Author: Adam Svitek
Mail: xsviteka@stuba.sk

AZA - Practical implementation and analysis of an algorithms

Adam Svitek

Author: Adam Svitek
Mail: xsviteka@stuba.sk

Contents of the document

1. Introduction....3
2. Tasks....3
   A. Scheduling with deadlines....3
      Algorithm outline....3
      Example....3
      Time Complexity analysis....4
   B. Scheduling with deadlines, Using DisjointSet....4
      Algorithm outline....4
      Time Complexity analysis....5
   C. File merging using Greedy algorithm....5
      Algorithm outline....5
      Time Complexity analysis....6
      Examples....6
   D. Huffman coding....,7
      Canonical coding....8
      Frequency-based Approximation....8
      Examples....9
      Time Compexity analysis....10
3. Future improvements....11
4. Conclusion....11
5. References....11

Author: Adam Svitek
Mail: xsviteka@stuba.sk

1.) Introduction

In this document I would like to present my implementation of the practical assignment consisting of four different algorithmic problems and their further analysis:

- Scheduling with deadlines algorithm
- Scheduling with deadlines algorithm using disjoint set
- Files merging using greedy Algorithm
- Generate (Canonical Huffman Coding)

2.) Tasks

A. Scheduling with deadlines

Scheduling with deadlines is a greedy algorithm that aims to maximize the total profit from set of jobs, where each job in the list has its own profit and deadline. The algorithm makes sure that each job from list can be completed within its deadline

Algorithm Outline:

First user is prompt to input list of jobs with its deadlines and profits (as many as he wants).

In next step the program sorts the array based on jobs profits. This step ensures that more profitable jobs are at the front of the list.

Next is to go through the jobs and add them to list so they can be completed no later than their deadline. Each job is being checked if there is empty space for it, if yes, the job is added to the list, if not, job is rejected, and program continues.

Output of this program is set of Jobs with total profit and total profit achieved.

Example:

The input would consist of 7 jobs with deadlines and profits accordingly:

Job	Deadline	Profit
1	2	40
2	4	15
3	3	60
4	2	20
5	3	10
6	1	45
7	1	55

The max deadline for our jobs is 4 so we can only take 4 jobs into consideration. After scheduling with deadlines algorithm, the optimal

sequence of the jobs would be: Job 7, Job 1, Job 3, Job 2 and the profit reached would be 170.

Time Complexity analysis:

Sorting: sorting function sorts jobs based on their profits (if profits are equal then deadlines are compared). Sort function has time complexity of **O(nlogn)**

Finding the max deadline: this operation compares deadlines of each item of the list, so time complexity is **O(n).**

Scheduling jobs: in scheduling jobs I am iterating through the whole list of jobs which is time complexity **O(n)** for n number of jobs. Each job is then added to the JobSchedule list on its free slot based on its deadline. If such slot is not free the job is skipped.

Printing the result: this function only iterates through list of scheduled jobs, so time complexity is **O(n).**

Overall Time Complexity:

The most time-consuming operations are sorting which is **O(nlogn)**

B. Scheduling with deadlines, use of Disjoint set

Scheduling with deadlines can be upgraded to use less time with use of disjoint set data. This method also makes sure that jobs are added as late as possible within their deadline.

With use of union-find we are able to cut down time complexity for finding optimal scheduling of jobs

Algorithm outline:

At the beginning disjoint set is created where each slot is represented as set pointing to itself. The number of sets is max deadline + 1(because we are going from 0)

The beginning of the code is similar to code in previous program where Jobs are sorted based on their profit. After this step program is trying to find latest available slot using small operation. If such slot is available program assigns this to the job and merge current slot with previous slot using unionSets operation.

The union and find operations are repeated until either all sets are full or there is no job to

The key difference in this program and the previous is that previous only sorted items into their slots. For example: If I have 5 empty slots and 4 jobs

with the same deadline 5 only one the job with highest profit would be done. On the other hand this implementation ensures that if there is empty space and deadline is not less than empty space, the job will be added

The output of this code is also sequence of scheduled jobs and total profit achieved.

Time complexity analysis:

Sorting: the sorting algorithm works as previous one with time complexity **O(nlogn)**

Scheduling: in scheduling algorithm program is iterating thorough each job in the list what is time complexity **O(n)**. Inner operations of this loop are small and unionSets.

Small function: is function for finding smallest available slot and compresses the path for future calls of small function. With path compression the time complexity is **O(a(n))** = Inverse Ackermann function which is close to constant O(1).

unionSets: has simple time complexity of **O(1)** because it only merges two sets by making root of one sat point to another set

Overall time complexity:

The most time-consuming part of the code is again sorting with time complexity of **O(nlogn)**.

C. File merging using Greedy algorithm

In this task the goal is to merge files(represented as integers) with minimal number of record. In each step of merging 2 files are merged together until all files are merged.

Memory calculation is the output of the program. The memory is sum of every merge operation.

To minimize memory usage program combines 2 smallest files each time

Algorithm outline:

At the beginning of program user is prompted to input the size of array which stores the files. After that user inputs the numbers representing the file sizes. They do not have to be in order.

The next step is while function which goes until a the size of array is 1(all files are merged) inside the function while, at the beginning, array is sorted as first thing.

The main logic takes 2 smallest files(first 2) and sum the together then program erases them from the array and adds new file (2 combined) at the and. This continues until all files are merged.

Author: Adam Svitek
Mail: xsviteka@stuba.sk

Time complexity analysis:

Input numbers: The time complexity of this part of the code is **O(n)** because its simple for function that iterates n time based on the number of files

Sorting and merging: The sorting function is imported from library algorithm. This function only sorts elements from smallest to biggest with time complexity of **O(nlogn)** but since this is in while loop the overall time complexity for sorting is **O(n^2logn)**. Merging, erasing and finally adding is used in every iteration so the time complexity is **O(n^2)** because it traverses through vector each time.

Overall time complexity:

The overall time complexity is **O(n^2logn)** where n is number of files in array

Examples:

The part of the assignment is to show 2 different inputs and analyse each step along the way.

Example 1:

Array size: 10

File sizes: 3,6,12,15,28,1,20,5,2,100

Memory: 0

Steps:

Sort(1,2,3,5,6,12,15,20,28,100)

Take first 2 merge and add ad the end =(3,5,6,12,15,20,28,100,3)

Sort(3,3,5,6,12,15,20,28,100), memory = 3

Take first 2 merge and add ad the end = (5,6,12,15,20,28,100,6)

Sort(5,6,6,12,15,20,28,100), memory = 9

Take first 2 merge and add ad the end = (6,12,15,20,28,100,11)

Sort(6,11,12,15,20,28,100) , memory = 20

Take first 2 merge and add ad the end = (12,15,20,28,100,17)

Sort(12,15,17,20,28,100) , memory = 37

Take first 2 merge and add ad the end = (17,20,28,100,27)

Sort(17,20,27,28,100) , memory = 64

Take first 2 merge and add ad the end = (27,28,100, 37)

Sort(27,28,37,100) , memory = 101

Take first 2 merge and add ad the end = (37,100,55)

Sort(37,55,100) , memory = 156

Take first 2 merge and add ad the end = (100,92)

Sort(92,100) , memory = 248

Take first 2 merge and add ad the end = (192)

Memory = 440(Output)

Example 2:

Array size: 6

File sizes: 12,8,3,1,6,20

Memory: 0

Steps:

Sort(1,3,6,8,12,20)

Take first 2 merge and add ad the end = (6,8,12,20,4)

Sort(4,6,8,12,20), memory = 4

Take first 2 merge and add ad the end = (8,12,20,10)

Sort(8,10,12,20) , memory = 14

Take first 2 merge and add ad the end = (12,20,18)

Sort(12,18,20) , memory = 32

Take first 2 merge and add ad the end = (20,30)

Sort(20,30) , memory = 62

Take first 2 merge and add ad the end = (50)

Memory = 112(Output)

D. Huffman coding

One of the last exercises is to make Huffman code without constructing Huffman tree and be able to code and decode text.

Task was to create 2 algorithms how codes will be generated, first being canonical Huffman coding where frequency of letters is combined from smallest to largest and based on this the depth of each letter(length of code) is calculated. The second approach was Frequency-based Approximation.

Frequency-based Approximation is simplified method how to generate Huffman codes where each letter has calculated height by its own frequency and total frequency what is sum of each letter frequency

Author: Adam Svitek
Mail: xsviteka@stuba.sk

Canonical Huffman Coding:

The canonical as mentioned is more complex approach but gives better results than Approximation method.

Algorithm Outline:

The program at the beginning creates an array of Items consisting of letter, frequency, height and code.

First the depths of each letter are calculated to represent the length of each letter in bits. Program takes 2 smallest frequencies and sums them together and adds +1 depth to both letters. These letters are then deleted from the list and substituted by the combination of the letters with new freq. This continues until all items are merged into one and all depths are calculated.

Next step is to encode letters according to their depths. At first letters are sorted in nondecreasing order. The second step is to generate codes. Each code is unique and can't be repeated. Code's length as mentioned above are generated based on the depth. For example:

Letter A, depth 1, code 0
Letter B, depth 3, code 100
Letter C, depth 3, code 101
Letter D, depth 3, code 110
Letter E, depth 4, code 1110
Letter F, depth 4, code 1111

At the end user is prompted to enter combination of available letter to be encoded and program outputs the encoded binary combination and the decoded original sequence (same for Frequency-Based Approximation)

Frequency-based Approximation:

This method is easier to execute but not as reliable as the Canonical because the calculations are only (as name suggests) approximations

Algorithm outline:

The Frequency based function is implemented in the same code so beginning stays the same as before. The difference is that items are not merged but their depth value is only calculated through simple equation.

The equation: depth = $-\log_2(\text{frequency}/\text{frequency\_all})$

This equation gives us number and after rounding up it gives us approximate depth(length) of the letter.

In the next step codes are generated for items based on their depth(length) as in previous algorithm

Encoding:

The process where from users input program generates code in binary based on letters and codes.

Author: Adam Svitek
Mail: xsviteka@stuba.sk

Example: User input: ABACF; encoded sequence: 010001011111

Decoding:

The process which work in reverse to encoding. The encoded sequence is again rephrased back into original form. The buffers stores bits until it matches any valid Huffman code then code is replaced with corresponding letter.

Examples:

Canonical:

Letter: A,B,C,D,E,F

Frequencies : 45, 13, 12, 16, 9, 5

Sort(F:5,E:9,B:13,C:12,D:16,A:45)

Merge(C:12,B:11,D:16,A:45,FE:14) depths:A:0,B:0,C:0,D:0,E:1,F:1

Sort(C:12,B:13,FE:14,D:16,A:45)

Merge(FE:14,D:16,A:45,BC:25) depths:A:0,B:1,C:1,D:0,E:1,F:1

Sort(FE:14,D:16,BC:25,A:45)

Merge(BC:25,A:45,DEF:40) depths:A:0,B:1,C:1,D:1,E:2,F:2

Sort(BC:25,DEF:40,A:45)

Merge(A:45,BCDEF:65) depths:A:0,B:2,C:2,D:2,E:3,F:3

Sort(A:45,BCDEF:65)

Merge(ABCDEF:110) depths:A:1,B:3,C:3,D:3,E:4,F:4

Codes:

A = 0, B = 100, C = 101, D: 110, E: 1110, F:1111

Phrase: ABBCAF ,code = 0 100 100 101 0 1111

Author: Adam Svitek
Mail: xsviteka@stuba.sk

Frequency-based Approximation:

Letter: A,B,C,D,E,F

Frequencies : 45, 13, 12, 16, 9, 5

A = $-\log_2(45/110)$ = 1,15 rounded up = 2

B = $-\log_2(45/110)$ = 2,94 rounded up = 3

C = $-\log_2(45/110)$ = 3,06 rounded up = 4

D = $-\log_2(45/110)$ = 2,64 rounded up = 3

E = $-\log_2(45/110)$ = 3,47 rounded up = 4

F = $-\log_2(45/110)$ = 4,32 rounded up = 5

Codes:

A = 00, B = 010, C = 011, D: 1000, E: 1001, F:10100

Phrase: ABCDEF ,code = 0 010 011 1000 1001 10100

Time complexity analysis:

Depth Canonical: The sorting of elements inside this function has time complexity of **$O(n^2 \log n)$** because sorting is repeated until only one element remains. Updating depth takes **$O(n^2)$** because after merging it has to find item in list of items and update its height

Depth Frequency-based Aprox: this method is more time efficient because it only iterates through the list once and calculate the depth. This means that the time complexity is only **O(n)**

Encoding text: For each character, the program iterates through the n items in the list to find corresponding code. In worst case time complexity is **O(m.n)**where m is list of characters and n list of codes

Decoding text: For each bit the program is trying to match the buffer with n codes in the list of codes. If text has k bits it has time complexity of O(k.n). If we assume that k is approximately equal to m.logn(each character has logn bits) this means that time complexity is **O(m.logn.n)**

Overall time complexity:

In canonical the dominant aspect is **$O(n^2 \log n)$** if n dominates m(n = number of characters, m = number of characters to encode). But if m dominates m the time complexity is based on decoding with time complexity **O(m.logn.n)**

Author: Adam Svitek
Mail: xsviteka@stuba.sk

In Frequency-based Approximation the dominant part is always decoding with time complexity **O(m.logn.n)** because the code generating method is O(n).

3.) Future improvements

While working on assignments I noticed that the key time limitation is sorting which can be improved thus improving the time complexity of whole code. The use of heap could help reduce this problem and speed up the process.

4.) Conclusion

This document presents a practical implementation and analysis of algorithms related to scheduling and encoding, focusing on efficient solutions for real-world problems. The key achievements were to be able to implement algorithms and be able to analyse their functions and time complexity.

5.) References

https://www.geeksforgeeks.org/introduction-to-disjoint-set-data-structure-or-union-find-algorithm/
https://www.geeksforgeeks.org/job-sequencing-problem-using-disjoint-set/
https://www.geeksforgeeks.org/canonical-huffman-coding/
https://www.geeksforgeeks.org/time-complexities-of-different-data-structures/