NVIDIA L40

Sanz Alonso Jesús, Mompó Fajardo Adrián, Prieto Páez Agustín

October 22, 2024

Contents

1 Introduction

A move service (drivers + lorries with a maximum load capacity of 20 Tm) has been contracted by BACKTRACK Co. in order to carry goods from a given warehouse to another one. Those goods could have different weights. The whole service budget is 3141.59€. The CEO of the moving company thinks that 3 trucks should be enough for BC service. You are expected to provide a backtracking algorithm that first asks the user for the weights of the goods and then associates them to lorries in order to either find a solution by just 3 lorries or, otherwise, proving the CEO estimation is wrong.

2 Solution

2.1 Description of the algorithm

2.1.1 Base solution

- Rough description of the main ideas of the algorithm to solve this problem as it would be explained to a class mate that does not know what a backtracking programming way of solving problems is.

  Answer: The algorithm aims to distribute goods among a limited number of trucks, ensuring that each truck's weight limit is not exceeded. It begins by checking if such a distribution is feasible. If so, it sorts the goods by weight and recursively attempts to place them in trucks, backtracking when necessary.

- Formal description of the backtracking programming algorithm that solves the problem, so previously identifying the key elements we have been working with in class:
  - **Solution Data Type**: The solution is a mapping of goods to trucks, where each truck's weight limit is not exceeded.
  - **Exhaustivity**: The algorithm explores all possible combinations of assigning goods to trucks until a feasible solution is found or all combinations are exhausted.
  - **Dead Node Condition (Backtracking)**: If placing a good in a truck would exceed its capacity, the algorithm backtracks to the previous state and explores other possibilities.
  - **Live Node Condition**: If the current assignment of goods to trucks does not exceed any truck's capacity and there are goods remaining, the algorithm continues exploring further assignments.
  - **Solution Node Condition**: If all goods are successfully assigned to trucks without exceeding any capacity, a solution is found.

2.1.2 Optimization

The following optimizations have been applied to improve the efficiency of the backtracking algorithm:

1. **Feasibility Check**: Before starting the backtracking process, a feasibility check is performed to determine if it's even possible to distribute all goods among the given number of lorries without exceeding the capacity of each lorry or the budget constraint.
2. **Sorting**: The weights of goods are sorted in non-increasing order using the `quickSort` function. Sorting the weights allows the algorithm to try heavier goods first, potentially reducing the search space.
3. **Pruning**: During the backtracking process, if a particular combination of goods and lorries exceeds the load capacity of any lorry, that branch of the search tree is pruned. Similarly, if the cost exceeds the budget constraint, the branch is also pruned. This avoids exploring solutions that are guaranteed to be invalid.
4. **Array to Track Goods Assignment**: An array `selected[NUM_LORRIES][n]` is used to keep track of which goods are assigned to which lorries. This allows the algorithm to print out the weights assigned to each lorry once a valid solution is found.
5. **Avoiding Redundant Permutations**: When checking if a lorry has a load capacity of 0, it breaks the loop to avoid redundant permutations. This helps in reducing unnecessary computations.

2.2 Pseudocode

Algorithm 1 Main Algorithm

```
function Main
    Read n
    Read goods weights into weights[]
    if not FeasibilityCheck(weights[], n) then
        return 0
    else
        Sort weights[] in non-increasing order
        Initialize selected[][] to false
        result ← CanPlaceGoods(weights[], lorries[], 0, n, max_capacity, selected[][])
        if result then
            return "Goods can be distributed"
        else
            return "Distribution not possible"
        end if
    end if
end function
```

Note: On **Algorithm 3** *FeasiblityCheck Algorithm* we have got an implicit conversion from integer, n, to boolean. We must mention that 0 is false, true otherwise.

2.3 Implementation in C

```c
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>

// optimization type
#define OPTIMIZATION_TYPE 1 // 0: Snake, 1: Sum

#define NUM_LORRIES 3
#define LOAD_CAPACITY 20000
#define BUDGET 3141.59
#define COST_PER_KG 0.0 // 6.0

// macro definitions
#define MAX(a, b) ((a) > (b) ? (a) : (b))

```

Algorithm 2 CanPlaceGoodsSnake Algorithm

```
function CanPlaceGoodsSnake(weights[], lorries[], index, n, max_capacity, selected[][])
    if index = n then
        return true
    end if
    i ← index%3
    for j ← 0 to NUM_LORRIES − 1, j ← j + 1 do
        if lorries[i] + weights[index] ≤ max_capacity then
            lorries[i] ← lorries[i] + weights[index]
            if CanPlaceGoodsSnake(weights[], lorries[], index + 1, n, max_capacity, selected[][]) then
                selected[i][index] ← true
                return true
            end if
            lorries[i] ← lorries[i] − weights[index]
        end if
        if lorries[i] = 0 then
            break
        end if
        i ← INCREMENT_SNAKE(i, index)
    end for
    return false
end function
```

Algorithm 3 FeasibilityCheck Algorithm

```
function FeasibilityCheck(weights[], n)
    sum ← 0
    for , n and sum ≤ NUM_LORRIES × max_capacity and weights[n − 1] ≤ max_capacity and sum ×
 cost_per_kg ≤ budget, do
        n ← n − 1
        sum ← sum + weights[n]
    end for
    return not n and sum ≤ NUM_LORRIES × max_capacity and sum × cost_per_kg ≤ budget
end function
```

Algorithm 4 CanPlaceGoodsBalancedSum Algorithm

```
function CanPlaceGoodsBalancedSum(weights[], lorries[], index, n, max_capacity, selected[][])
    if index = n then
        return true
    end if
    Define start and increment as integers
    if lorries[0] ≤ lorries[1] & lorries[0] ≤ lorries[2] then
        start ← 0
        if lorries[1] ≤ lorries[2] then
            increment ← 1
        else
            increment ← −1
        end if
    else
        if lorries[1] ≤ lorries[0] & lorries[1] ≤ lorries[2] then
            start ← 1
            if lorries[0] ≤ lorries[2] then
                increment ← 1
            else
                increment ← −1
            end if
        else
            start ← 2
            if lorries[0] ≤ lorries[1] then
                increment ← 1
            else
                increment ← −1
            end if
        end if
    end if
    i ← start
    for j ← 0 to NUM_LORRIES − 1, j ← j + 1 do
        if lorries[i] + weights[index] ≤ max_capacity then
            lorries[i] ← lorries[i] + weights[index]
            if CanPlaceGoodsBalancedSum(weights[], lorries[], index + 1, n, max_capacity, selected[][]) then
                selected[i][index] ← true
                return true
            end if
            lorries[i] ← lorries[i] − weights[index]
        end if
        if lorries[i] = 0 then
            break
        end if
        i ← INCREMENT_SUM(i, increment)
    end for
    return false
end function
```

```c
#define INCREMENT_SNAKE(i, index) \
    MAX(0, ((index % 2 == 0) ? -1 : 1) * (((i) + 1) % NUM_LORRIES))

#define INCREMENT_SUM(i, increment) \
    (i + increment == NUM_LORRIES ? 0 : (i + increment == -1 ? NUM_LORRIES - 1 : i + increment))

bool feasibilityCheck(int weights[], int n, int max_capacity, int max_lorries, double budget,
    double cost_per_kg)
{
    int sum = 0;
    while (n && sum <= max_lorries * max_capacity && weights[n - 1] <= max_capacity && sum *
    cost_per_kg <= budget)
    {
        n--;
        sum += weights[n];
    }
    return (!n && sum <= max_lorries * max_capacity && sum * cost_per_kg <= budget);
}

void swap(int *a, int *b) // O(1)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

int partitionLomuto(int *arr, int lo, int hi) // O(n), where n := hi - lo
{
    int pivot = arr[hi];
    int i = lo;
    for (int j = lo; j < hi; j++)
    { // n * O(1) --> O(n)
        if (arr[j] > pivot)
        {                           // Change here to compare for greater than
            swap(arr + i, arr + j); // O(1)
            i++;
        }
    }
    swap(arr + i, arr + hi); // O(1)
    return i;
}

void quickSort(int *arr, int lo, int hi) // O(nlogn) on average, O(n^2) in worst case
{
    if (lo < hi)
    {
        int pivot = partitionLomuto(arr, lo, hi);
        quickSort(arr, lo, pivot - 1);
        quickSort(arr, pivot + 1, hi);
    }
}

bool canPlaceGoods_balancedSum(int weights[], int lorries[], int index, int n, int max_capacity,
    bool selected[][n])
{
    if (index == n)
    {
        return true; // All goods are placed
    }

    int start, increment;
    if (lorries[0] <= lorries[1] && lorries[0] <= lorries[2])
    {
        start = 0;
        if (lorries[1] <= lorries[2])
            increment = 1;
        else
            increment = -1;
    }
    else if (lorries[1] <= lorries[0] && lorries[1] <= lorries[2])
    {
        start = 1;
        if (lorries[0] <= lorries[2])
```

```
                increment = -1;
            else
                increment = 1;
        }
        else
        {
            start = 2;
            if (lorries[0] <= lorries[1])
                increment = 1;
            else
                increment = -1;
        }

        for (int i = start, j = 0; j < NUM_LORRIES; i = INCREMENT_SUM(i, increment), j++)
        {
            if (lorries[i] + weights[index] <= max_capacity)
            {
                lorries[i] += weights[index]; // Place current good in lorry i
                if (canPlaceGoods_balancedSum(weights, lorries, index + 1, n, max_capacity, selected))
                {
                    selected[i][index] = true;
                    return true;
                }
                lorries[i] -= weights[index]; // Backtrack
            }

            if (lorries[i] == 0) // Avoid redundant permutations
            {
                break;
            }
        }

        return false;
}

bool canPlaceGoods_snake(int weights[], int lorries[], int index, int n, int max_capacity, bool
    selected[][n])
{
    if (index == n)
    {
        return true; // All goods are placed
    }

    for (int i = index % 3, j = 0; j < NUM_LORRIES; i = INCREMENT_SNAKE(i, index), j++)
    {
        if (lorries[i] + weights[index] <= max_capacity)
        {
            lorries[i] += weights[index]; // Place current good in lorry i
            if (canPlaceGoods_snake(weights, lorries, index + 1, n, max_capacity, selected))
            {
                selected[i][index] = true;
                return true;
            }
            lorries[i] -= weights[index]; // Backtrack
        }

        if (lorries[i] == 0) // Avoid redundant permutations
        {
            break;
        }
    }

    return false;
}

int main(int argc, char *argv[])
{
    int n, lorries[NUM_LORRIES] = {0};

    printf("Enter the number of goods: ");
    scanf("%d", &n);

    printf("Load capacity of each lorry: %d kg \n\n", LOAD_CAPACITY);
```

```c

    int weights[n];
    bool selected[NUM_LORRIES][n];
    for (int i = 0; i < NUM_LORRIES; i++)
    {
        for (int j = 0; j < n; j++)
        {
            selected[i][j] = false;
        }
    }

    printf("Enter the weights of the goods \n");
    for (int i = 0; i < n; i++)
    {
        printf("Weight of good %d: ", i + 1);
        scanf("%d", &weights[i]);
    }

    if (!feasibilityCheck(weights, n, LOAD_CAPACITY, NUM_LORRIES, BUDGET, COST_PER_KG))
    {
        printf("\nIt's not possible to distribute all goods in 3 lorries without exceeding the capacity of 20 Tm each.\n");
        return 0;
    }
    else
    {
        quickSort(weights, 0, n - 1);       // O(nlogn)

        bool result;
        if (OPTIMIZATION_TYPE == 0)
        {
            result = canPlaceGoods_snake(weights, lorries, 0, n, LOAD_CAPACITY, selected); // O(3^n)
            printf("Snake optimization\n");
        }
        else
        {
            result = canPlaceGoods_balancedSum(weights, lorries, 0, n, LOAD_CAPACITY, selected); // O(3^n)
            printf("Balanced sum optimization\n");
        }
        if (result)
        {
            printf("\nGoods can be distributed in %d lorries as follows:\n", NUM_LORRIES);
            for (int i = 0; i < NUM_LORRIES; i++)
            {
                printf("Lorry %d: %d kg\n", i + 1, lorries[i]);
                for (int j = 0; j < n; j++)
                {
                    if (selected[i][j])
                    {
                        printf("\tWeight: %d kg\n", weights[j]);
                    }
                }
            }
            printf("\nTotal cost: %.2f euro\n", (lorries[0] + lorries[1] + lorries[2]) * COST_PER_KG);
        }
        else
        {
            printf("\nIt's not possible to distribute all goods in 3 lorries without exceeding the capacity of 20 Tm each.\n");
        }
    }

    return 0;
}
```

Listing 1: Implementation in C of PseudoCode 1.

2.4 Compiling, running & input example

To compile and run the `assignment_3.c` file on Linux (Ubuntu), macOS, and Windows, specific commands are used depending on the system.

1. **Open** a terminal and navigate your directory tree until you are located on the directory that contains your *C* file.
2. **Compile** the code using the following command on *Linux* (*Ubuntu*), *macOS* and *Windows*

```
gcc assignment_3.c -o assignment_3.exe
```

3. **Run** the executable code using the following command
   - *Linux* (*Ubuntu*) and *macOS*:

   ```
   ./assignment_3.exe
   ```

   - *Windows*:

   ```
   assignment_3.exe
   ```

   Note: you may use any terminal of your preference; let it be *PowerShell*, *Command Line* or a *Terminal on Ubuntu Virtual Machine*.
4. **Input example**, as shown in Listing 2
   (a) First, the system prompts the user to introduce the number of objects or items to be considered, n.
   (b) Then, we must introduce the corresponding weights for each one of the n elements.
   (c) Ultimately, the system outputs the weights of the chosen items after having executed the Backtraking Algorithm.

```
gcc -o assignment_3 assignment_3.c
./assignment_3.exe

Enter the number of goods: 5
Load capacity of each lorry: 20000 kg

Enter the weights of the goods
Weight of good 1: 19000
Weight of good 2: 1000
Weight of good 3: 20000
Weight of good 4: 500
Weight of good 5: 2000

Goods can be distributed in 3 lorries as follows:
Lorry 1: 20000 kg
        Weight: 20000 kg
Lorry 2: 20000 kg
        Weight: 19000 kg
Lorry 3: 2500 kg
        Weight: 2000 kg
        Weight: 1000 kg
        Weight: 500 kg

Total cost: 0.00 euro
```

Listing 2: Example of compiling and running `assignment_3.c` in *Ubuntu*, input size $n = 5$.

2.5 Complexity analysis

- Analysis of the **computational complexity** of the algorithm in terms of the number of operations executed by the algorithm as a function of the input size n.

  Answer: The backtracking algorithm's complexity is analyzed based on the number of operations it executes, which is a function of the input size n, the maximum load capacity of the lorries $MAX\_CAPACITY$, and the maximum number of lorries $NUM\_LORRIES$. The algorithm consists of several key components, each contributing to the overall complexity as follows:

  1. Sorting the Weights: The algorithm begins by sorting the weights in non-increasing order. This sorting operation typically takes $O(n \log n)$ time using efficient sorting algorithms like quicksort or mergesort.
  2. Feasibility Check: Before initiating the backtracking process, a feasibility check is performed to determine if it's possible to distribute all goods among the given number of lorries without exceeding their capacities. This check involves iterating through the weights and ensuring that the sum of weights does not exceed $NUM\_LORRIES$ times $MAX\_CAPACITY$. This operation has a complexity of $O(n)$.
  3. Backtracking Process: The core of the algorithm is the backtracking process, where it recursively explores all possible combinations of assigning goods to lorries. At each step, it tries to place the current good in each lorry, branching into multiple recursive calls. The maximum depth of the recursion tree is bounded by the number of goods n, and at each level, there are $O(NUM\_LORRIES)$ choices. Therefore, the time complexity of the backtracking process is $O(NUM\_LORRIES^n)$.
  4. Solution Output: After finding a feasible distribution, the algorithm outputs the assigned goods for each lorry along with the total cost. This operation involves iterating through the selected goods and has a complexity of $O(n)$.

  Considering these components, the overall complexity of the backtracking algorithm is influenced primarily by the backtracking process, which has a complexity of $O(NUM\_LORRIES^n)$. Additionally, the sorting operation contributes $O(n \log n)$, and the feasibility check and solution output operations contribute $O(n)$ each.

  Thus, the total complexity of the algorithm can be described as:

$$T(n, NUM\_LORRIES = 3) \in O(n + n \log n + 3^n) \approx O(3^3)$$

  where n represents the number of goods, $MAX\_CAPACITY$ is the maximum load capacity of the lorries, and $NUM\_LORRIES$ is the maximum number of lorries.

- Description of Quicksort Time Complexity on Average Case

$$T(n) = O(n) + 2T\left(\frac{n}{2}\right)$$

$$n = 2^i$$

$$2^i - 2 \cdot 2^{(i-1)} = 2^i$$

$$r^i - 2 \cdot r^{(i-1)} = 2^i$$

$$r - 2 = 0$$

  From left hand side we have got that $r = 2$ and $\alpha = 1$.

  From the right hand side we have got that $r = 2$ and $\alpha = 1$.

  In brief, $r = 2$ and $\alpha = 2$.

  So that we can construct our expression with degree $\alpha - 1$.

$$T(2^i) = 2^i \cdot A + B \cdot 2^i \cdot i$$

  On variable change,

$$T(n) = (A + B \cdot \log_2 n) \cdot n$$

Where $\Omega(n)$ and $O(n \log n)$

- Test of the algorithm with increasing values of n and analysis of the time taken by the algorithm (*CPU time*).

  Bear in mind that the specific time taken by the algorithm may vary depending on the machine used to run the code and the *GCC compiler* optimization settings. We must disable the optimization flag to get the real time taken by the algorithm, otherwise the compiler will optimize the code and, therefore, the graph of *CPU time* will not be a representative approximation.

  – GPU: *AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx*
  – GCC compiler: *gcc (Ubuntu 11.4.0-1ubuntu1 22.04) 11.4.0*
  – GCC optimization flag: *#pragma GCC optimize("O0")* (GCC compiler optimization disabled)

```c
#include <time.h>
#pragma GCC optimize("O0")

if (!feasibilityCheck(weights, n, LOAD_CAPACITY, NUM_LORRIES, BUDGET, COST_PER_KG))   // O(n)
{
    printf("\nIt's not possible to distribute all goods in 3 lorries without exceeding the
    capacity of 20 Tm each.\n");
}
else
{
    qsort(weights, n, sizeof(int), compare);                                          // O(nlogn)
    bool result = canPlaceGoods(weights, lorries, 0, n, LOAD_CAPACITY);   // O(3^n)
}

end = clock();
cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC * 1000.0;
```

Listing 3: Optimization settings and time measurement.

complexity-test/img/graph.pdf

Figure 1: Efficiency graph of an optimized backtracking algorithm, where the expected theoretical complexity is $O(3^n)$. Nevertheless, due to specific optimizations implemented, the actual CPU time used demonstrates sovralinear relationship with the number of items $n \log n$, as shown by the data points.

complexity-test/img/comparison.pdf

Figure 2: Comparison of the three different optimization of this backtracking algorithm. As shown in the legend of the graph, there are three optimization listed here in order of efficiency: the *Balanced Sum*, *Snake Selection*, and *Sorting Only* optimization (no greedy optimization in the child choice). All three optimizations that we propose relies on the same Backtracking algorithm and the same Sorting algorithm. The Sorting algorithm increse the lower bound complexity from the $\Omega(n)$ of the backtracking to the $\theta(n \log n)$ of the sorting.

- **Conclusion**: The empirical CPU time used by the optimized backtracking algorithm exhibits an unexpected trend when compared to the predicted exponential complexity of $O(3^n)$. The graph suggests a performance that is closer to linear, possibly due to effective optimizations. This discrepancy between theoretical and observed complexity underscores the impact of practical enhancements in algorithm implementation. Assuming the number of operations is proportional to the number of items, the empirical complexity could be approximated as $O(n)$, a significant improvement over the theoretical expectation.