Project 2 - Testing Solution Quality

Overview

Project 2 involves testing various bin packing algorithms experimentally to determine the quality of the solutions they produce. Specific bin-packing algorithms to be implemeted and tested are the following five algorithms:

1. Next Fit (NF)
2. First Fit (FF)
3. First Fit Decreasing (FFD)
4. Best Fit (BF)
5. Best Fit Decreasing (BFD)

NOTE: Algorithms above except Next Fit algorithm have two implementations. The naive version has time complexity O(N^2). The faster implementation version uses balanced binary search tree (E.g. AVL Tree) ans has time complexity O(N logN).

Bin Packing Algorithms

This section first introduces bin packing problems ans then explains the implementation and time/space complexity of five different bin packing algorithms. Pseudocode/C++ code of algorithms will be shown.

Bin Packing Problem Definition

Given n items with sizes s1, s2, ..., sn such that $0 < si < 1$ for $1 \leq i \leq n$, pack them into the fewest number of bins of capacity 1. It is safe to assume that all items have weight smaller than bin capactiy and larger than 0.

There is no known polynomial time algorithm for its solution and it is conjectured that none exists.

Applications include:

- Loading of containers like trucks
- Placing data on multiple disks
- Storage a large collection of music onto tapes
- Job scheduling

Next Fit (NF)

Loop through the list of items in order. Check to see if the current item can fit into the current bin. If so, assign it to the current bin and update the remaining free capacity of the current bin. Otherwise, record the final free capacity of the current bin, start a new bin, assign the current item to the new bin, and update the free capacity of the new bin.

C++ code:

```
void next_fit(const vector<double>& items, vector<int>& assignment,
vector<double>& free_space) {
    int n = items.size();
    int bin = 0;
    double cap = 1.0;
    double fre = cap;

    for (int i = 0; i < n; ++i) {
        if (items[i] < fre || fabs(items[i] - fre) < __DBL_EPSILON__) {
            assignment[i] = bin;
            fre -= items[i];
        } else {
            ++bin;
            assignment[i] = bin;
            free_space.push_back(fre);
            fre = cap - items[i];
        }
    }
    if (fabs(fre - cap) >= __DBL_EPSILON__) { // NOT_EQUAL
        free_space.push_back(fre);
    }
}
```

This algorithm processes each item in the list only once so it has time complexity O(N). As of space complexity, it does not use extra space to store items, so it has O(1) space complexity.

First Fit (FF)

First fit algorithm has two implementations. The slower version implementation scans the list of bins in order to find the first bin that is large enough to hold the current item. I The faster version implementation stores bins in balanced binary search tree and uses the tree to search for the first bin that is large enough to hold the current item.

1. Slow version:

   Loop through each item in the list in order. Place the new item in the first bin that is large enough to hold it. A new bin is created only when the current does not fit in the previous bins.

   C++ code:

   ```
   void first_fit(const vector<double>& items, vector<int>& assignment,
   vector<double>& free_space) {
       int n = items.size();
       int bin = 0;
       double cap = 1.0;

       for (int i = 0; i < n; ++i) {
           int j;
           for (j = 0 ; j < bin; ++j) {
               if (items[i] < free_space[j] || fabs(items[i] -
   ```

```
 free_space[j]) < __DBL_EPSILON__) {
                free_space[j] -= items[i];
                assignment[i] = j;
                break;
            }
        }
        if (j == bin) {
            free_space.push_back(cap - items[i]);
            assignment[i] = bin;
            ++bin;
        }
    }
}
```

The inner loop of the above implementation, in the worst case, processes O(N^2) bins since for each item, it scans every previous bins. Overall, it has time complexity O(N^2) and space complexity O(1) since no extra space is used.

2. Fast version:

   TODO: AVL Tree

First Fit Decreasing (FFD)

A problem with the First Fit algorithms is that packing large items is difficult, especially if they occur late in the sequence. The solution is to first sort the items by size, from largest to smallest, then run the First Fit algorithm.

Pseudocode:

```
def first_fit_decreasing (items, assignment, free_space):
    sort(items) # in decreasing order
    first_fit(items, assignment, free_space)
```

This algorithm also has slow and fast implementation, since it is based on the First Fit algorithm. Its time complexity is depended on how First Fit algorithm is implemented. Sorting will not affect the overall time complexity because efficient sorting algorithms has time complexity O(N logN) which is less than O(N^2)

Best Fit

First fit algorithm has two implementations. The slower version implementation scans the list of bins in order to find the first bin that is large enough to hold the current item. I The faster version implementation stores bins in balanced binary search tree and uses the tree to search for the first bin that is large enough to hold the current item.

1. Slow version:

2. Fast version:

   TODO: AVL Tree

Best Fit Decreasing (BFD)

Experimental Testing

Test Method

Test Result Analysis