0.1 The Development of The Packing Algorithm

The algorithm was developed from the theory described in section **??**. The theory has been extended from the experiences when packing the program. These extensions have made the packing more effective, which result in that more items can be packed. To understand the different methods, the algorithm as described in section **??** must be extended a bit. The extension will be describes in this section, with a roundup at the end. To describe the code, this section will be divided in different subsections to understand the different parts of the code.

0.1.1 Classes

To handle the information and functions needed to handle suitcases and items to pack, the classes luggage and luggage_item has been made. They do both inherit from the class Cube_Shape, with the variables width, depth, height and name. The Cube_Shape class provides the information of a cube, which both suitcases and items can be seen as. It is the dimensions and a name.

The class "luggage"

The class diagram of the luggage class can be seen on Figure 0.1. There are a lot of private fields (starting with "_") which can be accessed through properties. It contains all the values needed: the maximum weight, the weight of the suitcase etc. It contains also a set of methods which is used through the packing process. The properties which should only be accessed in the methods in the class has a private setter, so functions outside the class only can get the value, and not change the properties by a mistake. An example of this is the property "weight", which indicates the total weight of the items in the suitcase and the suitcase' weight. This value should only be set by the class it self in the methods placing an item.

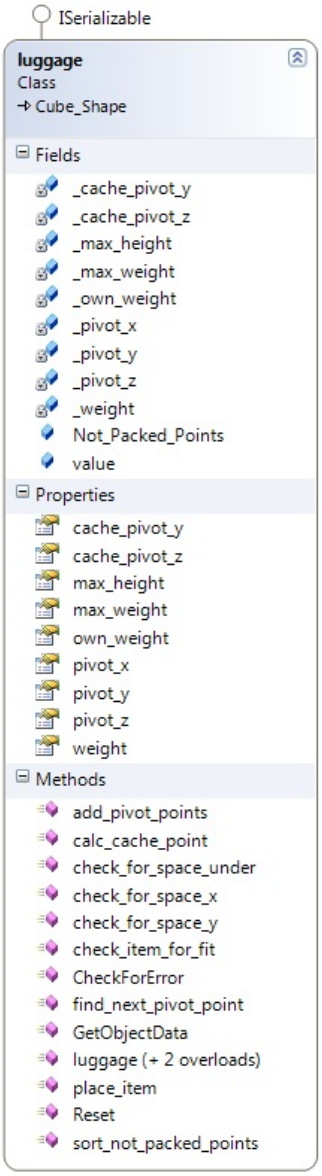

Figure 0.1: The class diagram of the luggage class

As seen on Figure 0.1 there are a set of methods in the class "luggage". These methods provide the necessary functions to handle the suitcase while packing.

The class "luggage_item"

The class luggage_item is for the items to pack in the luggage. The class diagram can be seen on Figure 0.2. As in the luggage class, this class has some variables, which can only be accessed through properties. Those properties, which should only be set inside the class have a private setter. There are also a private method "Rotate_Lug_Item", which should only be called inside the class.

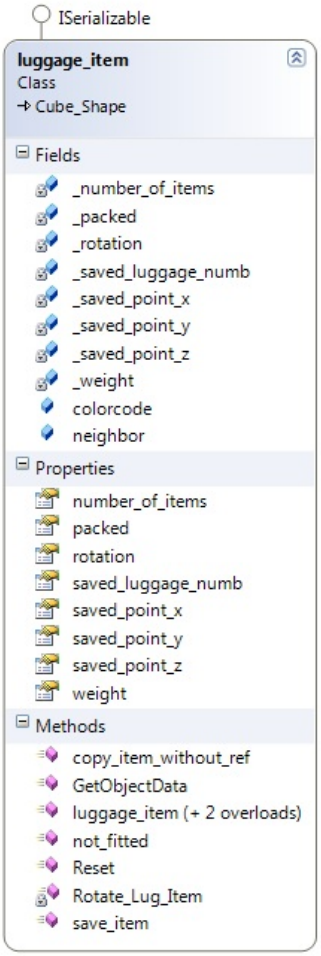

Figure 0.2: The class diagram of the luggage_item class

Using the classes

The luggage class and the luggage_item class is used in the program with a list. These lists of the classes is made by the initialization of the program as seen on Listing , and is passed thought to the other form. These lists contains the information of every suitcase, item and the result of the packing.

```
1 public List<luggage_item> luggage_items_to_pack = new List<luggage_item>();
2
3 public List<luggage> luggages_to_pack = new List<luggage>();
```

Listing 1: The initialization of the list containing the suitcases and luggages. (From the Main form)

0.1.2 Description of the code

The algorithm to pack has been develloped from the flowchart on Figure **??**. The following will describe the code and the extensions of the flowchart.

Reset and Sorting

As seen on the figure, the first thing which is done is to sort the suitcases and the items. Before sorting the suitcases and luggages, it resets the values changes when packing. This

is to ensure the algorithm has the right values when running it twice. The reset methods for the items and the suitcases are in their respective classes as seen on Figure 0.1 and Figure 0.2. A loop will call the reset methods in all suitcases and items in the lists. This is for example to reset the value of where there are saved items in the luggage. The items and luggage is also rotated so the luggage has the longest side as width as seen on Listing **??**. This is done, so the algorithm packs as optimal as possible. The items is likewise rotated, so the height is the smallest side. This will ensure that the algorithm will try to pack items lying, and not upright.

```
1                //Rotate so width is the longest side
2                int temp;
3                if (depth > width)
4                {
5                    temp = depth;
6                    depth = width;
7                    width = temp;
8                }
9
10               if (height > depth)
11               {
12                   temp = height;
13                   height = depth;
14                   depth = temp;
15               }
16
17               if (depth > width)
18               {
19                   temp = depth;
20                   depth = width;
21                   width = temp;
22               }
```

Listing 2: The code to rotate the luggage until the longest side is the width. The rotation is made by switching the values of 2 sides. (From the luggage class in the Main form.)

The method to sort the luggages and items will sort them by size, so the largest items will be attempted to be packed first in the biggest suitcases. This will give a better packing, because the smallest items are easier to fit than the big items.

Weight Distribution

The average distribution of weight in the luggage is calculated before going in the loop of the packing algorithm. In the flowchart on Figure **??** it will "Find next suitcase" when packing a item. This is not as simple as shown in the algorithm. The next suitcase is found by not only size, but also weight. It will start with the largest luggage and see if the weight of the suitcase, its content and the item to pack will exceed the average weight per luggage. If it does it will try to use the next luggages. If the item cannot be fitted in any luggage using the average weight, it will try again with only the maximum weight of the luggage in mind, so a item can be packed even though it exceeds the average weight per suitcase. The code of this can be seen on Listing **??**, where the stop_check_for_avg_weight variable checks if there should still be check for average weight. The lug_id variable if the index in the luggages list of the suitcase which is checked now. The lug_items_counter is the index of the item to pack. This loop will continue until the item is packed, or every possibility has been tried.

```
1 if (stop_check_for_avg_weight == true || (luggages_to_pack[lug_id].weight +
      luggage_items_to_pack[lug_items_counter].weight <= weight_per_luggage))
2 {
```

```
3     if (luggages_to_pack[lug_id].weight +
          luggage_items_to_pack[lug_items_counter].weight <
          luggages_to_pack[lug_id].max_weight)
4     {
```

Listing 3: The statements checking if the item can be fitted in this luggage by weight. (From the method pack_items in the Main form)

Checking For Fit in Pivot Points

If the item could be packed in a suitcase by weight, the algorithm will try to fit the items by weight. The next point is calculated by a sorted list of packing points. The list is sorted so the item is tried packed at the lowest points first. This is done in the find_next_pivot_point function in the luggage class. If the item placed in the given pivot point without exceeding the dimensions of the suitcase it will call the method in "check_item_for_fit" in the luggage class. This method will run through all the effected point and check if there are placed any items in the points. This method can be seen on Listing **??**, where the function will check every point in the array value. This value is an indicator of how many elements which is placed in the point (the value should of course never be higher than one).

```
1  bool check_for_fit = true;
2  for (int count_z = pivot_test_point.z; count_z < pivot_test_point.z +
       test_item.height; count_z++)
3  {
4      for (int count_y = pivot_test_point.y; count_y < pivot_test_point.y +
           test_item.depth; count_y++)
5      {
6          for (int count_x = pivot_test_point.x; count_x < pivot_test_point.x
               + test_item.width; count_x++)
7          {
8              if (value[count_x, count_y, count_z] != 0)
9              {
10                 check_for_fit = false;
11             }
12         }
13     }
14 }
15 return check_for_fit;
```

Listing 4: The method to check if the item can be placed at the pivot_test_point (from the luggage class in the Main form)

If the item cannot be fitted in the point in the luggage, the item will be rotated and tried fitted again. If every rotation possibilities has been tried, it will use the next packing point - or if no more packing points, the next suitcase.

Placing the item

If the item could not be packed in any suitcase, it will be added to a list of not packed items, which will be printed to the user later on. It will then be removed from the packing list, and the next item will be packed. If the item could be packed, it will be tried to move it lower in the suitcase until it meet another item in the check_for_space_z in the luggage class. This will ensure an optimal packing. The item will then be packed in the suitcase be adding 1 to the value in every point the item will fill out in the suitcase, using the place_item method in the luggage class. The packing points is saved to the item, so it have the information of, where it is saved. This is done by the function save_item from

the luggage_item class. The corners of the item is added to the list of packing points, and the list is sorted. The item is now saved and the next item can be packed.

When every item is packed

When all items have been packed (or placed in the list of not packed items), T´the next step in the algorithm is to generate the colors each item should be display with in the 3D atal: ret section Viewer. This code can be seen in section . The algorithm will at the end check if there has been a mistake during packing, so one of the values in the luggage has exceed 1. If it has, it will throw an exception. It will then check, if some of the items could not be packed. This will show an error message to the user, with the information of which elements which could not be packed as seen on Listing **??**.

```
1            if (Not_Packed_Items.Count > 0)
2            {
3                string error_message;
4
5                error_message = "All your items could not be packed. This
                     could be because of lack of space or too heavy items. The
                     rest of the items, has been packed. The items are:\r\n";
6
7                foreach (luggage_item not_packed_item in Not_Packed_Items)
8                {
9                    error_message += not_packed_item.name + "\r\n";
10               }
11
12               MessageBox.Show(error_message, "Fatal error",
                     MessageBoxButtons.OK, MessageBoxIcon.Error);
13           }
```

Listing 5: The code will return an error message to the user if some items could not be packed (from the method check_error_after_packing in the form Main)

0.1.3 Round up

The algorithm checks for a lot of different criteria, also a few more than descibed here. This will ensure a optimal packing of the items, so the most posible number of item can be fitted in the luggage. The algorithm has therefore been extended since the first design in section **??**. The extended, but still a bit simplified, flowchart can be seen on Figure 0.3.

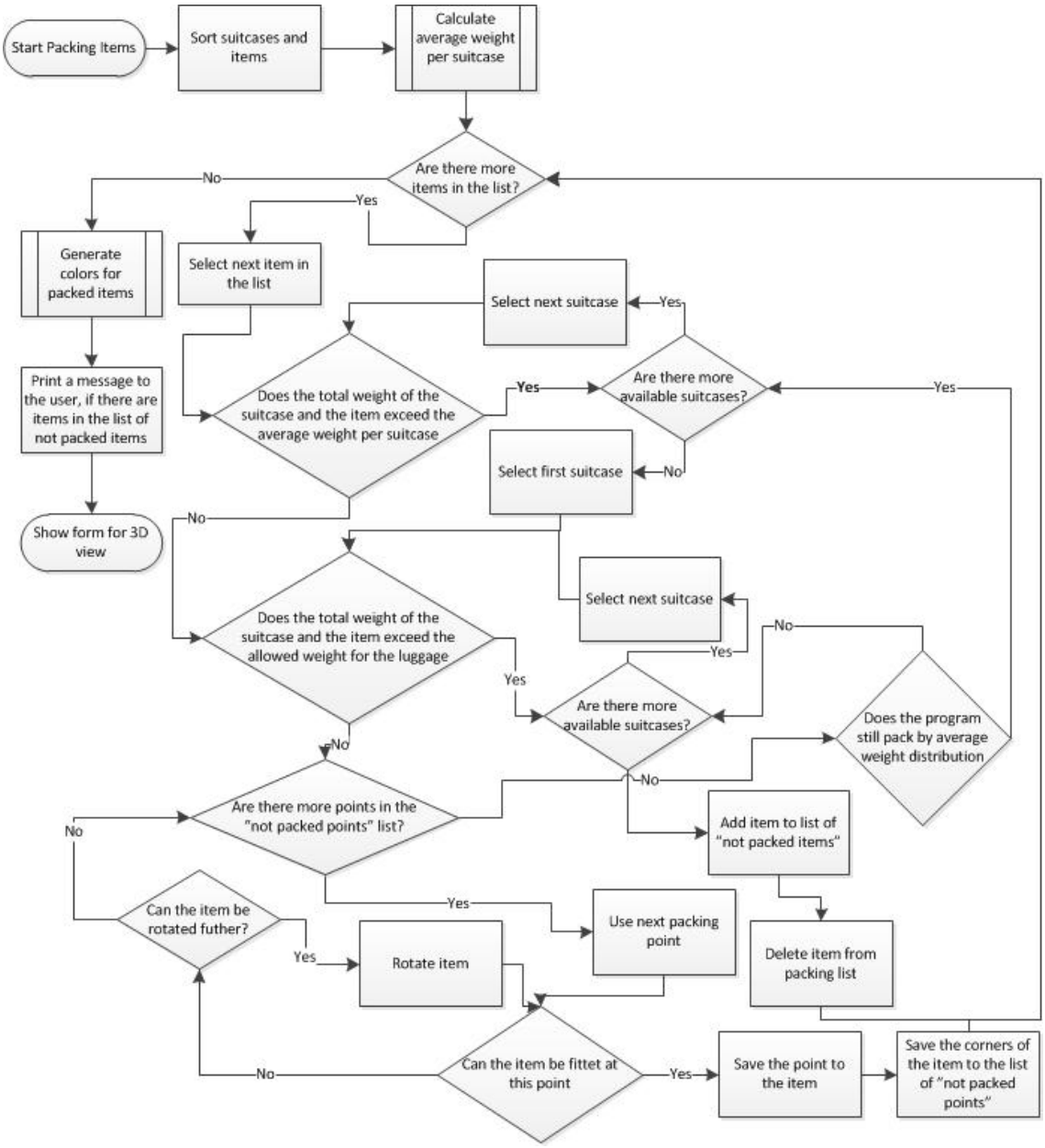

Figure 0.3: The flowchart for the packing algorithm