The solution consists of multiple projects:

- **Packing**, which manages creating a packing solution from packing vector and packing input.
- **PackingTests**, which are referencing **Packing** and testing more complex classes and methods of that project.
- **Evolution**, which references **Packing** and manages evolution of packing vector through generations.
- **SettingForm**, which references **Evolution** and handles user setting of the evolution and the packing.
- **RandomInputGenerator**, a simple project that references **Packing** and helps create a JSON packing input.
- **App**, that is referencing both **Evolution** and **SettingForm** and handles the program itself, meaning getting user input through form, the loading the packing input JSON file, running the evolution and the saving the output.

App

PackingInputLoader

Class responsible for loading packing input data from a JSON file.

- **PackingInput LoadFromFile(string fileName)** – Loads a PackingInput from a JSON file.

ContainerExport

Record representing the container after packing that is going to be saved to JSON.

- **int ContainerID** – Unique identifier of the container.
- **long CurrentWeight** – Current total weight of all packed boxes in the container.
- **long OccupiedVolume** – Total volume occupied by packed boxes.
- **IReadOnlyList<PackedBoxExport> PackedBoxes** – List of exported packed boxes in the container.

PackedBoxExport

Record representing a single packed box that is going to be exported to JSON.

- **int BoxID** – Unique identifier of the box.
- **string Rotation** – Rotation of the box within the container.
- **Region OccupiedRegion** – Region occupied by the box in the container.

PackedBoxExtensionForExport

Static class providing extension methods for exporting packed box data.

- **PackedBoxExport ExportPackedBox(this PackedBox packedBox)** – Converts a PackedBox instance into a PackedBoxExport.

ContainerExtensionForExport

Static class providing extension methods for exporting container data.

- **ContainerExport ExportContainer(this ContainerData container)** – Converts a ContainerData instance into a ContainerExport.

PackingOutputSaver

Static class responsible for saving the exported packing results to a file.

- **void SaveToFile(IReadOnlyList<ContainerData> containers, string fileName)** – Exports a list of containers and writes them to a JSON file.

Program

Main entry point of the packing application. Coordinates the loading of input, execution of the packing algorithm, and saving of output.

- **static void Main**() – Application entry point. Steps:
    - Opens the program settings form (FormProgram.Run()) and retrieves user-provided settings. If the setting is null, program returns.
    - Loads packing input from a JSON file using PackingInputLoader.LoadFromFile().
    - Executes the evolutionary packing algorithm via EvolutionProgram.Run() using the loaded input and settings.
    - Saves the resulting packed containers to a JSON file using PackingOutputSaver.SaveToFile().

SettingForm

This project handles the user setting through windows forms.

SettingsForm

Graphical user interface for configuring program settings before running the packing algorithm.

Provides fields for selecting input/output JSON files, choosing placement and packing order heuristics, selecting algorithm type, and defining evolutionary parameters.

- **ProgramSetting ProgramSetting** – Holds the final configuration entered by the user, which is passed to the program when the form is confirmed.
- **SettingsForm()** – Initializes the form, sets title, size, colors, and creates all controls. Also applies consistent styling to labels, buttons, checkboxes, and combo boxes.
- **InitializeComponents()** – Creates and positions all input fields, labels, and buttons on the form.

FormProgram

Static class responsible for launching and managing the **SettingsForm** graphical interface for program configuration. It provides a Run method that displays the settings dialog and returns user-defined program settings.

- **ProgramSetting? Run()**
  Displays the SettingsForm to the user and returns the configured program settings if confirmed.
  - If the user confirms with **OK**, returns the configured ProgramSetting.
  - If the user cancels, returns null.

Evolution

This project contains everything necessary for fitness evaluation of a packing solution and evolution of the packing vector.

ContainersFitnessEvaluator

It is responsible for evaluating the fitness of a solution in the packing problem based on container utilization.

- **double EvaluateFitness(IReadOnlyList<ContainerData> containers)** Evaluates the fitness of the given list of containers.
  - For each container, calculates the value as the maximum of:
    - relative weight capacity usage of the container.
    - relative volume capacity usage of the container.
  - Tracks the minimum utilization value across all containers.
  - Returns the fitness score as this value plus number of containers:

PackingVectorFitnessEvaluator

Class responsible for evaluating the fitness of packing vectors. It uses a PackingVectorSolver to transform a packing vector into a container solution and then evaluates that solution with an IFitnessEvaluator<IReadOnlyList<ContainerData>> (that is interface implemented by ContainerFitnessEvaluator).

- **PackingVectorFitnessEvaluator(IFitnessEvaluator<IReadOnlyList<ContainerData>> containersFitnessEvaluator, PackingVectorSolver packingVectorSolver)**
  Initializes the evaluator with a container fitness evaluator and a packing vector solver.
- **double EvaluateFitness(PackingVector packingVector)**
  - Solves the given packing vector into a list of ContainerData using _packingVectorSolver.
  - Evaluates the resulting containers with _containersFitnessEvaluator.
  - Returns the fitness score as a single double value.
- **IReadOnlyList<double> EvaluateFitnesses(IReadOnlyList<PackingVector> packingVectors)** – Evaluates multiple packing vectors in parallel for performance.
- **static PackingVectorFitnessEvaluator Create(PackingInput inputData, PackingSetting packingSetting)**
  - Factory method for constructing a new evaluator.
  - Internally creates:
    - A PackingVectorSolver from input data and settings.
    - A ContainersFitnessEvaluator.
  - Returns a fully initialized PackingVectorFitnessEvaluator.

ProgramSetting

A configuration record that encapsulates all the necessary settings for running the packing program.It is an immutable data structure passed to the main program execution.

- **string SourceJson** – Path to the **input JSON file**, which contains the definition of containers and boxes to be packed.
- **string OutputJson** – Path to the **output JSON file**, where the packing results (final container states) will be saved.
- **PackingSetting PackingSetting**
- **string AlgorithmName** – Name of the **evolutionary algorithm** (or other algorithm) that will be used to solve the packing problem.
- **int NumberOfIndividuals** – Size of the population in the evolutionary algorithm.
- **int NumberOfGenerations** – Number of iterations (generations) that the evolutionary algorithm will run.

EvolutionaryAlgorithms

A static factory class that provides access to available evolutionary algorithms for solving the packing problem. It maintains a dictionary of algorithms identified by their names.

- **EvolutionaryAlgorithmDictionary** (private static readonly Dictionary<string, Func<…>>)
    - Internal mapping between a string identifier (algorithm name) and a factory function.
    - Each factory function instantiates an implementation of **IEvolutionary<PackingVector>** given:
        - **IReadOnlyList<PackingVector>** initial population
        - **IMultipleFitnessEvaluator<PackingVector>** fitness evaluator
        - **IEvolutionData<PackingVector>?** optional evolution data (may be null)
        - **double stopValue** (termination criterion).
    - Currently supported algorithm:
        - "Differential Evolution" → PackingVectorDifferentialEvolution
- **string[] EvolutionaryAlgorithmsArray** – Returns an array of all available evolutionary algorithm names.
- **GetEvolutionaryAlgorithm(string name, IReadOnlyList<PackingVector> initialPopulation, IMultipleFitnessEvaluator<PackingVector> fitnessEvaluator, IEvolutionData<PackingVector>? data, double stopValue)**
    - Returns an instance of the requested evolutionary algorithm.

IEvolutionary<T>

Interface defining the contract for all evolutionary algorithms. It provides access to the current state of the evolution process, including populations, fitness values, and best individuals.

- **int CurrentGeneration** – The current generation number of the evolutionary process.
- **IReadOnlyList<T> CurrentGenerationPopulation** - The collection of individuals (solutions) in the current generation.
- **IReadOnlyList<double> CurrentGenerationFitness** – The fitness values corresponding to the individuals in CurrentGenerationPopulation.
- **(T individual, double fitness) CurrentGenerationBest** – The best individual in the current generation along with its fitness score.
- **(T individual, double fitness) GlobalBest** – The best individual found across all generations of the evolutionary process.

- **void Evolve(int numberOfGenerations)** – Runs the evolutionary process for the specified number of generations.

EvolutionaryBase<T>

Abstract base class implementing the IEvolutionary<T> interface. It provides the common functionality for evolutionary algorithms such as tracking generations, populations, fitness values, and best individuals.

- **int CurrentGeneration**
- **IReadOnlyList<T> CurrentGenerationPopulation**
- **IReadOnlyList<double> CurrentGenerationFitness**
- **(T individual, double fitness) CurrentGenerationBest**
- **(T individual, double fitness) GlobalBest**
- **void Evolve(int numberOfGenerations)** – Evolves the population for the specified number of generations. For each generation:
    - Calls NextGeneration() (abstract, must be implemented by subclass).
    - Updates CurrentGenerationBest and potentially GlobalBest.
    - Records progress via _data.Update(...).
    - Stops early if HardStopNow() is satisfied.

Protected:

- **double? _hardStop** – Optional hard stop value for fitness. If reached, evolution terminates early.
- **IEvolutionData<T> _data** – Data collector for tracking evolutionary progress (e.g., statistics or visualization).
- **IComparer<double> _fitnessComparer** – Comparator used to determine whether fitness is minimized or maximized.
- **IMultipleFitnessEvaluator<T> _fitnessEvaluator** – Evaluator responsible for computing fitness values of individuals.
- **abstract void NextGeneration()** – Defines how the next generation is created (e.g., crossover, mutation, selection).

DifferentialEvolution<T>

Class implementing the Differential Evolution algorithm for evolving a population of individuals. It extends EvolutionaryBase<T> and defines the process of creating new generations through tournament selection and uniform mutation.

- **protected override void NextGeneration()** – Selects individuals using tournament selection, then mutates the selected individuals with a random partner (tournament of size 1). Evaluates the fitness of mutated individuals and replaces the current population with the mutated one.

TournamentSelector<T>

Class implementing tournament selection for evolutionary algorithms. It selects individuals from a population based on fitness, optionally minimizing or maximizing.

- **TournamentSelector(bool minimizing)** – Initializes a selector that compares fitness either to minimize (true) or maximize (false).
- **TournamentSelector(IComparer<double> fitnessComparer)** – Initializes a selector with a custom fitness comparer.
- **(IReadOnlyList<T> individuals, IReadOnlyList<double> fitnesses) SelectMultiple(IReadOnlyList<T> population, IReadOnlyList<double> fitness, int number, int tournamentSize)** – Selects multiple individuals from the population using tournament selection. Returns a tuple of selected individuals and their fitnesses.
- **(T individual, double fitness) Select(IReadOnlyList<T> population, IReadOnlyList<double> fitness, int tournamentSize)**
    - Selects a single individual using tournament selection.

PackingVectorPopulationFactory

Class for generating populations of random PackingVector individuals.

- **PackingVectorPopulationFactory(int packingVectorLength)** – Initializes the factory with the desired packing vector length.
- **IReadOnlyList<PackingVector> CreatePopulation(int numberOfIndividuals)** – Generates a population of random PackingVector instances.

EvolutionProgram

Static class responsible for orchestrating the execution of the evolutionary packing algorithm. It manages population creation, fitness evaluation, evolution, solution generation, and validation.

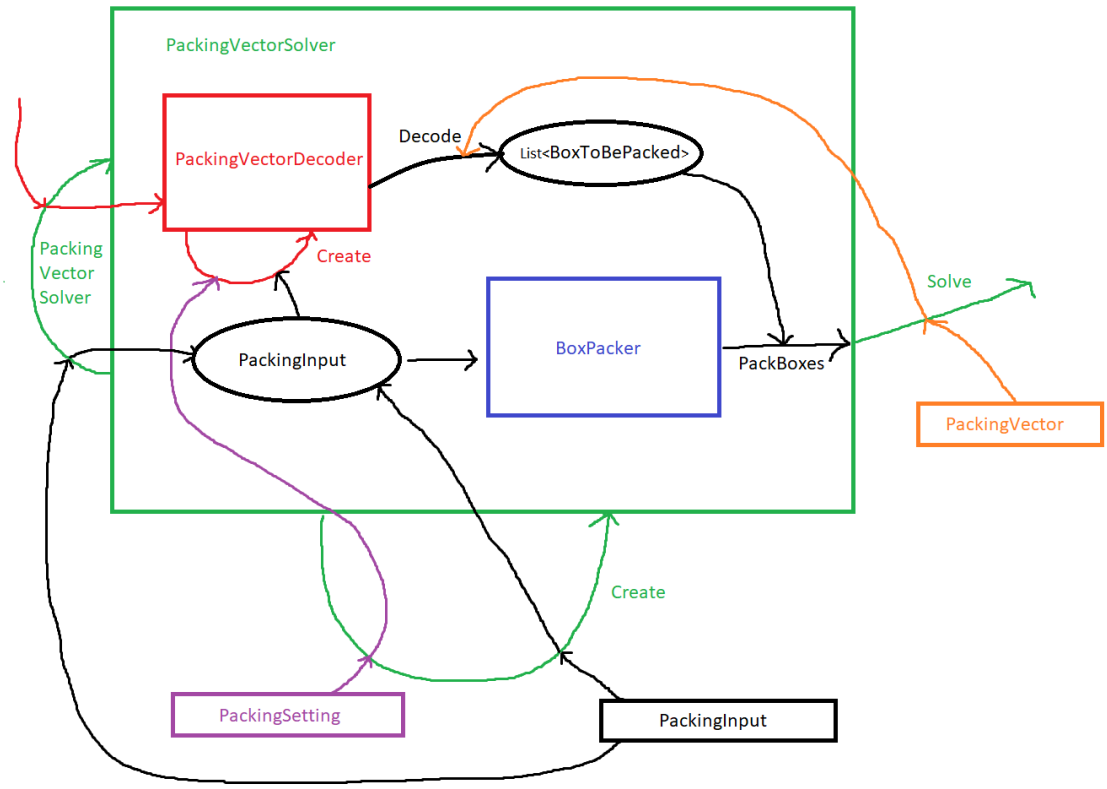
- **IReadOnlyList<ContainerData> Run(ProgramSetting setting, PackingInput packingInput)** – Executes the full evolutionary packing process and returns the resulting container solution.
    - **Process:**
        - Creates an initial population of PackingVector individuals using CreateInitialPopulation().
        - Creates a PackingVectorFitnessEvaluator for evaluating packing vectors.
        - Determines a stopValue based on a lower bound of the problem.
        - Retrieves the selected evolutionary algorithm via EvolutionaryAlgorithms.GetEvolutionaryAlgorithm().
        - Evolves the population for the specified number of generations.
        - Retrieves the best individual (GlobalBest) from the evolutionary process.

- Solves the packing problem with PackingProgram.Solve() using the best vector.
- Validates the solution with PackingValidityChecker().

- **IReadOnlyList<PackingVector> CreateInitialPopulation(PackingInput packingInput, PackingSetting packingSetting, int populationSize)** – Creates a population of random packing vectors.
- **void PackingValidityChecker(IReadOnlyList<ContainerData> containers)** – Validates that no boxes intersect within containers.

Packing

This project includes everything needed for the packing, mainly a method of converting a packing vector and the packing input into unique packing solution.



PackingInput

Represents the input for a packing problem, containing container properties and a list of boxes properties.

- **ContainerProperties ContainerProperties** – Properties of the container (sizes and maximum weight).
- **IReadOnlyList<BoxProperties> BoxPropertiesList** – List of properties of all boxes that we need to pack.
- **int GetLowerBound()** – Calculates a lower bound on the number of containers needed to pack all boxes. Considers both total weight and total volume of all boxes relative to the container's capacity. Returns the ceiling of the maximum of weight-based and volume-based requirements.

ContainerProperties

Represents the properties of a container.

Properties:

- **Sizes Sizes** – Sizes of the container.
- **int MaxWeight** – Maximum allowed weight for the container.

BoxProperties

Represents the properties of a box.

Properties:

- **int ID** – Unique identifier of the box.
- **Sizes Sizes** – Sizes of the box.
- **int Weight** – Weight of the box.

PackingSetting

Represents the settings for packing, including heuristics and rotation options.

Properties:

- **string[] SelectedPlacementHeuristics** – Array of names of placement heuristics selected for packing (e.g., "FirstFit", "BestFit").
- **bool AllowRotations** – Indicates whether box rotations are allowed during packing.
- **string? SelectedPackingOrderHeuristic** – Optional name of a heuristic for determining the order in which boxes are packed (if not chosen, sorting is done using the packing vector instead).

PackingVectorSolver

Class responsible for solving a packing problem based on a given packing vector, input data, and settings. It uses a decoder to interpret the packing vector and a box packer to perform the packing.

- **PackingVectorDecoder PackingVectorDecoder** – Decoder used to convert a packing vector into a list of **BoxToBePacked**.
- **PackingInput PackingInput** – Input data for the packing problem, including container properties and a list of box properties.
- **PackingVectorSolver(PackingVectorDecoder packingVectorDecoder, PackingInput packingInput)** – Initializes a new instance of PackingVectorSolver with the given decoder and packing input.
- **IReadOnlyList<ContainerData> Solve(PackingVector packingVector)** – Solves the packing problem for a given PackingVector. Steps:
    - Decodes the packing vector into a list of BoxToBePacked.
    - Creates a BoxPacker using the container properties.
    - Packs all boxes according to the decoded instructions.
    - Returns a read-only list of ContainerData representing the packed containers.
- **PackingVectorSolver CreateSolver(PackingInput packingInput, PackingSetting packingSetting)** – static factory method for creating a PackingVectorSolver from packing input and settings. Internally creates a PackingVectorDecoder using the provided settings and input and returns a new instance of PackingVectorSolver.

PackingProgram

Static helper class providing high-level methods for creating solvers and solving packing problems, including random packing. Intended as a convenient entry point for applications using the packing library.

- **PackingVectorSolver CreateSolver(PackingInput packingInput, PackingSetting packingSetting)** – Creates a **PackingVectorSolver** for the given packing input and settings.
- **IReadOnlyList<ContainerData> Solve(PackingVector packingVector, PackingInput packingInput, PackingSetting packingSetting)** – Solves a packing problem using a specified **PackingVector**.
- **IReadOnlyList<ContainerData> SolveRandom(PackingInput packingInput, PackingSetting packingSetting)** – Generates a random packing vector and solves the packing problem.
- **int GetPackingVectorExpectedMinimalLength(PackingInput packingInput, PackingSetting packingSetting)**
    - Calculates the expected minimal length of a packing vector based on the number of boxes and packing setting.

a) ContainersAndPacking repository

This repository contains everything needed for a converting a list of **BoxToBePacked** to a valid packing solution, meaning a list of **ContainersData** (all containers used, including all boxes packed in them).

Coordinates
Represents a 3D point with integer coordinates and provides utilities for working with 3D positions.

- **Properties X, Y, Z** – The integer coordinates along the respective axes.
- **Constructor Coordinates(int X, int Y, int Z)** – Initializes the coordinates with the given values.
- **GetEuclideanDistanceTo(Coordinates coordinates)** – Computes the Euclidean distance from this point to another Coordinates instance.

Sizes
Represents the dimensions of a 3D box, with utilities for volume calculation, region conversion, and rotation.

- **Properties X, Y, Z** – The dimensions along the respective axes. Must be greater than 0.
- **Constructor Sizes(int X, int Y, int Z)** – Initializes the sizes with the given values. Throws an exception if any value is ≤ 0.
- **GetVolume()** – Returns the volume of the object as a long value (X × Y × Z).
- **ToRegion(Coordinates start)** – Converts the size into a **Region** starting from the given **Coordinates**.
- **GetRotatedSizes(Rotation rotation)** – Returns a new **Sizes** instance representing the dimensions after applying the specified **Rotation**.

Rotation
Represents one of the six possible axis-aligned rotations of a 3D object. Used to generate different orientations of a box for packing or placement.

- **XYZ** – No rotation; axes remain in original order.
- **XZY** – Swap Y and Z axes.
- **YXZ** – Swap X and Y axes.
- **YZX** – Rotate axes so that Y becomes X, Z becomes Y, X becomes Z.
- **ZXY** – Rotate axes so that Z becomes X, X becomes Y, Y becomes Z.
- **ZYX** – Reverse order of all axes; Z becomes X, Y becomes Y, X becomes Z.

Region
Represents a rectangular region in space, defined by its start and end coordinates. Used to represent occupied or available space in a container or packing scenario.

- **Start** – Coordinates of the minimal corner of the region (smallest X, Y, Z).
- **End** – Coordinates of the maximal corner of the region (largest X, Y, Z).
- **Region(Coordinates start, Coordinates end)** – Creates a standardized region where Start is always the minimal corner and End is the maximal corner, regardless of input order.
- **bool IntersectsWith(Region anotherRegion)** – Returns true if this region overlaps with another region in 3D space.
- **bool IsSubregionOf(Region region)** – Returns true if this region is fully contained within the specified region.
- **bool IsOverregionOf(Region region)** – Returns true if this region fully contains the specified region.
- **Sizes GetSizes()** – Returns the dimensions of the region as a Sizes object (X, Y, Z).
- **long GetVolume()** – Returns the volume of the region.

EmptyMaximalRegions
Represents a collection of all maximal empty regions in a container that are not yet occupied by any box. Maintains and updates the list as new boxes are placed.

- **IReadOnlyList<Region> EmptyMaximalRegionsList** – Returns a read-only list of all current empty maximal regions.
- **EmptyMaximalRegions(Region initial)** – Initializes with a single initial region representing the whole empty space.
- **EmptyMaximalRegions(IEnumerable<Region> initials)** – Initializes with a collection of initial empty regions.
- **void UpdateEMR(Region newOccupied)** – Updates the list of empty maximal regions after a new box (region) is placed.
    - Throws an exception if the new region is not fully contained in any existing empty region.
    - Splits intersecting regions using a region splitter.
    - Removes subregions that are no longer maximal.

BoxToBePacked
Represents a box that is intended to be packed into a container. Contains information about the box's properties, its rotation, and the placement heuristic used to determine how it will be positioned.

- **BoxProperties BoxProperties** – The properties of the box, including its original dimensions and weight.
- **Rotation Rotation** – The rotation applied to the box for packing.
- **PlacementHeuristic PlacementHeuristic** – The heuristic used to decide into which container and at which place should be the box packed.
- **BoxToBePacked(BoxProperties boxProperties, Rotation rotation, PlacementHeuristic placementHeuristic)** – Creates a new BoxToBePacked with the given properties, rotation, and placement heuristic.
- **PackedBox ToPackedBox(PlacementInfo placementInfo)** – Converts the box to a PackedBox once it has been placed.
    - Validates that the occupied region dimensions match the rotated box dimensions.
    - Throws an exception if there is a mismatch.
- **Sizes GetRotatedSizes()** – Returns the dimensions of the box after applying the specified rotation.
- **long GetVolume()** – Returns the volume of the box.

PlacementInfo
Represents the placement of a box within a container. Contains information about which container the box is placed in and the specific region it occupies.

- **int ContainerID** – Identifier of the container where the box is placed.
- **Region OccupiedRegion** – The **Region** within the container that the box occupies.
- **PlacementInfo(int containerID, Region occupiedRegion)** – Creates a new PlacementInfo instance with the specified container ID and occupied region.

Container
Represents a single container in the packing process. Manages the boxes packed inside it, keeps track of weight and volume, and maintains the list of empty maximal regions available for packing.

- **int ID** – Unique identifier of the container.
- **long CurrentWeight** – Current total weight of all packed boxes.
- **long OccupiedVolume** – Total volume currently occupied by boxes in the container.
- **IReadOnlyList<PackedBox> PackedBoxes** – List of all boxes packed in this container.
- **ContainerData Data** – Snapshot of the container's current state, including ID, weight, volume, empty regions, packed boxes, and container properties.

- **Container(int iD, ContainerProperties containerProperties)** – Creates a new container with the given ID and properties. Initializes empty maximal regions based on the container size and sets weight and occupied volume to 0.
- **void PackBox(BoxToBePacked boxToBePacked, PlacementInfo placementInfo)**
  Packs a box into the container at the specified placement. Updates the current weight, occupied volume, and empty maximal regions. Throws exceptions if:
    - The placement's container ID does not match this container.
    - Adding the box would exceed the container's maximum weight.

ContainerData
Immutable record representing a snapshot of a container's current state. Stores weight, volume, packed boxes, empty maximal regions (EMR), and container properties.

Properties:

- **int ID** – Unique identifier of the container.
- **long CurrentWeight** – Total weight of all packed boxes at the time of snapshot.
- **long OccupiedVolume** – Total volume occupied by boxes at the time of snapshot.
- **IReadOnlyList<Region> EMR** – Current list of empty maximal regions (available space) in the container.
- **IReadOnlyList<PackedBox> PackedBoxes** – List of boxes packed in the container.
- **ContainerProperties ContainerProperties** – Properties of the container.
- **ContainerData(int id, long currentWeight, long occupiedVolume, IReadOnlyList<Region> emr, IReadOnlyList<PackedBox> packedBoxes, ContainerProperties containerProperties)**
  Creates a new immutable snapshot of a container's state.
- double **GetRelativeVolume()** – Returns the fraction of the container volume currently occupied by boxes.
- double **GetRelativeWeight()** – Returns the fraction of the container's maximum weight currently used.

PlacementHeuristic

A delegate that defines a heuristic function for choosing where a box should be placed in available containers.

Parameters:

- **BoxToBePacked boxToBePlaced** – The box that needs to be packed.
- **IEnumerable<ContainerData> containersData** – Collection of container snapshots to evaluate.

Returns:

- **PlacementInfo?** – A PlacementInfo object specifying the container and region to place the box, or null if no valid placement is found.

PlacementHeuristics

Static class providing a set of predefined placement heuristics for packing boxes into containers.

- **IReadOnlyList<string> PlacementHeuristicsList** – List of available heuristic names.
- **PlacementHeuristic GetPlacementHeuristic(string placementHeuristicName)** – Returns the heuristic function corresponding to the given name. Throws an exception if the name is unknown.
- **IReadOnlyList<PlacementHeuristic> GetMultiplePlacementHeuristics(string[] placementHeuristicsNames)** – Returns multiple heuristic functions based on the provided names.

Heuristic Methods:

- **FirstFit(BoxToBePacked boxToBePlaced, IEnumerable<ContainerData> containersData)** – Selects the first container and EMR region where the box fits.
- **BestFit(BoxToBePacked boxToBePlaced, IEnumerable<ContainerData> containersData)** – Chooses the region that minimizes the remaining empty space in the selected region.
- **MaxDistance(BoxToBePacked boxToBePlaced, IEnumerable<ContainerData> containersData)** – Places the box in the region farthest from the end corner of the container.
- **MinDistance(BoxToBePacked boxToBePlaced, IEnumerable<ContainerData> containersData)** – Places the box in the region closest to the start corner of the container.

BoxPacker

The class is responsible for creating containers and packing boxes into them based on the heuristic given by each box.

- **IReadOnlyList<ContainerData> ContainersData** – Read-only list of snapshots of all containers, containing current state information (weight, occupied volume, EMRs, packed boxes).
- **BoxPacker(ContainerProperties containerProperties)** – Initializes a new BoxPacker instance with the given container properties, starts with an empty list of containers.
- **void PackBoxes(IEnumerable<BoxToBePacked> boxesToBePacked)** – Packs a collection of boxes sequentially into containers using **PlacementHeuristic** of the box. If there is not place where to pack the box, a new **Container** is opened.

b) PackingVectorAndDecoding repository

To enable effective use of evolutionary algorithms, a packing solution needs a vector representation. This repository defines a **PackingVector**, which can be deterministically decoded into a list of **BoxToBePacked**, resulting in a unique packing solution.

PackingVectorCell

Represents a single, PackingVector element with a value in the range [0,1).
Stores the value internally as a ushort to save memory while supporting high precision.

- **Constructor PackingVectorCell(double value)** – Creates a cell from a double in [0,1). Throws if out of range.
- **ToDouble()** – Converts the cell back to a double.
- **CompareTo(PackingVectorCell other)** – Compares two cells based on their internal value.
- **Implicit conversion to double** – Any cell can be used as a double.
- **Explicit conversion from double** – Must be explicit because the value must be [0,1).

PackingVector

Represents an immutable vector of **PackingVectorCell** values.
Provides utilities for creating, slicing, and converting vectors to common collection types.

- **Count** – Gets the number of elements.
- **this[int index]** – Accesses an element at the given position.
- **Slice(start, length)** – Returns a sub-vector.
- **CreateRandom(length)** – Creates a vector with random values.
- **CreateEmpty()** – Creates an empty vector.
- **CreateZeros(length)** – Creates a zero-filled vector.
- **Implicit conversions** – To double[] and List<double>.
- **Explicit conversions** – From double[] and List<double>.

PackingVectorDecoder

Decodes a PackingVector into usable packing instructions for boxes and containers.
Handles multiple vector parts (placement heuristics, rotations, containers) and sorts boxes according to the specified heuristics.

- **Constructor PackingVectorDecoder(...)** – Initializes the decoder with specific part decoders, box sorter, and packing input.

- **PackingVectorMinimalLength** – Minimum required length of the packing vector so all decoders can operate.
- **Decode(PackingVector packingVector)** – Returns a sorted list of BoxToBePacked based on the packing vector.
- **DecodeContainers(PackingVector packingVector)** – Returns a list of ContainerProperties decoded from the packing vector.
- **Create(PackingSetting packingSetting, PackingInput packingInput)** – Factory method that sets up the decoder with heuristics and decoders according to the given settings.

HeuristicalBoxSorter

Sorts boxes using a predefined heuristic rather than the packing vector.

- **Constructor HeuristicalBoxSorter(IComparer<BoxToBePacked> boxComparer)** – Initializes the sorter with a custom comparer.
- **IsUsingPackingVector** – Indicates whether the sorter uses a packing vector (always false for this class).
- **Sort(IReadOnlyList<BoxToBePacked> unsortedBoxes, PackingVector _)** – Returns a new sorted list of boxes using the heuristic.

OrderHeuristics

Provides predefined heuristic (IComparer) to order boxes for packing.
Includes methods to list available heuristics and obtain comparers for sorting.

- **OrderHeuristicsList** – Returns the names of all available heuristics.
- **GetOrderHeuristic(string orderHeuristic)** – Returns an IComparer<BoxToBePacked> corresponding to the given heuristic name. Throws if the name is unknown.

Built-in heuristics:

- **HighVolumeFirstHeuristic** – Sorts boxes by volume in descending order.
- **HighAreaBaseFirstHeuristic** – Sorts boxes by base area (X×Y) in descending order.
- **LongestFirstHeuristic** – Sorts boxes by their longest dimension in descending order.

PackingVectorUsingBoxSorter

Sorts boxes based directly on a packing vector. Each cell in the vector corresponds to one box; boxes are sorted according to the cell values (between 0 and 1).

- **IsUsingPackingVector** – Indicates that this sorter uses a packing vector (true).

- **Sort(IReadOnlyList<BoxToBePacked> unsortedBoxes, PackingVector packingVector)** – Returns a new list of boxes sorted according to the given packing vector. Throws an exception if the vector is too short.

Internal details:

- Uses a PackingPairComparer to compare (PackingVectorCell, BoxToBePacked) pairs by cell value.

PackingVectorNonUsingPartDecoder<T>
Decoder that ignores the values in a PackingVector and always returns a predetermined value for all elements.

- **Constructor PackingVectorNonUsingPartDecoder(T possibility)** – Initializes the decoder with a specific value that will always be returned.
- **IsUsingPackingVector** – Indicates whether the decoder uses values from the vector (always false for this class).
- **DecodeMultiple(PackingVector packingVector)** – Returns a list of values with the same length as the packingVector, where each element is _possibility.
- **Decode(PackingVectorCell cell)** – Returns _possibility, regardless of the input PackingVectorCell.

Usage:
This decoder is suitable for parts of the solution that are static and do not need to change during evolutionary optimization, e.g., if only one type of rotation or container is allowed.

PackingVectorUsingPartDecoder<T>
Decoder that selects a value from a predefined list of possibilities based on the corresponding PackingVectorCell value.

- **Constructor PackingVectorUsingPartDecoder(IReadOnlyList<T> possibilities)** – Initializes the decoder with a list of possible values. Each vector cell is used to select one of these values.
- **IsUsingPackingVector** – Indicates that this decoder uses the packing vector (true).
- **DecodeMultiple(PackingVector packingVector)** – Returns a list of decoded values corresponding to the vector, where each element is selected from _possibilities based on the cell value.
- **Decode(PackingVectorCell packingVectorCell)** – Converts the vector cell value (in [0,1)) into an index in _possibilities and returns the corresponding value.

Details:

- The cell value is interpreted as a fraction of the _possibilities array length. For example, a cell value of 0.5 in a list of 4 items selects index 2 (0.5 * 4 = 2).

- This decoder is suitable for parts of the packing solution where multiple discrete options exist (e.g., rotations, placement heuristics) and should be determined by the vector.