|  |
| --- |
| UCLan Cyprus |
| CO1406 – Algorithms and Data Structures Assignment Report |
| A thorough analysis of an assignment regarding algorithms and data structures |

|  |
| --- |
| Prepared by: Panayiotis Theodorou  3-17-2023  Tutor: Panagiotis Andreou |

Table of Contents

Table of Contents

[Abstract 2](#_Toc130226633)

[Introduction 2](#_Toc130226634)

[Research: 2d bin packing problem – knapsack problem 2](#_Toc130226635)

[2D Bin Packing Problem 2](#_Toc130226636)

[Knapsack Problem 2](#_Toc130226637)

[Several Potential Solutions 2](#_Toc130226638)

[Heuristic Solutions Researched through the 2D Bin Packing Problem 3](#_Toc130226639)

[Heuristic Solutions Researched through the Knapsack Problem 3](#_Toc130226640)

[Algorithm Explanation 3](#_Toc130226641)

[Loading of the Configurations and Error Control 3](#_Toc130226642)

[Helper Functions 3](#_Toc130226643)

[CanPlace 3](#_Toc130226644)

[Place 4](#_Toc130226645)

[Backtrack 4](#_Toc130226646)

[findNextBox 5](#_Toc130226647)

[SwapToNextConfig 5](#_Toc130226648)

[Solver 6](#_Toc130226649)

[Datatype Selection 6](#_Toc130226650)

[Complexity Analysis 7](#_Toc130226651)

[Load packer problem 7](#_Toc130226652)

[Problem solver 7](#_Toc130226653)

[Time Complexity Analysis Charts 7](#_Toc130226654)

[Analysis conclusion 8](#_Toc130226655)

[Heuristic methods 8](#_Toc130226656)

[Sorting boxes – applied method 8](#_Toc130226657)

[Results 9](#_Toc130226658)

[Studied heuristics – not applied methods 9](#_Toc130226659)

[Study Material 10](#_Toc130226660)

[Materials used for this assignment 10](#_Toc130226661)

[‌Materials that were just looked upon 10](#_Toc130226662)

# Abstract

The Container Packing problem is this year’s CO1406 – Algorithms and Data Structures Assessment. All important details regarding this project will be explained in the following report, meaning that the origin, logic, identity and complexity of this problem will be analyzed. The report aims to satisfy whatever question may the program arise to its users.

# Introduction

The Container Packing Problem (CPP) falls into the category of optimization problems and is a problem like the 2D Bin Packing Algorithm and the Knapsack Problem, that functions with a stack to utilize backtracking. The goal of this problem is to pack a number of boxes into a container without leaving any box behind and without any of the boxes overlapping the container edges. Real world uses are often seen in transportation of heavy or important goods. Most CPPs are done in a 2d or 3d environment and use different shapes and weights.

# Research: 2d bin packing problem – knapsack problem

As mentioned above two of the most common optimization problems are the 2d bin packing problem and the knapsack problem. Both problems are **NP-hard** meaning that there is never an efficient enough solution, therefore these are problems that often require heuristic methods to provide an almost optimal solution.

## 2D Bin Packing Problem

This problem involves packing a certain number of rectangular items into the minimum number of bins possible, while minimizing the excess space. This problem is most seen in fields like transportation and logistics.

## Knapsack Problem

This problem on the other hand is very similar to the CPP that we need to create in this assignment, meaning that we typically must pack a number of items with given values (weight, size etc.) into a container, but if some items do not fit in the given configurations, then they are not placed at all, unlike the CPP.

## Several Potential Solutions

There can never be a solution that would match the requirements of the brief directly, since the program must run without the use of heuristics. Although, since the problem is very similar to the two problems mentioned above there are many heuristic solutions that can be used in this scenario as well. Therefore, if we apply similar approaches like those applied in the 2DBPP and the knapsack problem, we will have several different solutions, varying in both time and size complexity.

These are the some of the potential solutions that can be applied in this scenario:

### Heuristic Solutions Researched through the 2D Bin Packing Problem

* First-Fit Algorithm
* Best-Fit Algorithm
* Next-Fit Algorithm

### Heuristic Solutions Researched through the Knapsack Problem

* Greedy Algorithm
* Dynamic Programming Algorithm

# Algorithm Explanation

## Loading of the Configurations and Error Control

The loadPackerProblem is responsible for the loading of configurations and the error control.

**Signature:** PackerProblem\* loadPackerProblem(string filename)

**Parameters:** The function takes as a parameter the file name which is given to it in the main function.

**Explanation:**

This function takes as a parameter a file which it breaks down into pieces to assign all of the mandatory variables. It starts by reading the container width, length, and the number of boxes, then it creates a Box type array and initializes each box. Later on a 2D pointer array is initialized that represents visually the container and is filled with zeros. Finally, the function does error control for all mentioned above.

## Helper Functions

### CanPlace

CanPlace is a function that checks whether a box can fit or not, it also checks if it can fit rotated or not.

**Signature:** bool canPlace(PackerProblem\* problem, Coordinates& coords, short& currentBox)

**Parameters:** The PackerProblem object, the Coordinated object, and a reference to a short that resembles the current box.

**Explanation:**

The function starts by assigning two Boolean variables (canPlace and verticalPlace) to true. Then, it checks if the given box can be placed vertically or horizontally and it returns false if neither is true. If one of these conditions is true, then the function checks whether the box will be in contact with any other boxes upon placement. If it does then return false, else the box can be placed.

### Place

The place function is responsible for placing a box in the container based on the given coordinates and rotation.

**Signature:** void place(PackerProblem\* problem, Coordinates& coords, short& currentBox)

**Parameters:** The PackerProblem object, the Coordinated object, and a reference to a short that resembles the current box.

**Explanation:**

If the box needs to be placed horizontally then the function will loop up to the rows and columns that will be occupied by set box and changes the zeros in those positions to the box’s name. The same logic is applied for the vertical placement as well, finally decrement the total number of boxes. The rotation is specified through the coords.

### Backtrack

The backtrack function is used to undo the placement of boxes in the container when a solution cannot be found.

**Signature:** void backtrack(PackerProblem\* problem, Coordinates& coords, Stack<State>\* stack)

**Parameters:** The function takes as parameters the PackerProblem object, the Coordinates object, and a pointer to the stack of type <State>

**Explanation:**

The backtrack function starts by checking if the stack is empty, then it retrieves the coordinates and the current box from the stack. Then placed is set to false for the box. The function then loops through the container so that it can remove every instance of the box. Later it removes the coordinates and the box index from the stack, and it increments the number of boxes to be placed. Further the backtracked Boolean is set to true for that box so that it does not place it in the same position. This function is key to trying different placements.

### findNextBox

The findNextBox function finds the next unplaced box.

**Signature:** void findNextBox(PackerProblem\* problem, short& currentBox, unsigned short& numOfBox)

**Parameters:** The PackerProblem object and two unsigned shorts current box and number of boxes

**Explanation:**

The function first checks if the current box is greater than 0 and decrements it if it is, so that it moves to the previous box. Then the function loops through the array of boxes and it checks for each box whether it is placed or not. If the current box has been placed, the function decrements the current box numbers and continues checking. If it reaches the beginning of the array, then it sets the current box to the number of boxes. Overall, this function finds the next unplaced box.

### SwapToNextConfig

The SwapToNextConfig function swaps the current box with the box at a certain index in the boxArray. This swapping is done so that a new configurations of boxes is done for the algorithm to try.

**Signature:** void swapToNextConfig(PackerProblem\* problem, short& currentBox, unsigned short& numOfBox, unsigned short& traversal)

**Parameters:** The PackerProblem object, the current box, the number of boxes and the traversal

**Explanation:**

This function swaps the current box with another box in an array, creating a new configuration for the packing algorithm. The traversal variable keeps track of the current configuration being explored, and the function resets this variable once all possible configurations have been explored.

## Solver

This is the function responsible to use all the helper functions in order to figure out a correct configuration based on the file provided.

**Signature:** PackerSolver\* solver(PackerProblem\* problem, bool debug = false)

**Parameters:** The PackerProblem object and a Boolean debug

**Explanation:**

The first step that the solver does is to create the PackerProblem object and a Stack. The solver engages in a while loop that goes until all boxes have been placed. Within the loop several checks are done to determine the correct action to take.

1. If the current box can be placed at the current coordinates, it is placed, and the function continues to the next iteration of the loop.
2. If the current coordinates are empty and not all boxes have been checked for a possible placement, the function calls the findNextBox function to find the next unplaced box.
3. If the current coordinates are not empty, the function increments the column coordinate.
4. If none of these conditions are met, the function backtracks to the previous state and tries a different configuration.

While still in the loop during each iteration of the loop, the coordinates are updated, and if they have reached the end of the container, they are reset to the beginning. Once all boxes have been placed, the function outputs the number of steps taken and returns the solvedProblem object.

# Datatype Selection

Throughout the entire program I follow this set of rules:

**If a variable needs to be of numeric value:** **unsigned short (2 bytes)**.

**Other possible data types:** int(4 bytes), double (8 bytes), float (4 bytes), short (2 bytes)

**Reasoning:** The unsigned short is the data type that takes the fewest amount of all other numeric data types, it takes just 2 bytes, but it doesn’t allow negative numbers.

**Exceptions:** Wherever indexing is required I use int.

**If a variable needs to be a text-based variable:** **char(1 byte).**

**Other possible data types:** String (depends)

**Reasoning:** It’s the smallest datatype available for text.

**Exceptions:** In the error control segment where I use strings to manage sufficient error management.

# Complexity Analysis

## Load packer problem

Since the Loading function is always dependent on the input size, the time complexity of it is O(n). That is because of the while loop which always goes from 0 to n.

**Worst Case Scenario: O(n)**

## Problem solver

The problem solver is of worst-case time complexity O(n^2) since there is a while loop that is dependent on the number of boxes inputted(n), and then the solver calls the findNextBox function which also initializes a similar while loop, that traverses the boxes(n).

**Worst Case Scenario: O(n^2)**

## Time Complexity Analysis Charts

For more detail on the time needed for each example file, charts are provided below for when the boxes are sorted and for when they are not.

## Analysis conclusion

When applying sorting the time needed for execution is greater, which is not a negative indicator since it is expected. Although sorting adds time for this scenario the objective is just to provide the time needed for the files. The importance of sorting will be analyzed later.

# Heuristic methods

As mentioned above the problem on hand is an NP-hard problem meaning that there cannot be an efficient enough solution for all scenarios, therefore the only way to make it more efficient is through the use of heuristics. Heuristics are methods that are referred to as trade offs in the computer science world since exchange precision with speed. Int this case they do not make the program lack in precision since the heuristics applied only help it in speed.

## Sorting boxes – applied method

One simple enough method to create a heuristic algorithm is to sort whatever it is that the program is examining, therefore in this instance Merge Sort is used to sort the boxes before anything else is done. **Has worst case complexity of O(nlogn).**

The following charts show the difference in steps needed when sorting is not applied and when it is.

### Results

Each of the charts were created after a minimum of 10 executions with and without sorting for each file. All files are relatively easy to solve since they don’t have many boxes and the container doesn’t have much area to cove, but in the one instance (example file 6) where the numbers are higher the algorithm without sorting reached 18405 steps, with sorting it had a maximum of 105. This absolutely proves that sorting the boxes helps the algorithms tremendously.

## Studied heuristics – not applied methods

The studied heuristics are the methods mentioned in the research section, these are methods that although prove efficient were not applied since I did not wish to center my program around it only working with heuristics and since a requirement is that the program must work only with a stack and backtracking while heuristics only amplify the performance, I did not use them.

For 2DBPP

* First-Fit Decreasing (FFD) heuristic: This heuristic sorts the items by decreasing size and then assigns them to the bin using the First-Fit heuristic, which places the item in the first bin that can accommodate it.
* Best-Fit Decreasing (BFD) heuristic: Like FFD, this heuristic sorts the items by decreasing size, but instead of using the First-Fit heuristic, it uses the Best-Fit heuristic, which places the item in the bin that leaves the least amount of unused space after the item is placed.

For Knapsack Problem

* Greedy heuristic: This heuristic selects items with the highest value-to-weight ratio and packs them into the knapsack until no more items can be added.
* Dynamic Programming (DP) heuristic: This heuristic breaks the problem down into subproblems and uses the solutions to smaller subproblems to solve the larger problem. DP can be used to solve the 0-1 Knapsack Problem and the Unbounded Knapsack Problem.

# Study Material

## Materials used for this assignment.

* Balodi, T. (n.d.). *A Complete Guide to Solve Knapsack Problem Using Greedy Method | Analytics Steps*. [online] www.analyticssteps.com. Available at: https://www.analyticssteps.com/blogs/complete-guide-solve-knapsack-problem-using-greedy-method [Accessed 20 Mar. 2023].
* blog.boot.dev. (2020). *Examples of Heuristics in Computer Science*. [online] Available at: https://blog.boot.dev/computer-science/examples-of-heuristics-in-computer-science/.
* Google Developers. (n.d.). *The Knapsack Problem | OR-Tools*. [online] Available at: https://developers.google.com/optimization/pack/knapsack [Accessed 20 Mar. 2023].
* Weisstein, E.W. (n.d.). *NP-Hard Problem*. [online] mathworld.wolfram.com. Available at: <https://mathworld.wolfram.com/NP-HardProblem.html>.

## ‌Materials that were just looked upon

* <https://uphf.hal.science/hal-03723754/document>
* https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=2cb8247534c9e889ac42b2362f0ad96c8c6b8c77