May 20, 2019

BIN-PACKING ALGORITHM

1. OVERVIEW

Description: In the bin packing problem, objects of different volumes must be packed into a finite number of bins or containers whose volume is V. The solution aims to find a best way that minimizes the number of bins used.

This project aim to testing various bin packing algorithm experimentally to determine the quality of the solutions they produce. The algorithms include Next Fit, First Fit, Best Fit, First Fit Decreasing (FFD), Best Fit Decreasing (BFD), Optimized First Fit and optimized Best fit. In this project, we set all objects we are packing floating point number between 0 and 1 which are generated uniformly at random. Also, all the bins are given fixed size 1.

Input: For each algorithm, the input of it includes items, assignment, and free space. Item is a vector of doubles between 0 and 1, representing the items need to be packed. Assignment is a vector of integers who have the same size as items and all numbers are initialized to be zero. The algorithm should modify this vector's element to store the assignment of bin for each item. The index of the vector corresponds to each item in Items vector, and the integer stored in Assignment implies which bin it should be put in. The free space is a vector of doubles, representing the amount of space left in each bin that each algorithm creates. The index of this vector represents the bin number in Assignment and double represents the number of space left.

Plot: After testing each algorithm on list of items of length n, we will draw a plot of the performance for each algorithm. The x-axis is the length of items N, where N increases from 1 to 50,000 with gap 100. The y-axis is the corresponding waste W(A) to each input size n, where **W(A)** is the number of bins used by the algorithm minus the total size of all items in the list – estimated how much space is wasted given a list of n items.

2. ALGORITHM ANALYSIS

This section will introduce the mechanism of implementation of each algorithm. Pseudocode code of some algorithms will be shown.

Next Fit

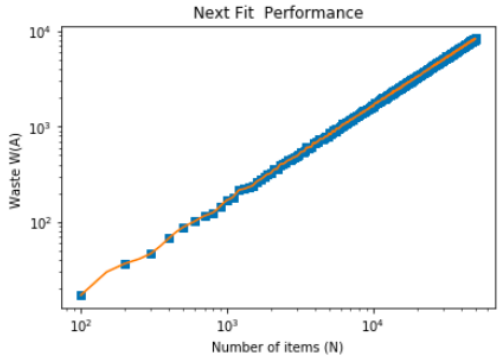
This algorithm checks to see if the current item fits in the current bin. If so, then place it there, otherwise start a new bin.

In pseudo-code, we have the following algorithm:

```
create the first bin and set it size as 1.0
current_bin  = 0  #integer represent the current bin number

for All objects i = 1, 2, . . . , n do
   if object i fits in current bin
       Pack object i in current_bin then pack the next object
   end if
   if Object i did not fit in any available bin then
       Create new bin and pack object i.
       Current bin += 1  #set current bin to the new created bin
   end if
end for

if freespace[0] eaqual to 1.0  # means no item is packed to the bin
    remove the first bin
end if
```

Algorithm implementation:

Create a bin first and set it size as 1.0. Mark the first bin as current bin we are packing. For each item in the list, test if the item fit the current bin we are using. If item fits in current bin, pack object in current bin then pack the next object. If not, create a new bin and pack the item into the new bin. Then set the new bin as current bin. Repeat procedures above until we pack all items. After the packing is finished, check if the first bin empty. If so, remove the first bin since no items is packed into any bin.



Plot analysis: We can easily tell from the plot that the waste of Next fit increases proportionally to the number of items being packed. We will compare the waste plot and linear regression of all 5 algorithms later in the final section.

First Fit

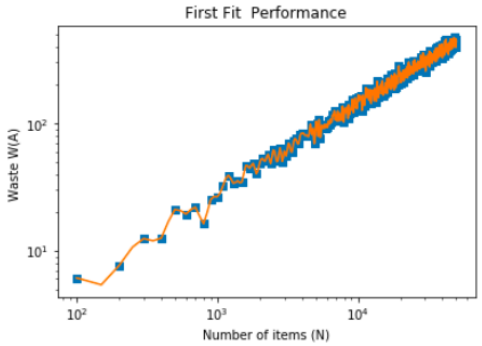
This algorithm check to see if an item fit in any bin. To find the bin, iterate through the bin list and pack the item into the first bin it finds. Start a new bin only if there is no available bin.

In pseudo-code, we have the following algorithm:

```
for All objects i = 1, 2, . . . , n do
   for All bins j = 1, 2, . . . do
       if Object i fits in bin j then
           Pack object i in bin j.
           Break the loop and pack the next object.
       end if
   end for
   if Object i did not fit in any available bin then
       Create new bin and pack object i.
   end if
end for
```

Algorithm implementation:

For each item in the list, iterate through all the existing bins to find a bin whose free space is larger than or equal to the item's weight. If we can find a bin that can hold the item, add the item to the bin, then break the loop and pack the next object. After iterating through all the existing bins, if we can't find any bin to hold it, create a new bin and pack object in the new bin. Repeat above procedure until we pack all the items.



Plot analysis: We can easily tell from the plot that the waste of First fit increases proportionally to the number of items being packed. The performance of it is better than the next fit since for each item, we check all the bins to find an available bin to pack rather than only checking the current bin in the next fit algorithm.

First Fit Decreasing

This algorithm first order the items by size in decreasing order, then run the First Fit Algorithm.

In pseudo-code, we have the following algorithm:

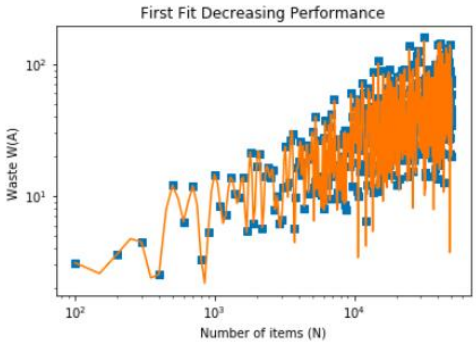
```
Store object and it's index as tuple in a vector # to keep the index of item even after sorting
Sort the vector in decreasing order by object's value

Apply First-Fit to the sorted list of objects
 for All sorted objects i  do
    for All bins j = 1, 2, . . . do
        if Object i's weight fits in bin j then
            Pack object i in bin j.
            assignment[object i's index] = j
            Break the loop and pack the next object.
        end if
    end for
    if Object i did not fit in any available bin then
        Create new bin and pack object i.
        assignment[object i's index] = size of bin
    end if
 end for
```

Algorithm implementation:

This algorithm is an optimized version of First Fit. For each item in the item list, I use a self-defined struct called my-pair to store the item's value and index. Then use a vector called sorted item to hold those pairs. After that, sorting the vector by the item's weight(value) in decreasing order. Then, apply the First Fit algorithm to the sorted vector as explained above. Below is the struct I use to keep index of item consistent before and after sorting.

```
struct mypair {
    int index;
    double items;
};
```

Plot.



Plot analysis: We can easily tell from the plot that the waste of First fit Decreasing increases proportionally to the number of items being packed. The performance of it is better than the First Fit since before packing, we sort the items in decreasing order, which means the first bin we find would be the tightest bin that fit the item, we will make full use of bin's space.

Best Fit

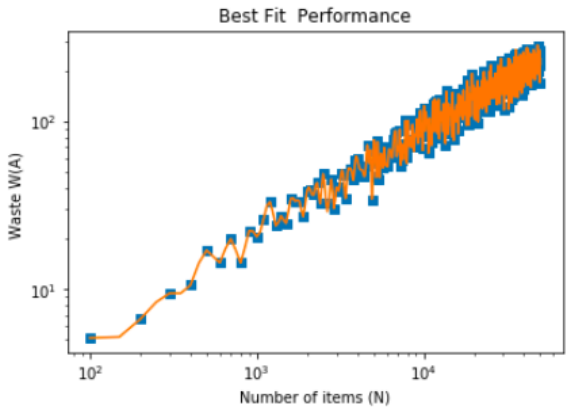
This algorithm will place an item in a bin where it fits the tightest. If it does not fit in any bin, then start a new bin.

In pseudo-code, we have the following algorithm:

```
for All objects i = 1, 2, . . . , n do
    best_bin = −1 (integer representing best bin for current item)
    best_remaining = 100(the remaining capacity of the best bin so far)
    for All bins j = 1, 2, . . . do
        if Object i fits in bin j
            if the remaining capacity of bin j < best_remaining so far
                best_remaining = RC of bin j
                best_bin = j
        end if
    end for
    if best_bin = −1 # means object fits no bin
        Create new bin and pack object i.
    end if
    if best_bin != −1   # means there is a best bin that object i can fit in
        Pack object i in best_bin
    end if
end for
```

Algorithm implementation:

Set variable best bin = -1 and best remaining = 100. For each item in the list, iterate through all the existing bins to find a bin whose free space is larger than or equal to the item's weight. If we find a bin that can hold the item, compare the RC of bin to the best remaining. If RC of the bin is less than the best remaining, set best remaining to current bin's RC and best bin to current bin. If not, continue iterating the remaining bins. After iterating through all the existing bins, if we can't find any bin to hold it (test if best bin = -1), create a new bin and pack object in the new bin. Repeat above procedure until we pack all the items.



Plot analysis: We can easily tell from the plot that the waste of Best fit Decreasing increases proportionally to the number of items being packed. The performance of it is better than the First Fit since we always find the tightest available bin to pack the items, which means we will make full use of existing bin's space.

Best Fit Decreasing

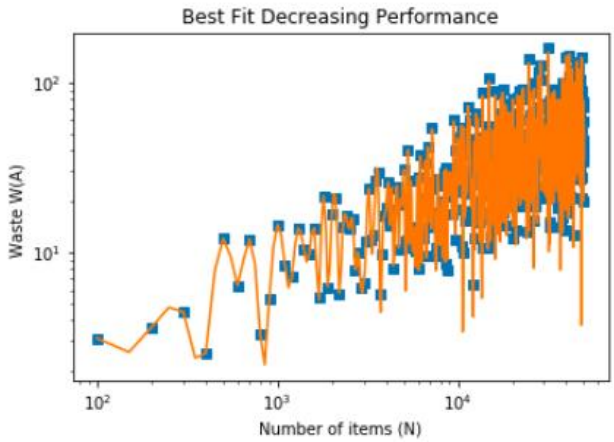
This algorithm first order the items by size in decreasing order, then run the Best Fit Algorithm.

In pseudo-code, we have the following algorithm:

```
Store object and it's index as tuple in a vector # to keep the index of item even after sorting
Sort the vector in decreasing order by object's value.

for All objects i in sorted vector do
    best_bin = -1 (integer representing best bin for current item)
    best_remaining = 100(the remaining capacity of the best bin so far)
    for All bins j = 1, 2, . . . do
        if Object i's weight fits in bin j
            if the remaining capacity of bin j < best_remaining so far
                best_remaining = RC of bin j
                best_bin = j
        end if
    end for
    if best_bin = -1 # means object fits no bin
        Create new bin and pack object i.
    end if
    if best_bin != -1 # means there is a best bin that object i can fit in
        Pack object i in best_bin
        Assignment[object i's index] = best_bin
    end if
end for
```

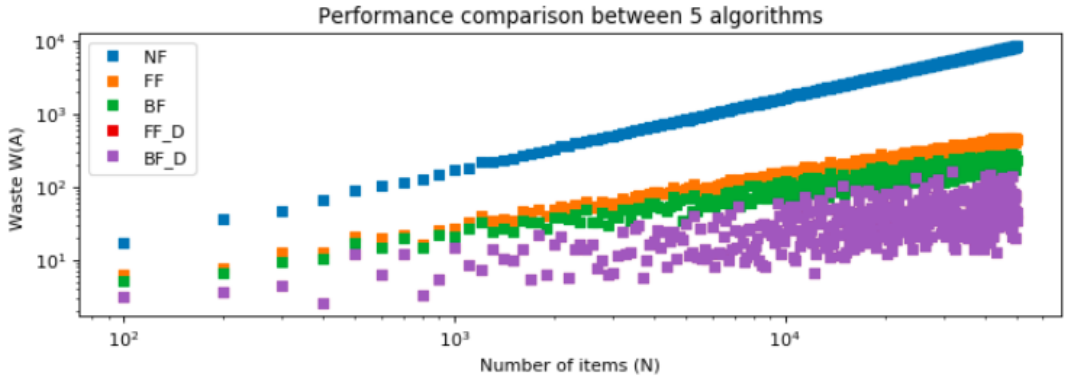
Algorithm implementation:

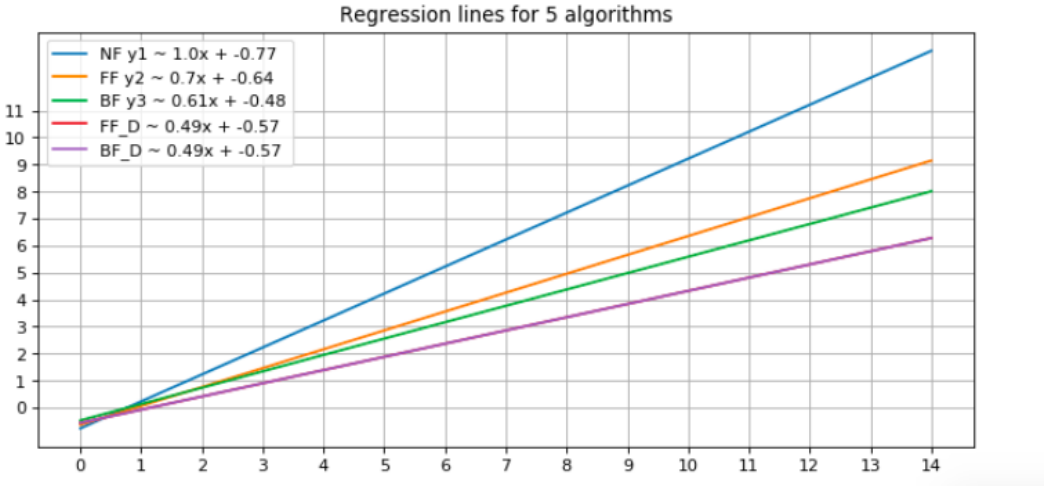
This algorithm is an optimized version of Best Fit. For each item in the item list, I use a self-defined struct called my-pair to store the item's value and index. Then use a vector called sorted item to hold those pairs. After that, sorting the vector by the item's weight(value) in decreasing order. Then, apply the Best Fit algorithm to the sorted vector as explained above.



Plot analysis: We can easily tell from the plot that the waste of First fit Decreasing increases proportionally to the number of items being packed.

3. PLOT ANALYSIS





The first graph shows again the waste of each algorithm. The second one showing the regression line of each algorithm in log-log scale.

Analysis:

FF Decreasing vs BF Decreasing: as we can tell from the graph, the plots of these two algorithms are the same so only the plot of BF_D shows up. The reason for that is they both sort the items list first in decreasing order. The first fit bin in decreasing order list is the exact best fit bin. The bins after the first fit bin would have larger free space since we have sorted the list in decreasing order. So the two algorithms have the same performance.

Five algorithms comparison: As in the waste-test-method said, log-log scale will be easier to compare each algorithm because it makes the power to appear in the slope. So we can tell from the second figure that the performance order(best to worst) is:

Best fit decreasing = First Fit decreasing > Best Fit > First Fit > Next Fit

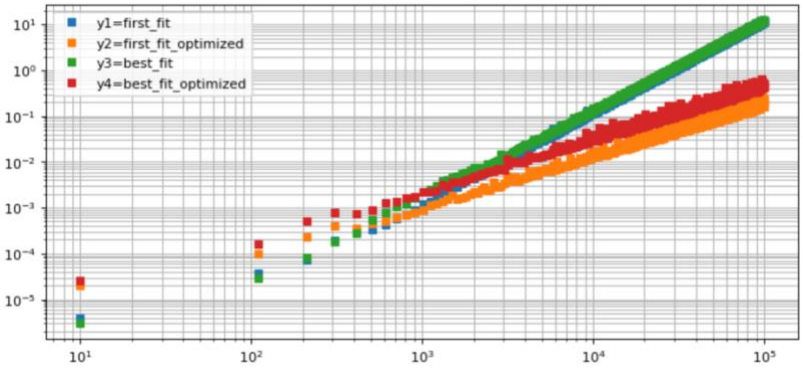
Next Fit is the worst since it only checks the current bin. If current bin doesn't fit, it doesn't check any other existing bin but create a new bin. The First Fit is the second worst one since although it does check the existing bin after the current bin fails, it only finds the first bin that fit which might not the best bin. The Best bin is better than First Fit since it always find the best bin among the existing bins. FF-D and BF-D are the best since it sort the list to make we pack large items first then smaller item. So we are always packing small items into bins that have "tightest" free space, which means we make full use of our existing bin.

4. CONCLUSION

Through this process, testing various bin packing algorithm experimentally to determine the quality of the solutions they produce. As our result shows, the performance of the five algorithms given the same input items can be ordered as below(best to worst) in most cases:

Best fit decreasing = First Fit decreasing > Best Fit > First Fit > Next Fit

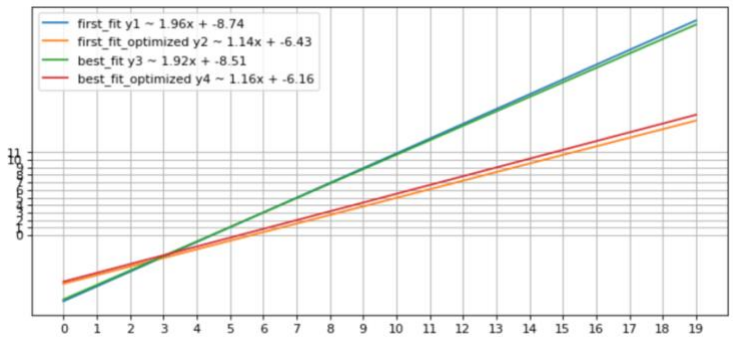
5. EXTRA PART: OPTIMIZED ALGORIT



HM

Performance of algorithms and optimized algorithm

Performance of algorithms and optimized algorithm(linear regression)



Two figures above shows the time performance for Best fit, Next fit and their optimized version. We can easily see from the plot that both optimized version of algorithm is better than the basic algorithms since the optimized version only takes O(nlogn) time while the basic version takes O(n^2) time.

Implementation analysis:

I used the AVL tree that provided in Piazza and made some changes to it. I will explain several places I modified and how I construct the AVL tree.

```
class FFNode : public forest::AVLTreeNodeBase<FFNode> {
public:
  FFNode() = default;
  FFNode(const int new_key, const double rc, const double brc ) : key(new_key), rc(rc), brc(brc){};
  ~FFNode() = default;

public:
  bool operator<(const FFNode &other) const { return key < other.key; }
  bool operator>(const FFNode &other) const { return key > other.key; }
  bool operator=(const FFNode &other) const { return key == other.key; }
public:
  void SetKey(const int new_key) { this->key = new_key; }
  void Setrc(const double  cap) { this->rc = cap; }
  void Setbrc(const double  sub) { this->brc = sub; }
  int getkey(){return key;}
  double get_rc(){return rc;}
  double get_brc(){return brc;}

private:
  int key;
  double rc;
  double brc;
};

class BFNode : public forest::AVLTreeNodeBase<BFNode> {
public:
  BFNode() = default;
  BFNode(const int new_key, const double rc, const double brc ) : key(new_key), rc(rc), brc(brc){};
  ~BFNode() = default;

public:
  bool operator<(const BFNode &other) const
  { if( rc == other.rc){return key < other.key;}
  return other.rc > rc ;}
  bool operator=(const BFNode &other) const
  { return key == other.key && brc == other.brc;
  }

public:
  void SetKey(const int new_key) { this->key = new_key; }
  void Setrc(const double cap) { this->rc = cap; }
  void Setbrc(const double sub) { this->brc = sub; }
  int getkey(){return key;}
  double get_rc(){return rc;}
  double get_brc(){return brc;}

private:
  int key;
  double rc;
  double brc;
};
```

I first define two node classes for First Fit optimized and best fit optimized, which are child class of AVL tree node base. They only have minor different in compare operator and equal operator. Since for FF AVL tree, we construct the tree by bin number of each node and for BF AVL tree, we construct the tree by BRC of each node.

Insertion

I made a minor modification on this part. The tricky part of this function is to maintain the BRC of each node. There are two places we need to update the BRC of each node: 1. after each time we call a recursion on children or after we add a leaf 2. each time we balance the tree.

Remove

We also need to adapt remove by updating BRC of each node. There are two places we need to update BRC of each node : 1. after we successfully find and remove this node 2. Each time we balance the tree.

```
root->Setbrc(largest_brc(brcval(root->mLeft), root->get_rc(),brcval(root->mRight)));
```

This is the way I update BRC of each node. Compare the BRCs of the node itself, the left node and right node and update the largest BRC to current node's value.

It will take too much space to illustrate how the tree is constructed in detail, I will explain it in algorithm implementation.

```
int size = items.size();
if(!size)
{
    return;
}

FFNode new_bin = FFNode(0,1.0,1.0);
free_space.push_back(1.0);
//There is at least one item insert new bin to the tree
AVLTree<FFNode> tree;
tree.insert(new_bin);
for(int i =0; i < size; i++)
{
    double bestrc  = tree.Largestrc();
    //check if there is a bin in the tree that fit the item
    if(items[i] > bestrc)
    {
        int new_Bin_rc = free_space.size();
        FFNode new_node = FFNode(new_Bin_rc,1.0,1.0);
        tree.insert(new_node);
        free_space.push_back(1.0);
    }
    //traverse the tree to put the item in the right place
    int key = tree.FFtraverse(items[i]);
    free_space[key] -= key;
    assignment[i] = key;
}
```

Above is the algorithm for FF optimized. If we can't find any item in item list, do nothing. If not, create a AVL tree and add a new bin node to it. Iterate through the item list. Tree.largestrc() returns the BRC of root node. If current item's weight is bigger than the BRC of root node, that means there is no bin fit the item. So create a new bin and insert to the tree. Then using FFtraverse function to return the bin number(key) of the right bin. Remaining work is trivial.