Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

DIPLOMA THESIS ASSIGNMENT

Student:	Bc. Adam H o r k ý
Study programme:	Open Informatics
Specialisation:	Artificial Intelligence
Title of Diploma Thesis:	Multi-agent Solver for Multi-dimensional Bin Packing Problem

Guidelines:

1. Study the problematics of bin packing problem.
2. Study solution approaches with respect to multi-agent systems.
3. Design and implement algorithms for one dimensional problem.
4. Extend the algorithm for multi-dimensional problem.
5. Experimentally validate and evaluate implemented algorithms.

Bibliography/Sources: Will be provided by the supervisor.

Diploma Thesis Supervisor: Ing. Jiří Vokřínek, Ph.D.

Valid until: the end of the summer semester of academic year 2012/2013

[illegible]

prof. Ing. Vladimír Mařík, DrSc.
Head of Department

L.S.

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta elektrotechnická
-4-

[illegible]

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, January 10, 2012

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Cybernetics

Diploma thesis

Multi-agent Solver for Multi-dimensional Bin Packing Problem

Adam Horký

Supervisor: Ing. Jiří Vokřínek, Ph.D.

Study Programme: Open Informatics

Field of Study: Artificial Intelligence

May 10, 2012

Acknowledgements

I would like to thank to my supervisor, Ing. Jiří Vokřínek, Ph.D. for the interesting topic, encouragement and advice he has provided. Further I must thank to my brother Petr, his wife Vanessa and their son Mario for language corrections.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V praze dne 10.5.2012 .... [illegible]

Abstract

This work focuses on multi-agent solution of bin packing problem. Firstly the architecture of multi-agent solver with three types of agents is defined and from this architecture an abstract algorithm is generated. The actual optimization process consists of three parts: allocation, delegation and negotiation. The proposed model was applied to the solution of one-dimensional, two-dimensional and three-dimensional version of bin packing problem. No additional restrictions were taken into account. Various optimization techniques have been used in the negotiations - from simple heuristics through the application of exact algorithms to ILP optimization of related sub-problems. The results demonstrate that some implementations are able to compete with the best current solutions either by the quality of solutions, or by computation time.

(Tato práce se zaměřuje na multi-agentní řešení Bin Packing Problemu. Nejprve je popsána architektura multi-agentního solveru jež definuje tři typy agentů a z ní je poté odvozen abstraktní algoritmus. Samotný optimalizační proces sestává ze tří částí: alokace, delegace a vyjednávání. Navrhnutý model byl aplikován na řešení jedno-dimenzionální, dvou-dimenzionální a tří-dimenzionální verze Bin Packing Problemu. žádná přídavná omezení nebyla brána v úvahu. Různé optimalizační techniky byly využity v rámci vyjednávání - od jednoduchých heuristik přes aplikaci exaktního algoritmu až po ILP optimlizaci dílčích podproblémů. Dosažené výsledky ukazují, že některé implementace jsou schopné konkurovat nejlepším současným řešením buď podle kvality řešení, nebo podle výpočetního času.)

Contents

1 Introduction

Bin Packing Problem (BPP) belongs to standard optimization problems. It is a combinatorial NP-hard problem where objects of different volumes are supposed to be packed into a minimal number of bins (can be called also containers). Although the original definition is about one-dimensional bins and items many other multi-dimensional and multi-constrained derived versions of BPP have been emerging all the time. It is because the problem can be found in different forms almost everywhere not only in the area of packing but also in e.g. cutting (cutting shapes out of sheets of metal, plastic etc.). Since even the simplest variant of BPP is still NP-hard, optimal solutions can not be found for all instances in a polynomial time. Many approaches to finding the best possible solution of such problem in a reasonable time exist. One of the approaches is based on multi-agent planning which is performed in a multi-agent environment.

Bin packing problem can be described in terms of multi-object optimization and therefore it fits into a multi-agent environment. In that environment multiple agents are coordinating and competing to optimize their partial plans that compose a global plan which represents the solution of the optimization problem. This text describes how bin packing problem can be adapted to multi-agent environment and how a multi-agent solver can be used to solve different dimensions variants of BPP. Success of implementations is assessed by evaluating on benchmark data and comparing to state-of-the-art best known solutions.

In the rest of this chapter the problem is addressed more in detail including important variants and related problems which have been encountered. Most of the principles are integrated in the final implementation. The implementation is addressed in Chapter 2. It does not contain any implementation details but rather general ideas described by mathematical expressions or in pseudo-code. The actual implementation with source code can be found on the attached CD (Appendix A). Chapter 3 contains a description of experiments on benchmark data with a discussion of the obtained results which follows in Chapter 4. The final chapter also contains outlook on future work reflecting the gained results.

1.1 Bin Packing Problem

In this section the basic principles and concepts of the subject of this paper – bin packing problem are presented. Main focus is on simple heuristic algorithms and approaches because their general concepts are very important for the implementation of the multi-agent solver. Its allocation and negotiation model counts with a non-complex, quick way of solving related sub problems. The quality of the result is then achieved thanks to the diversity of the multi-agent model. In consequence it means that the literature may be old-dated as new heuristic approaches have not appeared recently. Complex or meta-heuristic (including genetic) algorithms are briefly discussed.

The classical version of the Bin Packing Problem is defined as follows:

Given a bin size V and a list $a_1, \ldots, a_n$ of sizes of the items to pack, find an integer B and a $B - partition\ S_1 \bigcup \ldots \bigcup S_B$ of $\{1, \ldots, n\}$ such that $\sum_{i \in S_k} a_i \leq V$ for all $k = 1, \ldots, B$.

The mathematical foundation of bin packing began in the early 70's. It appeared to be an extremely rich research area: it soon turned out that this simple model could be used for a variety of different practical problems, ranging from a large number of cutting stock applications to packing trucks with a given weight limit, assigning commercials to station breaks in television programming, or allocating memory in computers [6].

Nowadays the term bin packing encapsulates many derived specific problems which can be classified in many different ways. The simplest categorizing of the bin packing problem is according to the spatial dimension – from one-dimensional to three-dimensional. They can moreover contain any other constraints (item rotation, forbidden positions etc.) and weighting (corresponds to Knapsack problem). This work takes into consideration only spatial and item rotation constraints. In the rest of the text I abbreviate the different types of the bin packing problem as dDBPP where d represents the spatial dimension (for instance 1DBPP).

Not many approaches that deal with pure Bin Packing Problem using multi-agent principles exist. The research is more shifted to more specific variants as in [1] which combine container loading algorithms with agent-based simulation to optimize various aspects in relation to the cargo - such as stability and fragility - or vehicle routing problem (VRP) in [18] or [22].

1.1.1 Upper and lower bounds

Research of the bin packing problem includes examination of general guarantees and bounds on best/worst case behavior which can be expected and can also be used in algorithms.

One of these bounds is the lower bound[1] of the optimal solution – it tries to estimate the optimal solution when it is not known, or can't be proved. It can be then used as a starting point of optimization algorithms. A lower bound function for bin packing takes a problem instance and efficiently computes a lower bound on the minimum number of bins needed. If we find a solution that uses the same number of bins as the lower bound, then we know that the solution is optimal, and we can terminate the search [8]. The computation of lower bounds can be very simple (ratio of sum of items' volumes and the container's volume) or more complex. We want to have as good estimation as possible because the better starting value we have the better computation time we can achieve.

Worst case performance is a proof of how good the algorithm is in a worst case. It is usually compared with optimal value of the instance - OPT(I). It is good for comparing different approaches. Although these values are usually proved only for simple algorithms and problems, it can be important for us anyway, because the approaches (heuristics) can be adopted or combined in various ways to solve more sophisticated problems and the complexity is multiplied in those situations.

1.1.2 One-dimensional bin packing problem

In the classical version of the 1DBPP one is given a list $L = (a_1, \ldots, a_n)$ of items (or elements) and an infinite supply of bins with capacity C. A function $s(a_i)$ gives the size of item a_i, and satisfies $0 < s(a_i) \leq C$, $1 \leq i \leq n$. The problem is to pack the items into a minimum number of bins under the constraint that the sum of the sizes of the items in each bin is no greater than C. In simpler terms, a set of numbers is to be partitioned into a minimum number of blocks subject to a sum constraint common to each block [6].

The 1DBPP algorithms can be divided into on-line and off-line. In the case of on-line algorithms, items are packed in the order they are encountered while going through the given list L (the list containing all items). The bin in which an item is packed is chosen without knowledge of other items not yet encountered in L. These algorithms are the only ones that can be used in certain situations, where the items to be packed arrive in sequence and have to be assigned to a bin as soon as they arrive. Off-line algorithms, on the other hand, have complete information about the entire

[1]Lower bound in terms of minimization, upper bound as to maximization.

list throughout the packing process [6]. In my work I use both principles, but I don't explicitly distinguish them as I don't find it important in the context of the assignment and in general the principles are related as the off-line algorithms include the on-line part, only sort the list L of items in desired order in the preprocessing phase.

The most famous simple heuristic algorithms are described below.

- **Next-Fit** (NF) – The item is packed next to the previous item. If it does not fit to the bin, new one is open and the item is packed there.
- **First-Fit** (FF) – The item is packed to the first open bin it fits into. If there is no such bin, new one is open and the item is packed there.
- **Best-Fit** (BF) – The item is packed into the open bin with the largest content it fits into. If there is no such bin, new one is open and the item is packed there.
- **First-Fit-Decreasing** (FFD) – The off-line version of FF, where the items are firstly sort in non-increasing order.
- **Best-Fit-Decreasing** (BFD) – The off-line version of BF, where the items are firstly sort in non-increasing order. On average, BFD performs slightly better than FFD.

Table 1.1 contains comparison of the above mentioned algorithms as to complexity and worst case performance ratio (APR).

Heuristic algorithm	Time complexity	Worst case performance ratio (APR)
Next-Fit (NF)	$O(n)$	$2 \cdot OPT(I)$
First-Fit (FF)	$O(n \cdot \log(n))$	$\frac{17}{10} \cdot OPT(I)$
Best-Fit (BF)	$O(n \cdot \log(n))$	$\frac{17}{10} \cdot OPT(I)$
First-Fit-Decreasing (FFD)	$O(n \cdot \log(n))$	$\frac{11}{9} \cdot OPT(I)$
Best-Fit-Decreasing (BFD)	$O(n \cdot \log(n))$	$\frac{11}{9} \cdot OPT(I)$

Table 1.1: 1DBPP algorithms comparison.

The heuristics were improved many times (according to [6] Refined First-Fit Decreasing, Modified First-Fit Decreasing, Best Two-Fit etc.), but they are usually very complicated, have worse time complexity and the benefit as better APR is not that significant. Other approaches are Martello and Toth algorithm (see [11] – algorithm for finding optimal solution), better-fit heuristic (see [3]), randomized and genetic algorithms. The simplest case of 1DBPP without any other constraints is not commonly extensively studied nowadays as the old algorithms are sufficient for real applications.

1.1.3 Two-dimensional bin packing problem

The 2DBPP seeks to pack a set R of n rectangles with dimensions $w_i \cdot h_i$ into identical larger rectangular bins with dimensions $W \cdot H$ using the fewest bins possible [17]. The problem is the two-dimensional extension of the classic (one-dimensional) Bin Packing Problem and is one of the most studied problem in the so called Cutting & Packing category [9].

According to [9] there are 4 classes of the problem which originated from two of the most common requirements:

- the orientation of the items and
- the guillotine cutting (items must be obtained through a sequence of edge-to-edge cuts parallel to the edges of the bin – see Figure 1.1).

The classes are:

- **2BP/O/G**: the items are oriented (O), and guillotine cutting (G) is required;
- **2BP/R/G**: the items may be rotated by 90′ (R) and guillotine cutting is required;
- **2BP/O/F**: the items are oriented and cutting is free (F);
- **2BP/R/F**: the items may be rotated by 90′ and cutting is free.

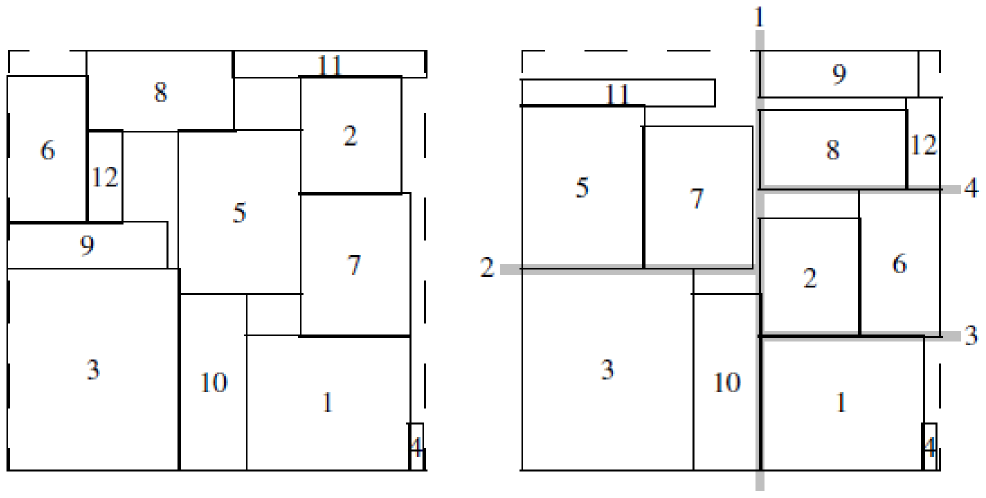

Figure 1.1: A Non-Guillotine Pattern. Right: A Guillotine Pattern ([17]).

Most of greedy algorithms can be classified in two families: One phase algorithms which directly pack the items into the finite bins and two-phase algorithms that start by packing the items into a strip (bin having width W and infinite height). In the second phase, the strip solution is used to construct a packing into finite bins.

Basic approach to dealing with bin/strip packing is to pack items into so-called shelves. It means that the placed items form levels which define horizontal constraints and are put on each other. Three classical strategies for the shelf packing have been derived from famous algorithms for the one-dimensional case (see Subsection 1.1.2). In each case, the items are initially sorted by non-increasing height and packed in the corresponding sequence. Let j denote the current item, and s the last created shelf [9]:

- **Next-Fit Decreasing Height** (NFDH) strategy: item j is packed left justified on shelf s, if it fits. Otherwise, a new shelf is created, and j is packed left justified into it;
- **First-Fit Decreasing Height** (FFDH) strategy: item j is packed left justified on the first shelf where it fits, if any. If no shelf can accommodate j, a new shelf is initialized as in NFDH;
- **Best-Fit Decreasing Height** (BFDH) strategy: item j is packed left justified on that shelf, among those where it fits, for which the unused horizontal space is a minimum. If no shelf can accommodate j, a new shelf is initialized as in NFDH.

The strategies are illustrated on the Figure 1.2.

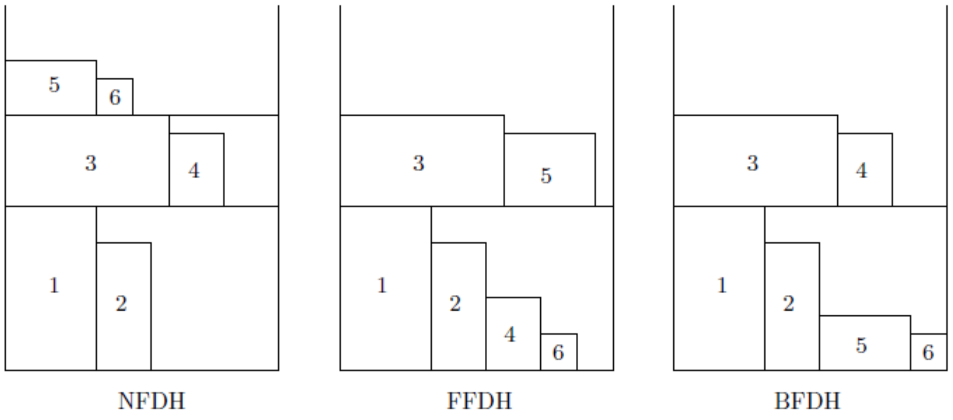

Figure 1.2: Shelf packing strategies [9].

The Table 1.2 contains comparison of the above mentioned algorithms as to complexity and worst case performance ratio (APR) in terms of strip packing (OPT represents the minimal possible height of the strip). A reader can compare the table with the Table 1.1 to notice a similarity with 1DBPP.

More sophisticated approaches of constructing shelves are:

- FC (Floor Ceiling) - items can be put also on the ceiling of a shelf. Refer to Figure 1.3.

Heuristic algorithm	Time complexity	Worst case performance ratio (APR)
NFDH	$O(n \cdot \log(n))$	$2 \cdot OPT(I) + 1$
FFDH	$O(n \cdot \log(n))$	$\frac{17}{10} \cdot OPT(I) + 1$
BFDH	$O(n \cdot \log(n))$	$\frac{17}{10} \cdot OPT(I) + 1$

Table 1.2: Strip packing algorithms.

- KP (Knapsack Packing) – contains knapsack sub-problem where each item has its profit defined as its area ($w_i \cdot h_i$). The tallest item is chosen and creates a shelf – the goal of the sub-problem is to add other items to the shelf according to the knapsack problem optimization. As knapsack problem also belongs to NP-hard problems, we get outside the polynomial time complexity, however the sub-problem is usually so small that it is solved very quickly with an optimal solution.

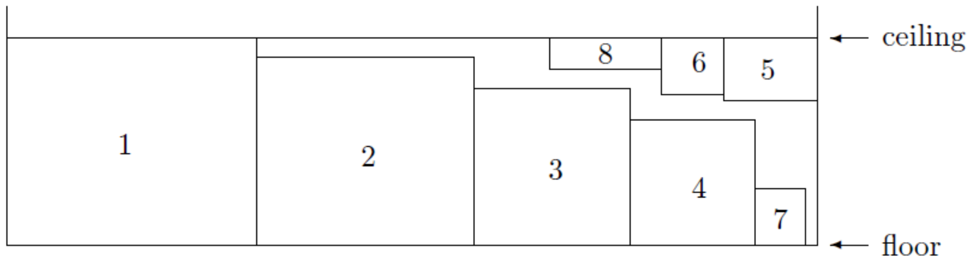

Figure 1.3: Floor ceiling algorithm [9].

When the shelves are prepared, next step is to arrange them into bins. At the first sight it looks like 1DBPP and actually it is – we can use any exact or heuristic algorithm to solve the 1DBPP.

The simplest two-phase algorithms use a heuristic 1DBPP strategy to fulfill bins with shelves i.e. one-dimensional items of certain height. Two most famous two-phase algorithms are listed below.

- **Hybrid First-Fit** (HFF) – Introduced by [5]. Uses FFDH for shelves construction and FFD (first fit decreasing) as 1DBPP approach. The APR is $\frac{17}{8} \cdot OPT(I) + 5$.
- **Finite Best-Strip** (FBS) – Introduced by [2]. Uses BFDH for shelves construction and BFD (best fit decreasing) as 1DBPP approach.

One of features of shelf-based algorithms is the guillotine *cuttability* and it can solve problem in 2BP/*/G (except of shelves constructed by FC if it is not explicitly ensured).

One-phase algorithms construct bins directly. They can be shelf-based – the items are placed in shelves, but contrary to two-phase algorithms, the shelves are being created continuously inside the bins. However one-phase algorithms are generally worse than two-phase algorithms. The second type are non-shelf algorithms which place items to the whole space of bins – one of them is Alternate Directions (AD) – for basic understanding see Figure 1.4.

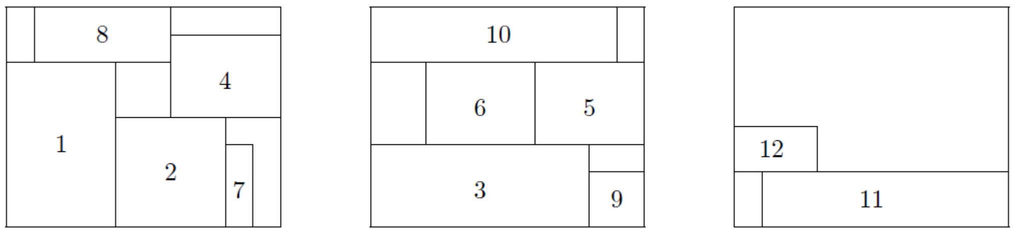

Figure 1.4: Alternate Direction algorithm [9].

Experiments on benchmark data (for more details look at [9]) shows that KP, FC and AD provide similar results' quality.

1.1.4 Three-dimensional bin packing problem

The 3DBPP packs a set R of n rectangles with dimensions $w_i \cdot h_i \cdot d_i$ into identical larger rectangular bins with dimensions $W \cdot H \cdot D$ using the fewest bins possible.

As it is not common to take care about guillotine constraint in as in 2DBPP (it has not such practical usage) only orientation constraint is taken into account when distinguishing 3DBPP classes – they are 3DBPP/O and 3BP/F. In benchmarks, however, the 3DBPP/O version is preferred.

Heuristics for this problem are usually derived from those for 2DBPP but are not very successful. An exact algorithm for 3DBPP that uses shelves and 1DBPP subproblem solving is introduced in [10]. It is composed from two-levels - in the first level the items are assigned to bins without specifying their actual position, while a specialized algorithm is used to test whether a subset of items can be placed inside a single bin and to determine the placing when the answer is positive.

The most popular variant of the problem which has the most useful application is *container loading* which is discussed in the next Subsection 1.1.5.

1.1.5 Container loading

Container Loading Problem (CLP) is a special case of 3DBPP where the goal is not to pack all items into the smallest number of containers, but we take into consideration only one container (typically with dimension reflecting industrial standards) and we want to maximize its used volume. This problem is very practically oriented and tries to satisfy real world constraints which logistic companies face. Much of the recent work has moved away from pure knapsack or bin-packing formulations of the container loading problem and has paid increasing attention to various additional factors which may affect the task in practice. Orientation constraints on individual types of cargo and container weight capacity limits represent simple examples of such factors. Other problem definitions include the weight distribution within a container as a critical factor and forms aspects of cargo stability which have been explicitly considered in several approaches as attributes of solution quality [1].

The existing 3D-CLP methods are based on different heuristic packing approaches that determine the structure of generated packing plans (cf. [16]):

1. Wall building approach The container is filled by vertical cuboid layers ("walls") that mostly follow along the longest side of the container.
2. Stack building approach The boxes are packed in a suitable manner in stacks, which are themselves arranged on the floor of the container in a way that saves the most space. Characteristic of this approach is that the stacks do not themselves form walls as defined before.
3. Horizontal layer building approach The container is filled from bottom to top through horizontal layers that are each intended to cover the largest possible part of the (flat) load surface underneath.
4. Block building approach The container is filled with cuboid blocks that mostly contain only boxes of a single type with the same spatial orientation. The approach is related to approach (3), but the main motive here is to fill the largest possible sub-spaces of the container without internal losses.
5. Guillotine cutting approach This approach is based on a slicing tree representation of a packing plan. Each slicing tree corresponds to a successive segmentation of the container into smaller pieces by means of guillotine cuts, whereby the leaves correspond to the boxes to be packed.

One of the features of the container loading is that it usually tries to deal with same type of boxes. Naturally it is more convenient to place such boxes beside each other as they don't waste the space – critical condition while maximizing the utilized space. Forming columns or layers (further referenced as block of boxes - only one box can also form a block) from those boxes is a typical task of CLP. See Figure 1.5.

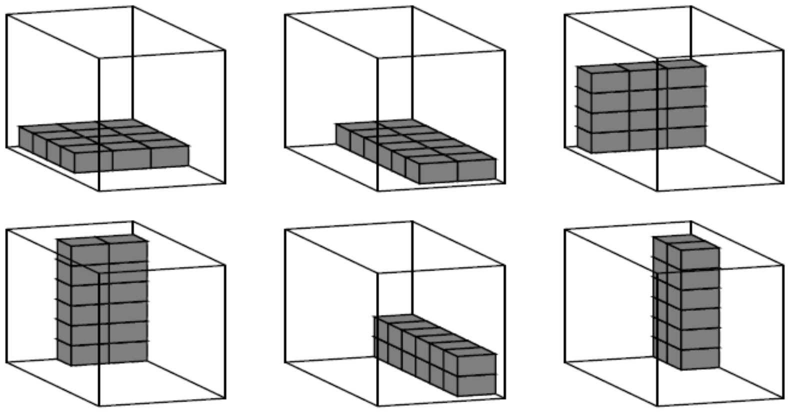

Figure 1.5: Different blocks of boxes (layers) in CLP. What is missing - only one box can also form a block.

Important part of CLP algorithms is how to deal with unused space – let call it free space. Classical approach was to split the free space (while new item is being placed) to disjoint parallelepiped spaces (for instance in [13, 20]). [14] introduced concept of maximal spaces where the empty remaining space is split to parallelepiped non-disjoint spaces. This approach seems to be quite successful and produces better solutions. You can see maximal spaces on Figure 1.6.

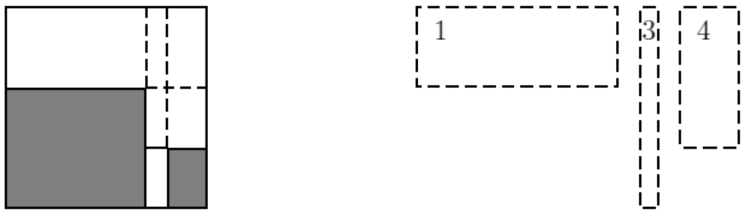

Figure 1.6: Maximal free spaces [14].

Based on layers and maximal spaces a constructive algorithm was proposed in [14]. The algorithm finds a good admissible solution which can be then improved by various techniques – GRASP in [13] or VNS (Variable Neighborhood Search) in [15]. The constructive algorithm's skeleton is captured in Algorithm 1.

The skeleton contains some points - for instance choosing the free maximal space, the block of boxes – which can be implemented variously according to different heuristics. It has an important impact on algorithm's behavior, results and efficiency. Different approaches are discussed in Section 2.5 which describes the final implementation. Many of the approaches are adopted from [14].

CLP differs from 3DBPP in items' rotation possibilities. While in 3DBPP/F items

```
Input  : Container dimension $C$, set of boxes $\{b_1, ..., b_m\}$, numbers of boxes $N_i$.
Output: Boxes' positions.
function solve ($C, \{b_1, ..., b_m\}, N_i$) begin
    $S \leftarrow \{C\}$ ;                         // The set of free maximal spaces
    $B \leftarrow \{b_1, ..., b_m\}$ ;             // The set of boxes still to be packed
    $Q_i \leftarrow N_i$ ;                         // Number of boxes of type $i$ to be packed
    while any box is packed do
        $chosenSpace \leftarrow$ chooseFreeSpace ($S$);
        $chosenBlockOfBoxes \leftarrow$ chooseBlockOfBoxes ($chosenSpace, B, Q_i$);
        putBoxesToSpace ($chosenSpace, chosenBlockOfBoxes$);
        update free spaces;
    end
end
```

Algorithm 1: CLP constructive algorithm.

can be rotated in all possible directions and 3DBPP/O can't be rotated anyway CLP reflects logistic demands - items may be restricted to be rotated in vertical direction, but they can be freely rotated horizontally. Benchmark data for CLP contains these constraints.

1.2 Multi-agent solver

In the work, paradigms of multi-agent planning and multi-agent solver are supposed to be used. In this section principles of a solver introduced in [19] are described.

First, we define the multi-agent problem from [19] as:

"Task decomposition and allocation to the number of autonomous agents, where the allocation is based on individual agents commitments to the joint solution using private constraints and motivation."

In simplified words it means that a complex problem is divided into smaller subproblems, these are solved by individual computational units – agents and the results are then composed into the final solution. The agents use communication (more precisely task delegation or negotiation) to improve their partial solutions (plans).

The main objective function of multi-agent problem is defined as maximization of agents' social welfare:

$$s_w = \sum_{a \in A} u_a \tag{1.2.1}$$

where $A = a_1, \ldots, a_n$ is a population of agents and u_a is the utility of agent a. After deriving the final objective function of the multi-agent solver is [19]

$$\sum_{t \in T} cost(t, a) \tag{1.2.2}$$

where $cost(t, a)$ is the cost of the agent a to perform the task t.

According to [19] the abstract multi-agent solver architecture is defined as a composition of three types of agents (followed by Figure 1.7)

- **Task Agent** for pre-processing of the problem. This agent should use a domain specific heuristic, generic ordering strategy or randomized ordering method.
- **Allocation Agent** for problem decomposition into tasks and delegation of the tasks to Resource Agents. This agent maintains task allocation and result synthesis. This agent's strategies and algorithms are domain-independent.
- **Resource Agent** for individual case-specific resource planning. In case of further decomposition, the task is handed over to another Task Agent.

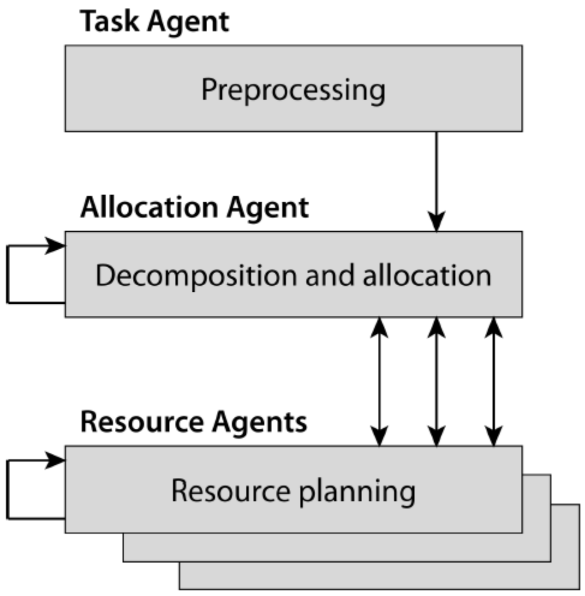

Figure 1.7: Multi-agent solver architecture [19].

Multi-agent solver consists of a number of interacting resource agents – each having an ability to maintain his individual plan of assigned tasks. These agents are coordinated by allocation agent. The whole plan of the problem is then a fusion of such partial plans. In the domains where the optimization/planning problem can be decomposed into independent task the multi-agent approach shows its benefits. Such a task can be allocated and executed by different agents with low or no influence on each other [19].

The architecture and the representation of the multi-agent problem, which is handled by a set of resource agents in the lowest level, naturally allow us to use parallelization (i.e. distributed system) to solve such problems.

1.2.1 Abstract Algorithm

The abstract algorithm representing the presented multi-agent solver minimizing objective function defined by Equation 1.2.2 is captured by Algorithm 2. According to the abstract architecture denoted in Figure 1.7 it contains three phases [19]:

- The first phase of the function solve is task **pre-processing** provided by the Task Agent. The ordering heuristic represents case-specific sorting of the tasks to increase the solver's efficiency in the particular domain. In some cases the ordering has no influence, but in others it may provide significant improvement especially in domains with stronger task dependencies.
- The second phase is iteration over all tasks and **allocation** (*allocateAll*) performed by the Allocation Agent minimizing the insertion cost computed by Resource Agents. As a part of this iteration, the **dynamic improvement** (*delegateAll*) based on cooperation of Allocation Agent and all Resource Agents may take place – the improvement strategy is applied to every Resource Agent after allocation of each task (see below for the description of improvement strategies).
- The third phase of the *solve* function is the **final improvement** (*negotiateAll*) of the solution. After allocation of all tasks the improvement strategy is executed by all Resource Agents.

The algorithm is based on local optimization of a single task insertion and subsequent improvement. Each iteration of the algorithm provides a greedy (order-dependent) task allocation supported by locally-optimized solution of resources utilization (which can be seen from global point of view as hill-climbing search). The algorithm does not use any backtracking mechanism or exhaustive search of the state space. It has a significant impact on the algorithm's computational complexity but it is susceptible to finding locally efficient solution only. The global solution quality is improved by execution of incremental version of improvement strategies - the improvement runs as long as the solution is being improved [19]. If the solution can not be found for all tasks, the number of agents is increased and the process is repeated[2].

The main methods of the second and third phases are:

- ***Allocate*** – the allocation agent allocates an unallocated task $t \in T$ to a resource agent (tries to choose the best agent according to a predefined heuristic) – minimizing $cost(t, a)$.
- ***Delegate*** - it encapsulates the process in which an agent a_i delegates task t to an agent a_j. It should satisfy the improvement condition:

  $$cost^{remove}(t, a_i) - cost^{insert}(t, a_j) > 0 \tag{1.2.3}$$

  .

  It appeared to be an effective optimization tool in terms of dynamic improvement.

  The *delegate* method as well as its concrete implementations is clearly defined in [19]. The Delegate All strategy of the *delegate* method (refer to [19]) is included in the final implementation.
- ***Negotiate*** – the *negotiate* method is used in final improvement. It can simply call the *delegate* method, but it is better to define the final improvement method in a different way. In Bin Packing it is more clear, because if you examine the above mentioned *delegate* method, you will find out that it is not suitable as the final improvement because few free space remains in the fulfilled bins after the *phase 2*.

  For the bin packing problem *swap* method which deals with swapping of tasks between two agents was introduced. It is described in the chapter of implementation.

  The method contains also remaining tasks as a parameter and the agents can use it in the negotiation process.

Except the *delegation* and *swapping*, other improvement strategies are implemented and will be discussed in this work. In the further text I use the term *delegation* while talking about dynamic improvement and *negotiation* to encapsulate final improvement methods.

In [19] the resource agents' insertion and removal estimations of single task are introduced as a kind of a resource planning heuristics. The functions are:

[2]There is usually implemented a cache mechanism that helps to reuse the partial plans of the previous set of agents which can save the computation time.

```
Input  : Set of tasks $T$, set of Resource Agents $R$
Output: $T$ allocated on $R$ and local plans on Resource Agents exist
function solve ($T, R$) begin
    apply ordering heuristic on $T$;                                  // Phase 1
    allocateAll ($T, R$);                                             // Phase 2
    while allocation of any $t : T$ successful do                     // remaining tasks
        negotiateAll ($R, T$);                                        // Phase 3
        allocate ($t, R$);
    end
end
function allocateAll ($t, R$) begin
    forall the $t : T$ do
        allocate ($t, R$);
        if allocation successful then
            remove $t$ from $T$ else mark $t$ as not allocated and continue;
        end
        delegateAll ($R$);                      // Phase 2 - dynamic improvement
    end
end
function allocate ($t, R$) begin
    forall the $a : R$ do
        find winner with the lowest $cost^{estI}(t, a)$;
    end
    if winner is found then
        assign $t$ to the winner;
    end
end
function delegateAll ($R$) begin
    while delegated do
        forall the $a : R$ do delegate ($a, R$)
    end
end
function negotiateAll ($R, T$) begin
    while negotiated do
        forall the $a : R$ do negotiate ($a, R, T$)
    end
end
```

Algorithm 2: The abstract algorithm of the multi-agent solver.

- **Insertion estimation** $cost^{estI}(t,a)$ - the estimation of the cost of the task insertion. It represents the increase of the agent's cost function caused by undertaking the task t.
- **Insertion** $cost^{insert}(t,a)$ - the real cost of the task insertion. This value is determined by adding a new task t to the plan of the agent a in the current state. It is the result of the particular resource planning algorithm of the resource agent.

The allocation is based on the determination of the *winner* agent – the resource agent with the lowest insertion estimation cost of the task t:

$$winner = \underset{a \in A}{argmin}\, cost^{estI}(t,a) \tag{1.2.4}$$

The opposite functions used by improvement strategies are:

- **Removal estimation** $cost^{estR}(t,a)$ - the estimation of the cost of the task removal. It represents the decrease of the agent's cost function caused by dropping the task t.
- **Removal** $cost^{remove}(t,a)$ - the real cost of the task removal. This value is determined by removing the task t from the plan of agent a in the current state. It is the result of the particular resource planning algorithm of the resource agent.

Later it turned out that the removal cost and negotiation based on difference between insertion and removal cost is not completely suitable for for all of the presented multi-agent implementations of the bin packing problem. So it can be omitted in some cases and the introduced model is used on a more abstract level.

1.2.2 Complexity analysis

The worst complexity of the abstract algorithm is upper-bounded by

$$\mathcal{O}(n \cdot \log(n) + n \cdot (O^{alloc} + m \cdot O^{delegate}) + m \cdot O^{negotiate}) \tag{1.2.5}$$

where n denotes the number of tasks and m is the number of resource agents. Each addend represents a phase of the algorithm's basic model. The $n \cdot \log(n)$ part represents the ordering heuristic and corresponds to the complexity of standard sorting algorithms. The $n \cdot \mathcal{O}^{alloc}$ part is the complexity of the allocation with the dynamic improvement and the last part is about the final improvement phase. For simplification of the complexity analysis we assume only one iteration of *while* loop while performing improvement methods.

The improvement part can be further decomposed as

$$O^{delegate} = m \cdot fi^{delegate}(\frac{n}{m}) \tag{1.2.6}$$

$$O^{negotiate} = m \cdot fi^{negotiate}(\frac{n}{m}) \tag{1.2.7}$$

where $\frac{n}{m}$ is the average number of tasks allocated to a particular resource agent and $fi(\frac{n}{m})$ is the factor representing the complexity of the implemented agents‘ improvement strategy. It describes more precisely the fact that each agent tries to negotiate (e.g. delegate all his tasks) with all other m agents to improve the overall solution.

After that slight concretization of the improvement strategy we get the final complexity:

$$\mathcal{O}(n \cdot O^{alloc} + n \cdot m^2 \cdot fi^{delegate}(\frac{n}{m}) + m^2 \cdot fi^{negotiate}(\frac{n}{m})) \tag{1.2.8}$$

If we don't include the dynamic improvement we get

$$\mathcal{O}(n \cdot O^{alloc} + m^2 \cdot fi^{negotiate}(\frac{n}{m})) \tag{1.2.9}$$

It could be decomposed even more while taking into account principles described in [19], but in relation to the rest of this work, the Expression 1.2.8 is the least admissibly general representation of the complexity and concrete adjustments are discussed in related sections. In general, the *allocation*, *delegation* and *negotiation* time complexities can vary depending on an implementation of the abstract model. However, the solver should solve problems in a polynomial time, hence ensuring the polynomial complexity of each of the parts is crucial.

2 Implementation

This chapter contains all details about the multi-agent solver for bin packing problem. The ideas are presented more or less chronologically. It means that the order of included sections and subsections reflects how the work evolved over time. All of the BPP spatial variants are adapted to the multi-agent solver introduced in Section 1.2 starting with the 1DBPP over shelf-based 2D/3D packing to the universal multi-dimensional packing based on container loading volume optimization described in Subsection 1.1.5. As stated in Section 1.1 no other constraints except of the spatial and rotation restrictions were taken into account in terms of Bin Packing Problem.

In the sections some results are referenced for a discussion over implementation details. However all gained results as well as benchmark data descriptions comprise the Chapter 3 and Appendix A.

The key task of the work was to adapt the bin packing problem to multi-agent paradigms. There are two suitable approaches to this problem. The first approach is to treat items (boxes) as agents and the negotiation would be based on finding the best place in any of bins for the agents. This approach can be seen for instance in [1]. However more natural seems to be an implementation of bins as agents. Moreover it fits better to the multi-agent solver abstract model described in Section 1.2 because the tasks' description and the *allocation* and *negotiation* principles can be handled better by assigning them to items than in case of placing agents into a space as in the first approach.

Main difficulties occur when evaluating the quality of task allocation – as introduced in [19] – an agent tries to delegate its task to other agent where it would have a better value. But it is not easy to determine such value when placing items in multi-dimensional space. That is the reason why the abstract model of multi-agent solver from [19] is kept more general and *estimateRemove* value is left out in Algorithm 2. However it is used in some of the others introduced implementations based on one-dimensional space optimization.

Another limitation of the original solver was its *delegate* method. The method is together with dynamic improvement a part of final improving strategies and its goal is to delegate a task (or all tasks) from an agent i to an agent j in such way that global solution is closer to optimum. The problem is that it is not very efficient in final improvement as all bins' capacities are nearly full and not much free space exists to delegate any task to somewhere else. Furthermore when we use BFD for the allocation – after non-increasing ordering as preprocessing – there is nearly no such optimization possible, because large items are allocated first. It led me to separate dynamic and final improvement strategy which can now represent different functionality. In ILP based 1DBPP solver the *delegate* is omitted and the *negotiate* is redefined into ILP optimization, in the heuristic 1DBPP solver and in shelves creation sub-solver (see Section 2.3) a more suitable *swap* method which tries to swap the items between agents is introduced and in universal multi-dimensional solver implementation (Section 2.5) the *negotiate* simply reallocates all items of chosen agents together with remaining items trying to find better volume ratio of the agents.

2.1 Preprocessing

In the preprocessing phase, the tasks can be sorted according to a heuristic which can improve the overall solving process. In [18] it is stated that ordering of the tasks doesn't have to have significant importance - as for VRP (Vehicle Routing Problem). However for BPP it has. From basic observations, the average computation time is lower when non-increasing strategy is used because less delegations and negotiations are needed. In all other solver implementations non-increasing ordering strategy - according to parameter which is optimized - is performed in preprocessing phase and it is no more discussed.

2.2 Multi-agent solver for 1D Bin Packing Problem

The one-dimensional bin packing problem version is the simplest case of the bin packing. The bins and the items have only one dimension and there is no need to take care about a position in a bin where to put the item – one is straightforwardly placed beside other and when an item is removed, the rest of the load can be imaginarily pushed to one side to make the free space maximal. This fact is very important because it causes the optimization space to be much smaller than in a more dimensional setting. However, 1DBPP is still NP-hard. All phases that represents the abstract Algorithm 2 as well as their concrete implementations are discussed below.

2.2.1 Allocation

After the preprocessing we get the sorted list L of unallocated items and we first try to allocate them among all resource agents. We can adapt the problem to the multi-agent solver from Section 1.2 according to Equation 1.2.4 after defining the insertion estimation $cost^{estI}(t,a)$ function. This function reflects how convenient is to place the task t inside the bin represented by the agent a. If one looks at the 1DBPP heuristics described in Subsection 1.1.2 she can find out that the winner agent function can handle the Best-Fit strategy - the best agent for the task t is the one that has the largest content and the item fits into it. Therefore we can define the insertion estimation cost as the amount of free space after adding the item (i.e. the smaller the better). If the item does not fit in the bin, the function returns infinite cost:

$$cost^{estI}(t,a) = \begin{cases} C-(h(t)+uc), & h(t)+uc \leq C \\ \infty & h(t)+uc > C \end{cases} \tag{2.2.1}$$

where C refers to the bin's capacity, $h(t)$ returns a size[1] of the item represented by the task t and uc is used capacity - already allocated space.

The solver's abstract model also contains tasks pre-processing which sorts the tasks before the allocation and delegation phases. If we sort the items in a non-increasing order a Best-Fit Decreasing (BFD) comes out. Hence, after the allocation we have the solution's quality of $\frac{11}{9} \cdot OPT(I)$ as stated in Table 1.1.

The time complexity of the allocation phase is $n \cdot m$ which is generally worse than the BFD optimal implementation $(n \cdot \log n)$. In general we don't need to improve it. First, we suppose quite small number of agents anyway as discussed in Section 1.2, hence the complexity is sufficient. Second the following improvement strategies are supposed to process in worse time complexity - in the context the quadratic behavior in number of tasks can be treated as a lower bound of the overall solver's complexity.

2.2.2 Delegation

The dynamic improvement phase (called *delegation* here) is handled by Delegate All strategy as described in the *original* multi-agent solver in [19]. It tries to follow the optimization condition written in Expression 1.2.3.

To suit the $cost^{estI}$ function to perform BFD allocation strategy as well as ensuring the optimization during improvement phase, the estimation functions are defined as:

[1]It is labeled $h(t)$ meaning *height* to be compatible with following multi-dimensional versions.

$$cost^{estI}(t,a) = \begin{cases} (C-(h(t)+uc))\cdot h(t), & h(t)+uc \leq C \\ \infty & h(t)+uc > C \end{cases} \tag{2.2.2}$$

$$cost^{estR}(t,a) = (C-uc)\cdot h(t) \tag{2.2.3}$$

This ensures that the items will be placed to the fullest bins in the allocation phase and that it will optimize the solution according to the improvement conditions stated in Expression 1.2.3 and Expression 2.2.4 in the improvement phase.

2.2.3 Negotiation

After the allocation phase the final improvement phase comes to optimize the agents' load and try to allocate the rest of unallocated items into the predefined number of bins. If it can't be done, the number of agents (bins) must be increased. The beginning of this chapter contains a discussion that delegation as the final improvement is not fully efficient in bin packing because of insufficient free space in other bins after allocation. Swapping of the items between the agents appears more suitable for this task.

Swap method iterates through each agent and each of its tasks and tries to find other task owned by a different agent to trade it. Every agent only tries to maximize its own load – it means that it exchanges smaller items for larger. The improvement condition is:

$$cost^{estR}(t,a) - cost^{estI}(t_{other},a) > 0 \;\wedge\; cost^{estI}(t,a_{other}) \neq \infty \tag{2.2.4}$$

t_{other}represents the task from the agent a_{other} who negotiate with the agent a.

2.2.4 Algorithm

The template representing the abstract algorithm of the multi-agent solver (see Algorithm 2) is used for its specific implementation. In this case, we only implement the *delegate* and *negotiate* methods. The implementation of these methods is contained in Algorithm 3.

The *delegate* method corresponds to the Delegate All strategy from [19] - every agent tries to all reallocate all his tasks to other agents where they fit best - the *winner* is found according to Equation 1.2.4 and Equation 2.2.2. In final improvement the *swap* method is called where the agent tries to exchange its tasks for other tasks

```
function allocate ($t, R$) begin
    forall the $a : R$ do
        find winner with the lowest $cost^{estI}(t, a)$;
    end
    if winner is found then
        assign $t$ to the winner;
    end
end

function delegate ($a, R$) begin
    forall the $t$ : tasks of agent $a$ do
        forall the $a_{other} : R \setminus a$ do
            find winner with the lowest $cost^{estI}(t, a_{other})$;
        end
    end
    if winner is found then
        assign $t$ to the winner;
    end
end

function negotiate ($a, R, T$) begin
    forall the $t$ : tasks of agent $a$ do
        swap ($t, R$) ;
    end
end

function swap ($t, R$) begin
    forall the $a_{other} : R \setminus a$ do
        forall the $t_{other}$ : tasks of agent $a_{other}$ do
            find best task $t_{other}$ and $a_{other}$ to swap $t \leftrightarrow t_{other}$;
        end
    end
    if $t_{other}$ found then
        remove $t$ from $a$ and assign $t_{other}$ to $a$;
        remove $t_{other}$ from $a_{other}$ and assign $t$ to $a_{other}$;
    end
end
```

Algorithm 3: Basic implementation of Algorithm 2.

such that it optimizes its load according to the improvement condition captured by Expression 2.2.4. Equations 2.2.2 and 2.2.3 are used inside.

The time complexity of the algorithm comes from Expression 1.2.8. We can further decompose

$$O^{alloc} = m \tag{2.2.5}$$

because the costs estimation computation is trivial. While calculating $fi(\frac{n}{m})$ we can get to two possible branches - *delegation* or *negotiation* (*swapping*). We get

$$fi^{delegate}(\frac{n}{m}) = \frac{n}{m} \cdot m = n \tag{2.2.6}$$

$$fi^{negotiate}(\frac{n}{m}) = \frac{n}{m} \cdot m \cdot \frac{n}{m} = \frac{n^2}{m} \tag{2.2.7}$$

After putting all together we get the final Expression 2.2.8 of the upper bound of time complexity (we assume $n > m$):

$$\mathcal{O}(n \cdot m + m^2 \cdot n + m \cdot n^2) = \mathcal{O}(m^2 \cdot n + m \cdot n^2) = \mathcal{O}(m \cdot n^2) \tag{2.2.8}$$

2.2.5 ILP based negotiation

This implementation was designed to find better solutions - ideally the optimal solutions. The basic idea is that we optimize many small sub-problems using ILP and the overall solution comprised from those optimal plans can be optimal as well. It can be easily adapted on the negotiation model - agents negotiate with each other to optimally reload their content to enable not yet allocated items to be placed inside one of them. The implementation of Algorithm 2 is captured in Algorithm 4.

```
function delegateAll ($R$) begin
    // leave it blank
end

function negotiate ($a, R, T$) begin
    forall the $a_{other}$ : $R \setminus a$ do
        optimizeILP ($a, a_{other}$);
    end
end
```

Algorithm 4: Implementation of abstract algorithm for ILP based negotiation.

The *optimizeILP* method gets a set of bins (agents) as input and rearranges their load such that free space of one of them (lets say the first in the set) is maximized - ideally the free space of one of the agents after the optimization will be

$$s_{a_1} = \min\Big(\sum_{a \in A'} s_a, C_{a_1}\Big) \tag{2.2.9}$$

i.e. overall free space of the set of agents will move to the first agent (s_{a_1}). It enables to allocate other not allocated items in that bin. The optimization sub-problem is solved by an ILP solver and it is defined in Expression 2.2.10:

$$\begin{aligned} \min \sum_{t \in T'} allocated_{t,a_1} \cdot h(t) & \qquad such\ that \\ \sum_{t \in T'} allocated_{t,a} \cdot h(t) & \leq C_a,\ \forall a \in A' \\ \sum_{a \in A'} allocated_{t,a} & = 1,\ \forall t \in T' \\ allocated_{t,a} & \in \{0,1\},\ \forall a \in A',\ \forall t \in T' \end{aligned} \tag{2.2.10}$$

T' is a set of all items got by unloading of all agents from the input set, $allocated_{t,a}$ is a boolean variable that represents that the task t is allocated in the agent a, $h(t)$ returns a size of the item t and C_a is the capacity of the agent a.

The whole negotiation part optimizes extensively and it is even able to iteratively totally reorganize the already allocated items. Therefore dynamic improvement appeared to be excessive and so the *delegateAll* method is left blank.

We must be aware that the ILP optimization part is NP-hard and so we get outside the polynomial time. However the sub-problems are so small that the optimal partial solutions are found very quickly.

2.2.6 Fitness based selection of agents

In the previous subsection an approach which optimizes the global solution by rearranging items between two agents was introduced. Basic case is to simply iterate through all agents and run the optimization for every couple. However, the ILP optimization (including problem definition creation) is not the cheapest in terms of computation time and for many couples small or even no improvement is gained. It is because one of them may be and usually is (nearly) full and the time is wasted in that way. More convenient is to form these couples only from agents that are less full. I took an inspiration from genetic algorithms and their selection principles. We can imagine that we have a population of agents and in every iteration we select two random agents The ones with higher fitness function (defined in Equation 2.2.11) are more likely to be chosen. According to what have been stated above the fitness function reflects the free space:

$$f(a) = s_a \tag{2.2.11}$$

Altogether the more free space the agent have the more probable is that he will participate in the negotiation process. In the implementation it means that the *negotiateAll* method in Algorithm 2 is overwritten to ensure the selection (tournament selection strategy is implemented). The pseudo-code of that corresponds to Algorithm 4 with the addition captured in Algorithm 5.

```
function negotiateAll $(R, T)$ begin
    while optimization unsuccessful and counter $<$ $MAX\_ATTEMPTS$) do
        $\{a_1, ..., a_n\} \leftarrow$ choose $n$ agents according to tournament selection strategy;
        optimizeILP $(\{a_1, ..., a_n\})$;
    end
end
```

Algorithm 5: Selection of agents.

The *optimizeILP* as well as the ILP formula (see Equation 2.2.10) is defined generally for any number of agents which opens new possibilities for the negotiation model. Any number n of agents can be chosen to participate in the negotiation and results show (see Section 3.1) that better solutions can be obtained with more than two negotiators in some benchmark instances. However there is no general benefit visible and furthermore a big number of negotiators can bring a very long computation time due to the ILP module.

The selection approach appeared to be very successful in some benchmark instances (refer to Section 3.1) and sometimes it is able to find better solutions then the iterative ILP based negotiation model and even much faster. However not always. The problem is that it contains randomization and so we can't predict the behavior. Sometimes it is able to find the optimal solution very quickly, but other time it does not and it searches until maximum of allowed iterations is reached and new agent is added. The $MAX\_ATTEMPTS$ constant must be set by the user and it is not easy to determine this value. If we have enough time and we really want to try to find the optimum, we can set it to a high value with the risk of very long computation time without finding the optimal solution anyway.

2.3 Two-phase multi-agent solver for 2DBPP

Many possibilities appeared when dealing with solving 2DBPP. Also an ILP approach as in 1DBPP came into consideration, but it turned out that it is not computationally manageable because the problem is too complex to solve using ILP. Therefore it is inappropriate in the multi-agent environment where the optimization is processed many times. This section contains a description of how the multi-agent solver was adjusted to solve 2DBPP inspired by two-phase shelf-based algorithms described in Subsection 1.1.3. Different approach which can also deal with 3DBPP is covered in Section 2.5

In terms of the final two-phase multi-agent implementation we have two separate sub-solvers - each responsible for one phase. One solver for shelves creation and any of the 1DBPP solvers from the previous Section 2.2 as the second. The main task is thus to define a multi-agent solver for a preparation of shelves. These become the input for the 1DBPP solver - each shelf then represents an one-dimensional item with the height as its size.

2.3.1 Multi-agent solver for shelves

While talking about shelves items must be placed in a row of defined width so that sum of their heights is minimized. If we imagine items as tasks and each shelf as an agent we can define it as a multi-agent problem where each agent tries to *delegate* and *negotiate* its tasks - items - to minimize its overall height which equals the maximal height of the loaded items. The formal definition of the optimization problem is (Equation 2.3.1):

$$\begin{aligned} \min \sum_{a\in A} \max_{t\in T_a} h(t) & \qquad \textit{such that} \\ \sum_{t\in T_a} w(t) & \leq W, \ \forall a \in A \end{aligned} \tag{2.3.1}$$

where $h(t)$ and $w(t)$ are height and width of the item t, W is the width of the bin, A is a set of agents (bins) and T_a corresponds to a set of allocated items in the bin a.

In general there is no need to minimize the number of shelves - the minimal sum of heights does not suppose the minimal number of shelves as can be seen on Figure 2.1. In case a) the minimum sum of heights is reached, but not the minimum number of agents. On the other hand in case b) the minimum possible number of agents is used, but the sum of heights is worse than in the first case. Hence the width of the items is

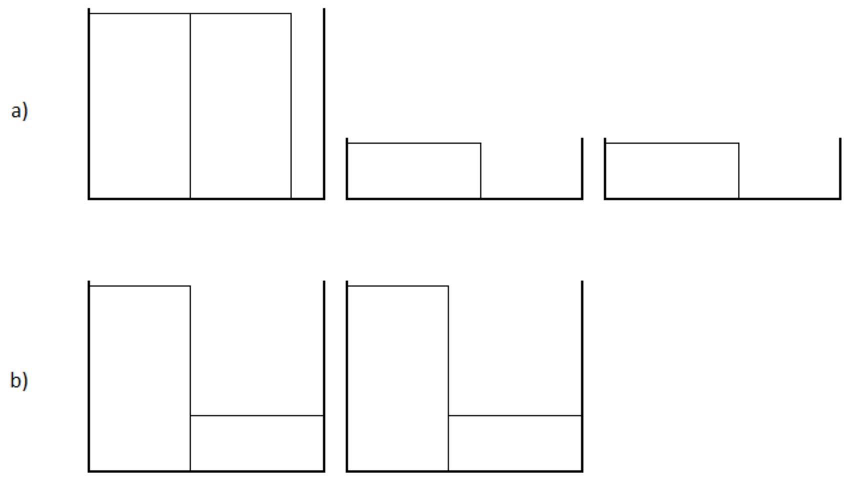

Figure 2.1: Minimal sum of heights of shelves.

not reflected in the the optimization phases but only height is. The exception being the initialization, i.e. the initial number of agents is based on items' width.

The optimization task (Expression 2.3.1) can be easily adapted on the implementation described by Algorithm 3 and Equation 1.2.4. Estimation costs must be defined and they are as follows (Equations 2.3.2 and 2.3.3):

$$cost^{estI}(t,a) = \begin{cases} \max\limits_{t' \in T_a \bigcup \{t\}} h(t') - \max\limits_{t' \in T_a} h(t'), & w(t) + uc \leq W \\ \infty, & w(t) + uc > W \end{cases} \quad (2.3.2)$$

$$cost^{estR}(t,a) = \max_{t' \in T_a} h(t') - \max_{t' \in T_a \setminus \{t\}} h(t') \quad (2.3.3)$$

$h(t)$ and $w(t)$ are height and width of an item t, W is width of the bin, uc is its used space as to width and T_a is a set tasks allocated to agent a. In normal words estimate add is a difference between maximal heights before and after the adding of the item t, if it fits to its width and estimate remove cost returns a difference between maximal heights before and after the removing of the item t. When a smaller item than the highest in the bin is added it costs nothing on the other hand if an item which is not the highest in the bin is removed no profit is gained. Only when the maximum changes it is reflected in the costs.

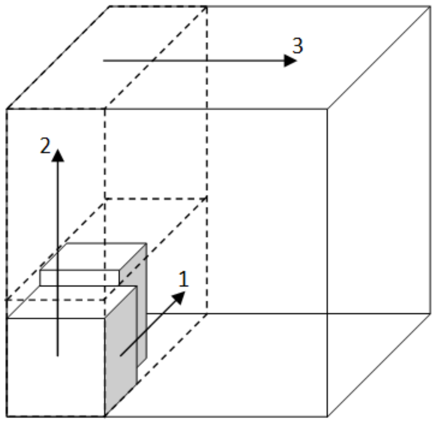

Figure 2.2: Three-phase optimization process.

2.4 Three-phase multi-agent solver for 3DBPP

Same reflection as for 2DBPP led to compose a joint three-phase multi-agent solver. Each phase of such approach in fact reduces one dimension and so there is no reason why 3DBPP couldn't be solved in this way too. The task now is to reduce 3DBPP to 2DBPP and send it to any 2DBPP solver.

The overall optimization process realized by three-phase multi-agent model is illustrated in Figure 2.2. In the first level the agents negotiate about 3D items to create so-called strips (two-dimensional alternative for shelves), these are then processed by shelf-agents to produce shelves (phase 2) which are used in the third phase as one-dimensional items and standard bin agents optimize their placing according to 1DBPP model.

The strips creation phase is very similar to the shelves creation in 2DBPP. The same implementation is used except for a little modification of its estimation costs. Now not only height but also width of strip needs to be optimized. In other words an area of maximal height and maximal width is supposed to be minimized with respect to depth constraint (Expression 2.4.1).

$$\begin{aligned} \min \sum_{a \in A} \max_{t \in T_a} h(t) \cdot \max_{t \in T_a} w(t) \qquad & \textit{such that} \\ \sum_{t \in T_a} d(t) \quad \leq \quad & D, \ \forall a \in A \end{aligned} \tag{2.4.1}$$

$d(t)$ is depth of task t whereas D is depth of the container.

Other values are same as in Expression 2.3.1. The estimate costs are derived similarly to 2DBPP:

$$cost^{estI}(t,a) = \begin{cases} \max\limits_{t' \in T_a \bigcup \{t\}} h(t') \cdot \max\limits_{t' \in T_a \bigcup \{t\}} w(t') \\ \quad -\max\limits_{t' \in T_a} h(t') \cdot \max\limits_{t' \in T_a} w(t'), & d(t) + uc \leq W \\ \infty, & d(t) + uc > W \end{cases} \tag{2.4.2}$$

$$cost^{estR}(t,a) = \max_{t' \in T_a} h(t') \cdot \max_{t' \in T_a} w(t') - \max_{t' \in T_a \setminus \{t\}} h(t') \cdot \max_{t' \in T_a \setminus \{t\}} w(t') \tag{2.4.3}$$

The time complexity of all phases is the same as in Expression 2.2.8 with the difference in number of agents and items because it decreases after each phase. The model is easy and is supposed to solve instances very quickly, however the solutions quality may not be very high because much free space is wasted and it sums in every level. The quality is expected to be much worse than for the same approach in 2DBPP.

2.5 Universal multi-dimensional Multi-agent solver

This implementation is independent of number of dimensions and can be freely used for 2DBPP as well as for 3DBPPTheoretically for 1DBPP too, however probably it wouldn't perform very well because of exact placing which is ensured automatically in 1DBPP and thus it is not optimal in that environment.. The idea is based on volume optimization - agents try to maximize their collective used volume to load all the items according to Proposition 1.
Proposition 1. *Collective optimization has bigger influence than individual.*

It is not important to fully load one agent, but to spread the items between a set of agents - two at minimum. This proposition is supported by Figure 2.3. In case a) the first agent's used volume is maximized, however the remaining load is bigger than in case b). In such way the optimization can be distributed among pairs of agents which are able to consume the load better than the individuals.

After allocation of tasks between agents, agents try to maximize sum of their volume usage pair by pair which causes that sum of not allocated items volume is being minimized until all items are loaded. This process is very suitable for the *negotiation* phase of the multi-agent algorithm (Algorithm 2).

Heuristic based container loading algorithm which is mentioned in Subsection 1.1.5 and Algorithm 1 adapted for collective optimization was used as the volume usage maximization unit. The implementation of the CLP algorithm for individual

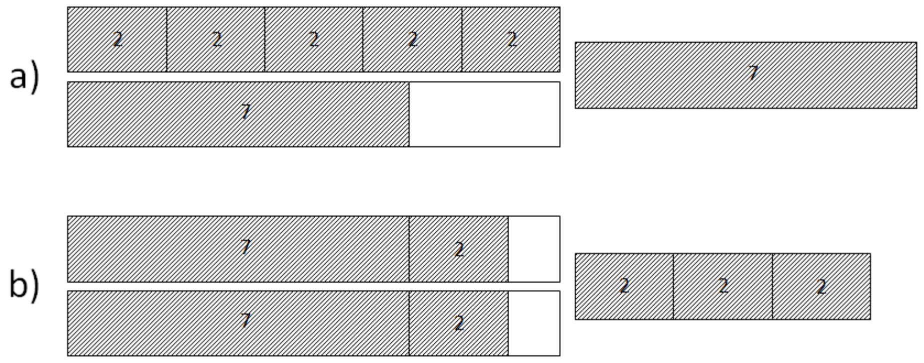

Figure 2.3: Collective optimization against individual.

optimization is covered in the following Subsection 2.5.1, its adaptation to multi-agent environment is then a part of Subsections 2.5.2 and 2.5.3 which deal with the *allocation* and *negotiation* processes. The implementation of the universal multi-dimensional algorithm is then captured in Algorithm 6.

2.5.1 Container loading algorithm implementation

The skeleton of the algorithm is in Algorithm 1. As stated in the related subsection (Subsection 1.1.5) there are two points which can be implemented variously and has an impact on algorithm's behavior and results. In other words these points represent heuristics of the algorithm. They are:

- choosing the free space (more precisely free maximal space, further called only free space) where item will be put in and
- choosing the block of boxes (layers of identical boxes or the box itself).

Main heuristics are adopted from [14].

To choose a free space a measure of its distance to the container's corner is used. For every two points in R^n, $a = (x_{11}, \ldots, x_{1n})$ and $b = (x_{21}, \ldots, x_{2n})$, the distance $d(a, b)$ is defined as the vector of components $|x_{11} - x_{21}|, \ldots, |x_{2n} - x_{2n}|$ ordered in non-decreasing order. For instance, if $a = (3, 3, 2)$ and $b = (0, 5, 10)$, $d(a, b,) = (2, 3, 8)$. For each new free space, the distance from every corner of the space to the corner of the container nearest to it is computed and keep the minimum distance in the lexicographical order:

$$d(S) = min\{d(a, c), a\,\text{vertex of}\,S, c\,\text{vertex of container}\,C\} \tag{2.5.1}$$

The free space and its corner with the minimum distance to any container's corner, or the volume of the space as a tie-breaker, is the place where the boxes will be packed. The reason behind that decision is to first fill the corners of the container, then its sides and finally the inner space [14]. The heuristic is further referenced as **Nearest Corner** heuristic.

Once a free space has been chosen, a block of boxes is chosen. The block of boxes means one box or putting more identical boxes beside each other to form a layer (see Subsection 1.1.5 or Figure 1.5). Two criteria are considered to select the block of boxes [14]:

- **Best Volume** - the block of boxes producing the largest increase in the objective function. This is a greedy criterion in which the space is filled with the block producing the largest increase in the volume occupied by boxes.
- **Best Fit** - the block of boxes which fits best into the free space. Distances from each side of the block to each side of the free space are computed and ordered in a vector in non-decreasing order. The block is chosen using again the lexicographical order.

After adding a block of boxes free spaces must be update - to split spaces which are affected and merge them when a subspace appears. Correct free space handling is a critical part of the algorithm's performance.

2.5.2 Allocation

The allocation phase could be implemented variously. The easiest approach is to fill the containers one by one by the algorithm from previous Subsection 2.5.1. Another possibility is to extend the algorithm to choose free spaces from all the agents - then the boxes would be allocated among all agents. However, to meet the abstract Algorithm 2 different implementation was chosen. Best Fit heuristic defined in previous Subsection 2.5.1 is used in the $cost^{estI}(t, a)$ value. Thus in every iteration a task (after non-increasing pre-ordering) is allocated in the free space in which any block of the box fits best. The selected free space represents the *winner* agent.

When non-increasing presorting of boxes is used, the preprocessing and allocation process is actually similar to Best Fit Decreasing (BFD) algorithm for 1DBPP except the specified Best Fit definition. Hence all that can be now designated as BFD in multi-dimensional space.

```
function delegateAll (R) begin
    // leave it blank
end

function negotiate ($a, R, T$) begin
    forall the $a_{other}$ : $R \setminus a$ do
        $T' \leftarrow T \bigcup$ tasks of $\{a, a_{other}\}$;
        forall the $boxHeuristic : definedHeuristics$ do
            forall the $s$ : free spaces of $\{a, a_{other}\}$ do
                $chosenSpace \leftarrow$ best Nearest Corner free space $s$;
            end
            forall the $box : T$ do
                forall the $blockOfBoxes$ of box that fits to $chosenSpace$ do
                    $chosenBlockOfBoxes \leftarrow$ best acc. to $boxHeuristic$;
                end
            end
            put $chosenBlockOfBoxes$ to $chosenSpace$;
            update free spaces;
        end
        preserve the allocation with $max \sum_{a' \in \{a, a_{other}\}} V_{used}(a')$;
    end
end
```

Algorithm 6: Implementation of the universal multi-dimensional algorithm.

2.5.3 Negotiation

The *negotiation* phase is the only improvement phase because the dynamic improvement is omitted. The reason is that the *negotiation* is realized in a different way without $cost^{estI}$ and $cost^{estR}$ computation and no reasonable *delegation* method was invented. The problem is that the allocated boxes are usually being blocked up by other boxes and when one is removed, no better free space, according to Best Fit strategy (or any other), usually exists because the space just freed is naturally the *best fit*. The optimization possibilities are thus limited in this case.

The method is using Algorithm 1 as described in Subsection 2.5.1 except that it is extended to perform on a set of containers (agents). When a free space is selected according to the nearest corner heuristic the best from all agents is chosen. Then block of boxes are sought inside this space. No clear statement can be said about which heuristic of choosing the block of boxes from the two introduced - Best Volume and Best Fit - is better. Sometimes Best Fit is better sometimes Best Volume is better depending on the set of input items. One can even invent many other heuristics and it may bring the best result on certain instances. The best strategy is to run many different heuristics on the same sub-problem and choose the best result. And that is exactly how the optimization in this case works.

Every pair of the agents is negotiating by unloading all their boxes which are then, together with the remaining (not allocated) items, reallocated between the two agents in many ways according to various input heuristics. Each pair has an unique set of boxes and so different heuristic can be successful in different *negotiation*s. The allocation with the best used volume of the two agents is preserved to other iterations. In such way the volume of remaining boxes is being decreased while the optimization is successful.

The implemented box choosing heuristics are:

- Best Fit - see Subsection 2.5.1
- Best Volume - see Subsection 2.5.1
- Randomized Best Volume - similar to Best Volume, but the block of boxes is chosen randomly from n best (i.e. biggest that fit) blocks with probability

  $$pr_{bl} = \frac{V(bl)}{\sum\limits_{bl' \in B_n} V(bl')}$$

  i.e. the probability corresponds to the volume ratio in the set of n best blocks.

A nice thing about this model is that one can add new heuristics and it can improve the results' quality (it can't be worsened). Even not very good heuristic can bring some improvement because the negotiation is processed many times and so the probability that it can find the best solution of a sub-problem is quite high. Good randomization in box choosing is a challenge that could be realized in a future improvement of the algorithm.

3 Experiments

In this chapter experiments and interesting results are discussed. Results on the benchmark data serve us to assess and compare the introduced approaches. All gained results can be found on the attached CD. Inside the chapter, results from different tables are put together to see differences between algorithms or algorithms' settings. I have chosen only some benchmark data instances - usually the biggest where margins are more visible - or averages on the same data classes. To compare the quality of solutions deviation from lower bound defined as:

$$\frac{ub - lb}{lb} \times 100[\%] \tag{3.0.1}$$

is used - *ub* is the solution found by a solver and *lb* represents the best[1] lower bound usually defined in articles from which the benchmark data originate. All experiments have been run on a standard laptop with 3GB of RAM and 2,16GHz dual core processor. The solver has been implemented as JAVA application with slight performance optimization.

3.1 Experiments on 1DBPP

For evaluation of the multi-agent algorithm adapted on solving 1DBPP, benchmark data from [7] were used. There are 2 classes of bin packing instances. The first class, files *binpack1* to *binpack4* (problem identifiers beginning with *u*) consists of 120, 250, 500 or 1000 items of sizes uniformly distributed in (20,100) to be packed into bins of size 150. The second class, files binpack5 to binpack8 (problem identifiers beginning with *t*) consists of "triplets" of items from (25,50) to be packed into bins of size 100.

For the "uniform" class, the *Best Known* value is the one found by algorithm in the [7]. Except for problems *u120_08, u120_19, u250_07, u250_12* and *u250_13*[2], this is

[1]Usually many different computations of lower bounds exist and the best is the maximum of them.
[2]Four of them were proved as optimal by this multi-agent solution.

also the smallest number of bins capable of accommodating all the items (i.e. the lower bound), so the value is the proved optimum.

For the “triplets” class, the instances were constructed with a known global optimum of $n/3$ bins, i.e. the guaranteed optimal solution has exactly three items per bin. The “triplets” class contains decimals and a three items precisely sum to value of 100. This is a special case and the multi-agent model is not suitable for finding exact “triplets”. Only uniform class, which more naturally simulates real bin packing problems, was taken into account in benchmarking. The only experiment on “triplets” shows that every time a solution worse by exactly 1 bin is found.

Table 3.1 displays results of impact of improvement strategies described in Section 2.2 on the 1DBPP biggest benchmark instances (*binpack4*). $Solve_{Nego}$ represents full heuristic algorithm (see Algorithm 3), $Solve_{NoImpr}$ is a variant where no improvement methods are called and only allocation is performed - it corresponds to BFD (see 1.1.2). $Solve_{NoFinalImpr}$ and $Solve_{NoDynamicImpr}$ represent cases in which no final improvement is called and only dynamic improvement is performed and vice versa. It shows that the dynamic improvement gives the best solution enhancement, but the final improvement also plays its role and processed both the best results are gained.

Table 3.2 contains results of different settings of ILP based multi-agent solver, the standard heuristic based implementation comparing to the best known solutions when the benchmark data originated (see [7]). $Solve_{Nego}$ represents basic heuristic implementation (as in the previous measurement), $Solve_{ILPIter}$ is a ILP based negotiation approach described in Subsection 2.2.5 and the last two column headers contain variants of selection based ILP approach (see Subsection 2.2.6) where 2 or 3 agent are selected in terms of tournament selection strategy.

The heuristic approach is much faster in average and produces quite reasonable results. What is not seen in the table is that the ILP negotiation model with selection mechanism can be very effective - it is able to find optimal solutions very quickly (even faster then the heuristic approach), but very ineffective too - in cases when it can’t find the optimum the $MAX\_ATTEMPTS$ value is reached and it can consume very long time. Thus the variance of the computation time of this approach is very high and the average value is meaningless (furthermore it depends on the value of the $MAX\_ATTEMPTS$ constant). There is no big difference in results’ quality when two or three agents are selected - it appears that selecting three agents could be advantageous for small instances, however it consumes much more time with larger input data.

	Best Known	$Solve_{NoImpr}$		$Solve_{NoDynamicImpr}$		$Solve_{NoFinalImpr}$		$Solve$	
Instance	# bins	# bins	time(s)	# bins	time(s)	# bins	time(s)	# bins	time(s)
u1000_00	**399**	403	0,14	401	2.28	400	3.09	**399**	5.51
u1000_01	**406**	411	0,07	408	1.49	**406**	3.18	**406**	5.64
u1000_02	**411**	416	0,06	413	1.82	412	3.17	**411**	5.65
u1000_03	**411**	416	0,05	414	1.51	414	3.38	412	5.89
u1000_04	**397**	402	0,04	400	1.51	398	3.33	398	6.00
u1000_05	**399**	404	0,04	402	1.80	400	3.18	400	5.98
u1000_06	**395**	399	0,03	397	1.20	**395**	3.14	**395**	5.40
u1000_07	**404**	408	0,16	406	1.22	**404**	3.33	**404**	5.94
u1000_08	**399**	404	0,02	402	2.04	400	3.32	**399**	5.91
u1000_09	**397**	404	0,02	401	1.60	400	3.43	398	6.58
u1000_10	**400**	404	0,01	402	1.36	**400**	3.36	**400**	6.07
u1000_11	**401**	405	0,01	404	1.69	**401**	3.45	**401**	5.89
u1000_12	**393**	398	0,02	395	1.31	**393**	3.43	**393**	5.65
u1000_13	**396**	401	0,02	399	3.62	397	3.54	**396**	5.89
u1000_14	**394**	400	0,02	397	1.41	395	3.46	395	5.99
u1000_15	**402**	408	0,02	405	1.40	404	3.70	403	6.21
u1000_16	**404**	407	0,01	406	1.36	**404**	3.67	**404**	6.03
u1000_17	**404**	409	0,02	406	0.98	406	3.34	405	6.49
u1000_18	**399**	403	0,01	401	1.12	**399**	3.45	**399**	6.10
u1000_19	**400**	406	0,02	402	2.78	402	3.50	**400**	6.00
	400.55	405.4	**0.04**	403.05	1.67	401.5	3.37	400.9	5.94

Table 3.1: Comparison of the 1DBPP algorithm's improvement strategies.

	Best	$Solve_{Nego}$			$Solve_{ILPIter}$			$Solve_{ILPSelect2}$			$Solve_{ILPSelect3}$		
data	$\emptyset bins$	$\emptyset bins$	%dev	$\emptyset t$(s)	$\emptyset bins$	%dev	$\emptyset t$(s)	$\emptyset bins$	%dev	$\emptyset t$(s)	$\emptyset bins$	%dev	$\emptyset t$(s)
bp1	49.15	49.3	0.35	0.03	49.1	-0.1	1.09	**49.05**	-0.2	0.2*	**49.05**	-0.2	0.53*
bp2	101.7	101.95	0.34	0.15	101.8	0.1	13,75	101.8	0.1	11.08*	**101.6**	-0.1	74.82*
bp3	**201.2**	201.55	0.17	0.75	201.3	0.05	55.69	**201.2**	0	9.7*	201.35	0.07	21.88*
bp4	**400.55**	400.9	0.09	5.94	**400.55**	0	187.46	400.6	0.01	14.02*	400.85	0.07	108.3*
$\emptyset dev$			0.24			0.04			0.03			0.035	

Table 3.2: Comparison of the 1DBPP approaches.

3.2 Experiments on 2DBPP

The goal is to assess the presented algorithms for 2DBPP test instances provided by Berkey and Wang [2] and Martello and Vigo [12], consisting of ten classes of problems. In each problem class there are 50 instances: 10 with 20 rectangles, 10 with 40 rectangles, 10 with 60 rectangles, 10 with 80 rectangles, and 10 with 100 rectangles. Items rotation is not allowed in the basic version. Problem classes I–VI were proposed by Berkey and Wang [2], while classes VII–X are due to Martello and Vigo [12].

The results gained on the benchmark data are presented in Table 3.3 for Berkey and Wang instances and Table 3.4 for classes from Martello and Vigo. In the tables approaches for solving 2DBPP introduced in Section 2.3 and Section 2.5 are compared to the best known solutions. The best known solutions were obtained from [21]. The article was not available at the time of this work, however the results are accessible on the project's website and are said to be the state of the art because their Space Defragmentation algorithm outperforms all leading meta-heuristic approaches by a significant margin [3]. All values are compared to lower bounds and the deviation is presented. If the deviation is 0 the value is also the proved optimum.

$Shelf_{NegoNego}$ is the two phase multi-agent solver which uses only heuristic based multi-agent sub-solvers to optimize the task. It is compared also to $Shelf_{KpIlp}$ where the first phase - shelves creation - is provided by solving of Knapsack Problem (refer to Subsection 1.1.3) defined as ILP program. The second phase is then when done by the $Solve_{ILPIter}$ from previous Section 3.1. Both phases are supposed to generate better partial solutions[4] than by $Shelf_{NegoNego}$. Other combinations were tested too - $Shelf_{NegoIlp}$ and $Shelf_{KpNego}$ - but the $Shelf_{KpIlp}$ produces naturally the best results on average in shelf-based implementations.

Universal represents the Universal multi-dimensional solver from Section 2.5. The universal solver contains feature that builds block of boxes of same dimensions which helps to use free space better. It was inspired by container loading which benchmark data are structured differently and the same boxes are one of the problem's supposition. However it is not in the bin packing benchmark instances. The items are generated randomly and there are no (or very few) items of the same dimension. Thus the solver's power is slightly weaken in the pure bin packing environment. The second feature which is not used in benchmarking is items rotations which is prohibited in basic version. It is the same in 3DBPP in Section 3.3.

[3]In other articles I found worse results so it is very probable.

[4]Both shelve creation implementations were compared as strip packing instances and KP performed better than $Self_{Nego}$, however the results have not been recorded. The comparison of $Solve_{ILPIter}$ and $Solve_{Nego}$ is in Table 3.2.

		LB	Best Known		$Shelf_{NegoNego}$			$Shelf_{KpIlp}$			$Universal$		
Class	n	∅*bins*	∅*bins*	%dev	∅*bins*	%dev	∅t(s)	∅*bins*	%dev	∅t(s)	∅*bins*	%dev	∅t(s)
class1	20	7.1	**7.1**	**0**	7.7	8.45	0.02	7.5	5.63	0.05	**7.1**	**0**	0.29
	40	13.4	**13.4**	**0**	14.2	5.97	0.04	13.9	3.73	0.13	**13.4**	**0**	0.35
	60	19.7	**20**	**1.52**	21.1	7.11	0.04	20.6	4.57	0.06	20.1	2.03	0.79
	80	27.4	**27.5**	**0.37**	28.8	5.11	0.04	28.5	4.01	0.18	**27.5**	**0.37**	1.91
	100	31.7	**31.7**	**0**	33.8	6.62	0.05	33	4.1	0.21	31.8	0.32	2.14
∅*dev*				**0.38**		6.65			4.41			0.54	
class2	20	1	**1**	**0**	1.4	40	0	**1**	**0**	0	**1**	**0**	0
	40	1.9	**1.9**	**0**	2.1	10.52	0	2	5.26	0	2	5.26	0.03
	60	2.5	**2.5**	**0**	3.1	24	0.01	2.8	12	0.01	2.6	4	0.07
	80	3.1	**3.1**	**0**	4	29.03	0.01	3.3	6.45	0.02	3.3	6.45	0.16
	100	3.9	**3.9**	**0**	4.9	25.64	0.01	4	2.5	0.02	4	2.5	0.23
∅*dev*				**0**		25.84			5.26			3.66	
class3	20	5.1	**5.1**	**0**	5.6	9.8	0	5.4	5.9	0.01	5.2	1.96	0.05
	40	9.2	**9.3**	**1.09**	10.1	9.78	0.01	10.1	9.78	0.03	9.6	4.35	0.29
	60	13.6	**13.9**	**2.21**	15.2	11.76	0.02	14.7	8.1	0.06	14	2.94	0.81
	80	18.7	**18.9**	**1.07**	20.4	9.1	0.02	20.2	8.02	0.13	19.1	2.14	1.89
	100	22.1	**22.3**	**0.91**	23.9	8.14	0.05	23.8	7.69	0.22	22.7	2.71	3.26
∅*dev*				**1.05**		9.72			7.89			2.82	
class4	20	1	**1**	**0**	1.2	20	0	**1**	**0**	0	**1**	**0**	0
	40	1.9	**1.9**	**0**	2	5.26	0.01	2	5.26	0.01	**1.9**	**0**	0.03
	60	2.3	**2.5**	**8.7**	2.9	26.1	0.01	2.7	17.39	0.02	**2.5**	**8.7**	0.12
	80	3	**3.2**	**6.67**	3.9	30	0.01	3.4	13.33	0.04	3.3	10	0.27
	100	3.7	**3.7**	**0**	4.4	18.92	0.02	4	8.11	0.04	3.8	2.7	0.44
∅*dev*				**3.07**		20.05			8.82			4.28	
class5	20	6.5	**6.5**	**0**	6.9	6.15	0	6.7	3.08	0.01	6.6	1.54	0.09
	40	11.9	**11.9**	**0**	12.4	4.2	0.01	12.5	4.2	0.04	**11.9**	**0**	0.42
	60	17.9	**18**	**0.56**	18.6	3.91	0.01	18.9	5.59	0.2	18.1	1.11	1.45
	80	24.1	**24.7**	**2.49**	25.6	6.22	0.04	25.7	6.64	0.3	24.7	2.49	3.1
	100	27.9	**28.1**	**0**	30.1	7.89	0.05	29.8	6.81	0.4	28.6	2.51	5.58
∅*dev*				**0.75**		5.68			5.43			1.53	
class6	20	1	**1**	**0**	**1**	**0**	0	**1**	**0**	0	**1**	**0**	0
	40	1.5	**1.6**	**6.67**	2	33.33	0	2	33.33	0.01	1.9	26.67	0.05
	60	2.1	**2.1**	**0**	2.5	19	0.01	2.3	9.52	0.02	2.2	4.76	0.12
	80	3	**3**	**0**	3.1	3.33	0.02	**3**	**0**	0.04	**3**	**0**	0.23
	100	3.2	**3.4**	**6.25**	3.9	21.9	0.02	3.5	9.38	0.06	3.5	9.38	0.72
∅*dev*				**2.58**		15.52			10.45			8.16	

Table 3.3: Results on 2DBPP benchmark data Berkey and Wang class 1-6.

		LB	Best Known		$Shelf_{NegoNego}$			$Shelf_{KpIlp}$			$Universal$		
File	n	$\emptyset bins$	$\emptyset bins$	%dev	$\emptyset bins$	%dev	$\emptyset t$(s)	$\emptyset bins$	%dev	$\emptyset t$(s)	$\emptyset bins$	%dev	$\emptyset t$(s)
class7	20	5.5	**5.5**	**0**	5.8	5.45	0	5.8	4.45	0	**5.5**	**0**	0.06
	40	10.9	**11.1**	**1.83**	11.7	7.34	0.01	11.5	5.5	0.01	11.2	2.75	0.48
	60	15.6	**15.8**	**1.28**	16.6	6.41	0.02	16.4	5.13	0.01	15.9	1.92	1.37
	80	22.4	**23.2**	**3.57**	23.8	6.25	0.03	23.6	5.36	0.15	**23.2**	**3.57**	5.22
	100	26.9	**27.1**	**0.74**	27.7	2.97	0.07	27.9	3.72	0.05	27.3	1.49	8.33
$\emptyset dev$				**1.49**		5.69			5.03			1.95	
class8	20	5.8	**5.8**	**0**	6.4	10.34	0	6.1	5.17	0.01	**5.8**	**0**	0.07
	40	11.2	**11.3**	**0.89**	12.6	12.5	0	11.9	6.25	0.06	11.4	1.79	0.56
	60	15.9	**16.1**	**1.26**	17.3	8.81	0.01	16.9	6.29	0.11	16.2	1.89	1.73
	80	22.3	**22.4**	**0.45**	23.9	7.2	0.03	23.3	4.48	0.29	22.5	0.9	4.12
	100	27.4	**27.8**	**1.46**	29.9	9.12	0.04	28.4	3.65	0.57	**27.8**	**1.46**	10.03
$\emptyset dev$				**0.81**		9.59			5.17			1.21	
class9	20	14.3	**14.3**	**0**	14.4	0.7	0	14.4	0.7	0.13	**14.3**	**0**	0.31
	40	27.8	**27.8**	**0**	**27.8**	**0**	0.01	28	0.72	0.76	**27.8**	**0**	2.8
	60	43.7	**43.7**	**0**	43.9	2.8	0.03	43.9	0.46	3.02	**43.7**	**0**	12.1
	80	57.7	**57.7**	**0**	**57.7**	**0**	0.07	57.8	0.17	6.83	**57.7**	**0**	34.4
	100	69.5	**69.5**	**0**	69.8	0.43	0.13	69.7	0.29	12.78	**69.5**	**0**	70
$\emptyset dev$				**0**		0.32			0.47			**0**	
class10	20	4.2	**4.2**	**0**	4.7	11.9	0	4.6	9.52	0.01	4.3	2.38	0.04
	40	7.4	**7.4**	**0**	8.1	9.46	0	7.8	5.41	0.02	**7.4**	**0**	0.19
	60	9.8	**10**	**2.04**	11.2	14.29	0.01	10.7	9.18	0.08	10.3	5.1	0.69
	80	12.3	**12.8**	**4.07**	14.1	14.63	0.02	13.7	11.38	0.07	13	5.7	1.25
	100	15.3	**15.9**	**3.92**	17.4	13.73	0.03	16.5	7.84	0.08	16.3	6.54	2.2
$\emptyset dev$				**2.01**		12.8			8.67			3.94	

Table 3.4: Results on 2DBPP benchmark data Martello and Vigo class 7-10.

From results it can be said that shelf based approach with only heuristic realization ($Shelf_{NegoNego}$) does not produce very high-quality results. The reason is that loss of optimality sums on each level - the result of the first are non-optimal shelves which are then placed in a non-optimal way. On the other hand, shelf-based algorithms are not supposed to be the best. Its strength is in its simplicity and in the meeting the guillotine cutting requirement (see Subsection 1.1.3). Taken this into account this the results are, except for some instances, quite reasonable in comparison to the $Shelf_{KpIlp}$, which represents one of the top shelf-based algorithm's implementations, and with very good solving time which could be more apparent on bigger instances.

The universal multi-dimensional solver performed well when time was not taken as the main criterion. It is not able to compete with the state of the art best algorithm [21], but the difference is not big. The solver can be, furthermore, easily enhanced by adding new reasonable box choosing heuristics. Only on some Berkey and Wang instances with small number of bins the margin is more visible. It is because the *negotiation* is very limited or even suppressed in instances with 1 or 2 bins as a solution and only *allocation* is responsible for the quality of the solution. The application of heuristics on different items' subsets - which is the optimization - does not occur. These instances are generally not suitable for any multi-agent solver and should be addressed by explicit algorithms like for container loading problem.

3.3 Experiments on 3DBPP

For the 3D Bin Packing Problem, 320 instances generated by Martello et. al. ([10]) were used. This set of instances consists of 8 classes each class further divided into 4 groups. Every group consists of 10 instances with same number of items - the number of items per instance in the 4 groups are 50, 100, 150 and 200, respectively. Rotations of items is prohibited. The dataset, beside lower bounds values, also contains *best known* solutions which were probably achieved by the exact 3DBPP algorithm presented in [10]. Because I did not found any other results on these datasets I use it for comparison. Beside this, state-of-the-art best results were obtained from a website of a project related to [21] as for 2DBPP. Results of the experiments are in Table 3.5.

$Strip_{NegoNegoNego}$ represents the three phase 3DBPP implementation with only heuristic optimization (no ILP is used). And *Universal* is the universal multi-dimensional solver this time applied on 3DBPP.

Same as for 2DBPP - the universal multi-dimensional solver performs well with quite high computation time. The reason may be non-optimal implementation of free space managing (cutting of unused free spaces could be a solution), or just too complex algorithm inside *negotiation* process. More lightweight version of container

		LB	Best Known 2003		Best Known		$Strip_{NegoNegoNego}$			*Universal*		
Class	*n*	*∅bins*	*∅bins*	%dev	*∅bins*	%dev	*∅bins*	%dev	*∅t*(s)	*∅bins*	%dev	*∅t*(s)
class1	50	12.5	13.8	10.4	**13.4**	**7.2**	14.2	13.6	0.22	13.5	8	2.68
	100	25.1	27.4	9.16	**26.6**	**5.98**	28.1	11.95	0.18	26.9	7.17	17.4
	150	34.7	37.9	9.22	**36.5**	**5.19**	39.2	12.97	0.58	37	6.63	61.95
	200	48.4	52.9	9.3	**50.9**	**5.17**	53.8	11.16	1.7	51.2	5.79	143.56
∅dev				9.52		**5.88**		12.42			6.9	
class2	50	12.7	14.1	11.02	**13.8**	**8.66**	15.1	18.9	0.02	14.1	11.02	3.29
	100	24.1	26.4	9.54	**25.6**	**6.22**	28.3	17.42	0.14	25.9	7.47	17.87
	150	35.1	38.7	10.26	**36.8**	**4.84**	41.6	18.52	0.49	37.3	6.27	59.36
	200	47.5	51	7.37	**49.5**	**4.21**	55.3	16.42	1.39	49.9	5.05	131.79
∅dev				9.55		**5.98**		17.82			7.45	
class3	50	12.3	13.6	10.57	**13.3**	**8.13**	14.6	18.7	0.01	13.6	10.57	3.11
	100	24.7	27.3	10.53	**25.9**	**4.86**	28.7	16.19	0.08	26.2	6.07	21.13
	150	36	39.5	9.72	**37.6**	**4.44**	40.9	13.61	0.22	37.7	4.72	49.95
	200	47.8	51.6	7.95	**50.1**	**4.81**	55.1	15.27	0.51	50.2	5.02	108.74
∅dev				9.69		**5.56**		15.94			6.6	
class4	50	28.9	29.6	2.42	**29.4**	**1.73**	29.9	3.46	0.02	29.5	2.08	2.35
	100	57.6	59.2	2.78	**59**	**2.43**	59.9	3.99	0.21	**59**	**2.43**	14.96
	150	85.2	87.5	2.7	**86.8**	**1.88**	87.9	3.17	0.79	**86.8**	**1.88**	44.99
	200	116.3	119.5	2.75	**118.8**	**2.15**	120.2	3.35	2.33	**118.8**	**2.15**	117.34
∅dev				2.66		**2.05**		3.49			2.13	
class5	50	7.4	10	35.14	**8.3**	**12.16**	9.4	27.03	0.01	8.4	13.51	3.01
	100	12.9	17.6	36.43	**14.9**	**15.5**	16.4	27.13	0.06	15.2	17.83	21.51
	150	17.4	24	37.93	**19.9**	**14.37**	22.7	30.46	0.17	20.5	17.82	86.15
	200	24.4	31.7	29.92	**27.2**	**11.48**	30.6	25.41	0.38	27.4	12.3	149.27
∅dev				34.85		**13.38**		27.51			15.36	
class6	50	9.1	10.3	13.19	**9.8**	**7.69**	11.1	21.98	0.01	10	9.89	1.59
	100	17.5	20.2	15.43	**19**	**8.57**	21.2	21.14	0.06	19.2	9.71	12.52
	150	26.9	32.3	20.07	**29.2**	**8.55**	32.1	19.33	0.19	29.3	8.92	35.37
	200	35	41.9	19.71	**37.5**	**7.14**	41.9	19.71	0.43	37.9	8.29	69.05
∅dev				17.1		**7.99**		20.54			9.2	
class7	50	6.4	9.2	43.75	**7.4**	**15.63**	8.6	34.38	0.01	**7.4**	**15.63**	2.42
	100	10.9	15.5	42.20	**12.2**	**11.93**	14.4	32.11	0.07	12.5	14.68	19.94
	150	13.7	19.9	45.26	**15.4**	**12.41**	17.6	28.47	0.19	15.9	16.06	57.68
	200	21	28.5	35.71	**23.4**	**11.43**	27.1	29.05	0.45	23.9	13.81	128.99
∅dev				41.73		**12.85**		31			15.04	
class8	50	8.3	10.1	21.69	**9.2**	**10.84**	10.4	25.3	0.01	9.4	13.25	2.87
	100	17.6	21.4	21.59	**18.9**	**7.39**	21	19.32	0.08	19.1	8.52	21.69
	150	21.3	28.3	32.86	**23.7**	**11.27**	26.9	26.29	0.24	24.2	13.62	76.97
	200	26.7	35	31.09	**29.5**	**10.49**	33.2	24.34	0.49	30	12.36	200.41
∅dev				26.81		**10**		23.81			11.94	

Table 3.5: Results on 3DBPP benchmark data Martello and Vigo [10].

loading optimization algorithm would speed up the multi-agent solver a lot. However the computation time is still tolerable and with respect to the results it is worth using. Blocks of boxes as well as boxes rotation features are not applied in benchmark as stated in Section 3.2. In real versions of problems the solver could perform even better.

As expected, the three-phase implementation is not very successful with respect to quality of results, however its main strength is the computation time and in an environment where solutions must be generated very quickly it is supposed to be useful. The improvement of the approach would be better strips and shelves filling - e.g similar to floor ceiling algorithm on Figure 1.3. However it is not easy to integrate it into the *allocation* and *negotiation* model and it would cause problem-specific modifications of the abstract algorithm.

4 Conclusions and Outlook

This work presents an application of multi-agent solver architecture introduced in [19] on well known Bin Packing Problem. Problematics of bin backing problem, its variants (including container loading problem) as well as the multi-agent solver architecture are examined in Chapter 1. Section 1.2 contains specification of the multi-agent algorithm with separated dynamic and final improvement - *delegation* and *negotiation* - and their incremental version. Slight generalization of the algorithm that allows to address problems differently without need of estimation removal cost - as used in the universal multi-dimensional solver - is also explained.

Implementations of bin packing problems are discussed in Chapter 2. First, one-dimensional bin packing problem (1DBPP) was addressed. After the simple heuristic approach which defines estimation costs according to free space of bins optimized version was introduced - the negotiation phase is performed by ILP optimization which ensures that the sub-problems are solved optimally. To enhance the improvement phase, fitness based selection of agents was adapted from genetic algorithm's principles.

Implementation of 2DBPP was inspired by shelf-based heuristic approach. Two-phase multi-agent solver which firstly composes shelves using again optimization based on estimation costs values minimizing the sum of heights and then uses the solver for 1DBPP. The same procedure was adjusted to 3DBPP. It has three phases each solvable by multi-agent solver with defined estimation costs minimizing values in related dimensions.

Finally the universal multi-dimensional solver which was tested on 2DBPP as well as on 3DBPP is discussed. It is based on collective volume optimization of couples of agents as *negotiation*. The underlying optimization process is derived from container loading algorithm of [14] using different heuristics. The best results are gained when many different heuristics are used inside the *negotiation* because each heuristic generates the best solutions on different subsets of items.

The experiments are covered in Chapter 3. Only basic version of bin packing problem was taken into account. No other constraints except dimensional and rotational were taken into account. Furthermore, rotation of items is prohibited in benchmark data so this feature as well as building blocks of same boxes is excessive in the final implementation of the universal multi-dimensional solver with regards to testing.

In 1DBPP experiments the improvement strategies were compared and it was proved that both phases - dynamic as well as final - are important in the optimization process. ILP approach produced naturally better results but with worse computation time. The selection principles appeared to be successful, however they can be ineffective too. The computation times can vary from milliseconds to tens or even hundreds of seconds depending on iteration limits that are set. All instances except one were proved as optimum (i.e. solutions equal to lower bounds were achieved) which outperforms the exact algorithm of [7] from which benchmark data originates. The heuristic approach generates slightly worse solutions but faster.

Experiments on 2DBPP revealed that the two-phase multi-agent approach can't compete with the best known solutions with regard to quality. The reason is that optimality is being lost in each of the levels and the losses are summed during the process. However, again, the main strength could be in its computation time and in dynamic environment where results must be generated very quickly it could be useful. The same thing, according to gained results, can be said about the three-phase multi-agent solver in 3DBPP.

The universal multi-dimensional solver performed well in 2DBPP benchmark instances as well as in 3DBPP. The results are slightly worse then the state-of-the-art best known solutions, but the deviation is small. Moreover, the solver can be easily enhanced by adding new box choosing heuristics - the results then can't be worsen, but only improved. Good randomization in choosing box procedure would possibly make the solver more powerful. The drawback of the implementation is the computation time which reaches more than 100 seconds per instance with 200 boxes. More lightweight version of the volume optimization algorithm without or with better free space handling would have to be included to speed up the solver. However one could find the ratio of quality and time reasonable.

Future work could have two parts. In the first part the introduced implementations could be improved trying to produce better solutions. Mainly shelves and strips optimization could be ensured different ways with differently defined estimation costs. The universal multi-dimensional solver could be reimplemented to reduce computation times - either replace the volume optimization algorithm or trying to define estimation costs and reuse the *delegate* and *swap* methods from one-dimensional space implementation. However for the second case additional techniques in *allocation* like shifting of allocated items or free space defragmentation would be probably needed which would make the problem even more complex. The interesting challenge is also an application of the solver in the distributed environment where each agent represents a computation unit (thread or process) and the whole solver acts as a distributed system. In implementations where the complexity of negotiation method is higher than of the communication overhead (probably the universal multi-dimensional solver) it could be beneficial.

The second branch of future work should contain adding more constraints to the problem. In some cases it should be very easy, for instance in one-dimensional two-constraint bin packing (the one-dimensional bin has together the capacity constraint also an additional one e.g. weight constraint) one could use the basic 1DBPP model adding the second constraint to the estimation costs resulting in sums or products. In more complex cases the solution would not have to be so easy and additional research would be needed.

A CD Attachment

The attached CD contains this data structure:

- data - benchmark data,
  - 1dbpp - data from [7],
  - 2dbpp - data from [2, 12],
  - 3dbpp - data from [10],
  - clp - container loading problem benchmark data from [4],
- results - results obtained from different implementations or configurations,
  - 1dbpp - results gained by mas-1bpp,
  - 2dbpp - results gained by mas-2bpp and mas-universal-bpp,
  - 3dbpp - results gained by mas-3bpp and mas-universal-bpp,
  - clp - results gained by CLP-constructive-algorithm,
  - bestKnown - best known results from [21],
- text - source text of the final pdf,
  - container_loading - text for container loading problem sub-project,
  - mas_bpp - LyX sources of the final pdf,
- workspace - Eclipse JAVA workspace (can be imported to Eclipse) containing standard Eclipse projects:
  - 3D-vis-framework - framework for visualization boxes in a container using JME3 (JAVA based game engine),
  - alite-mvn - a software toolkit helping with particular implementation steps during construction of multi-agent systems,

  - CLP-constructive-algorithm - implementation of container loading algorithm from [14],
  - CLP-main - test of CLP-constructive-algorithm including visualization,
  - mas-1bpp - multi-agent solver implementation for 1DBPP,
  - mas-2bpp - two-phase multi-agent solver implementation for 2DBPP,
  - mas-3bpp - three-phase multi-agent solver implementation for 3DBPP,
  - mas-bpp-benchmarks - testing of all ma-solver implementations,
  - ma-solver - abstract multi-agent solver,
  - ma-solver-vrp - original multi-agent solver from [18] applied on VRP,
  - mas-universal-bpp - the universal multi-dimensional solver implementation.
- horkyada_diploma_2012.pdf - the final pdf.

LIST OF FIGURES

List of Tables

List of Algorithms

BIBLIOGRAPHY

[1] *A Multi-methodology Agetnt-based Approach For Container Loading*, 2011.

[2] J. O. Berkey and P. Y. Wang. Two dimensional finite bin packing algorithms. *Journal of the Operational Research Society*, 38:423–429, 1987.

[3] A. K. Bhatia, M. Hazra, and S. K. Basu. Better-fit heuristic for one-dimensional bin-packing problem. *IEEE International Advance Computing Conference (IACC 2009)*, 2009.

[4] E.E. Bischoff and M.S.W. Ratcliff. Issues in the development of approaches to container loading. *OMEGA*, 23(4):377–390, 1995.

[5] F. K. R. Chung, M. R. Garey, and D. S. Johnson. On packing two-dimensional bins. *SIAM Journal of Algebraic and Discrete Methods*, pages 3:66–76, 1982.

[6] E. G.Jr Coffman, G. Galambos, S. Martello, and D. Vigo. Bin packing approximation algorithms: Combinatorial analysis. In *Handbook of Combinatorial Optimization* [6].

[7] E. Falkenauer. A hybrid grouping genetic algorithm for bin packing. *Working paper CRIF Industrial Management and Automation*, 1994.

[8] R. E. Korf. A new algorithm for optimal bin packing. *AAAI-02 Proceedings*, 2002.

[9] A. Lodi. *Algorithms for Two-Dimensional Bin Packing and Assignment Problems*. Universita Degli Studi Di Bologna - Doctoral thesis, 1996-1999.

[10] S. Martello, D. Pisinger, and D. Vigo. The three-dimensional bin packing problem. *Operation Research*, 48:256–267, 2000.

[11] S. Martello and P. Toth. Bin-packing problem. *Knapsack Problems: Algorithms and Computer Implementations*, pages chapter 8, 221Ű245, 1990.

[12] S. Martello and D. Vigo. Exact solution of the two-dimensional finite bin packing problem. *Management Sci.*, 44:388–399, 1998.

[13] A. Moura and J. F. Oliveira. A grasp approach to the container-loading problem. *IEEE Intell. Syst.*, 20:50–57, 2005.

[14] F. Parreno, R. Alvarez-Valdes, J. F. Oliveira, and J. M. Tamarit. A maximal-space algorithm for the container loading problem. *INFORMS J. Comput.*, 2008.

[15] F. Parreno, R. Alvarez-Valdes, J. F. Oliveira, and J. M. Tamarit. Neighborhood structures for the container loading problem: a vns implementation. *J Heuristics*, 16:1–22, 2010.

[16] D. Pisinger. Heuristics for the container loading problem. *European Journal of Operational Research*, 141:382–392, 2002.

[17] D. Pisinger and M. Sigurd. Using decomposition techniques and constraint programming for solving the two-dimensional bin-packing problem. *INFORMS Journal on Computing*, 19:36–51, 2007.

[18] J. Vokrinek, A. Komenda, and M. Pechoucek. Agents towards vehicle routing problems. In Wiebe van der Hoek, Gal A. Kaminka, Yves Lespérance, Michael Luck, and Sandip Sen, editors, *AAMAS 2010: Proceedings of the Ninth International Conference on Autonomous Agents and Multi-Agent Systems*, pages 773–780, Toronto, Canada, May 2010. IFAAMAS: Internatioal Foundation for Autonomous Agents and Multiagent Systems.

[19] J. Vokrinek, A. Komenda, and M. Pechoucek. Abstract architecture for task-oriented multi-agent problem solving. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 41(1):31–40, January 2011.

[20] Z. Wang and K. W. Li. A heuristic algorithm for the container loading problem with heterogeneous boxes. *IEEE International Conference on Systems, Man, and Cybernetics*, 2006.

[21] Z. Wenbin, Z. Zhang, W-C. Oon, and A. Lim. Space defragmentation for packing problems. *European Journal of Operational Research*, submitted.

[22] B. Zeddini, M. Temani, A. Yassine, and K. Ghedira. An agent-oriented approach for the dynamic vehicle routing problem. *Advanced Information Systems for Enterprises*, pages 70–76, 2008.