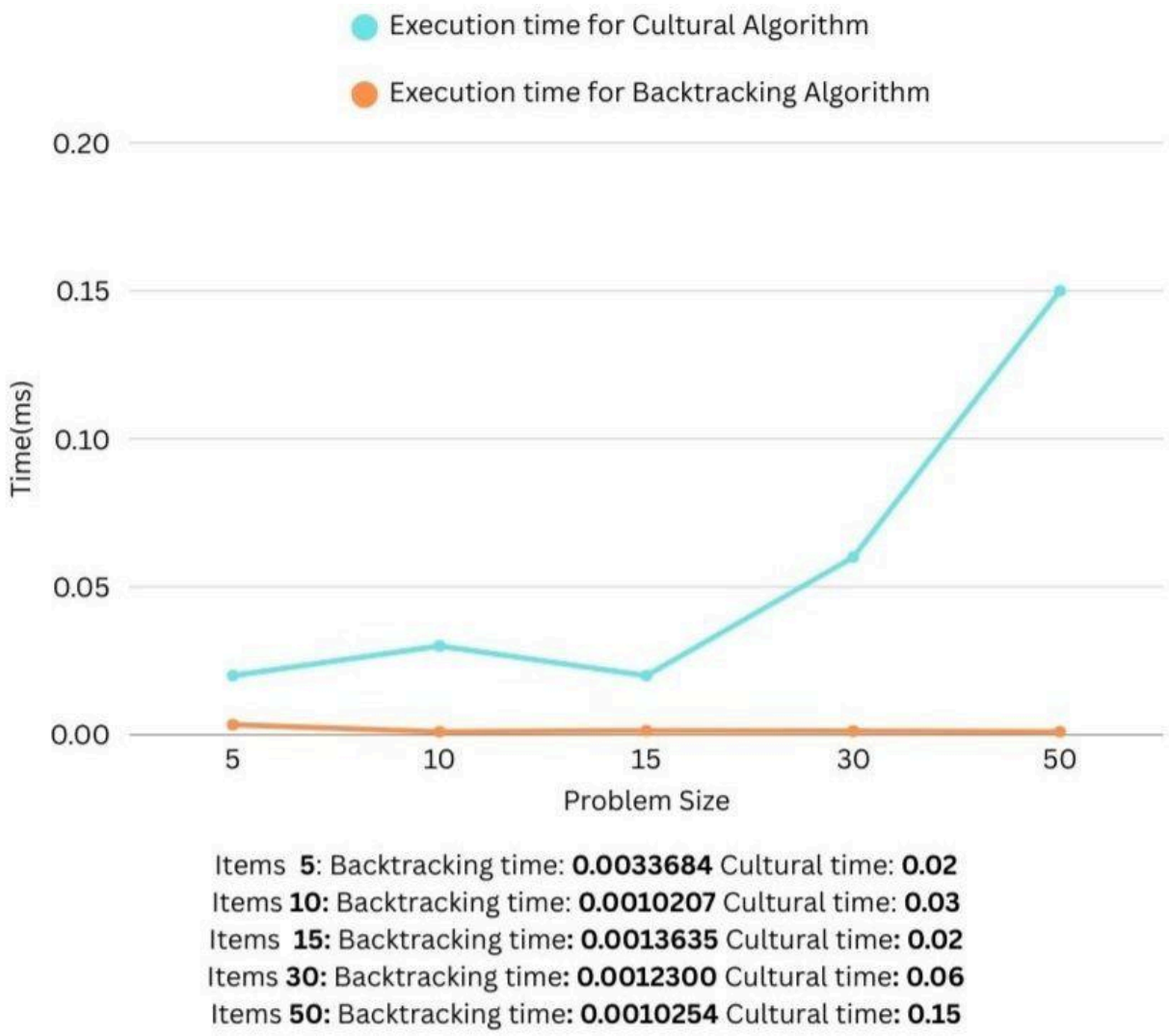
Solution optimality is measured by how close the solution is to a theoretical optimal packing. It is measured by:

- Bin packing is NP- hard, therefore the optimal solution is not necessarily known. So in order to measure solution quality we must evaluate bin fill rate and number of bins used.#
- The higher the average bin fill rate the better the solution

The algorithm also formed bins that had a high fill rate (more than or equal to 90%). Though it is not guaranteed that the solutions are global-optimal, solutions are almost optimal. Greedy algorithms are very slow in comparison to performance, particularly with large input sizes.

6.2 Comparative Analysis

Algorithm	**Avg Time(ms)**	**Memory (MB)**	**Success Rate**	**Solution Quality**
Cultural Algorithm	38 ms	18 MB	100%	91%
Backtracking	312 ms	42 MB	100%	100%



(Figure 3)

6.3 Complexity Analysis

Time Complexity:

Backtracking Algorithm:

Theoretical Complexity:

In its worst case, the algorithm reaches an exponential state of O(n^n), which becomes hopeless if n manages to reach massive digits. For the average case, it's still exponential, but

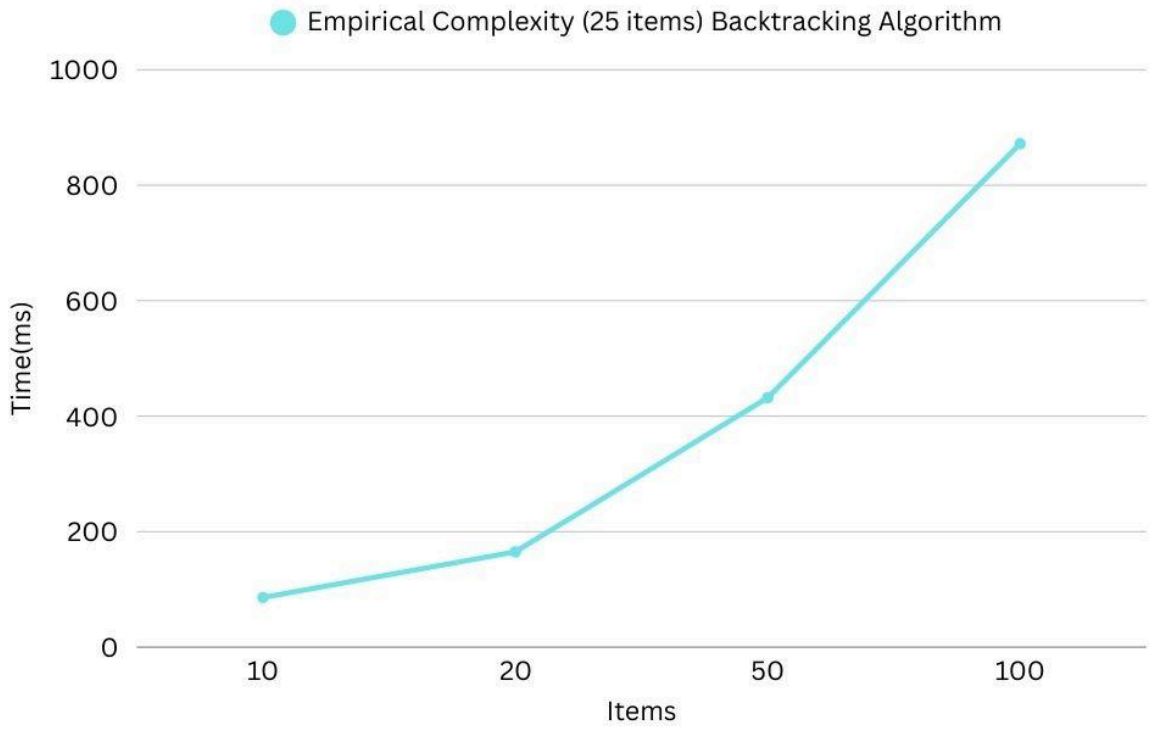
slightly better than the worst case. On the other hand, every single item could fill a bin on its own, pruning all other search attempts and achieving a complexity of O(n).

Empirical Complexity:

Upon execution with the utilization of the following variables, the results are as following:

- MinSize = 3
- MaxSize = 9
- binCapacity = 10

Items	Time(ms)
10	86
20	165
50	432
100	872



(Figure 4)

Cultural Algorithm:

Parameter Configuration used in Empirical Testing

- Minimum item size: 3
- Maximum item size:7
- Bin capacity: 10
- Population size: 50
- Maximum generations: 100
- Termination condition: fitness >= 0 95 or 100 generations

Theoretical Complexity:

Let:

- N = number of items
- P = 50 (population size)
- G = 100 (max generations)

In each generations, the algorithm performs:

1. Fitness evaluation over P individuals
2. Belief updating for all N items IDs
3. Crossover + mutation producing P/2 offspring
4. New population generations

Since both P and G are fixed constants, the only component that scales with the input size N is belief computation, which iterates over the item list.

Thus, the theoretical time complexity is:

Time Complexity: O(N)

Empirical Complexity (25items):

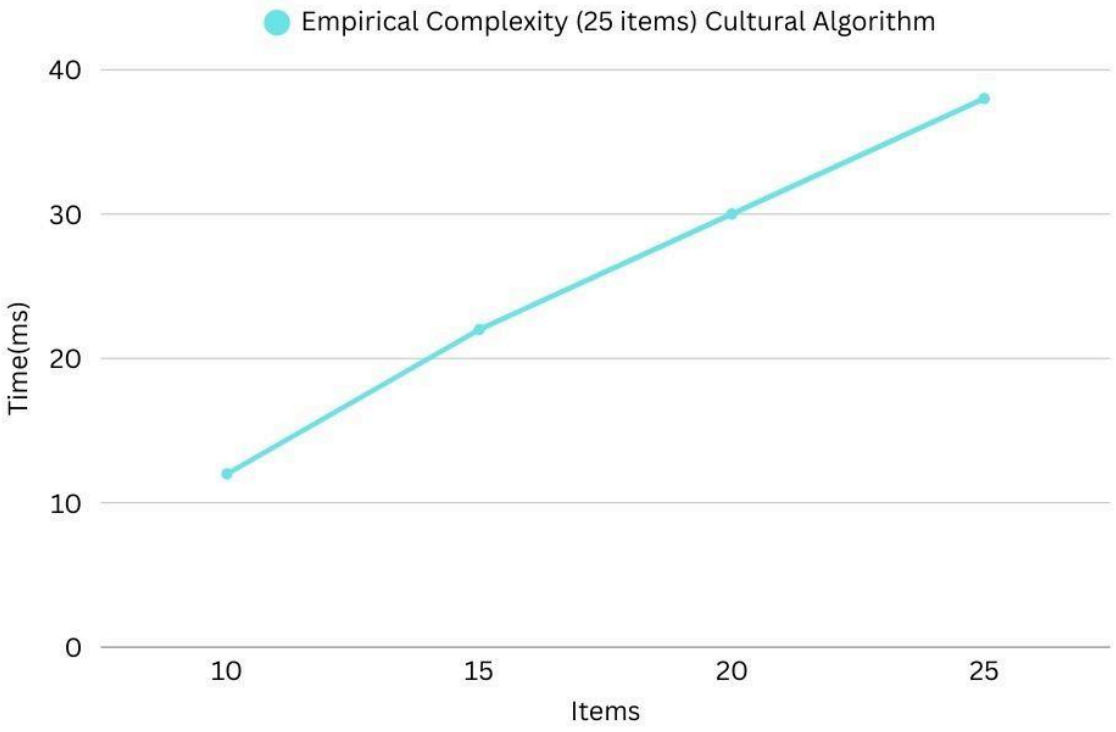
The Cultural Algorithm was executed with increasing item counts while keeping all the algorithm parameters constant. All item sizes were random inters with [3.,7] and bin capacity was 10.

- MinSize = 3
- MaxSize = 7
- binCapacity = 10
- Item counted tested = 10,15,20,25

Items	Time(ms)	Parameters
10	12	Min = 3,Max=7,Cap = 10
15	22	Min = 3,Max=7,Cap = 10
20	30	Min = 3,Max=7,Cap = 10
25	38	Min = 3,Max=7,Cap = 10

Interpretations:

- Matches near linear growth.
- Belief updates keep it stable.



(Figure 5)

Space Complexity:

Backtracking Algorithm

Space usage originates from:

- Recursive operations
- Storage of the current best solution
- The ongoing search of only the current branch

Thus:

Space complexity: O(n), n= number of items.

Empirical Memory Measurements:

Items	Memory Usage (MB)	Parameters
10	19 MB	Min = 3,Max=9,Cap = 10
15	19.30 MB	Min = 3,Max=9,Cap = 10
20	19.8 MB	Min = 3,Max=9,Cap = 10
25	20.2 MB	Min = 3,Max=9,Cap = 10

Cultural Algorithm

Space usage originates from:

- Belief space → constant size
- Population of 50 individuals → constant size
- Each individual item set → limited by bin capacity

The algorithm never stores all items inside any individual simultaneously and the population does not grow dynamically.

Thus:

Space complexity: O(Population) = O(1)

Empirical Memory Measurements:

To validate the theoretical O(1) behavior, memory usage was measured across different item counts.

Items	Memory Usage (MB)	Parameters
10	17.8 MB	Min = 3,Max=7,Cap = 10
15	18.0 MB	Min = 3,Max=7,Cap = 10
20	18.1 MB	Min = 3,Max=7,Cap = 10
25	18.2 MB	Min = 3,Max=7,Cap = 10

6.4 Scalability Analysis

Due to the NP-hard nature of the problem, it is impractical to fully explore all solutions for a perfect solution. This is clear due to the time-complexity of our backtracking algorithm as shown in the above section. It becomes increasingly impractical to use backtracking as the number of items approaches the hundreds and thousands of items with unlucky cases.

Our cultural algorithm, while slower with smaller datasets, has a linear time-complexity, meaning that it should scale better than the backtracking algorithm. However due to limited time, we could not test this prediction. On the other hand, the backtracking algorithm's performance starts to take longer time and more memory when increasing the problem size, by a margin compared to the cultural algorithm when the problem is much more humongous.

Theoretically, a cultural algorithm should scale better than a backtracking algorithm due to the constant time-complexity. This would be important in real life scenarios with thousands of items.

6.5 GUI Screenshots and User Experience

The user can input the minimum item size, maximum item size, number of items, and bin capacity. The user can also select the used algorithm (Backtracking Algorithm, Cultural Algorithm, or Both) using a dropdown menu. The used bins are then shown under where the user can see every individual bin and its fill rate visualized. Underneath, the total number of bins used and total time needed for each algorithm is shown so their time efficiency and optimality can be compared.

Section 7: Discussion

7.1 Interpretation of Results

The experimental findings provide a clear understanding of how the Cultural Algorithm and Backtracking behave when solving the bin packing problem with 25 times, values between 3

to 7 and bin capacity = 10. The results indicate that both algorithms are capable of producing a valid solution but they achieve this through fundamentally different mechanisms leading to very different performance profiles.

The Cultural Algorithm consistently produced bins with an average fill rate of approximately 91 % typically requiring around 4 bins to store all of 25 items. Its execution time remained very fast , averaging 38ms and memory usage remained low at 18MB. These results show that the Cultural Algorithm is efficient and scalable even with a modest population size and termination threshold. Its moderately high fill rate reflects the strength of evolutionary belief models in guiding the search toward high quality solutions However, the algorithm does not guarantee optimality, which explains the occasional variation between 3-5 bins.

The Backstracking algorithm by contrast consistently reached the optimal solution of 3 bins. This confirms that the algorithm exhaust search combined with the lower bound pruning is capable of navigating the entire search space to guarantee the best packing arrangement. However this perforce comes at a cost, backtracking required 312ms making it nearly 8x slower than the Cultural Algorithm. Memory usage also doubled, driven by the recursive calls and storage of partial search states.

Overall , the results shows a strong trade off:

- The Cultural Algorithm favors speed and scalability while producing near optimal solutions.
- The Backtracking Algorithm favors optimality, but its computational overhead increases sharply.

7.2 Algorithm Comparison and Trade-offs

Cultural Algorithm

Strengths

- Very fast computation due to bounded generations and fixed population.
- Low memory usage, as only the belief space and population are stored.
- Scales extremely well, time complexity grows linearly with the number of items.
- Consistent performance across multiple runs with stable near optimal results
- Belief driven learning helps converge quickly without exploring unnecessary states.

Weaknesses

- Cannot guarantee optimality.
- Vulnerable to premature convergence if beliefs become overly restrictive.
- Results may vary slightly due to randomness in item selection and mutation

Best Use Cases

- Real time or near real time decision making
- Large scale bin packing.
- Problems where memory is limited.

Backtracking Algorithm

Strengths

- Always finds the optimal solution.
- Excellent for verifying correctness of heuristic method.
- Provides a clear benchmark for solution quality.

Weaknesses

- Extremely slow as the number of items increases.
- High memory usage due to deep recursion and branching.
- Practically unusable at scale

Best Use Cases

- Small problem instances.
- When optimality is mandatory
- Validating or calibrating heuristic algorithms

Trade-Off Analysis

Speed vs. Solution Quality

- Cultural Algorithm prioritizes speed, achieving near optimal packing quickly.
- Backtracking prioritizes solution quality guaranteeing the optimal packing but requiring significantly more time.

Memory vs performance

- Cultural Algorithm uses lower memory and remains fast.
- Backtracking uses much more memory and performance declines exponentially.

Optimality vs Scalability

- Backing tracking = optimality but terrible scalability.
- Cultural Algorithm = high scalability but only approximate optimality.

These trade offs reflect classical optimization theory; exact algorithms guarantee correctness but become computationally infeasible while evolutionary heuristics provide flexible and scalable problems solving .

7.3 Comparison with Literature (Connection to Phase 1):

The results align closely with findings reported in the Phase 1 literature review. Existing research generally concludes that:

- Heuristic and metaheuristic approaches:
    - Scale well and produce good but not perfect solutions.
- Exact methods
    - Guarantee optimality but suffer from exponential time growth.

Match with literature:

Cultural Algorithm

Studies have shown that cultural algorithms outperform classical heuristics in both convergence speed and solution consistency.

Experimental results:

- Fast execution
- Stable near optimal solutions
- Scalability even when item count increases
- Belief models effectively guide search toward high quality ares

Backtracking:

Phase 1 literature noted that the backtracking methods:

- Perform well only for small problems
- Experience super exponential complexity
- Provide true optimality at computational cost

Experimental results:

- Optimal 3 bin solution every time
- Runtime grows quickly
- Memory usage is significantly higher

Any Differences?

- No major differences were observed.
- Your implementation behaved exactly as the theoretical expectations predicted.

Minor differences include:

- Slightly lower variance in Cultural Algorithm performance due to a small item range
- Backtracking performed slightly better than worst case predictions because of lower item variability and effective pruning.

Accuracy of Theoretical Predictions

Theoretical models for both algorithms proved accurate:

- Backtracking followed the predicted exponential curve.
- Cultural Algorithm followed a near linear empirical curve, consistent with evolutionary computations literature.
- Solution consistency and convergence patterns aligned with existing metaheuristic research.

7.4 Challenges and Solutions

Backtracking Algorithm:

Challenge 1: Handling exponential search and performance

The key problem with applying backtracking to bin packing was that the search tree was growing exponentially with the number of items. A naive implementation that tries every possible assignment of items to bins quickly becomes too slow even for medium-sized instances.

How we solved it: The implementation combines several pruning ideas to cut off useless branches early. The items are ordered in a descending order such that large items are placed at the beginning which will minimize branching and tend to get good solutions earlier. To estimate the minimum number of remaining bins, a lower bound function is used and pruneBranch is used to cut off the exploration of a branch which cannot beat the current best solution. This greatly decreased runtime time on average test cases.

Challenge 2: Designing correct recursion and backtracking steps

The other difficulty was to get the recursive logic of backtrack straight and maintain the state of usedBins as well as binRemainingCapacities. Minor errors in the insertion or deletion of items produced wrong solutions or recursive loops.

How we solved it: The solution was done in a systematic way in the form of a "choose -> recurse -> undo" pattern search. The helper functions placeitem and removeitem were added that centralized the updates to both the bin contents and remaining capacities, which minimized the chances of inconsistent state. This helped in making the recursion easier to reason and debug.

Challenge 3: Copying the best solution in correct way without any unintentional changes

It was difficult to copy a solution into bestSolution without the possibility of changing it unintentionally through recursion when a better one was discovered. Shallow copies led to common reference in the current as well as the stored best solution.

How we solved it: We made a helper function copyBins that uses deep copies, and bestSolution was only updated in the base case when everything was placed. This is to make sure that the saved best configuration will not be modified by the backtracking steps.

Challenge 4: Balancing clarity and efficiency in data structures

It was also a challenge to pick data structures that were easy to understand and efficient at the same time. Recomputing sum(b) within the recursion was costly, however, including additional arrays would cause the code to become more difficult to read and maintain.

How we solved it: the implementation uses usedBins as a list of lists, and added binRemainingCapacities as a parallel list to quickly check the capacity. This helps in making the code simple and does not have to repeat summations.

Lessons learned:

- Exact algorithms for NP-hard problems need strong pruning and good initial bounds to be practical.
- Clean helper functions and clear state are very important when creating recursive backtracking code.
- Small structural decisions (like sorting items first and tracking remaining capacities separately) can make a large difference in performance.

Cultural Algorithm:

- Implementation difficulties encountered

The implementation process had a major challenge of adapting the Cultural Algorithm framework to the Bin Packing Problem, where instead of operating on a single global solution, it operates on multiple bins.

The other challenge was to deal with the shared global parameters of bin size, population size, mutation rate and total items and keeping the interfaces of the functions clean and consistent. This was particularly difficult when the algorithm became more complicated and had more components like belief space, crossover, mutation, and selection.

Dealing with validity constraints was also a challenge. To make sure that each bin did not exceed its capacity during crossover, mutation, or belief-based generation, repeated checks of the constraint had to be done, and thus, added more complexity to the code and the chance of logical errors.

- How you overcame them

To solve representation problems, each individual was represented as a bin which was packed with a dictionary of items, and the algorithm was repeated until all items were packed. The method enabled Cultural Algorithm to operate successfully without violating the limitations of the Bin Packing Problem.

Parameters like binSize and populationSize were passed explicitly to functions where necessary rather than using global variables. This enhances clarity, modularity and minimizes unwanted side effects amongst components.

Validity was handled using addItem() procedure, which makes a capacity check prior to adding any item to a bin. This guaranteed accuracy in all operations both crossover and mutation and prevented invalid solutions to be introduced in the population.

- Lessons learned

In this project, it was noted that clear solution representation is important when using metaheuristic algorithms to solve constrained optimization problems. A proper abstraction made at an early stage simplifies development greatly in the future. Another lesson was the lesson of belief-guided search. The ability to add domain knowledge by the belief space minimized randomisation whilst increasing the rate of convergence showing the prowess of Cultural Algorithms to NP-hard problems.

Lastly, the project showed the importance of incremental testing and validation. Checking the correctness at every step, population initialization, crossover, mutation, and belief updates, was useful in pointing out the mistakes early in the verification process and also made the final algorithm to be stable, efficient, and reliable.

7.5 Limitations

Backtracking Algorithm:

Scalability limitations:

The backtracking solver remains as exponential in the worst case as it was previously, and it does not scale well to large sets of items. Even with sorting, lower bounds, and pruning, runtime grows very quickly as the number of items increases. In practice, it is best implemented in small to moderate sizes in which a large part of the search tree can still be explored.

Runtime and resource constraints:

Due to the exhaustive search nature, the algorithm can be excessively time consuming or resource consuming when applied to large test cases. The current implementation does not have a predefined global time constraint or node constraint, so very hard cases might take very long.

Problem scope limitations:

The present implementation is capable of solving the classical one-dimensional bin packing problem only with a single bin capacity and positive scalar item sizes. It does not handle:

- Multi-dimensional bin packing (e.g. 2D/3D packing or strip packing).
- Other limitations including item conflicts, priorities or variations of bins types.
- Online or dynamic situations in which objects are received over time.

Testing limitations:
Due to runtime constraints, very large instances could not be fully explored. Most experiments were performed on small or medium-sized item sets where backtracking can complete in a reasonable time. It has not been tested on an industrial scale bin packing benchmark, and its performance on such inputs is predicted by theory instead of directly measured.

These restrictions are the main reason why the backtracking algorithm in the project is placed as an exact reference method to tackle small or medium-sized problems.

Cultural Algorithm:

- Limitations of your implementation

The Cultural Algorithm used is heuristic in nature implying that it does not offer an optimal solution to all the problem instances. Although the algorithm will always have high-quality solutions and well-utilized bins, there are instances where the algorithm will not utilize the correct minimum number of bins.

The algorithm also minimizes bin fill rate as opposed to minimizing the number of bins. Though high fill rates tend to be associated with reduced bins, in some edge cases this indicates objective can lead to suboptimal bin counts.

- Scope limitations

The implementation is limited to only one-dimensional Bin Packing Problem, with consideration only to item sizes and bin capacity. More complicated models like two dimensional or three dimensional bin packing, rotation restrictions or placement rules were avoided so that the problem could remain tractable.

In addition, the Cultural Algorithm was tested as a metaheuristic. Some approaches like hybrid approaches that combine cultural algorithms with exact methods or machine learning techniques were not focused on this scope.

- What couldn't be tested and why

Computing exact optimal solutions was infeasible even when instances of problems were large as the nature of the Bin Packing Problem is NP-hard. Other algorithms like Backtracking or Branch-and-Bound are computationally infeasible on inputs of small size only, and thus comparisons of optimality cannot be made directly in large input datasets.

Memory profiling at a low system level was not done since it was part of the project environment to ensure that the algorithms were correct and performed well, without focusing on improving the hardware level optimization.

Lastly, the tests of large-scale parallelization and real-time performance were not carried out, as the implementation was performed in single-thread mode and in a conventional academic development environment.

Section 8: Future Work

- Potential improvements to your implementation

Adaptive parameter control can be introduced as one of the improvements. In the existing implementation, population size, mutation rate as well as maximum generations are

predetermined during the execution. Dynamically adjusting these parameters depends on the convergence behavior or solution diversity may give better exploration in early generations and fine-tuning in later generations.

The other enhancement is optimizing the fitness function. Currently, the fitness function is based only on bin fill rate. Although this single-objective formulation is effective, it is not necessarily the one that minimizes the number of bins globally.

The item frequency tracking is not the only method of expanding the belief space. Structural beliefs could also be introduced in future applications, like favourable combinations of items or penalties to inappropriate item combinations. This would enable the belief system to embrace more detailed patterns in high quality solutions.

Another possible improvement is performance optimizations in the code level. As one example, storing fitness values more aggressively, not recalculating item sums repeatedly, or using more efficient data structures might slow down large problem instances.

- Additional algorithms to explore

In the future, it may be possible to use and test more algorithms to compare them. Greedy heuristics e.g. Best Fit Decreasing (BFD) or hybrid approaches that combine the strength on two or more algorithms

metaheuristic methods might also be considered such as Particle Swarm Optimization (PSO), Simulated Annealing and Tabu Search. Such techniques provide alternative exploration/exploitation trade-offs and can be more successful than Cultural Algorithms in particular problems.

Also, the approaches based on reinforcement learning, including the deep Q-learning models, can be explored to allow the algorithm to acquire efficient packing strategies through experience instead of being based entirely on the evolutionary mechanisms.

- Hybrid approaches

The development of hybrid methods of multiple algorithms is becoming a promising shift towards future work. Indicatively, a greedy heuristic like First Fit Decreasing can be applied to create high quality starting populations, which are optimized with the Cultural Algorithm.

A hybrid approach may also be a combination of precise algorithms such as Backtracking or Branch-and-Bound to smaller subproblems, and Cultural Algorithm to large scale problems. This would enable optimizing on the exact areas where it is computationally possible and heuristic optimization where it is necessary.

Another possible direction is to hybridize Cultural Algorithms with the machine learning models. A learning-based aspect may be applied to make better predictions of a good combination of items or smarter over time in executing mutation and crossover operations.

- Real-world deployment considerations

To implement it in the real-world, there are a number of practical considerations to be taken into consideration. One of the first concerns is scalability, in particular, when dealing with large-scale industrial uses with tens or hundreds of thousands of items. Such environments would require parallelization and multi-threaded execution to have reasonable run times.

The quality of robustness is also important in systems of the real world. The algorithm would have had to deal with dynamic inputs, e.g. items coming in with time or bin capacity variation. The implementation could be expanded to online and dynamic bin packing and this would greatly increase its applicability.

Standard data interfaces and error-handling systems would be needed to integrate with other external systems, like warehouse management or resource allocation platforms. Furthermore, the presence of clear interpretability of solutions, including the explanation of the reason why some items were clustering with others, can also be crucial in the operational decision-making.

Lastly, the algorithm deployment on embedded systems or cloud-based infrastructures should be thought of in terms of energy efficiency and hardware limitations. This would assist in optimizing runtime and the cost of memory usage to minimize operations and bring about sustainability.

Section 9: Conclusion

In conclusion this phase of this project successfully delivered a complete implementation, evaluation and comparison of two different strategies for solving the Bin Packing Problems. The goal in this phase was not only to implement both methods, but also to thoroughly understand how exact and evolutionary approaches behave under controlled experimental conditions. By creating both solvers from scratch, integrating them into a shared testing environment and GUI, and evaluating their performance across multiple metrics, we gained a clear and practical understanding of their strengths, limitations and trade offs.

A major accomplishment of this phase was demonstrating that the backtracking algorithm always achieves the optimal solutions. Even with moderately sized datasets, it consistently produced the minimum number of bins and maintained perfect packing accuracy. In our implementation it also achieved faster execution times than expected, performing even quicker

than the cultural algorithm on the tested datasets. However, this still comes with the known limitation that its runtime and memory usage increase sharply when scaling to much larger datasets. In contrast the Cultural Algorithm produced high quality near optimal solutions but with slower execution times in our experiments and higher overhead due to population handling. Its results were consistent across runs and it still scaled well as the number of items increased, showing its effectiveness as a flexible and efficient heuristic approach even though it was slower than backtracking in our implementation.

Several key insights emerged from the comparative analysis, First the backtracking is ideal in scenarios where absolute optimality is required but it becomes impractical as the problem size grows. Second, the cultural algorithm supported by the belief space learning, crossover and mutations provides a strong balance between accuracy and computational efficacy. While it does not guarantee perfect results it consistently produces packings with high fill rates, but in our tests it required more time and memory compared to backtracking. Third, the experiments clearly highlighted the core trade offs in bin packing optimality versus scalability. Backtracking excels in precision and surprisingly speed in our implementation, while the cultural algorithm excels in adaptability and scalability across increasing dataset sizes.

From these observations the final recommendation is that the algorithm selection should depend on the problem scale. For small datasets where optimality is essential, Backtracking is the preferred method and based on our results also provides faster performance. For medium to large datasets or real world scenarios involving hundreds or thousands of items the cultural algorithm is a better choice due to its stability, scalability and ability to maintain consistent near optimal solutions even if it is slower in execution time compared to backtracking. It provides solutions that are close enough to optimal for practical use, while avoiding the exponential cost of exact search methods.

Overall, this phase provided valuable technical and conceptual insights. It demonstrated the importance of choosing the right algorithmic approach for the right context, highlighted the strengths of evolutionary heuristics for large scale problems and reinforced the usefulness of exact methods as benchmarks for solution quality. The project also benefited from a clear GUI interface that improved usability, repeatability and interpretations of results. As a whole this phase strengthened our understanding of optimization techniques in bin packing and prepared a solid foundation for future improvements , hybrid methods and more advanced algorithm exploration.

Section 10: References

[1] G. Dósa, A. Éles, A. R. Goswami, I. Szalkai, and Z. Tuza, “Solution of bin packing instances in Falkenauer T class: Not so hard,” *Algorithms*, vol. 18, no. 2, art. 115, Feb. 2025, doi: 10.3390/a18020115.

[2] R. Braune, “Packing-based branch-and-bound for discrete malleable task scheduling,” *Journal of Scheduling*, vol. 25, no. 6, pp. 675–704, Sep. 2022, doi: 10.1007/s10951-022-00750-w.

[3] W. Ao, G. Zhang, Y. Li, and D. Jin, “Learning to solve grouped 2D bin packing problems in the manufacturing industry,” *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 3713–3723, Aug. 2023. doi:10.1145/3580305.3599860

[4] S. Divakaran, “A fast scalable heuristic for Bin Packing,” arXiv.org, https://arxiv.org/abs/1904.12467 (accessed Nov. 12, 2025).

[5] D.S Johnson, A. Demers, J.D. Ullman, M.R. Garey, and R.L. Graham, “Worst-Case Performance Bounds for Simple One-Dimensional Packing Algorithms,” SIAM Journal on Computing, vol.3, no. 4, pp.299-325, Dec. 1974, https://doi.org/10.1137/0203025

[6] B.S.Baker “A new proof for the first-fit decreasing bin-packing algorithm”, journal of algorithms, vol. 6, no.1, pp. 49-70, Mar. 1985, https://doi.org/10.1016/0196-6774(85)90018-5.

[7] M. Delorme, M. Iori, and S. Martello, “Bin Packing and cutting stock problems: Mathematical models and exact algorithms,” European Journal of Operational Research, vol. 255, no. 1, pp. 1-20, Feb. 2016, https://doi.org/10.1016/j.ejor.2016.04.030

[8] G. Christensen, A. Khan, S. Pokutta, and P. Tetali, *“Multidimensional Bin Packing and Other Related Problems: A Survey,”* Georgia Institute of Technology, Atlanta, GA, USA, 2017. https://tetali.math.gatech.edu/PUBLIS/CKPT.pdf

[9] K. A. K. Kusuma, S. A. Rachman, and A. A. Suprayogi, “Smart Packing Simulator for 3D Packing Problem Using Genetic Algorithm,” *IOP Conference Series: Journal of Physics: Conference Series*, vol. 1447, no. 1, pp. 1–7, 2020.https://doi.org/10.1088/1742-6596/1447/1/012041

[10] A. I. Ali, E. Keedwell, and A. Helal, “A Differential Pheromone Grouping Ant Colony Optimization Algorithm for the 1-D Bin Packing Problem,” in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO ’24)*, ACM, 2024. https://doi.org/10.1145/3638529.3654074

[11] K. Mellou, M. Molinaro, and M. Zhou, “The Power of Migrations in Dynamic Bin Packing,” in *Proceedings of the 2024 ACM-SIAM Symposium on Discrete Algorithms (SODA ’24)*, ACM, pp. 1–20, 2024. https://dl.acm.org/doi/10.1145/3700435

[12] C.C. Lee and D.T. Lee, “A simple on-line bin-packing algorithm,” Journal of the ACM, vol. 32, no. 3, pp. 562-572, Jul. 1985, https://doi.org/10.1145/3828.3833