Advanced Data Structures and Algorithm Analysis

Project 6: Texture Packing

Date: 2024-11-30

2024-2025 Autumn&Winter Semester

Table of Content

Chapter 1: Introduction

1.1 Problem Description

The project require us to design **approximation algorithms** running in polynomial time to solve **Texture Packing** problem. We can regard it as a 2-dimension bin packing, with items("rectangle texture" in the problem) and bins("resulting texture" in the problem) having both width and height, but we only need a single bin with **bounded width** and **unbounded height**, and we should keep the bin with a (nearly) minimum height.

1.2 Backgound of Data Structures and Algorithms

1.2.1 FFDH Algorithm

Just like texture packing problem is the 2D version of bin packing problem, the **FFDH** (i.e. First-Fit Decreasing-Height) algorithm is also the 2D version of FFD algorithm in bin packing problem.

- It's an **offline algorithm**, which means that the algorithm doesn't process the input data unless it gets all input data, and in our algorithm, all items should be sorted by their height in a decreasing order.
- Before placing the current item, the algorithm scans the levels from bottom to top in the bin, then places the item in the first level where it will fit.
- A new level will be created only if the item does not fit in any previous ones.

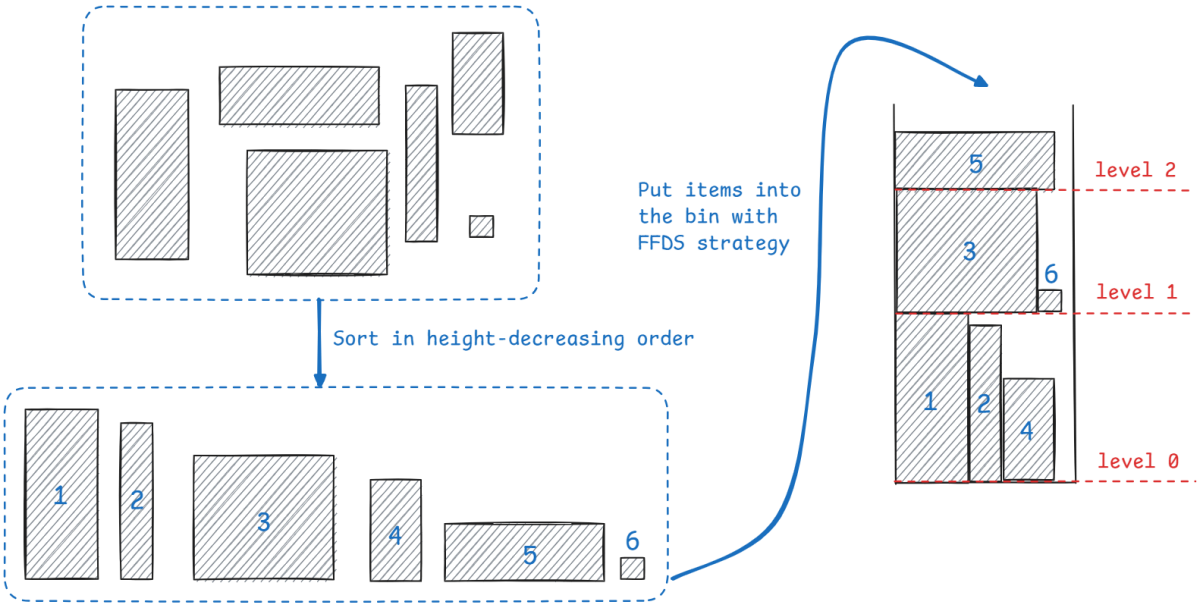

Figure 1: FFDH Approximation Algorithm

1.2.2 Advanced Approximation Algorithm

Chapter 2: Algorithm Specification

In this chapter, we will introduce approximation algorithms of FFDH(basic version) and (advanced version) in details, including aspects below, to enable readers have a comprehensive and thorough understanding of these algorithm.

- Description of approximation algotithms with pseudocodes.
- Calculation of approximation ratio of algorithm with proof.

Beware that our project doesn't use complex data structures and we only use arrays and structures in C, so it's meaningless to introduce them and we should only focus on the algorithm implementation.

2.1 FFDH Algorithm

Initially, let's take a look at the pseudocode of FFDH to gain a deeper insight into this kind of approximation algorithm.

Inputs:

- *W*: Fixed width of the bin(i.e. resulting texture)
- *n*: The number of items
- *rect*: Multiple rectangle texture, i.e. items
- *isDebug*: Flag of debug mode

Outputs:

- *curHeight*: the "minimum" height of the bin

Procedure: FFDH(*W*: **double**, *n*: **integer**, *rect*: **Item array**, *isDebug*: **bool**)

```
1  Begin
2      Sort rect[] by item's height in decreasing order
3      for item in rect[] do
4          for level in existing levels do
5              if level's width + item[i]→width ≤ W then
6                  put the item into this level and update the state
7                  break
8              end
9          if no level can fit the item then
10             create a new level
11             put the item into this level
12             update the state
13         end
14     end
15     print debug info if the user using the debug mode
16     return the current height of the bin as the "minimum" height
17 End
```

We can divide the procedure into four steps:

1. Sort all items by their height in decreasing order.
2. For all items, put them into the bin in the sorted order.
   - Scan all levels from bottom to top, find the first level that can accomodate the current item.
   - If no levels can fit it, then create a new level and put it into the new level.
3. (if necessary)Print the debug info, including:
   - the height-decreasingly sorted item data,

- the occupied-by-items width for each level,
- the positions of items.

4. Return the current height of the bin as the "minimum" height.

Now we should figure out the approximation ratio of this algorithm. We claim that FFDH algorithm is a **2-approximation algorithm**. Here is the proof:

Proof

Assume that:

- all items have been sorted by their height in decreasing order
- L: the list of items
- $OPT(L)$: the optimal solution(the actual minimum height) of texture packing
- $FFDH(L)$: the solution attained from the FFDH algorithm
- w_i: the width of the ith item
- h_i: the height of the ith item
  - h_0: the highest height
- W: the fixed width of the bin
- $A = \sum_{i=1}^{n} w_i \cdot h_i$: the total area of all items

First of all, we can determine the upper bound of the $OPT(L)$

- Obviously, $OPT(L) \geq h_0$
- $OPT(L) \geq \frac{A}{W}$, because the item on the right side represents the ideal result when all items fill the bin "tightly", i.e. no space waste in the bin.

Claim: $FFDH(L)$ satisfies the inequality below:

$$FFDH(L) \leq \frac{A}{W} + h_0 \leq 2OPT(L)$$

- let $h_r = FFDH(L) - h_0$, which is the remaining height of the bin in FFDH algorithm
- then we should prove that $h_r \leq \frac{A}{W}$. Because this approximation algorithm is also a **greedy algorithm**, which always put the highest item into the bin first, it ensures that the correctness of this inequality

As a consequence, the approximation ratio $\rho = \frac{FFDH(L)}{OPT(L)} \leq 2$, and we can guarantee that FFDH is a 2-approximation algorithm.

Beware that our proof is not very rigor, and the actual approximation ratio may be less than 2.

2.2 Advanced Approximation Algorithm

Chapter 3: Testing Results

3.1 FFDH Algorithm

3.1.1 Correctness Tests

3.1.2 Performance Tests

3.2 Advanced Approximation Algorithm

3.2.1 Correctness Tests

3.2.2 Performance Tests

Chapter 4: Analysis and Comments

4.1 Space Complexity

Conclusion:

- FFDH algorithm: $O(N)$
- advanced approximation algorithm:

Analysis:

- FFDH algorithm:
- advanced approximation algorithm:

4.2 Time Complexity

Conclusion:

- FFDH algorithm: $O(N^2)$
- advanced approximation algorithm:

Analysis:

- FFDH algorithm:
- advanced approximation algorithm:

4.3 Further Improvement

Appendix: Source code

5.1 File Structure

```
.
├── README.md
├── code
│   ├── Makefile
│   ├── README.pdf
│   ├── build
│   ├── scripts
│   │   ├── getStopWord
│   │   ├── getStopWord.cpp
│   │   ├── html2txt.py
│   │   ├── iist_diagram.py
│   │   ├── invIndexFunc.cpp
│   │   ├── invIndexHeader.h
│   │   ├── invIndexSearch.cpp
│   │   ├── invIndexTest.cpp
│   │   ├── search_main.cpp
│   │   ├── search_test.cpp
│   │   └── wordStem
│   └── data
│       ├── file_word_count.txt
│       ├── search_test
│       ├── shakespeare-master
│       ├── shakespeare_works
│       ├── stop_words.txt
│       ├── inverted_index_tests
│       ├── txt_title.txt
│       ├── word_count.txt
│       └── word_docs.txt
└── documents
    └── report-p1.pdf
```

5.2 ttpHeader.cpp

```
#define ITEMNUM 10000                          // Maximum number
of items
#define ITERATIONS 10000                       // Iteration time
#define INPUTDIR "../test/input.txt"           // Directory of the
input file

// structure of a single item
typedef struct item {
    double width;
    double height;
} Item;

// First Fit by Decreasing Height, a basic 2-approximation
algorithm
// W: Fixed width of the resulting texture
// n: The number of items
// rect: Items
// isDebug: Flag of debug mode
int FFDH(double W, int n, Item rect[], int isDebug);
```

5.3 ttpMain.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include "ttpHeader.h"

double Width;                  // Fixed width of the resulting texture
int n;                         // The number of items
Item rect[ITEMNUM];            // Items
FILE * fp;                     // File pointer
int isDebug;                   // Flag of debug mode
int isTiming;                  // Flag of timing mode
clock_t start, stop;           // Record of start and stop time of
the approximation algorithm

// Input handler
void getInput(int argc, char * argv[]);
```

```
// Print the timing infomation
void printTime(clock_t start, clock_t end);

int main(int argc, char * argv[]) {
    int i;
    double miniHeight;   // Result

    getInput(argc, argv);                                    // Input

    // Execution of approximation algorithm
    if (isTiming) {
        start = clock();                                     //
Start timing
        for (i = 0; i < ITERATIONS; i++)                     //
Multiple execution of algorithms
            miniHeight = FFDH(Width, n, rect, 0);
        stop = clock();                                      //
Stop timing
        printTime(start, stop);                              //
Print the timing infomation
    } else {
        miniHeight = FFDH(Width, n, rect, isDebug);
        printf("The minimum height: %.2f\n", miniHeight);    //
Output
    }

    return 0;
}

// Input handler
void getInput(int argc, char * argv[]) {
    int i;
    int choice = 0;    // Choice whether receiving terminal input
or file input

    // If using command arguments
    if (argc > 1) {
        for (i = 1; i < argc; i++) {
            // Handle debug mode
```

```
                if (!strcmp(argv[i], "-d") || !strcmp(argv[i],
"--debug"))
                    isDebug = 1;
                // Handle file input mode
                else if (!strcmp(argv[i], "-f") || !strcmp(argv[i],
"--file"))
                    choice = 1;
                // Handle timing mode
                    if (!strcmp(argv[i], "-t") || !strcmp(argv[i],
"--timing"))
                    isTiming = 1;
            }
        }

        if (!choice) {    // Terminal input
            scanf("%lf", &Width);
            scanf("%d", &n);
            for (i = 0; i < n; i++) {
                scanf("%lf%lf", &rect[i].width, &rect[i].height);
            }
        } else {          // File input
            fp = fopen(INPUTDIR, "r");      // Read the input file
            fscanf(fp, "%lf", &Width);      // Similar to terminal input
            fscanf(fp, "%d", &n);
            for (i = 0; i < n; i++) {
                fscanf(fp, "%lf%lf", &rect[i].width, &rect[i].height);
            }
        }
}

// Print the timing infomation
void printTime(clock_t start, clock_t end) {
    clock_t tick;       // Ticks
    double duration;    // Duration(unit: seconds)
    int iterations;

    iterations = ITERATIONS;    // Set iteration time, for obvious
timing result
    tick = end - start;          // Calculate tick numbers
```

```
    duration = ((double)(tick)) / CLOCKS_PER_SEC;       // Calculate
the total duration of multiple execution of the algorithm

    // Print the timing info
    printf("\nTiming Result:\n");
    printf("Iterations: %d\n", iterations);
    printf("Ticks: %lu\n", (long)tick);
    printf("Duration: %.2fs\n", duration);
}
```

5.4 FFDH.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include "ttpHeader.h"

double curWidth[ITEMNUM];                   // Array recording the
total item width in each level
int pos[ITEMNUM];                           // Array recording the
position(which level) of every item
int level = 0;                                // Current level of
the resulting texture

int cmp(const void * a, const void * b);    // Comparator for
decrease-order quicksort function
void printDebugInfo(int n, Item * rect);    // Print the debug
infomation when using "--debug" command argument

// First Fit by Decreasing Height, a basic 2-approximation
algorithm
// W: Fixed width of the resulting texture
// n: The number of items
// rect: Items
// isDebug: Flag of debug mode
int FFDH(double W, int n, Item rect[], int isDebug) {
    double curHeight = 0;                    // Current height of
the resulting texture
    int i, j;
```

```
    // Initially, sort all items by their heights in decreasing order
    qsort(rect, n, sizeof(rect[0]), cmp);

    // initialize elements in curWidth to zero
    for (i = 0; i < n; i++)
        curWidth[i] = 0;

    // Handle all items
    for (i = 0; i < n; i++) {
        // Find the first fit existing level
        for (j = 0; j < level; j++) {
            // Find it!
            if (curWidth[j] + rect[i].width ⩽ W) {
                curWidth[j] += rect[i].width;     // Update the width of current level
                pos[i] = j;                       // Record the position of the item
                break;
            }
        }
        if (j < level)
            continue;

        // If not found
        ++level;                                  // Create a new level
        curWidth[level - 1] = rect[i].width;      // Update the width of current level
        curHeight += rect[i].height;              // Update the current height
        pos[i] = level - 1;                       // Record the position of the item
    }

    // Print the debug infomation when using "--debug" command argument
    if (isDebug)
        printDebugInfo(n, rect);
```

```
    // Return the current height of the resulting texture as the
minimal height
    return curHeight;
}

// Comparator for decrease-order quicksort function
int cmp(const void * a, const void * b) {
    const Item dataA = *(const Item*)a;
    const Item dataB = *(const Item*)b;

    return dataB.height - dataA.height;    // Beware of the order
of subtraction!
}

// Print the debug infomation when using "--debug" command
argument
// n: The number of items
// rect: Items
void printDebugInfo(int n, Item * rect) {
    int i;
    printf("Debug Info:\n");

    // 1. Print the height-decreasingly sorted item data
    printf("Height-decreasingly sorted item data:\n");
    for (i = 0; i < n; i++) {
        printf("%d: %.2f, %.2f\n", i, rect->width, rect->height);
        ++rect;
    }

    // 2. Print the occupied-by-items width for each level
    printf("\nTotal level: %d\nWidth:\n", level);
    for (i = 0; i < level; i++) {
        printf("%d level: %.2f\n", i, curWidth[i]);
    }

    // 3. Print the positions of items
    printf("\nPosition:\n");
    for (i = 0; i < n; i++) {
```

```
        printf("Item %d: level %d\n", i, pos[i]);
    }

    // Deviding line
    printf("====================================\n");
}
```

Declaration

We hereby declare that all the work done in this project titled "Texture Packing" is of our independent effort as a group.