CAPSTONE PROJECT

Find Minimum Time to Finish All Jobs Using Binary Search and Backtracking

D Teja
(192210626)
Department of Computer Science and Engineering
Saveetha School of Engineering, Saveetha Institute of Medical and Technical Sciences
Saveetha University, Chennai, Tamil Nadu, India Pincode:602105.

Dr R Dhanalakshmi,
Project guide, Corresponding Author, Department of Computer Science and Engineering,
Saveetha School of Engineering, Saveetha Institute of Medical and Technical Sciences,
Saveetha University, Chennai, Tamil Nadu, India. Pincode:602105.

PROBLEM STATEMENT :

You are given an integer array of jobs, where jobs[i] is the amount of time it takes to complete the ith job.
There are k workers that you can assign jobs to.
Each job should be assigned to exactly one worker.
The working time of a worker is the sum of the time it takes to complete all jobs assigned to them.
Your goal is to devise an optimal assignment such that the maximum working time of any worker is minimized.
Return the minimum possible maximum working time of any assignment.

Example 1:

Input: jobs = [1,2,4,7,8], k = 2
Output: 11
Explanation: Assign the jobs the following way:
Worker 1: 1, 2, 8 (working time = 1 + 2 + 8 = 11)
Worker 2: 4, 7 (working time = 4 + 7 = 11)
The maximum working time is 11.
Constraints: 1 <= k <= jobs.length <= 12 1 <= jobs[i] <= 107

ABSTRACT:

This project addresses the challenge of distributing a set of jobs among multiple workers in such a way that the maximum time taken by any worker is minimized. Each job, represented by an integer array, has a specific duration, and we are tasked with assigning these jobs to exactly k workers. The complexity of the problem arises from the need to minimize the maximum working time of any individual worker, which in turn optimizes the overall job distribution. The solution presented in this project utilizes a combination of binary search and backtracking techniques to achieve an efficient distribution of jobs. Binary search helps narrow down the possible values of the maximum working time, while backtracking ensures that we explore different job assignments to find the optimal configuration. This approach balances performance and accuracy, making it well-suited for scenarios involving job scheduling and resource allocation. The project demonstrates the practical application of algorithmic optimization in reducing workloads and improving efficiency in a multi-worker environment.

KEYWORDS: Job assignment, worker optimization, binary search, backtracking, workload distribution, resource allocation, job scheduling, algorithmic optimization, computational efficiency.

INTRODUCTION:

In many real-world scenarios, effective job assignment is crucial to maximizing productivity and minimizing workload imbalance among workers. This project tackles the problem of assigning a given set of jobs, each with a specific time requirement, to a limited number of workers in such a way that the maximum working time of any worker is minimized. The challenge lies in distributing the jobs as evenly as possible while ensuring that no single worker is disproportionately burdened. This type of optimization is not only relevant in manual labor settings but also in computational tasks, resource allocation, and scheduling problems.

The problem is formally defined by an integer array representing job durations and a fixed number of workers. The objective is to assign each job to exactly one worker in a manner that minimizes the maximum working time across all workers. This ensures a more efficient distribution of workload, reducing bottlenecks and improving overall performance.

To solve this problem, the project employs a combination of binary search and backtracking techniques. Binary search helps narrow down the possible maximum working times, while backtracking ensures all possible job assignments are explored to find the optimal solution. This hybrid approach provides both efficiency and accuracy in achieving the desired job distribution, making it a practical solution for applications in job scheduling, task distribution, and load balancing.

By leveraging these techniques, the project demonstrates how algorithmic approaches can solve complex optimization problems in a variety of domains, from workforce management to computational load distribution.

CODE:

```
#include <stdio.h>

#define MAX_JOBS 12
time exceeding 'maxTime'
int canAssignJobs(int *jobs, int jobsSize, int k, int *workers, int workerIdx, int maxTime) {
    if (workerIdx == jobsSize) {
        return 1;
    }

    for (int i = 0; i < k; i++) {
        if (workers[i] + jobs[workerIdx] <= maxTime) {
            workers[i] += jobs[workerIdx];
            if (canAssignJobs(jobs, jobsSize, k, workers, workerIdx + 1, maxTime)) {
                return 1;
            }
            workers[i] -= jobs[workerIdx];  // Backtrack
        }

        if (workers[i] == 0) {
            break;
        }
    }

    return 0;  // No valid assignment found
}

int findMinTime(int *jobs, int jobsSize, int k) {
    int low = 0, high = 0;

    for (int i = 0; i < jobsSize; i++) {
        if (jobs[i] > low) {
            low = jobs[i];
        }
        high += jobs[i];
    }

    int result = high;
    while (low <= high) {
        int mid = (low + high) / 2;

        int workers[MAX_JOBS] = {0};

        if (canAssignJobs(jobs, jobsSize, k, workers, 0, mid)) {
            result = mid;
            high = mid - 1;
        } else {
            low = mid + 1;
        }
    }

    return result;
}

int main() {
    int jobs[] = {1, 2, 4, 7, 8};
    int k = 2;
    int jobsSize = sizeof(jobs) / sizeof(jobs[0]);

    int minTime = findMinTime(jobs, jobsSize, k);
    printf("Minimum possible maximum working time: %d\n", minTime);

    return 0;
}
```

Explanation:

1. **canAssignJobs():** This function checks if it's possible to assign jobs such that no worker has a workload exceeding the current maxTime. It uses backtracking to try all possible assignments.
2. **findMinTime():** This function uses binary search to find the minimum possible maximum working time. The lower bound (low) is initialized to the largest job, and the upper bound (high) is the sum of all jobs.

3. **Backtracking and Binary Search:** The algorithm repeatedly tries a mid-value between the low and high and checks if it's a valid solution using backtracking. If valid, it tries smaller values to find the minimum.

RESULTS:

The jobs are assigned as follows:

- **Worker 1**: 1, 2, 8 (Total time = 1 + 2 + 8 = 11)
- **Worker 2**: 4, 7 (Total time = 4 + 7 = 11)

The maximum working time for both workers is 11, and since this is the smallest possible maximum working time that can be achieved, the output is 11.

Key Outcomes:

- **Optimized Job Distribution**: The algorithm successfully assigns jobs to workers in such a way that the maximum working time is minimized, ensuring an even distribution of workload.

- **Minimum Maximum Working Time**: For the given input, the minimum possible maximum working time was determined to be 11, meaning that no worker's total workload exceeds this value.

- **Efficient Solution with Binary Search and Backtracking**: The solution uses a combination of binary search and backtracking, which provides an optimal and efficient way to solve the problem within the given constraints.

- **Scalability for Small Inputs**: The algorithm works effectively for smaller job arrays (up to 12 jobs) and up to k workers, ensuring that it fits within the problem's constraints.

- **Practical Application**: This method can be applied in real-world scenarios where task distribution and load balancing among workers or resources is critical.

Performance Metrics:

Time Complexity:

- The solution uses a combination of binary search and backtracking.
- The time complexity is approximately O(k^n), where `n` is the number of jobs and `k` is the number of workers. Backtracking takes into account all possible assignments, but binary search significantly reduces the search space for the maximum working time, making it more efficient than brute force.

Space Complexity:

- The space complexity is O(n) due to the storage of job assignments during the backtracking recursion. Here, `n` represents the number of jobs being tracked.

Scalability:

- The algorithm performs efficiently for smaller job arrays, handling up to 12 jobs as per the constraints. For larger inputs, however, the complexity could increase significantly, and performance may degrade.

Optimization:

- By using binary search and backtracking, the solution avoids unnecessary calculations, improving efficiency compared to a brute-force approach. The method balances optimality and performance, particularly for small to medium-sized inputs.

Execution Time:

- For inputs like `jobs = [1, 2, 4, 7, 8]` and `k = 2`, the algorithm runs efficiently, solving the problem in milliseconds. The binary search helps narrow down the solution space quickly.

Memory Usage:

- The memory usage remains moderate, as it primarily stores the job assignments and recursion stack during the backtracking process.

DISCUSSION:

The problem of minimizing the maximum working time when assigning jobs to workers is solved using a combination of binary search and backtracking. The binary search narrows down the possible maximum working times, while backtracking verifies feasible assignments. This method efficiently finds the optimal job distribution within given constraints, avoiding the need for exhaustive search. While the approach performs well for up to 12 jobs, larger datasets may require additional optimizations or alternative methods. The solution effectively balances workloads among workers, ensuring fairness and efficiency in job assignments.

Limitations:

- **Scalability**: The current solution is efficient for up to 12 jobs but may struggle with significantly larger datasets due to exponential growth in possible job assignments.
- **Complexity**: While binary search and backtracking are efficient for smaller inputs, they can become computationally intensive as the number of jobs or workers increases, requiring more advanced optimization techniques.
- **Memory Usage**: The recursive backtracking approach can lead to high memory consumption, especially for larger job arrays or when deep recursion occurs.
- **Assumptions**: The solution assumes that jobs and workers are static. For dynamic scenarios where jobs or workers are added or removed frequently, additional mechanisms would be needed to maintain optimal assignment.
- **Fixed Constraints**: The method is tailored for problems with up to 12 jobs. For scenarios beyond these constraints, modifications or entirely different strategies might be necessary.
- **Real-Time Applications**: The approach may not be suitable for real-time job assignment systems where rapid reallocation of jobs is required.

FUTURE SCOPE:

To enhance the solution's applicability and performance, several future developments could be pursued. Expanding the algorithm to handle larger datasets beyond the current constraint of up to 12 jobs could involve leveraging more sophisticated optimization techniques and heuristics to improve scalability. Implementing dynamic job and worker management features would allow the solution to adapt to real-time changes in job assignments and worker availability, making it more versatile for applications in rapidly evolving environments. Additionally, integrating parallel processing or multi-threading could significantly reduce computational time, especially for large-scale job distributions. Exploring alternative algorithms and data structures, such as advanced graph-based methods or hybrid approaches combining different optimization strategies, could further refine performance. Incorporating machine learning techniques for predictive job allocation based on historical data could offer insights and enhance the efficiency of job assignments. Finally, extending the solution for integration with real-time systems and big data frameworks could broaden its use cases, enabling it to tackle complex job scheduling challenges in diverse industries.

CONCLUSION:

The project effectively addresses the problem of minimizing the maximum working time across multiple workers when assigning a set of jobs. By utilizing a combination of binary search and backtracking algorithms, the solution achieves an optimal job distribution with a minimized maximum working time. This approach not only ensures that the workload is balanced among the workers but also handles the constraints efficiently, even with the given upper limit of job numbers. The implementation demonstrates that advanced algorithmic techniques can significantly improve performance over naive methods, making it suitable for practical applications where job assignment and workload balancing are critical. The successful application of these techniques highlights their potential in real-world scenarios, paving the way for further enhancements and adaptations to handle larger and more complex job scheduling problems.