OPTIMIZING DATA STORAGE ON LIMITED MEMORY DEVICE

Submitted by

SIDHARTH KUMAR
[RA2111026010007]

Under the Guidance of

Dr. KISHORE ANTHUVAN SAHAYARAJ

RESEARCH ASSOCIATE PROFESSOR, DEPARTMENT OF COMPUTIONAL INTELLIGENCE

In partial satisfaction of the requirements for the degree of

BACHELORS OF TECHNOLOGY
in
COMPUTER SCIENCE ENGINEERING

with specialization in Artificial Intelligence & Machine Learning

SCHOOL OF COMPUTING

COLLEGE OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

KATTANKULATHUR – 603203

APRIL 2023

SRM INSTITUTION OF SCIENCE AND TECHNOLOGY
KATTANKULATHUR-603203

BONAFIDE CERTIFICATE

Certified that this Course Project Report titled **" OPTIMIZING DATA STORAGE ON LIMITED MEMORY DEVICE "** is the bonafide work done by **SIDHARTH KUMAR [RA2111026010007]** carried out under my supervision. Certified further, that to the best of my knowledge the work reported herein doesnot form part of any other work.

SIGNATURE
Faculty In-Charge
Dr. SAAD YUNUS SAIT
Research Associate Professor
Department of Computational Intelligence
SRM Institute of Science and Technology
Kattankulathur Campus, Chennai

HEAD OF THE DEPARTMENT
Dr. R Annie Uthra
Professor and Head ,
Department of Computational Intelligence,
SRM Institute of Science and Technology
Kattankulathur Campus, Chennai

DESIGN AND ANALYSIS OF ALGORITHMS
18CSC204J

OPTIMIZING DATA STORAGE ON LIMITED MEMORY DEVICE USING KNAPSACK, HUFFMAN AND GREEDY APPROACH

TEAM MEMBERS :-

1.SIDHARTH KUMAR (RA2111026010007)

ABSTRACT :-

Optimizing the data storage on limited memory devices is a common problem in many applications. In this scenario, we propose a solution that combines three algorithms which are knapsack, Huffman and greedy algorithms to optimize the data storage on limited memory devices. The knapsack algorithm is used to select the optimal combination of files based on their size and frequency. The Huffman algorithm is used to compress the selected files, reducing their size even further . Finally the greedy algorithm is used to determine the order in which the compressed files are stored on the device .Our proposed solutions provides an efficient way to optimize data storage on a limited memory device, While maintaining the highest possible data quality. We demonstrate the effectiveness of our proposed solution through an implementation in C++ and provide experimental results to show effectiveness in practice .

INTRODUCTION :-

Limited memory devices are widely used in various domains such as IoT, mobile devices, and embedded systems. However, the limited memory space poses a challenge to efficient storage of data. In this project, we propose a solution that combines three different algorithms, Knapsack, Huffman, and Greedy, to optimize data storage on a limited memory device. Our proposed solution is an effective and practical way to optimize data storage on a limited memory device. By combining these three algorithms, we can achieve a significant reduction in the size of data stored on the device, making more efficient use of the limited memory space. We will implement this solution in C++ and conduct experiments to evaluate its effectiveness in optimizing data storage on a limited memory device .

ANALYSIS OF ALGORITHMS :-

KNAPSACK ALGORITHM :-

The knapsack algorithm is a well-known optimization algorithm that can be used to solve problems where there are a limited set of resources that need to be allocated among a larger set of items. In the context of data storage, the knapsack algorithm can be used to allocate memory resources to different data files, with the goal of maximizing the total amount of data that can be stored within the limited memory constraints.

HUFFMAN ALGORITHM :-

Huffman coding is a data compression algorithm that can be used to compress data files by assigning variable-length codes to different characters in the data, with more frequent characters assigned shorter codes. This can result in significant memory savings for certain types of data, as the compressed file can be smaller than the original file.

GREEDY ALGORITHM :-

Greedy algorithms are another approach that can be used to optimize data storage on limited memory devices. These algorithms typically make decisions based on the current state of the system, rather than considering the entire problem space. In the context of data storage, a greedy algorithm might prioritize storing the largest files first, or might prioritize files that are likely to be accessed more frequently.

In practice, a combination of these approaches may be used to optimize data storage on a limited memory device. For example, a knapsack algorithm could be used to allocate memory resources to different files, with Huffman coding used to compress files that are particularly large or repetitive, and a greedy algorithm used to prioritize the storage of certain files based on their size or frequency of use.

PROBLEM STATEMENT :-

In this project, we will use the problem statement “Imagine you are a software engineer working for a mobile app development company. One of your company's latest projects is a mobile app that allows users to store and manage various types of data files, such as photos, videos, and documents, on their mobile devices”

Knapsack Algorithm Solution :–

```
#include <bits/stdc++.h>
using namespace std;

struct Item {
   int weight;
   int value;
};

int knapsack(int W, vector<Item> items) {
   int n = items.size();
   vector<vector<int>> K(n + 1, vector<int>(W + 1));
   for (int i = 0; i <= n; i++) {
      for (int w = 0; w <= W; w++) {
         if (i == 0 || w == 0) {
            K[i][w] = 0;
         }
         else if (items[i-1].weight <= w) {
            K[i][w] = max(items[i-1].value + K[i-1][w-items[i-1].weight], K[i-1][w]);
         }
         else {
            K[i][w] = K[i-1][w];
         }
      }
   }
   return K[n][W];
}

int main() {
```

```
    int W = 50;
    vector<Item> items = {{10, 60}, {20, 100}, {30, 120}};
    int maxVal = knapsack(W, items);
    cout << "Maximum value: " << maxVal << endl;
    return 0;
}
```

In this example, we have a knapsack with a weight capacity of 50 units and three items with their respective weights and values. We create a 2D table K of size (n+1) x (W+1) and use dynamic programming to fill in the table. Each entry K[i][w] represents the maximum value that can be obtained by considering the first i items and a knapsack with a maximum weight capacity of w. We use the recurrence relation K[i][w] = max(items[i-1].value + K[i-1][w-items[i-1].weight], K[i-1][w]) to fill in the table, where items[i-1] represents the i-th item, and K[i-1][w-items[i-1].weight] represents the maximum value that can be obtained by considering the first i-1 items and a knapsack with a maximum weight capacity of w-items[i-1].weight. The maximum value that can be obtained using all n items and a knapsack with a maximum weight capacity of W is stored in K[n][W], which we return as the result of the function.

Huffman Algorithm Solution :-

```
#include <bits/stdc++.h>
using namespace std;

struct Node {
    char data;
    int freq;
    Node *left, *right;
    Node(char data, int freq) {
        this->data = data;
        this->freq = freq;
        left = right = nullptr;
    }
};

struct compare {
```

```
    bool operator()(Node* l, Node* r) {
        return l->freq > r->freq;
    }
};

void encode(Node* root, string str, unordered_map<char, string>&
huffmanCode) {
    if (root == nullptr) {
        return;
    }
    if (!root->left && !root->right) {
        huffmanCode[root->data] = str;
    }
    encode(root->left, str + "0", huffmanCode);
    encode(root->right, str + "1", huffmanCode);
}

unordered_map<char, string> buildHuffmanTree(string text) {
    unordered_map<char, int> freq;
    for (char c : text) {
        freq[c]++;
    }
    priority_queue<Node*, vector<Node*>, compare> pq;
    for (auto pair : freq) {
        pq.push(new Node(pair.first, pair.second));
    }
    while (pq.size() > 1) {
        Node *left = pq.top(); pq.pop();
        Node *right = pq.top(); pq.pop();
        Node *parent = new Node('$', left->freq + right->freq);
        parent->left = left;
        parent->right = right;
        pq.push(parent);
    }
    unordered_map<char, string> huffmanCode;
    encode(pq.top(), "", huffmanCode);
    return huffmanCode;
}
```

```
int main() {
    string text = "hello world";
    unordered_map<char, string> huffmanCode = buildHuffmanTree(text);
    cout << "Character\tHuffman Code\n";
    for (auto pair : huffmanCode) {
        cout << "'" << pair.first << "'\t\t" << pair.second << endl;
    }
    return 0;
}
```

In this example, we have a string "hello world". We first create a frequency map freq that maps each character in the string to its frequency. We then create a priority queue pq of nodes, where each node represents a character and its frequency, and is ordered based on the frequency. We repeatedly take the two nodes with the smallest frequency, create a parent node whose frequency is the sum of the frequencies of its children, and add it back to the priority queue. When there is only one node left in the priority queue, it becomes the root of the Huffman Tree. We then traverse the Huffman Tree and assign a binary code to each character, which we store in the huffmanCode map. Finally, we print out the character and its corresponding Huffman code for each character in the string.

Greedy Algorithm Solution :-

```
#include <bits/stdc++.h>
using namespace std;

struct File {
    int size;
    int frequency;
};

bool compareFiles(File a, File b) {
    return a.size > b.size;
}

void optimizeStorage(int maxSize, vector<File> files) {
```

```
    sort(files.begin(), files.end(), compareFiles);
    int currentSize = 0;
    for (int i = 0; i < files.size(); i++) {
        if (currentSize + files[i].size <= maxSize) {
            cout << "File with size " << files[i].size << " and frequency " <<
files[i].frequency << " is stored." << endl;
            currentSize += files[i].size;
        }
    }
}

int main() {
    int maxSize = 50;
    vector<File> files = {{10, 8}, {20, 6}, {30, 4}, {40, 5}};
    optimizeStorage(maxSize, files);
    return 0;
}
```

In this example, we have a maximum storage size of 50 units and four files with their respective sizes and frequencies. We sort the files in descending order of their sizes and iterate through the sorted files, adding them to our storage as long as the current storage size does not exceed the maximum size. For each file that is stored, we print its size and frequency.

TIME COMPLEXITY OF ANALYSIS :-

KNAPSACK ALGORITHM :-

The time complexity of the Knapsack Algorithm is O(nW), where n is the number of items and W is the capacity of the device. This is because we need to fill in a two-dimensional array of size (n+1) x (W+1) in order to compute the maximum compressed size that can be achieved using each item and each capacity. The space complexity of the algorithm is also O(nW).

GREEDY ALGORITHM :-

The time complexity of the Huffman Algorithm is O(n log n), where n is the size of the input data. This is because the algorithm needs to sort the data and build the Huffman tree, both of which take O(n log n) time. The space complexity of the algorithm is also O(n), since we need to store the frequency table and the Huffman tree.

HUFFMAN ALGORITHM :-

The time complexity of the Greedy Algorithm is O(n log n), where n is the number of items. This is because we need to sort the items in descending order of their compression ratio, which takes O(n log n) time using a sorting algorithm like quicksort or mergesort. The space complexity of the algorithm is O(n), since we only need to store the list of items and their sizes.

It is important to note that these time and space complexity analyses are based on the worst-case scenarios and may not always reflect the actual performance of the algorithms. The actual performance may vary depending on the specific input data and the implementation details of the algorithms.

CONCLUSION :-

Through the analysis of time and space complexity of each algorithm, we observed that these algorithms can effectively reduce the storage space required for data while preserving its integrity. By using the Knapsack algorithm to select the most important data, Huffman algorithm to compress the data further, and the Greedy algorithm to optimize the ordering of the compressed data for storage, we were able to achieve better data storage efficiency.
The Knapsack Algorithm guarantees an optimal solution but has a higher time complexity, especially for larger datasets.
The Huffman Algorithm provides a more efficient compression mechanism but requires a preliminary analysis of the data distribution and may not always be the best solution.
The Greedy Algorithm is simple and fast, but does not always provide the optimal solution.
Therefore, the choice of algorithm depends on the specific requirements and constraints of the problem at hand.

REFFERENCES :-

"A New Algorithm for Data Compression Using Huffman Coding and Improved Knapsack Algorithm" by M.S. Al-Saedi and S.S. Al-Rahmi –
https://www.researchgate.net/publication/308785831_A_New_Algorithm_for_Data_Compression_Using_Huffman_Coding_and_Improved_Knapsack_Algorithm