Author: Nikolas Knapik
Date: 6.12.2024

2024

Analysis of an algorithm

PRACTICAL ASSIGNMENT

NIKOLAS KNAPIK

Table of Contents

1. Scheduling with Deadlines

Problem description:

This problem involves scheduling a set of jobs. Each job is characterized by a deadline in which it needs to be finished, and a profit it generates. The objective of the code is to calculate an optimal sequence of jobs that maximizes the total profit. Two jobs cannot be set to one deadline. If a cannot meet its deadline, it is rejected.

Algorithm overview:

The solution employs a greedy approach to maximize profit. The key steps are:

1. **Initialization:** The list of jobs include ID, deadline, and profit for every job. An array representing time slots is initialized using maximum deadline which is acquired from the jobs.
2. **Sort jobs:** Jobs are arranged based on their profit to ensure that higher profit jobs are prioritized for scheduling.
3. **Schedule jobs:** For each job in sorted list, the program checks if the slot corresponding to its deadline is available. If the slot is free, the job is assigned to the deadline and its profit is added to total profit. If the slot is not free, the job is skipped.
4. **Output the results:** The program prints the sequence of scheduled jobs and displays the total profit from the schedule jobs.

Data structures:

1) **Job list:** Stores the details of each job. It includes the ID, deadline and profit.
2) **Scheduled array:** Represents available slots for scheduling. It is initialized with -1 to indicate empty slots. Each index corresponds to a specific time slot.

Time complexity:

1) **Sorting jobs:** Sorting the jobs by profit takes:

$$O(nlogn)$$

2) **Finding maximum deadline:** Iterating through the list of jobs to find maximum deadline takes:

$$O(n).$$

3) **Scheduling jobs:** For each job, checking and assigning a slot takes ***O(1).*** Total complexity for this step is:

$$O(n).$$

4) **Overall complexity:**

$$O(n \log n) + n) = O(n \log n)).$$

Example:

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 output for such input would be :

$Optimal\ sequence\ of\ jobs$: $Job7\ Job1\ Job3\ Job2$

$Total\ profit$: 170

2. Deadlines Using Disjoint Set Union:

Problem description:

The second program solves the job scheduling problem with an enhanced algorithm that uses the **Disjoint Set Union (DSU)** data structure to efficiently manage available slots. Each job must be completed before its deadline, and only one job can be scheduled in a slot. The objective is to maximize the total profit by scheduling jobs optimally.

Algorithm overview:

The solution uses a greedy approach combined with the DSU data structure.

1) **Sort jobs by profit:** Arrange jobs according to their profit to prioritize higher-value jobs.
2) **Initiate DSU:** Create a DSU structure where each slot is initially its own parent. The DSU tracks the latest available slot for each job.
3) **Scheduling jobs using DSU:** For each job the program uses the *find* operation to locate the latest available slot. If the slot is available, it assigns the job to that slot and merges the slot with the next earlier slot using the *union* operation. The program will then add the job's profit to the total profit
4) **Outputting the results:** The program prints the sequence of scheduled jobs in the order they were assigned to slots and displays the total profit.

Data structures:

1) **Disjoint set union:** Tracks the availability of time slots. It supports two operations *find(x)* which finds parent of the set containing x. And *union(x, y)* which merges the sets containing x and y.
2) **Job list:** Stores details for each job: ID, deadline and profit.
3) **Schedule array:** Used for storing the job IDs in their assigned slots.

Time complexity:

1) **Sorting jobs:** Sorting the jobs where n is the number of jobs takes :

$$O(n \log n)$$

2) **DSU operations:** Each *find* or *union* operation has a time complexity where *d* is the maximum deadline and α is the inverse Ackermann function:

$$O(\alpha(d))$$

3) **Overall complexity:**

$$O(n \log n)$$

Example:

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

Output for this input will be :

$Optimal\ sequence\ of\ jobs$: $Job7\ Job1\ Job3\ Job2$

$Total\ profit$: 170

3. Greedy Algorithm for Minimizing File Merge Operations

Problem description:

The program aims to minimize the number of operations to merge n files into a single file. Each operation involves merging two files, and the cost of each merge is equal to the sum of the sizes of the two files. The objective is to determine the minimum total cost of merging all the files.

Algorithm overview:

The problem is solved using a greedy algorithm combined with a priority queue.

1) **Initialization:** A list of file sizes is provided by the user. A min-heap is initialized to store file sizes. It ensures that the smallest file sizes are always at the top.
2) **File merging:** While the heap contains more than one file, the program extracts the two smallest files and computes the cost of merging them by calculating the sum of their sizes. The merged file is then put back to the heap and the cost of the merging is added to the total cost.
3) **Solution output:** Once the heap contains only one file, the program prints the total cost of all merge operations.

Data structures:

1) **Priority queue (Min-heap):** Maintains the file sizes in ascending order for efficient extraction of the smallest files.

Time complexity:

1) **Heap initialization:** Inserting n elements into the heap takes:

$$O(n \log n)$$

2) **File merging:** Each merge operations consists of :
   a. Extracting the two smallest elements:

$$O(\log n)$$

   b. Inserting the merged file back into the heap:

$$O(\log n\ )$$

   For n files, there are *n-1* merge operations.

   Total complexity:

$$O((n-1)\log n)$$

3) **Overall complexity:**

$$O(n \log n)$$

Example:

Input data: 5,6,9,20,50,3,70,5,1,30

1) **Heap initialization:** Heap is initialized with the input data
2) **File merging:**
   - Merge 1 and 3: cost = 1 + 3 = 4. Total cost = 4. Heap: 4,5,5,6,9,20,30,50,70
   - Merge 4 and 5: cost = 4 + 5 = 9. Total cost = 13. Heap: 5,6,9,9,20,30,50,70
   - Merge 5 and 6: cost = 5 + 6 = 11. Total cost = 24. Heap: 9,9,11,20,30,50,70
   - Merge 9 and 9: cost = 9 + 9 = 18. Total cost = 42. Heap: 11,18,20,30,50,70
   - Merge 11 and 18: cost = 11 + 18 = 29. Total cost = 71. Heap: 20,29,30,50,70
   - Merge 20 and 29: cost = 20 + 29 = 49. Total cost = 120. Heap: 30,49,50,70
   - Merge 30 and 49: cost = 30 + 49 = 79. Total cost = 199. Heap: 50,70,79
   - Merge 50 and 70: cost = 50 + 70 = 120. Total cost = 319. Heap: 79,120
   - Merge 79 and 120: cost = 79 + 120 = 199. Total cost = 518. Final heap: 199
3) **Final result :**

- **Total cost = 518**

Input data: 58, 23, 91, 76, 4, 39, 87, 19, 64, 72

1) **Heap initialization:** Heap is initialized with the input data
2) **File merging:**
    - Merge 4 and 9: cost = 4 + 9 = 23. Total cost = 23. Heap: 23, 23, 39, 58, 54, 72, 76, 87, 91
    - Merge 23 and 23: cost = 23 + 23 = 46. Total cost = 69. Heap: 39, 46, 58, 64, 72, 76, 87,91
    - Merge 39 and 46: cost = 39 + 46 = 85. Total cost = 154. Heap: 58, 64, 72, 76, 85, 87, 91
    - Merge 58 and 64: cost = 58 + 64 = 122. Total cost = 276. Heap: 72, 76, 85, 87, 91, 122
    - Merge 72 and 76: cost = 72 + 76 = 148. Total cost = 424. Heap: 85, 87, 91, 122, 148
    - Merge 85 and 87: cost = 85 + 87 = 172. Total cost = 596. Heap: 91, 122, 148, 172
    - Merge 91 and 122: cost = 91 + 122 = 213. Total cost = 809. Heap: 148, 172, 213
    - Merge 148 and 172: cost = 148 + 172 = 320. Total cost = 1129. Heap: 213, 320
    - Merge 213 and 320: cost = 213 + 320 = 533. Total cost 1662. Final Heap: 533
3) **Final result :**
    - **Total cost = 1662**

4. Generating Huffman Codes

Problem description:

The program generates Huffman codes for a set of characters based on their frequencies. Two methods are used to create and analyze the codes:

1) **Canonical Huffman coding:** A structured approach to generate Huffman codes with consistent rules.
2) **Frequency-based approximation:** An approximate method to generate codes based on the probability of occurrence of characters.

The objective is to generate efficient binary codes for characters and to demonstrate encoding and decoding of messages using the generated codes. I will then compare the two approaches based on their performance and characteristics.

Algorithm overview:

Canonical Huffman coding:

1) **Determine code lengths:** The program simulates the Huffman process by merging the least frequent characters iteratively. The code length on the merged characters increases at each step.
2) **Assign binary codes:** The characters are sorted by their code length and lexicographical order and. Then, binary codes are assigned in canonical order using the calculated lengths.

Frequency-based approximation:

1) **Calculate probabilities:** Calculates the probability od each character based on its frequency using the formula: $F = \frac{frequency}{Total\ frequency}$
2) **Determine code lengths:** Calculate the code length for each character using the formula where *P* is the probability of the character.

$$L = \lceil -log2(P) \rceil$$

3) **Assign binary codes:** Sorts characters by their calculated lengths and lexicographical order. Then, it assigns binary codes in canonical order using the calculated lengths.
4) **Encoding and decoding:** When encoding, the program converts a string of characters into a binary string using the generated codes. When decoding, the program converts a binary string back into the original message using a reverse mapping.

Data structures:

1) **Character list:** Stores characters, their frequencies, code lengths, and binary codes.
2) **Binary code mapping:** A mapping between characters and their binary codes for encoding and a reverse mapping for decoding

Time complexity:

1) **Canonical Huffman coding:** Simulating the Huffman process due to repeated merging of least frequent characters takes:

$$O(n^2)$$

And for sorting binary codes it takes:

$$O(n \log n)$$

2) **Frequency based approximation:** Calculating the probabilities takes:

$$O(n)$$

Assigning binary codes for sorting takes:

$$O(n \log n)$$

3) **Overall complexity:**

Canonical Huffman coding:

$$O(n^2)$$

Frequency based approximation:

$$O(n \log n)$$

Example:

Character	Frequency
A	45
B	13
C	12
D	16
E	9
F	5

Canonical huffman coding:

1) **Code lengths:**

Character	Code length
A	1
B	3
C	3
D	3
E	4
F	4

2) **Binary codes:**

Character	Binary codes
A	0
B	100
C	101
D	110
E	1110
F	1111

Frequency-based approximation:

1) **Code lenghts and probability:**

Character	Probability	Code length
A	0.45	1
B	0.13	3
C	0.12	3
D	0.16	3
E	0.09	4
F	0.05	4

2) **Binary codes:**

Character	Binary code
A	0
B	100
C	101
D	110
E	1110
F	1111

Encoding and decoding example:

Input: “ABCDE“

Encoded message: “010011011110“

Deacoded message: “ACBDE“

5. Conclusion:

This documentation analyses the implementation of various algorithms. It includes Structure and the thought behind each algorithm as well as the resources used and time complexity of each algorithm. It gives thorough exapmles to each algorithms with inside of how the algorithm compute their data.