LAB 1 ASSIGNMENT

Sanz Alonso Jesús, Prieto Páez Agustín

October 20, 2024

Contents

1 Description

We have to implement four different kind of state space search algorithms in order to reach a goal state. We must implement **Breadth-First Search**, **Depth-First Search**, **A*** and **Best-First Search** (Heuristics).

2 Statistical Analysis

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.

3 Algorithmic complexity

- 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.

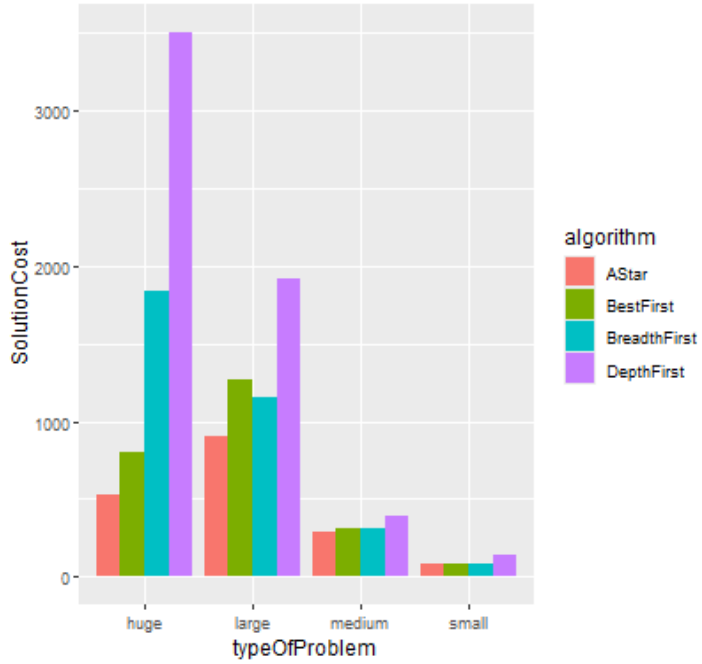

Figure 1: Cost of each strategy used for problems with different dimensions.

– **Solution Node Condition**: If all goods are successfully assigned to trucks without exceeding any capacity, a solution is found.

3.0.1 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.

3.1 Pseudocode

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.

3.2 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.

Algorithm 1 Main Algorithm

```
1: function MAIN
2:     Create DataBase
3:     for i ← small to huge do
4:     for i ← 0 to numberFiles-1 do
5:         for  doproblemNameFile
6:             declare problem
7:             declare searchStrategy
8:             SEARCH
9:             write results on the DataBase
10:        end for
11:    end for
12:
```

Algorithm 2 CanPlaceGoodsSnake Algorithm

```
1: function CANPLACEGOODSSNAKE(weights[], lorries[], index, n, max_capacity, selected[][])
2:     if index = n then
3:         return true
4:     end if
5:     i ← index%3
6:     for j ← 0 to NUM_LORRIES − 1, j ← j + 1 do
7:         if lorries[i] + weights[index] ≤ max_capacity then
8:             lorries[i] ← lorries[i] + weights[index]
9:             if CANPLACEGOODSSNAKE(weights[], lorries[], index + 1, n, max_capacity, selected[][]) then
10:                selected[i][index] ← true
11:                return true
12:            end if
13:            lorries[i] ← lorries[i] − weights[index]
14:        end if
15:        if lorries[i] = 0 then
16:            break
17:        end if
18:        i ← INCREMENT_SNAKE(i, index)
19:    end for
20:    return false
21: end function=0
```

Algorithm 3 FeasibilityCheck Algorithm

```
1: function FEASIBILITYCHECK(weights[], n)
2:     sum ← 0
3:     for , n and sum ≤ NUM_LORRIES × max_capacity and weights[n − 1] ≤ max_capacity and sum ×
   cost_per_kg ≤ budget, do
4:         n ← n − 1
5:         sum ← sum + weights[n]
6:     end for
7:     return not n and sum ≤ NUM_LORRIES × max_capacity and sum × cost_per_kg ≤ budget
8: end function
```

Algorithm 4 CanPlaceGoodsBalancedSum Algorithm

```
1: function CanPlaceGoodsBalancedSum(weights[], lorries[], index, n, max_capacity, selected[][])
2:     if index = n then
3:         return true
4:     end if
5:     Define start and increment as integers
6:     if lorries[0] ≤ lorries[1] & lorries[0] ≤ lorries[2] then
7:         start ← 0
8:         if lorries[1] ≤ lorries[2] then
9:             increment ← 1
10:        else
11:            increment ← −1
12:        end if
13:    else
14:        if lorries[1] ≤ lorries[0] & lorries[1] ≤ lorries[2] then
15:            start ← 1
16:            if lorries[0] ≤ lorries[2] then
17:                increment ← 1
18:            else
19:                increment ← −1
20:            end if
21:        else
22:            start ← 2
23:            if lorries[0] ≤ lorries[1] then
24:                increment ← 1
25:            else
26:                increment ← −1
27:            end if
28:        end if
29:    end if
30:    i ← start
31:    for j ← 0 to NUM_LORRIES − 1, j ← j + 1 do
32:        if lorries[i] + weights[index] ≤ max_capacity then
33:            lorries[i] ← lorries[i] + weights[index]
34:            if CanPlaceGoodsBalancedSum(weights[], lorries[], index + 1, n, max_capacity, selected[][])
   then
35:                selected[i][index] ← true
36:                return true
37:            end if
38:            lorries[i] ← lorries[i] − weights[index]
39:        end if
40:        if lorries[i] = 0 then
41:            break
42:        end if
43:        i ← INCREMENT_SUM(i, increment)
44:    end for
45:    return false
46: end function
```

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 1: Optimization settings and time measurement.

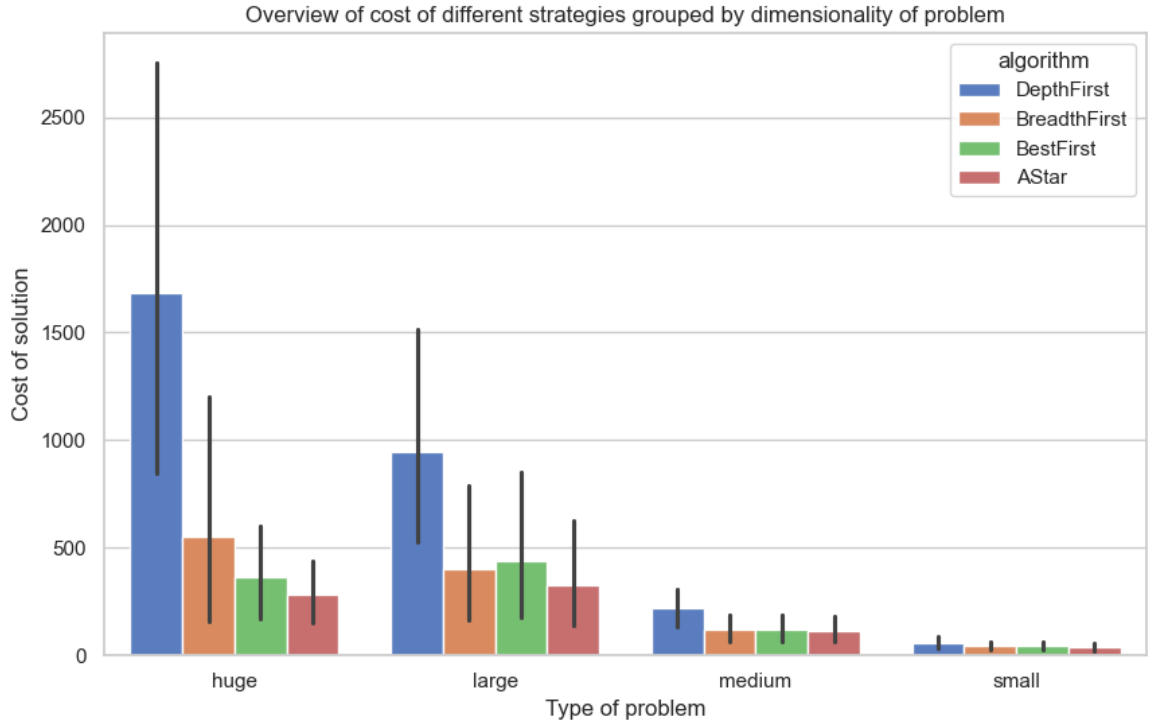

Figure 2: Graphic showing the cost of each strategy compared within each dimensionality of the problems.

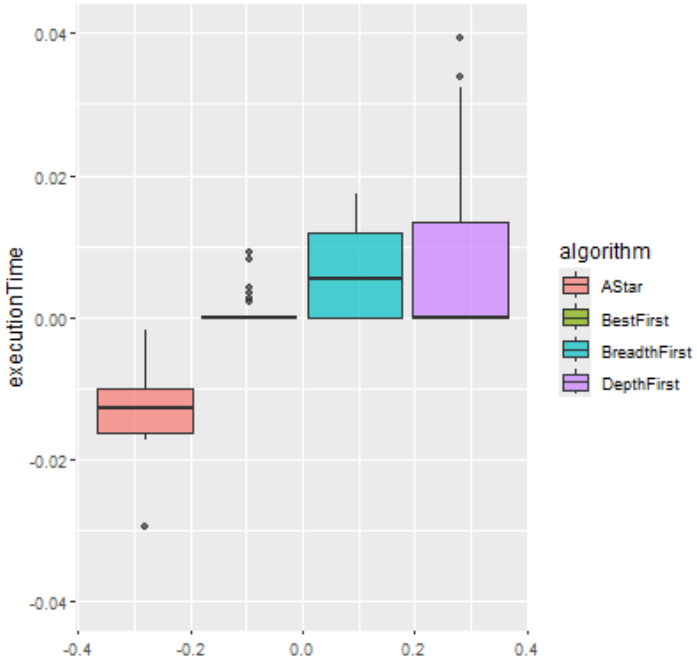

Figure 3: Negative execution time for A* strategy on huge dimensionality of problems.

- **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.