3D Placement
Design & Results

Chen Yinbo, Ding Youngseog, Kuang Zhengfei, Yu Zhijingcheng

June, 2016

1 Overview

This library is aimed at optimizing the volume of a box where a set of boxes of given sizes can be contained. In order to guarantee a maximal extensibility, large efforts have been made to keep the structure compatible with OOP design principles. Several OOP design patterns have been applied and in this way the use of this library can be highly flexible. A general optimization procedure is supported in this library, where the user is allowed to apply his or her own optimization methods. In addition, the algorithm described in [1] has been implemented and can be directly used as the default method.

2 Build

To build the library, you have to get the `Makefile` using `qmake`.

```
qmake 3d-placement.pro -o Makefile
```

Then simply execute `make` and the compilation will be automatically carried out.

3 Demo

The executable `3d-placement` provides a demonstration of this library. Four testcases are provided in the directory `testcase`, number from 0 to 3. The testcase number needs to be provided when you want to run the demonstration. For example, executing

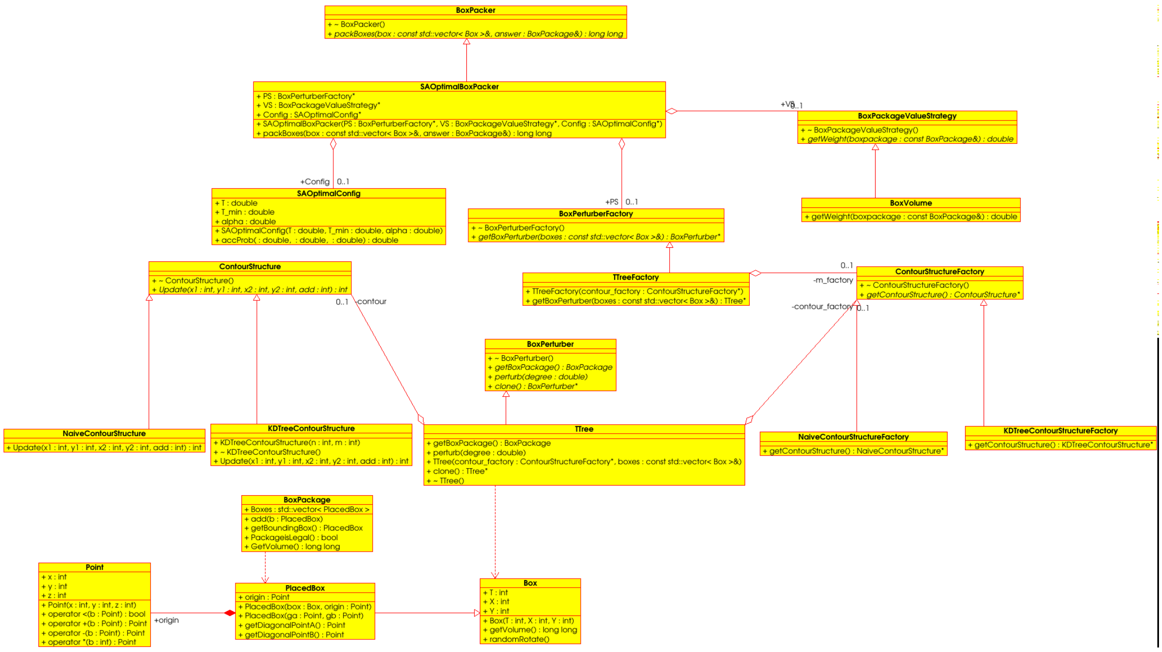
```
./3d-placement 3
```

will run the demonstration on the testcase numbered 3.

Some testcase is relatively large and may take minutes to process, and in this case you are advised to close your eyes and wait patiently.

4 Design

The structure of this library is shown in the UML diagram below.



4.1 Fundamental Elements

The fundamental elements include the objects that are visible in the original problem.

- A `Point` is a point with its position in 3D space.
- A `Box` is a 3D cuboid with three edge lengths specified.
- A `PlacedBox` is a `Box` with a fixed position. The position is actually expressed a `Point`, i.e, one of the corner points of the `PlacedBox`.
- A `BoxPackage` is a set of `PlacedBox`es. It is a legal solution to the original problem if and only if its `PackageisLegal()` function returns `true`, i.e, there is no mutual intersection among the set of `PlacedBox`es.

These are the four elements of the problem. All other classes in this library, are models for the procedure and the methods of the optimization task, but they are all merely operations on these elements.

4.2 `BoxPacker`

The `BoxPacker` is the entry point for users. Since the problem is to find a packing strategy for a set of boxes without speicified positions, the whole abstract process is just a function taking a set of `Box`es as input and returning a `BoxPackage` as output. The `BoxPacker` provides exactly such an abstract interface for users to use. The implementation of the process is left for its subclasses. One default implementation is shipped by its subclass `SAOptimalBoxPacker`.

The design here can be considered as an instance of Strategy pattern, which decouples the interfaces and implementations with the help of polymorphism.

The rest of the library is devoted to the implementation of `SAOptimalBoxPacker`, but we will see that much flexibility is availableeven within `SAOptimalBoxPacker` itself.

4.3 `SAOptimalBoxPacker`

The `SAOptimalBoxPacker` describes a framework for the application of the simulated annealing method in this optimization problem, which is flexible in three dimensions.

The first dimension is the perturbing models for the box package. This dimension is necessary in the simulated annealing because candidate neighbouring states are needed. The models are depicted by `BoxPerturber`s. The default implementation is provided in its subclass `TTree`, which is the T-tree structure proposed in [1].

The second dimension is the objective weight of optimization. One intuitive choice for the origin problem is the volume of the bounding box of the `BoxPackage`, but obviously there are far more choices than this. In addition, flexibility in this dimension allows easy adaptations of this library for other optimization objectives. This dimension is depicted by the `BoxPackageValueStrategy`. Its subclass `BoxVolume` is the intutive default choice.

The third dimension is the parameters of the simulated annealing process itself. These parameters are packed in `SAOptimalConfigs`.

These three dimensions are specified through function arguments. The `BoxPerturber` and the `BoxPackageValueStrategy` are both abstract classes. This is again an application of the Strategy pattern.

4.4 `BoxPerturber`

Its function `getBoxPackage()` converts the `BoxPerturber` into a real `BoxPackage`, and `perturb()` produces perturbance on itself of a dose specified by its argument.

Since what the `SAOptimalPacker` really wants is to generate `BoxPerturber`s rather than use a given one, it is necessary to wrap its generating process in the `BoxPerturberFactory`. This is an application of the Factory Method pattern.

The `TTree` is an implementation of the `BoxPerturber`, and its corresponding factory class is the `TTreeFactory`.

4.5 `TTree`

The `TTree` is still flexible. One stage of its `getBoxPackage()` process requires a structure that supports the maintaining of the maximal value of 2D rectangular areas. This structure is depicted by the abstract class `ContourStructure`. Two implementations are given in the `NaiveContourStructure` and the `KDTreeContourStructure`. The concrete type of the `ContourStructure` is specified by an argument of the constructor of the `TTreeFactory`, which is used to create `TTree`s. This is another application of the Strategy pattern.

Similarly, since `ContourStructure`s are what a `TTree` needs to generate, the factory class `ContourStructureFactory` becomes necessary, and this comprises a second application of the Factory Method strategy. The corresponding factory classes for the two implementations of the `ContourStructure` are `NaiveContourStructureFactory` and `KDTreeContourStructureFactory`.

5 Results

Tests have been performed on the the default method (the implementation of the algorithm in [1]). The inputs were `Boxes` with random edge lengths. The time limit was 10 minutes, and the score

$$t = \frac{\text{Total Volume of Boxes}}{\text{Volume of Bounding Box}}$$

was taken as the measure of the efficacy of the solution.

As n (i.e. the number of boxes) increased, t tended to decrease. However, we can see that for cases where $n \leq 100$, the solutions were reasonably good. Concretely, when $n = 40$, t varied in the range $[0.75, 0.80]$, and when $n = 100$, t fluctuated in the range $[0.70, 0.80]$.

References

[1] Ping-Hung Yul, Chia-Lin Yang, and Yao-Wen Chang. T-Trees: A Tree-Based Representation for Temporal and Three-Dimensional Floorplanning. *ACM Transactions on Design Automation of Electronic Systems, Vol.14*, 2009.